SYS

A Higher Level Parallel Programming Environment

Shakil Waiz Ahmed

YALEU/DCS/RR-1043
July 1994

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

A Higher Level Parallel Programming Environment

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Shakil Waiz Ahmed
July 1994

Abstract

A Higher Level Parallel Programming Environment

Shakil Waiz Ahmed
Yale University
July 1994

The Linda Program Builder is a higher-level programming environment that supports the design and development of parallel software. It is a window-oriented, menu-based system which provides coordination frameworks for program construction. It handles many of the bookkeeping details of parallel programming, allowing the user to concentrate on the computational aspects of the code. The LPB maintains a program-describing database which feeds information to the compiler for optimization, to a visualizer for enhanced program visualization, and potentially to other tools in the environment. The templates or coordination frameworks themselves can be custom-constructed by invoking a template-building template.

The ideas of the LPB are not restricted to Linda or even to parallel programming. Many of the ideas yield benefits in other environments. In fact, the LPB captures the idea of an "open" or "dynamic" preprocessor as an alternative to new programming languages.

In the name of God, Most Gracious, Most Merciful.

Read! In the name of thy Lord and Cherisher, who created —
Created man, out of a (mere) clot of congealed blood:
Read! And thy Lord is Most Bountiful,
He Who taught (the use of) the pen,
Taught man that which he knew not.

— The Holy Qur'an (96:1-5)

Acknowledgments

Although I officially have only one advisor, in reality I had two. Dr. David Gelernter is my official advisor, a tremendous source of inspiration and ideas. Dr. Nicholas Carriero is my unofficial advisor, an endless source of advice on technical issues. Nick patiently read through several drafts of this thesis and suggested numerous changes. I owe my sincere thanks to both David and Nick. I must also thank the other two members of my thesis committee, Dr. Suresh Jagganathan and Dr. Martin Schultz. Suresh has been helpful with advice at various stages of the project.

My former officemate, Dr. David Kaminsky (currently at IBM/Research Triangle Park) read through the very first draft and suggested changes which were incorporated into this thesis. Dr. Robert Bjornson (currently at Scientific Computing Associates, New Haven) read parts of the document and also had useful suggestions.

I could never have come this far without the support and encouragement from my parents, Dr. Jasimuddin Ahmed and Mrs. Sakina Ahmed. They instilled in me the desire to learn, to explore, and never ceased to believe in me. I know they've wanted this as much as I have and I dedicate this thesis to them.

My wife, Faria, was most patient while I worked on the final stages of my thesis. We got married during the most crucial stages of my work and she has been very understanding and supportive all the way.

I must thank my brothers (Shabbir and Sajjad) for the moral support through the years. My friends and officemates Susanne Hupfer and David Kaminsky made the countless days at work a lot more pleasant. My thanks also to all my friends (especially from the Islamic Student Association and the South Asia Society) who made graduate school such a fulfilling experience.

This research is supported by National Science Foundation grant CCR-8657615, by the Air Force Office of Scientific Research under grant number AFOSR-91-0098, and by Scientific Computing Associates, New Haven. Linda is a registered trademark of Scientific Computing Associates, New Haven, CT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Conventional (sequential) programmers have traditionally had a variety of tools at their disposal to aid in software development. With the increased demand for parallel programming comes a need for tools that provide support for development of parallel software. While there are a number of schools of thought on how to best express parallelism, a growing number of programmers are choosing explicitly parallel languages. Such languages allow users to explicitly express parallelism and thus take advantage of inherently parallel algorithms.

Given that explicitly parallel programming is gaining in popularity, how do we make the programming easier for programmers? The transition from sequential to parallel programming may be difficult for some users, and others will expect to find software development tools in a parallel programming environment. Both these problems can be addressed by a higher-level tool which eases parallel programming and provides parallel programming guidance to users. In particular, the tool should provide an integrated environment which guides the user through program construction; it should provide other supporting tools (such as an optimizing compiler and debugger); and it should be extensible.

For environments that use compiled languages, there are four key phases for program construction: coding, compiling, debugging and performance tuning. The higher-level tool could guide users in several different ways, but a few features should characterize it and cover the four phases mentioned above. These features should include: (1) the ability to capture programming experience and provide guidance based on this experience, and (2) the ability to use information from the construction phase to optimize the compiler and enhance program visualization.

Capturing programming experience requires identifying repeated program patterns. As more parallel programs are developed, some techniques and some kinds of algorithms will surface again and again. We can partition these into classes of programs

and add to these as new techniques and classes of algorithms are developed for programmer communities with specific needs. It would be useful to capture some of this programming experience and package it in a way that other programmers could use as a guideline, possibly even as some sort of "instant-program" package requiring only a few ingredients from the programmer to cook up a complete program. Hence, we need a framework which will offer users different paradigms that, upon selection, will lead the user through program construction. The same framework also should allow users to define guidelines and packages for future users, giving the system the adaptability to be customizable to different needs and biases. At the same time, the frameworks should not constrain the user. Traditionally, *structure editors*[1] have placed users in syntactic strait jackets, not allowing them to deviate from structured input. This approach can backfire; the objective should be to aid users, not to force them to adopt methodologies or programming styles they may not like. Hence, the parallel program building tool should be flexible in the sense that a user should be able to bypass guidelines at any point and enlist help from the system only when desired.

To enhance the environment as a whole, the guidance framework should be integrated with other tools. A programming environment should provide debugging aids to a user. Parallel debuggers by nature tend to be more complex than their sequential counterparts. To simplify their appearance and use, the debugging interface should track the level of abstractions that the user employs. If, for example, while interacting with the guiding system the user decides to use a particular distributed data structure, this information could be reflected in the feedback which the debugger provides. A visual debugger could have different representations for the different kinds of distributed data structures or processes that are used. The same applies to visual system for performance tuning. Turn the visualizer into a runtime system (most parallel program animations run on trace information) and the benefits to the programmer are quite significant.

The other important tool in a programming environment is the compiler. A compiler for an explicitly parallel programming language will recognize certain parallel constructs and translate these into the appropriate object code. Can we assist the optimizer in the compilation phase? As in the case of the debugging system, it may be possible to use some information from the code construction phase to optimize the compiler-generated code.

In summary, we want to construct a parallel programming environment which will guide the user through code development, capture programming experience, provide visualization and debugging aids, optimize the code, and also be flexible and adaptable.

[1] Structure editors require users to manipulate program structures and generally do not deal with free text

1.2 The Linda Program Builder

The Linda Program Builder (LPB) [AG92] [ACG94] was designed and built to address the needs discussed in the previous section. It is a higher-level programming environment that aids in the design and development of parallel software. The LPB is an Epoch-based[2], menu-driven, user-friendly system that supports incremental development of explicitly parallel C-Linda programs.

Epoch itself runs under X-windows and thus the LPB environment, by extension, runs under X-windows. The LPB environment is menu-driven, but allows the full flexibility of Epoch (Emacs) in editing all files. Several windows are open at all times, offering command menus and information on the developing program's current status.

With the LPB, programs can be constructed in one of three basic ways: (1) programs can be constructed interactively starting from scratch; (2) existing programs can be parallelized using LPB commands to insert parallelization code; and (3) programs can be freely constructed using conventional editing with perhaps an occasional supporting LPB command. The first option requires guidance from the LPB throughout program construction. The second option allows users to invoke a coordination framework and insert segments from an existing program. This feature would be particularly attractive to sites with sequential programs that need to be parallelized. Finally, the third option allows the user to freely edit and use the LPB only when desired.

How does the LPB assist program construction? It captures coordination frameworks for parallel programming. A coordination framework defines how the threads of parallel computation are tied together. There are concrete real world analogies: a soccer team follows a game plan specified by the coach. The game plan defines how the different soccer players will interact with each other to arrive at one common objective: to score goals and not let any in. Similarly, a coordination framework for a parallel program specifies how the various threads of computation are working towards a common objective. The LPB provides coordination frameworks to a user which are then used to construct parallel programs.

Coordination frameworks in the LPB are built in C-Linda, but nothing about the "parallel program builder" approach restricts it to Linda *per se*. By "program builder", we mean a tool which captures idiomatic use of a particular language, and guides the user through program development by offering higher-level constructs that are transformed into some base language. The sort of tool we describe might be built in the context of any explicitly parallel programming environment. On the other hand, the choice of Linda isn't arbitrary either. Linda is "perhaps the best-known parallel-processing language available today."[3]

Coordination frameworks take the form of *templates* in the LPB. Templates are program skeletons that look like hypertext documents. Users incrementally expand

[2]Epoch is a multi-window version of emacs developed by S. Kaplan of the University of Illinois, Urbana

[3]Digital Review Oct. 21, 1991

templates by clicking on buttons. Clicking on menu options paves the way for additional actions. An experienced programmer may choose to bypass many of the point-and-click facilities of the LPB. This is in sharp contrast to some related editing systems.

There have been many template-editor predecessors to the LPB, notably the Cornell Program Synthesizer[RT89a], but on the whole, they impose rigid frameworks on the programmer. Requiring the programmer to follow an imposed template guarantees syntactic correctness, but may cramp a creative programmer's flexibility. Expressivity is compromised because the user cannot develop any segments of code in his own style. The LPB offers similar features to the Cornell Program Synthesizer, but doesn't impose them. Consequently, the LPB cannot guarantee syntactic correctness for the program as a whole.

There is a tradeoff between expressivity and guaranteed syntactic correctness. To guarantee syntactic correctness, the system has to be in charge of the code under construction. Consequently, the system has to maintain a strong grip on any code the user develops. Structure editors such as those generated by the Cornell Program Synthesizer, require users to follow a strict framework under which the system and the user follow well-defined roles: the system constructs the expressions; the user fills in placeholders for expression segments. Clearly, this restricts expressivity. The LPB's goal is to serve as an aid to the user, not to be restrictive. By allowing coding flexibility, however, the system cannot maintain control on the portions of code that the user develops, and thus cannot guarantee syntactic correctness. The LPB serves as an apprentice that is consulted when needed. This allows the user to be creative in his code development, but also allows him to use as much help as he wants or needs. By sacrificing guaranteed syntactic correctness, we thus gain in expressivity and flexibility.

The most important features of the LPB are its support of templates, *distributed data structures*, *high-level operations,* and its construction of a *program database.* While templates represent entire program structures and direct control flow, the LPB also provides support at a data structure level. It supports constructing and manipulating distributed data structures of different types. High-level operations provide abstractions to a user which get transformed into the base language before compilation. Finally, the program database contains program-describing information which is acquired as the program is constructed. It forms the backbone of the LPB system and is the medium through which the LPB interacts with other tools in the environment.

By using the templates and other supporting features of the LPB during program construction, the user effectively makes semantic statements. This semantic information could be as broad as a "master-worker" coordination framework or as specific as the intention behind a particular operation. For example, the operation could be as simple as incrementing the value of a counter shared amongst processes. The LPB notes this semantic information and not only uses it during program construction, but passes it to other tools in the environment. In particular, the compiler receives this information and uses it to optimize the code. Similarly, the visualizer reads the information and enhances its display to reflect semantic information.

The LPB accomplishes the optimizations and visualizations mentioned above without sacrificing portability of the LPB's output. Both the Linda compiler and Tuplescope had to be modified to accommodate the semantic information from the LPB, but this does not limit portability of the code that the LPB generates. In fact, the LPB always generates an output file which can be passed on to any users with a standard Linda compiler or normal Tuplescope. The semantic information is passed to the modified tools separately and hence does not affect portability of the end product.

1.3 Thesis

Although motivated by concerns of parallel programming, the LPB addresses issues that are broader. In fact, the dynamic and adaptable nature of a program builder makes it an attractive alternative to new programming languages. Hence, taking all the issues into account, the thesis becomes:

> To show that a higher-level program building tool can aid the construction of parallel programs, improve their efficiency, enhance their visualization and run-time monitoring, and also serve as an "open" or "dynamic" preprocessor alternative to new programming languages.

1.4 Outline

The next chapter describes the LPB itself and discusses how to construct programs with the LPB. It goes through an example of constructing a database search program with the LPB and discusses some of the available templates. Following that, Chapter 3 describes how the LPB supports distributed data structures and high level operations. The chapter introduces the *abstractions* feature, and describes the program database.

Chapter 4 deals with the LPB's interaction with other tools. It discusses how information is passed to the compiler and the visualizer is used. Results are presented for some optimizations based on this information.

Flexibility and extensibility are the topics of Chapter 5. It discusses how the user can extend the LPB environment. In particular, this involves using the template-building template to construct templates which other users can use. The chapter goes through an example of constructing a master-worker template.

Chapter 6 presents the LPB as a "dynamic" preprocessor alternative to new programming languages. The argument relates program builders to "dynamic" preprocessors and discusses their advantages over continuously developing new programming languages to accommodate changing methodologies and needs.

Chapter 7 reports on some of the limitations of the current system due to the fact that the Linda compiler and Tuplescope preceded the LPB and thus were not built with the LPB interface in mind. The chapter presents a preliminary redesign of the compiler and Tuplescope for better interaction with the LPB.

To demonstrate how the LPB framework applies to other environments, Chapter 8 describes an environment that layers an object-oriented methodology on top of C. The LPB infrastructure was used to rapidly prototype the environment.

Finally, Chapter 9 discusses related work from different angles and Chapter 10 presents conclusions.

Chapter 2

Program Construction with the LPB

2.1 Templates

Coordination frameworks are expressed through *templates* in the LPB. A template is a program skeleton for a particular paradigm that serves as a blueprint for program construction. During program construction, users of a template expand code placeholders within the template to incrementally construct code segments. A template is different from high-level operations such as distributed data structure operations or abstractions (Chapter 3): templates are designed to direct control flow and yield complete programs; high-level operations and abstractions only yield partial program fragments.

Templates are designed to capture programming experience and pass the experience on to users who may construct programs which are similar in nature to those which a template encapsulates. Programming experience is classified according to repeated coding patterns or *methodologies*. A methodology for a class of programs is a framework which outlines the common characteristics of that class — additional pieces of information characterize individual programs. The template framework in the LPB can be adapted to various methodologies — template designers can organize the placeholders to yield a specific pattern of code development. The templates which are part of the existing LPB system, however, are all focused on explicitly parallel programming paradigms.

A number of templates supporting different parallel programming methodologies have been implemented for the LPB. The master-worker template, for example, guides the user through constructing parallel programs that use the master-worker coordination framework. This involves a single master process and several identical worker processes working on different segments of the problem. The Piranha template is somewhat similar in nature — it helps the user write adaptively parallel programs [Kam94].

The data parallel template, on the other hand, supports the owner-computes style of programming by providing filter, log and merge routines. A set of global reduction operators completes the data parallel package. The LPB interface for data parallelism thus smoothes the way for Linda programmers who use the data parallel model.

How does a user interact with templates? Templates are hypertext-like documents. Upon selection of a template, a user is presented with a skeleton of a program. The goal is to expand the skeleton by providing information to the LPB such that the latter can construct a complete program. A skeleton has a number of highlighted *buttons* in it. Clicking on a button with a mouse results in that particular button being expanded. An expansion could result in code replacing the button, potentially yielding further buttons. Alternatively, the expansion could result in an interactive session where the user is prompted for information. This interactive session will eventually yield code to replace the expanded button. Again, this code could contain yet more buttons. The buttons may be ordered in priority — i.e. some buttons may be dependent on information that can only be acquired by first expanding other buttons. Eventually, when all buttons have been expanded and the user has provided all the information that he was prompted for, the LPB will have constructed a complete program.

A programmer can choose to follow a template all the way through, but he is free to leave this framework whenever he wants and to return when necessary. There is a tradeoff here between flexibility and potential consistency problems. Information on the program being constructed is maintained by a *program-describing database* (see section 3.5) Should a programmer decide to bypass the LPB and choose to independently code program fragments that the LPB expects to be in control of, an inconsistency between the program database and the program itself might be created. This could happen in one of two ways: (1) The programmer may write code which the program-describing database does not know of, but expects to know of (e.g. the database expects to keep track of all distributed data structure manipulations); or (2) The programmer may delete code which the LPB has generated. The LPB will warn the programmer when either of these two cases happen, but cannot overrule what he has done. If the programmer's solo effort is syntactically and semantically correct, the resulting code will compile correctly, but will not benefit from enhancements that the LPB can provide (such as performance optimizations or visualizer enhancements – see Chapter 4).

How does the LPB prompt the user for information? Interactive sessions typically consist of menus or input windows with directions on how to specify the necessary information. Menus can be both context-dependent and context-independent. The former case appears in situations where the user is specifically asked to identify a selection. The latter case involves menus which are always available during an LPB session.

There are a number of different context-independent menus. The `Global Menu` contains a list of templates along with some basic commands that will save a file or parse it for inconsistences with the program database. The `Buffer Menu` lists all

currently open modules, and clicking on a particular module name causes the edit window to display that module. Even though a particular module is being displayed, some actions such as modifying a shared data structure may affect all references to that data structure scattered across all open modules. When this happens, all changes in the appropriate modules are highlighted. These highlights will be visible when the user displays the corresponding modules.

Linda uses tuples to coordinate parallel programs, and the `Tuple Menu` displays all currently defined tuples [CG89] [1]. The LPB keeps track of all references to the various tuples in a program. For every tuple, there are only certain operations that can be performed on it. Hence, when a tuple is selected, the context-dependent `Tuple Commands` menu lists only those operations which are permissible.

The list of permissible operations for a tuple will depend on the nature of the tuple. The LPB supports various distributed data structures and different tuples are dedicated towards supporting particular kinds of data structures. A tuple may be used as a shared counter, a shared variable, counting semaphore or simply as a plain Linda tuple. It may also be used as part of a distributed queue or shared linked list. Consequently, the list of commands permissible for a particular tuple will change according to the data structure that the tuple represents. If the currently selected tuple is used as a counter, for example, the only operations allowable on that tuple are increment and decrement operations and the commands menu reflects that.

The menus described above offer users support for building parallel program segments. These supporting features augment the services provided by templates. Users can benefit from this if they wish support to develop parallel code independently of templates, or in addition to code that an expanded template yields.

Finally, a context dependent information window completes the list of persistent windows in the LPB environment. This window is basically a comment window for tuples. Every time a tuple is selected in the `Tuples` menu, the information window will reveal the available information on that tuple. The default information in this window is type information on the fields of the tuple. However, this window is freely editable and the user can modify it to include his comments on a particular tuple. The information will always be carried with a tuple and helps users keep track of what tuples are used for.

To demonstrate the workings of how to construct a program with the LPB, an example follows. In particular, a master-worker template is used to construct a database search program.

[1] Strictly speaking, we should be speaking of tuple signatures of **in/out** patterns. "Tuples" actually refer to run-time instances of a particular tuple signature. We use the term "tuple" very loosely in the context of the LPB to refer to tuple signatures.

2.1.1 Using the master-worker template

Experience indicates that one of the most common parallel programming paradigms is the *master-worker* paradigm [Car87]. A master process starts a number of worker processes, generates a number of task descriptors, waits for results for the tasks, and having obtained results, tells the worker processes to stop. Each worker executes an infinite loop where it obtains a task descriptor, computes the task and generates the result for the task. In its simplest form, a master-worker coordination framework looks like the following:

```
master() {
    create workers
    create task descriptions
    gather results
    stop workers
}

worker() {
    while (1) {
        get task description
        break out of loop if told to stop
        compute task
        output result
    }
}
```

Many variations of this basic skeleton are possible. We could, for example, generate a worker for every task, and have the worker routines themselves evaluate to the result. We could have the master generate all the task descriptors itself, or we could have it generate just one, and each worker, upon receipt of a task descriptor, would then decide whether to generate further descriptors. All these variations and others are supported by the master-worker template.

To understand how templates in the LPB work, it will be helpful to examine an example in detail. We use the master-worker template to construct a simple database search program. Suppose we would like to search a large telephone directory for all entries matching a particular string. This string could be any combination of numbers and letters. Consequently, there could be multiple matches to our string. Our directory has no particular ordering and hence we will search through the whole directory file. This is a good case for the master-worker paradigm. The master process reads in the entire database file and breaks it into records. The data describing the chunks constitute the tasks descriptors for the workers. The workers search through the chunks for the target string and return result tuples indicating when matches are found.

When the LPB environment starts, an editing window and a number of menus appear. We select the `Master-Worker Template` option of the `Global Menu` to begin our program. This causes the corresponding template to appear in our editing window. Figure 2.1 shows the initial master-worker template. The template begins with only two buttons, one for the master routine and one for the worker routine. Clicking on either of these buttons will cause it to expand. A logical first step would be to read the phone database and generate the task descriptors, so we click on the `master routine` button. The code expands into what is shown in Figure 2.2.

The master routine becomes the `real_main` function in Linda. The LPB assumes that the number of workers spawned for this problem will be passed as a command line argument (but this can be changed by the user). The generated code in the `real_main` function checks the argument line and spawns off the worker processes by generating the eval loop. Two buttons appear in the expanded code. The first one of these is the `out tasks` button which will yield code to generate the task descriptors and gather the results. Following this, the `out poison` button will yield code that tells the workers to stop. Finally, the rest of the code in the function cleans up the finished worker processes.

The next step is to expand the `out tasks` button by clicking on it. This yields what is shown in Figure 2.3. To specify task descriptors, the user has to choose one of several different task distribution models. Consequently, a menu of model choices appears from which the user is asked to select a model. A message window appears simultaneously with the menu — this is a help window with information on how to proceed whenever input is required from the user. The message window disappears if it is clicked on. There are four possible model choices for the task descriptors generating phase.

One possible choice is a generic task bag such that the tasks descriptors are all dumped into an unordered bag from which workers grab task descriptors. If, however, the tasks must be completed in some order, then a task queue is the model to choose. In this case, the user can choose between single sources and multiple sources for the queue. Finally, the user may wish to monitor the number of tasks in the bag. If the number of tasks is potentially very large, then the bag could get filled with tasks such that memory limits are exceeded. The solution to this problem is to monitor the number of tasks in the bag, and every time a threshold value is reached, gather results until the number of tasks has dropped to a low level threshold. This is known as the "watermarked bag" approach and since we are considering a potentially very large telephone directory in our example, this is the model we choose. This results in the next input request which appears in figure 2.4.

The user is now asked to specify the loop boundaries for the task generation loop. In some cases, the number of tasks is identical to the number of workers. In fact, if the user chooses not to answer the boundaries question, this is what the LPB chooses to default to and the loop is generated accordingly. In the case of our example, however, we choose a different approach.

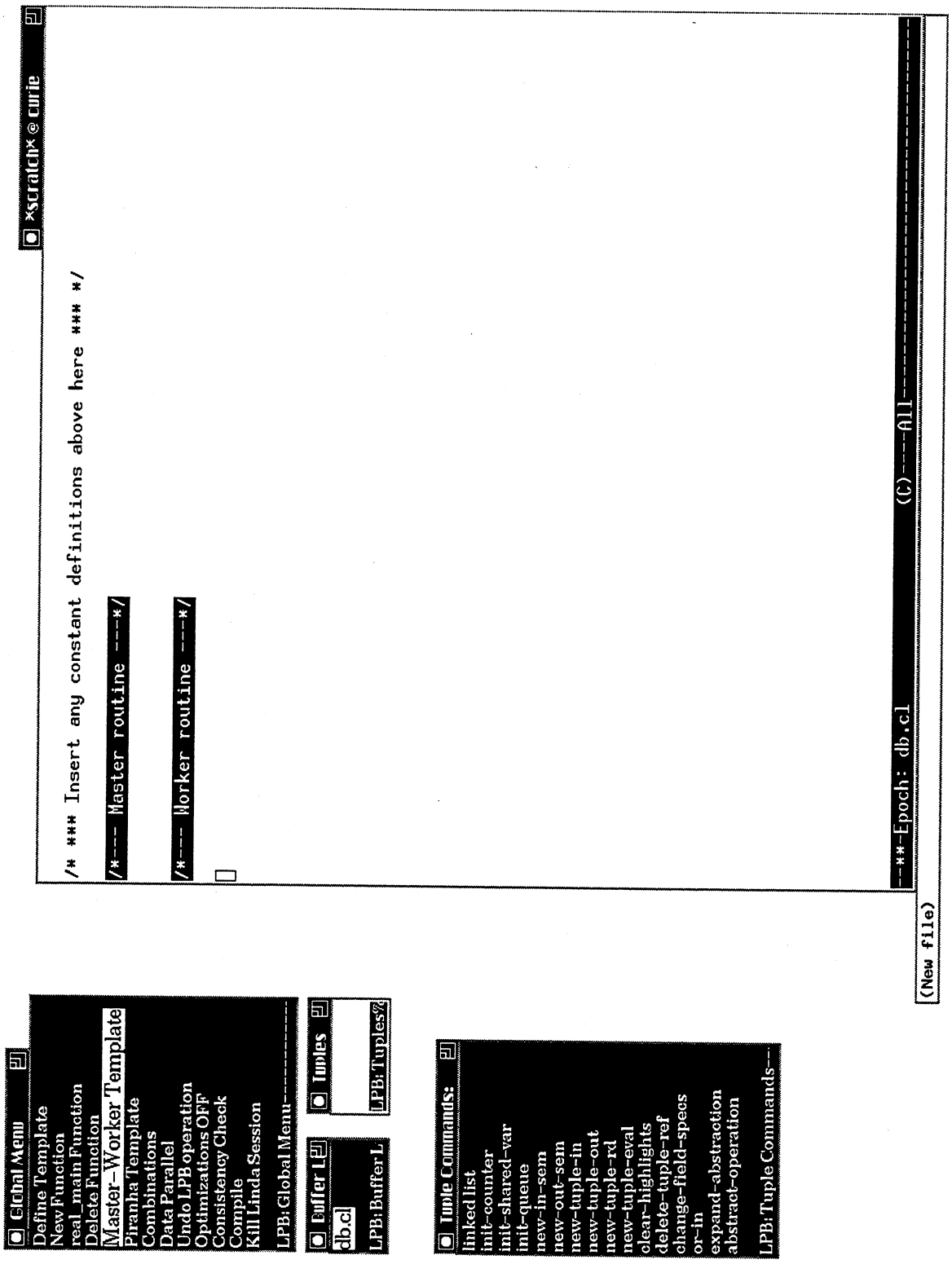

Figure 2.1: Master-Worker Template

Figure 2.2: Master-Worker: Master Routine

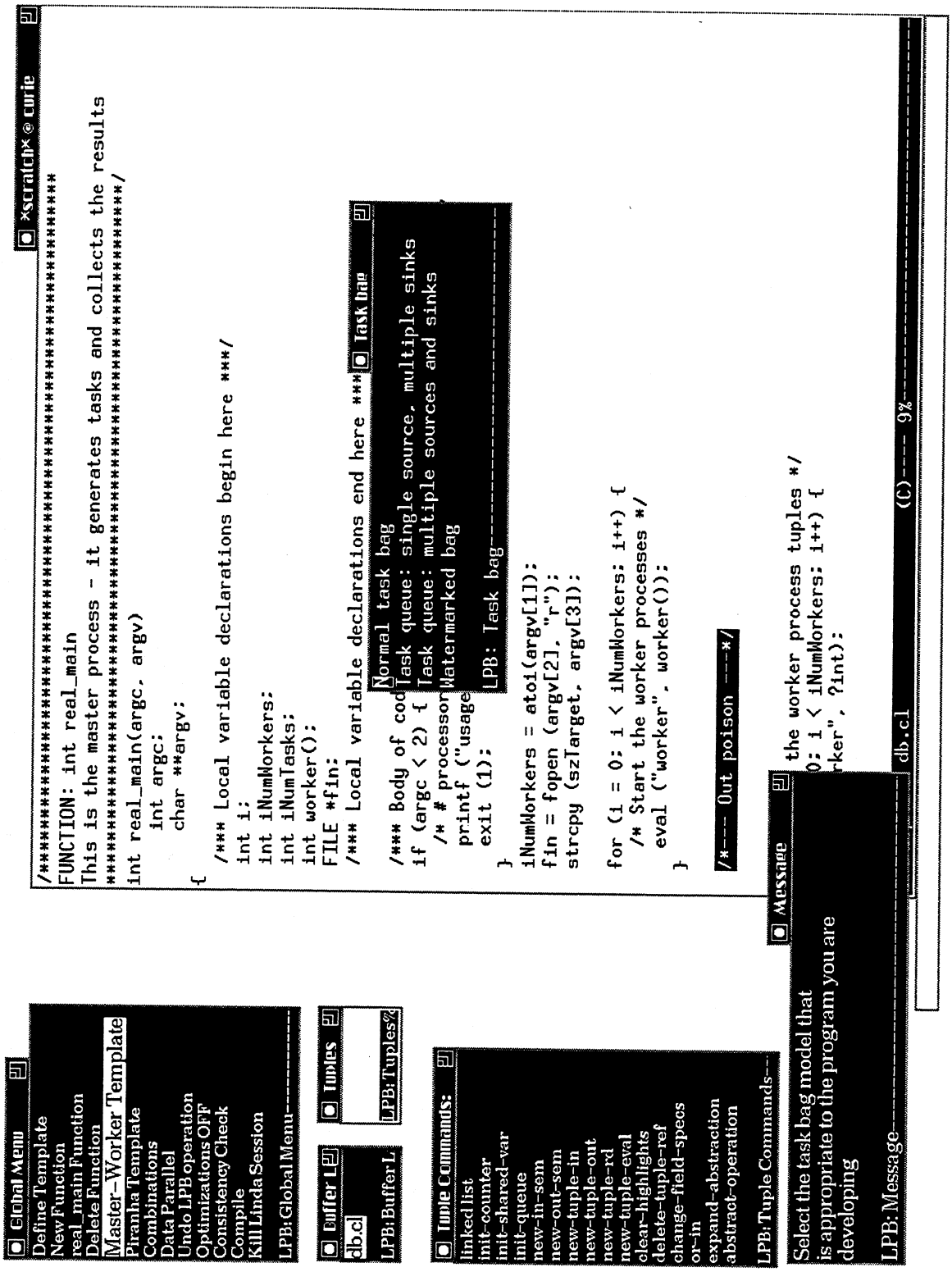

Figure 2.3: Master-Worker: Generating Tasks

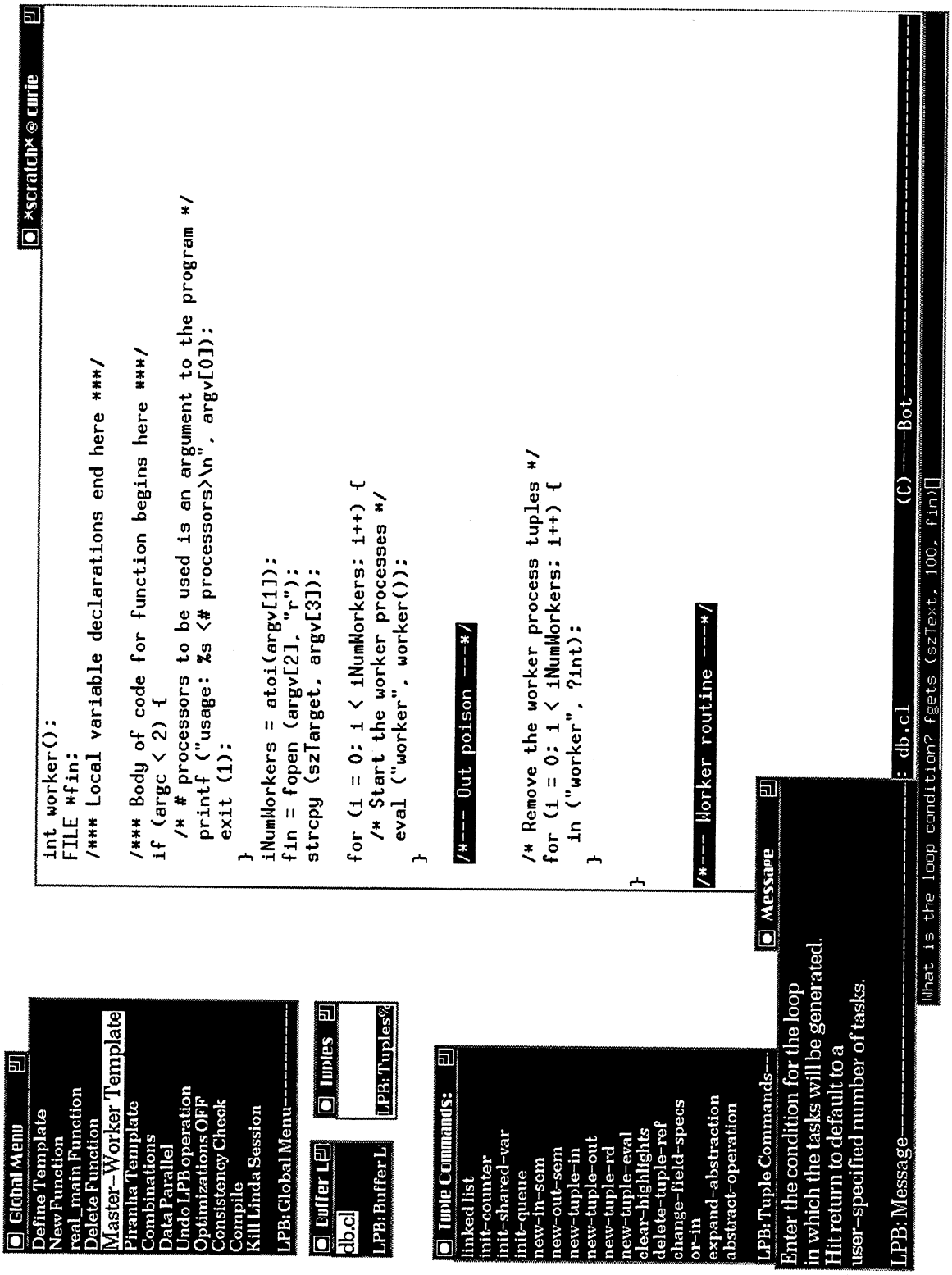

Figure 2.4: Master-Worker: Task Loop Condition

Our telephone file is organized such that we have one record per line, consisting of a name, a number and an address. Thus, it makes sense to issue one task descriptor for every line in the file. Workers will grab these task descriptors and work on one line at a time. Our loop condition therefore becomes dependent on whether the end of the input file has been reached. We express this as a file read operation. Consequently, we also need to open the file and here the flexibility of the LPB comes into play — we simply declare a file variable in our program and issue the appropriate file opening call early in our function. Having inserted the code for opening the file and having specified the loop condition, we now proceed to define the task descriptors themselves (figure 2.5).

The master-worker template asks the user to define the fields which constitute the task descriptor tuple. An input window appears titled "Tuple: task", and the user is asked to declare the variables for the fields of this tuple. We will describe each task by a line number and the line text. We thus declare an integer and an array of characters to hold the number and string respectively. Armed with these declarations, the LPB now has a definition for task descriptors and proceeds by asking the user to identify the *poison variable* (Figure 2.6).

A task descriptor needs to convey one of two things to a worker process: it will either give the worker data to work on, or it will tell the worker to stop computation and break out of its infinite loop. The latter is done through something known as a *poison pill.* A poison pill is simply an invalid value assigned to some field of the tuple which is recognized by the worker as being "impossible". The user is asked to identify which of the fields of the task descriptor carries the poison variable — i.e. the user has to identify the variable in the tuple that will be used to convey the poison pill. A menu appears listing the variables which have been declared for the task descriptor tuple. The user clicks on one of these (the line number variable is convenient in this example), and is prompted for a value. -999 is clearly an invalid line number and so it adequate for our poison pill value. Having specified this, the next step is to define the result tuples (Figure 2.7).

After the master is done generating task descriptors, it waits for results. These results also have to be conveyed through tuples which need to be defined. Much as in the case of defining tasks, an input window appears titled "Tuple: result", prompting the user for variable declarations for the fields of the result tuple. The results have to convey those lines in which matches were found. We will thus use a line number and a text string once again. In fact, in the case of lines where matches are found, the results will simply be returning the same values as the task descriptors. It thus makes sense to declare the exact same variables as in the case of the task descriptor tuple. Now that our result tuple has been defined, the loop for task generation and result gathering is complete and the appropriate code has been inserted into the program (Figure 2.8).

Several important things have happened. Two tuples have been defined as is apparent in the `Tuples` menu. The LPB has generated a loop which generates the task descriptors, monitors how many are put out into tuple space and gathers results as

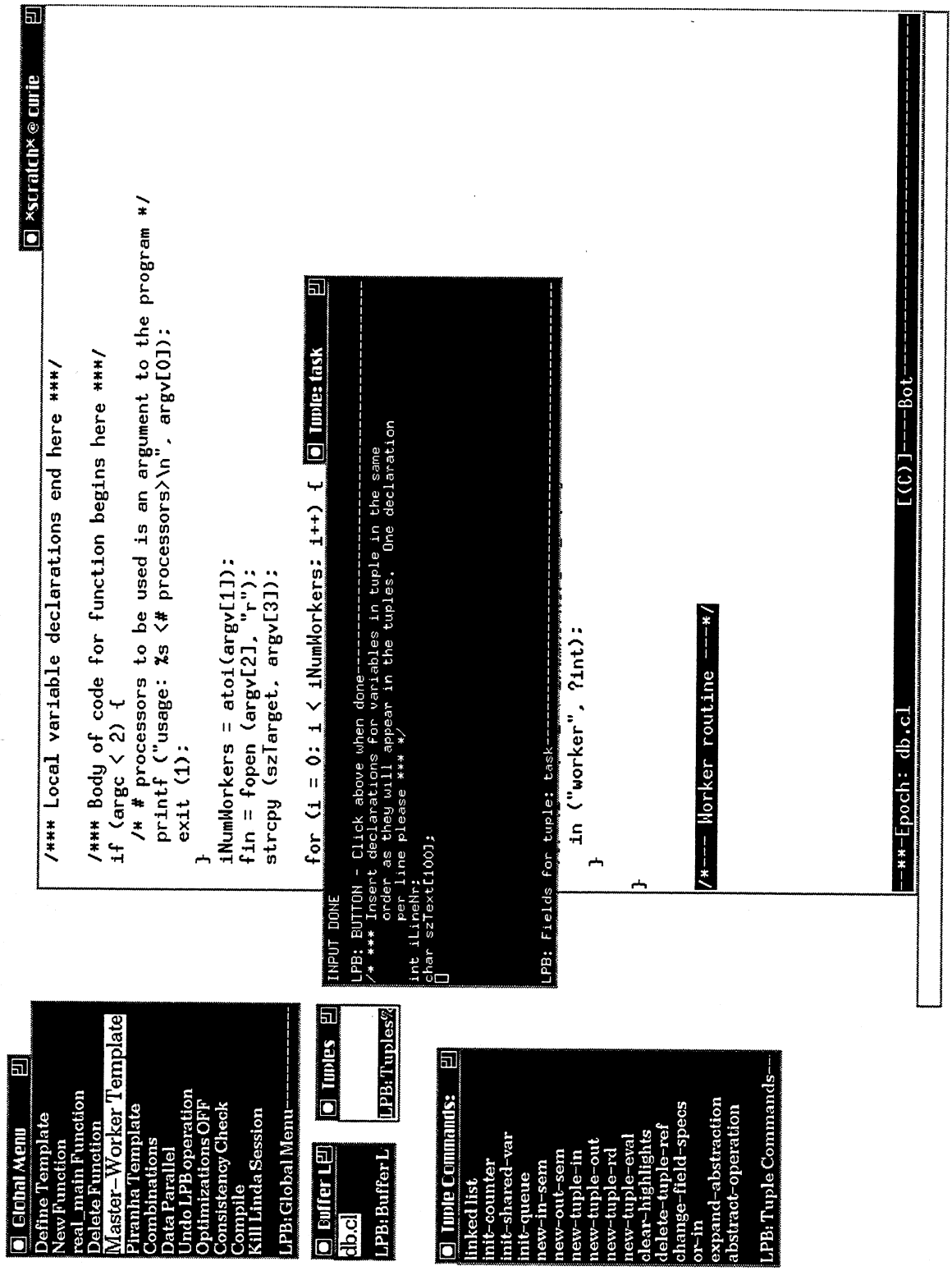

Figure 2.5: Master-Worker: Defining the Tasks

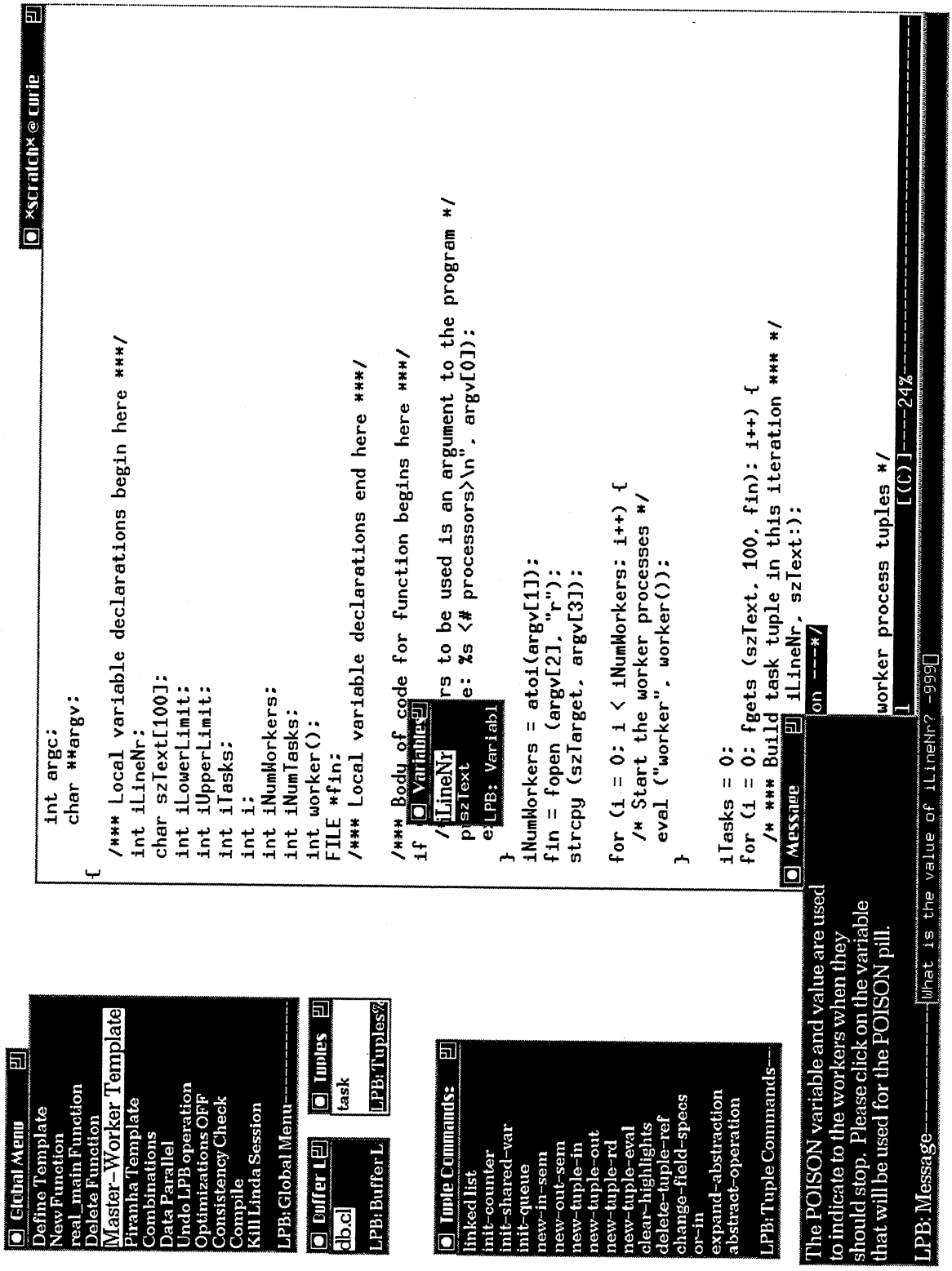

Figure 2.6: Master-Worker: Identifying the Poison Variable

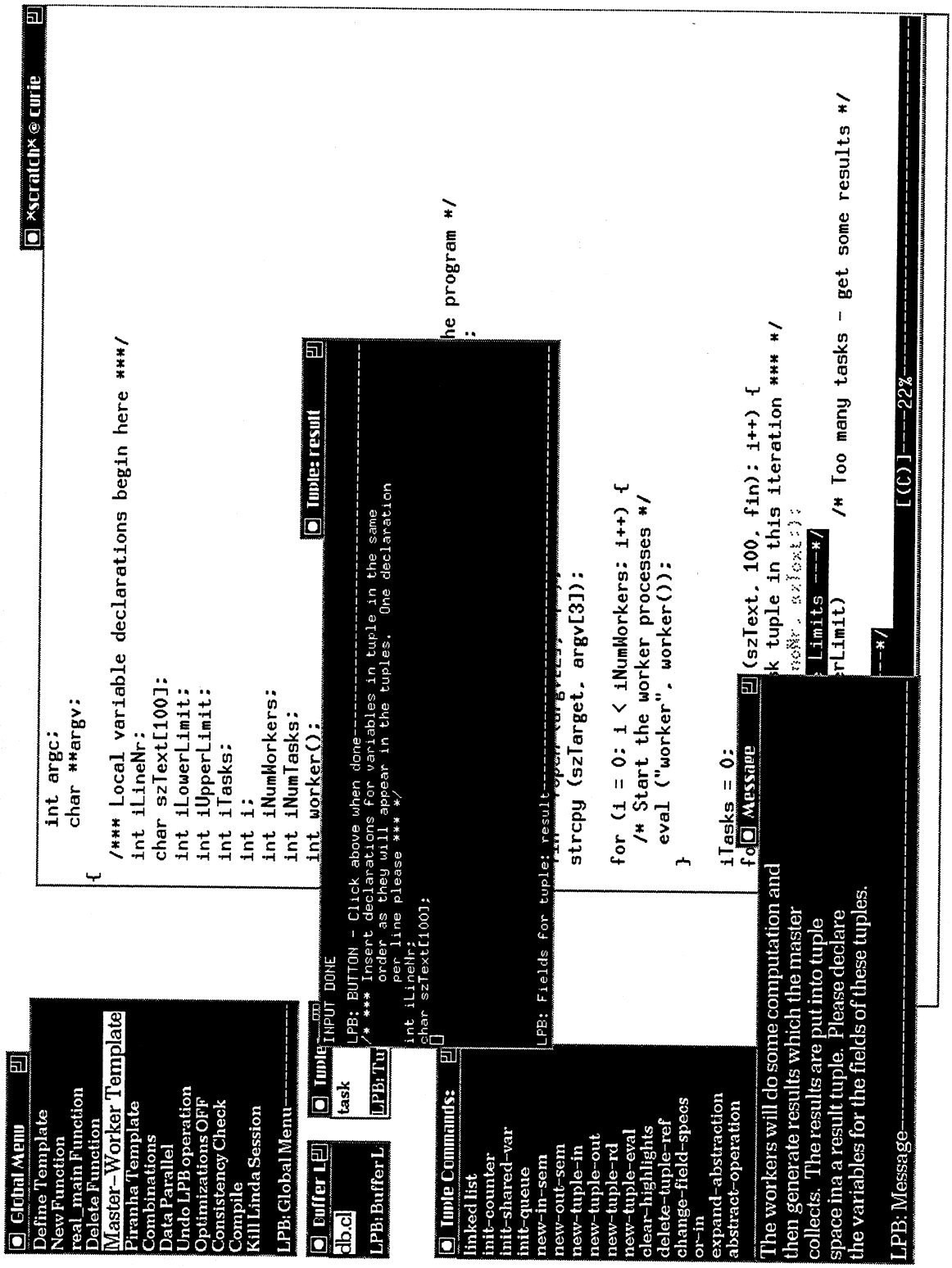

Figure 2.7: Master-Worker: Defining the Result Tuple

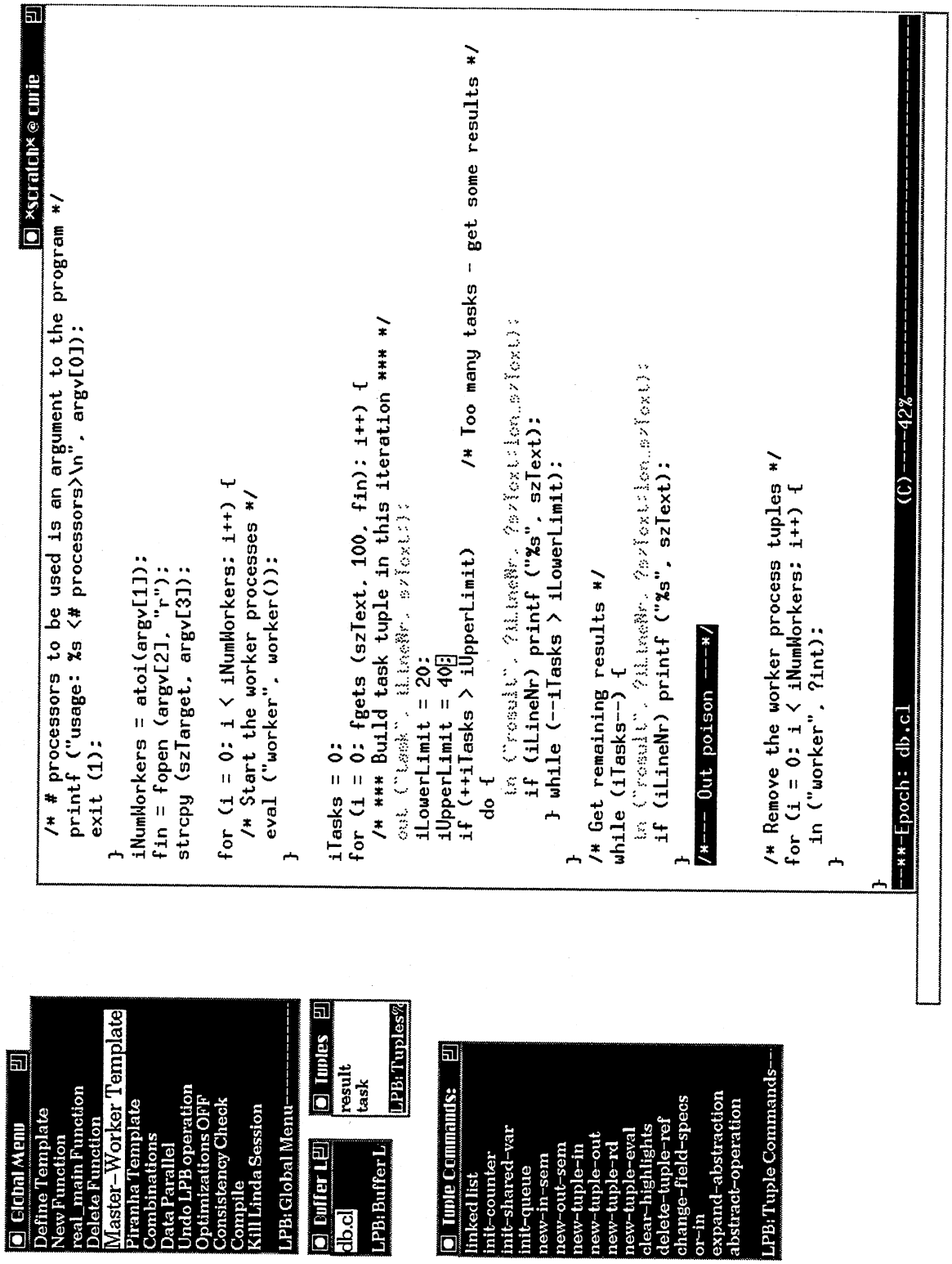

Figure 2.8: Master-Worker: The Task Generation and Result Gathering Loop

needed. Within the loop, the tuple operations are clearly highlighted in red.

Only a few more things need to be done in the master routine. The lower and upper bounds on the number of tuples to be held in the watermarked task bag need to be specified. This is merely a matter of clicking on two buttons which prompt for a number or expression. We also need to print out the results. If the line number of a result is positive, we print out the text string in that result; if it is zero, we ignore it. It may seem odd that we bother to return negative results, but this is necessary since we don't know *a priori* how many matches will be found. Consequently, we gather as many results as we initially generated task descriptors for. If we were streamlining this code, we could return negative result tokens that are small and don't hold an unnecessary string. There is now only one button left in the master routine which needs expansion and this is the poison button. Clicking on this automatically generates the appropriate tuple operation.

The last thing that still needs to be done for the master routine is to read and distribute the target string that will be searched for. We assume that the target string will be specified on the command line. To distribute the target to all the workers, we have to define a new tuple. To do so, we click on the `new-tuple-out` command in the `Tuple Commands` menu and the LPB responds by requesting a label for the tuple. We call the tuple "target" and are presented with an input window just like the one in the case of defining task or result tuples. In this case, we need only one field which is an array of characters to hold the target string. Once this has been defined, we are done and the appropriate **out** operation is inserted by the LPB into the code (Figure 2.9).

With our master routine complete, we can concentrate on the worker routine. Clicking on it gives us the skeleton for the worker shown in figure 2.10. The first thing we need the worker to do is to read the target string. For this, we make use of the tuple operation support that the LPB provides. Clicking on the tuple label for the tuple we want (the "target" tuple) in the `Tuples` menu causes the `Tuple Commands` menu to display a list of options for that tuple. We would like to read the target tuple from tuple space, so we click on the `tuple-rd` option in the `Tuple Commands` menu and the appropriate tuple operation is automatically inserted into the code. The corresponding variable declarations are also automatically made within the scope of the local function. Note that clicking on the target label in the `Tuples` menu also affects the `Tuple Info` window which displays information on the current tuple selection.

Expanding the remaining buttons of the worker routine is a mere formality. The task descriptor tuple, the poison pill, the result tuple — all of these have already been defined and clicking on the buttons automatically generates code. In fact, the only code segment which does need to be typed in is the actual computation (Figure 2.11). The computation is a straightforward string search. If a match is found, the tuple should be put out into tuple space. If no match is found, the line number must be set to zero and the tuple must then be put out. We enter the appropriate code for this and are now done with the program. We save it and hit the compile button. The LPB responds by giving us a choice of compiling with or without Tuplescope [BC90]

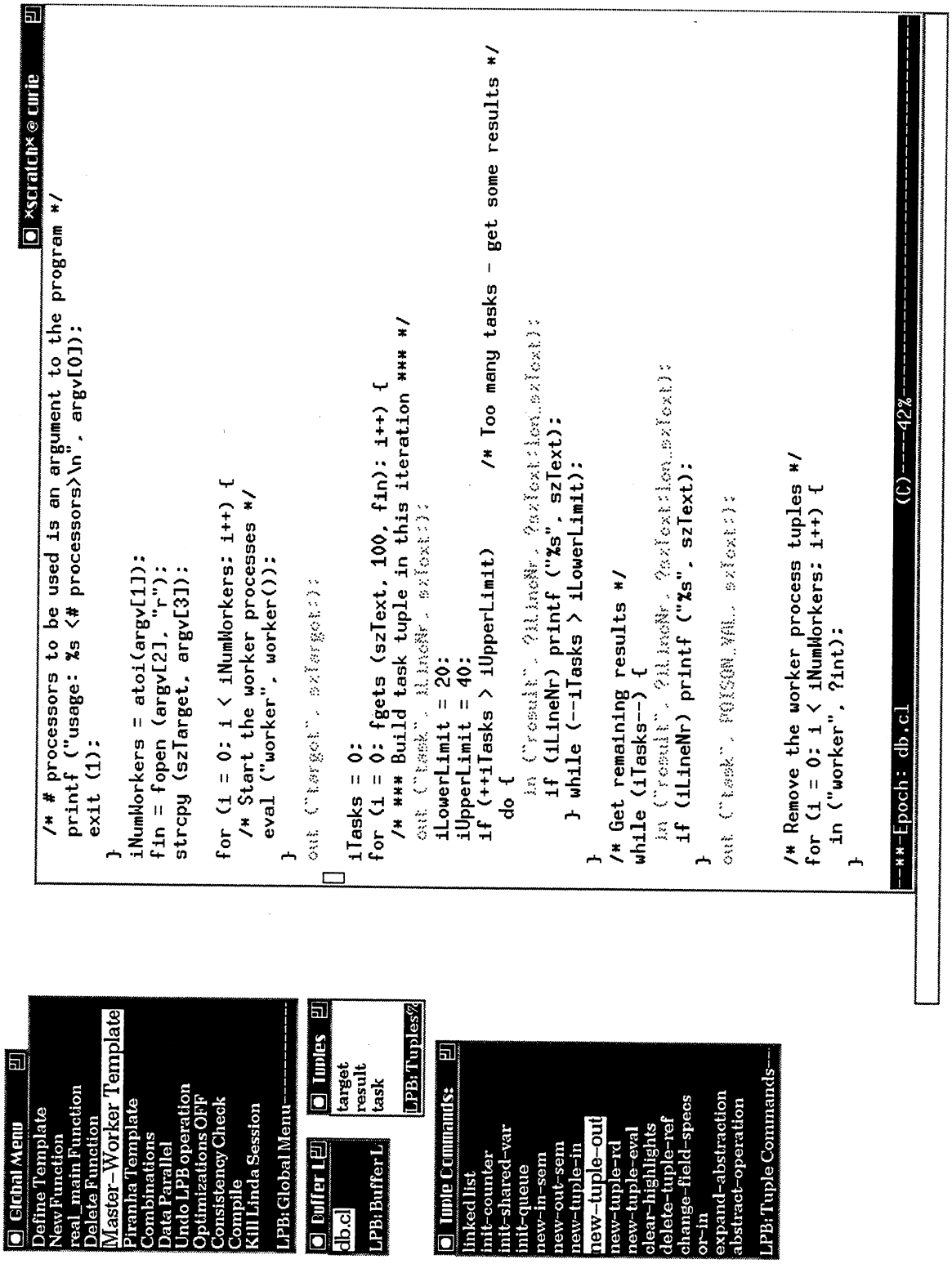

Figure 2.9: Master-Worker: Distributing the Target

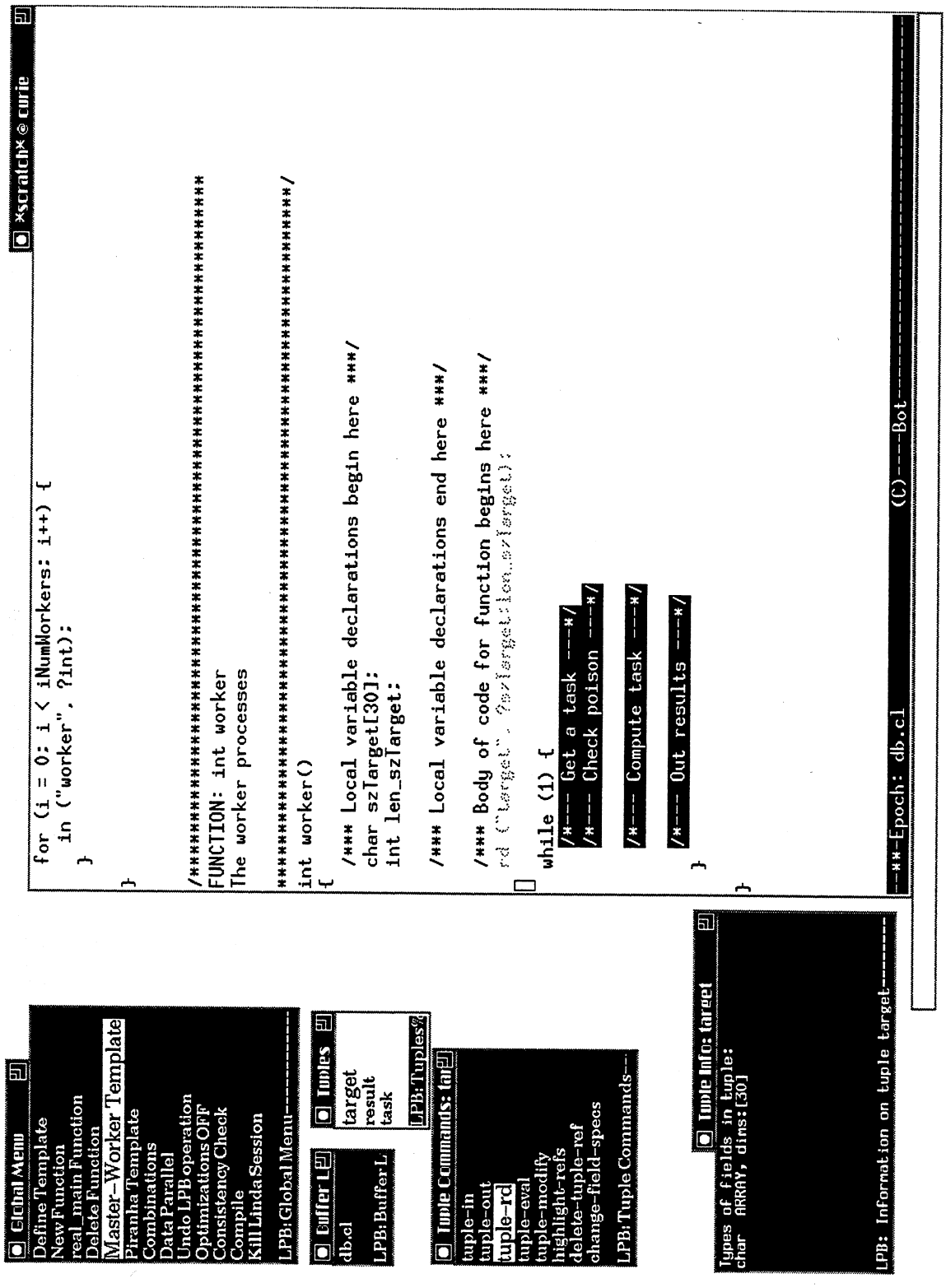

Figure 2.10: Master-Worker: Worker Routine

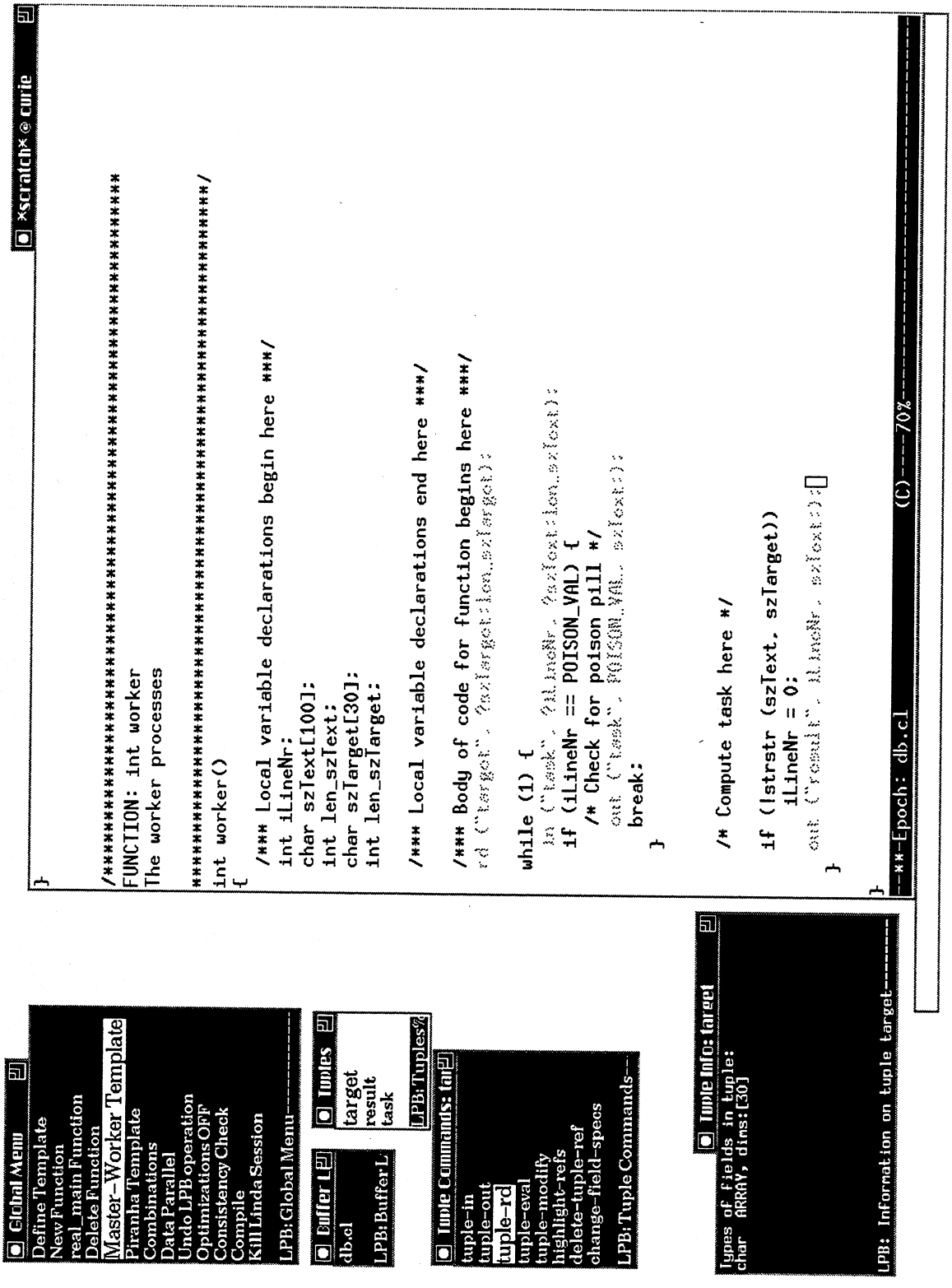

Figure 2.11: Master-Worker: Expanding the Worker Routine

(Figure 2.12). We decide to compile without the Tuplescope flag. We then run the program on an example in our process window (Figure 2.13).

We have constructed a parallel database search program using the master-worker template. The LPB handled much of the administrative efforts in the parallelization and guided the user through following the master-worker framework.

2.1.2 Hierarchical Templates

Templates could be hierarchical in nature. We can use existing templates as building blocks for other templates. Suppose, for example, that our site frequently constructs database search programs of the type we just built. We may wish to construct a template which generates parallel database search programs. What we want is a template that constructs parallel code for problems of the sort:

```
search target in all {...}
```

i.e. we want to search for some target in all elements of a large database. In order to construct such a template, we could use the master-worker template as a building block. One approach would be to simply have the template for database searches call upon the master-worker template and have the user fill it out in much the same manner as we have already seen in our example.

Alternatively, we could have our template call upon the master-worker template with some slots already filled in. All the user would need to specify would be his target and his database format. Users could thus very quickly generate database search programs that search through a particular database in a specific manner. Hence, customized templates could reduce software development time for different sites.

2.1.3 Other Templates

The LPB offers other templates in addition to the master-worker template. Two such templates are described below. Some of the design decisions and assumptions are also described. In general, the question of how and when to design and build a template is an interesting issue which is discussed in Chapter 5.

Piranha Templates

Piranha templates are designed to guide the user through constructing Piranha [Kam94] programs. The basic idea is similar in nature to the master-worker template of the previous section, but there are some important distinctions. At the top level, there are buttons for four different routines. The basic idea behind Piranha is that it enables users to write adaptively parallel programs. We have a *feeder* process which provides tasks to a number of *piranha* processes that run on different nodes of a network. The number of nodes working on a problem can change dynamically with availability.

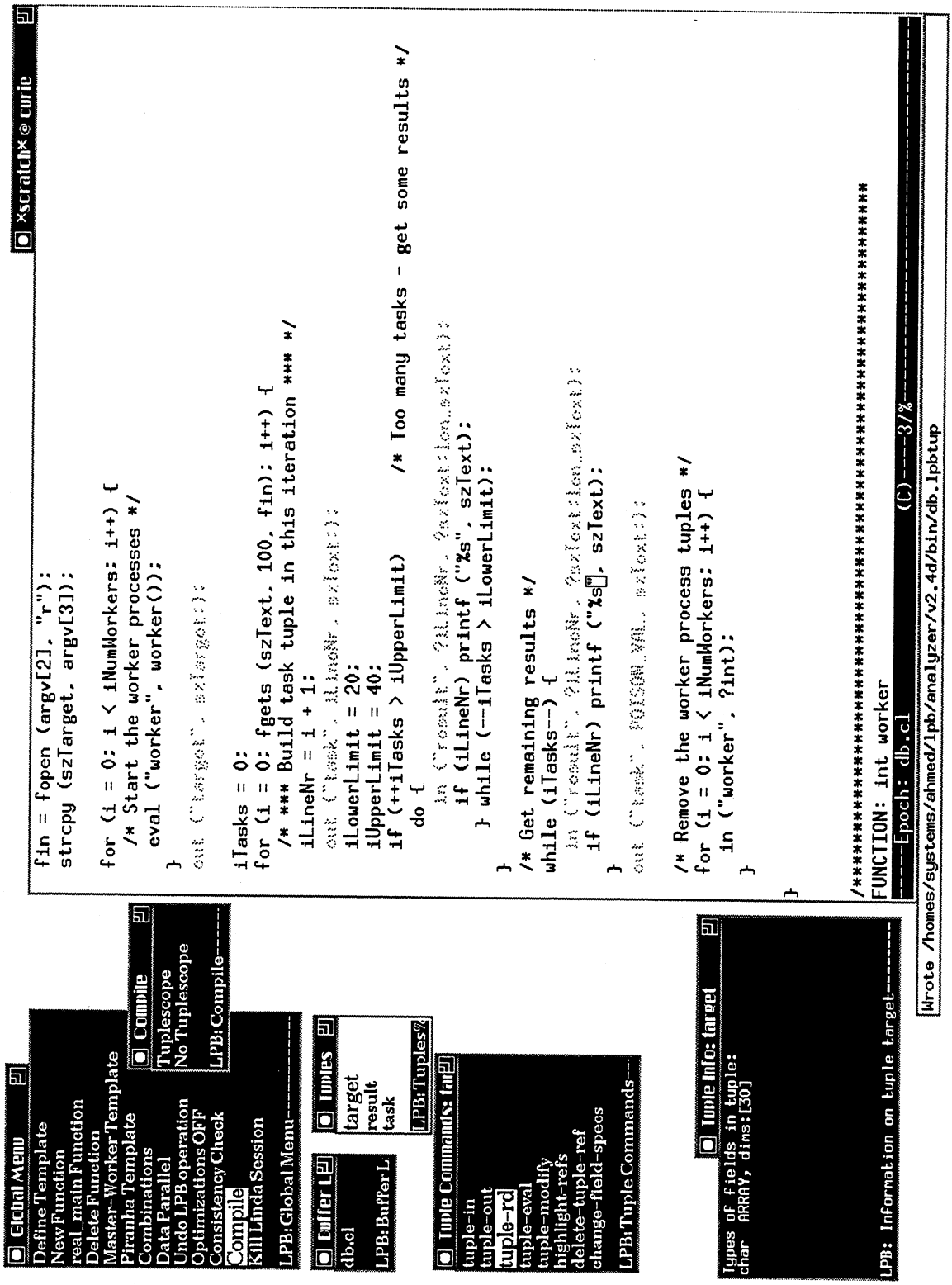

Figure 2.12: Master-Worker: Compiling the Program

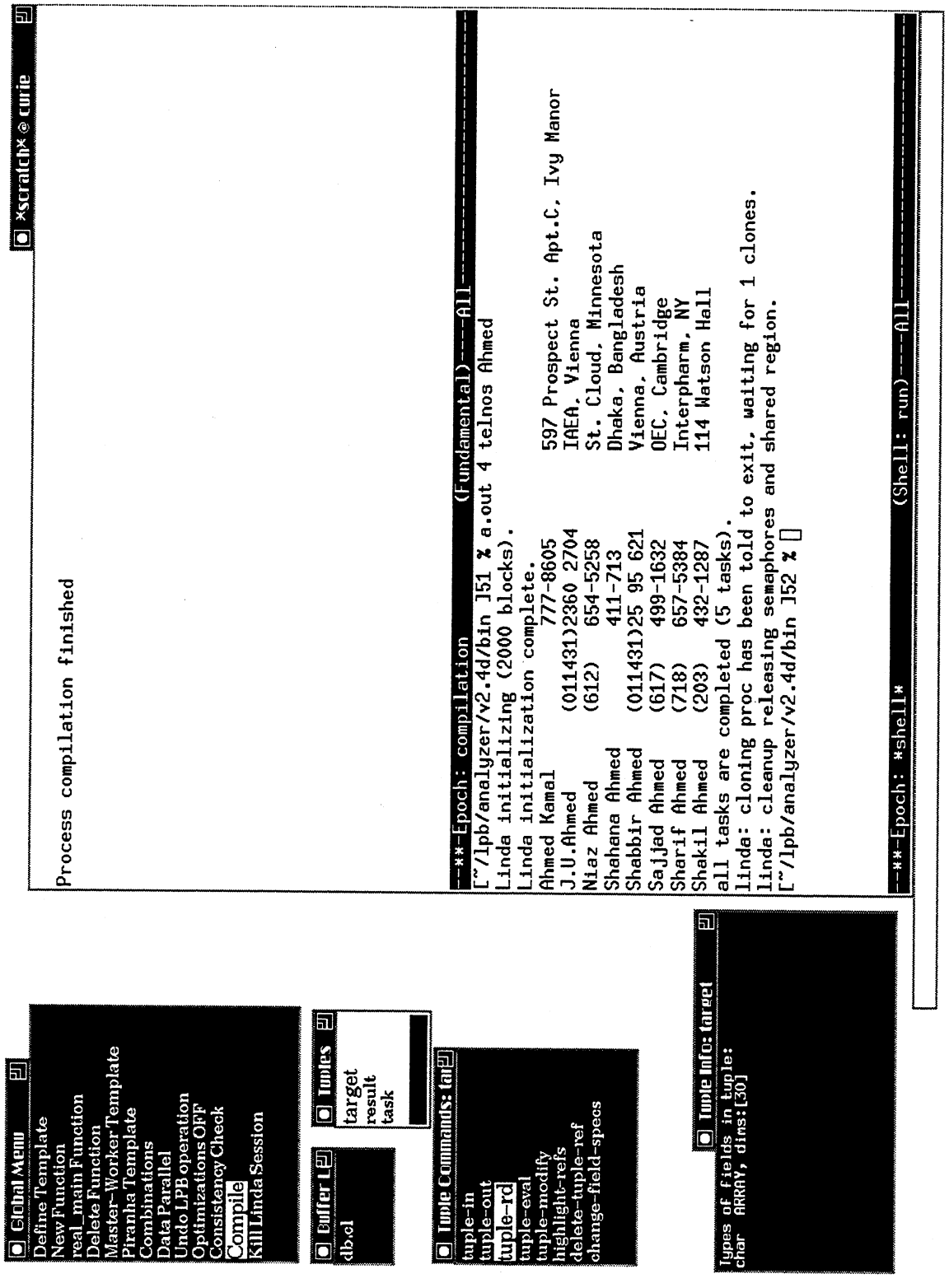

Figure 2.13: Master-Worker: Running the Program

The feeder routine is analogous to the master routine in the master-worker template and the piranha routine corresponds to the worker routine. The user expands these routines just as he would expand the corresponding routines in a master-worker program.

In addition to these two basic routines, Piranha programs also have a retreat routine and a copy routine. The copy routine copies the task data to local static variables which are needed by the retreat routine. When the user expands the copy routine button, it first generates some static variable declarations and then the actual copy function. The static variables are copies of the variables of the fields in the task tuple. The arguments to the copy routine are copies of the task tuple field variables and these are then copied into the static variables.

The retreat routine is invoked if a process running on a node has to be killed (Piranha processes are killed if certain conditions on a node are violated, e.g. the user of a workstation may just have returned from his lunch break and have touched his keyboard, indicating an intention to use his machine — in such a case foreign piranha processes become undesirable). Suppose a Piranha process has consumed a task from a pool of tasks and is working on it. If this Piranha process suddenly needs to be killed, it has to regenerate the task it has already consumed so that another Piranha can later work on it. The retreat routine regenerates the consumed task. In cases where there are no dependencies between different tasks, the retreat routine is quite simple — it merely involves doing an **out** of the consumed tuple. The LPB automatically generates such retreat routines. The original values of the fields of the task tuple have been stored in the static variables described above and can be used to regenerate the original task tuple.

Data Parallel Templates

The data parallel template supports the owner-computes style of programming [CG94]. The basic idea behind this style of parallel programming is that each processor holds some portion of a large, global data structure. The processor is responsible for all computation on its share of the data structure. Ultimately, the complete data structure itself will hold the end result, or the results from each portion need to be merged in some way to yield a final result.

At the top level of the data parallel template, there are buttons for three key routines: a filter routine, a log routine and a merge routine. The filter routine constructs a function which will determine which data belongs to particular processes. There are a number of ways to do this, but the most common one supported by the LPB is the *iterative chunk* approach. In this approach, a processor is responsible for some chunk of a larger data structure. It works on the chunk, one element at a time, until this chunk is done, at which point it proceeds to the next chunk that it is responsible for. The code below lists the `filter` routine. The first step is to claim a chunk of the data structure. This is done by doing an **in** on the "task" tuple — the `stop` variable will

hold the upper limit of the chunk that the process will work on. Once the `stop` value is acquired, it is incremented by `chunk` and put out to tuple space so that another process can claim this next chunk.

Each process goes through the list of elements in the global data structure and determines which of the elements are its responsibility by calling the `filter` routine. Given a chunk that the process has claimed, the filter routine's job is to check each element against this chunk. The counter variable `count` holds the number of the current element. If `count` falls within the range of the chunk, the return value is `true`, otherwise, this element does not belong to this processor. Consequently, in the latter case, the routine increments `count` and returns `false`.

```
static count = 0;
static chunk;
static LINDA_ID;

int filter()
{
  /*** Local variable declarations begin here ***/
  static next = 0, stop;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  if (!next) {
    in ("task", ?stop);
    next = stop;
    stop = next + chunk;
    out ("task", stop);
  }
  ++count;
  if (count == next) {
    if (++next == stop) next = 0;
    return 1;
  }
  else {
    return 0;
  }

}
```

The log routine fills a static data structure with results as they become available. When a process completes its computation, this data structure will hold all the results for the portion of the global data that the process was working on. When a user expands the log routine, he is asked to identify what kind of data structure he will use.

The data could be a simple variable, a structure, an array of variables, or an array of structures. Depending on the data structure type selected, the user is asked some questions regarding the names and types of the variables involved. Once this has been done, the routine is automatically generated.

Finally, the merge routine gathers the different parts of resultant data together to form the final answer. If the process happens to be the master routine, it will **in** the results from each of the workers. If it is a worker routine, it will put a result out to tuple space which contains its node identifier, and the static data structure constructed in the log routine. The latter contains the results that it has computed.

There are also a number of combination operators that can help users in merging data together in the final stage. These include simple combinators such as sum operators, global minimum operators, and global maximum operators. These operators work for any number of processors, not just powers of two. The user can also declare new operators by writing macros for the operators.

The combination operators complete the data parallel package. The data parallel template thus helps users write owner-computes style data parallel code in Linda.

Chapter 3

Distributed Data Structures and High-Level Operations

Linda programs generally make use of distributed data structures such as distributed arrays, task bags, shared variables and so on. Many of these crop up within the coordination-framework templates discussed above. Structures such as task bags, watermarked bags and distributed queues, to name a few, are often incorporated into the choices presented during the construction of a program through a template. But if a programmer needs to specify a data structure outside of a specific template, the LPB still provides support. Why would a programmer wish to do so? There are a number of scenarios where the situation could arise, but ultimately it stems from the basic need for flexibility.

One of the most important design decisions of the LPB was influenced by the goal to provide flexibility to the user in a manner that even experts would consider an aid rather than a burden. Consequently, as has been mentioned in Chapter 2, the user has the option of bypassing templates partially or even completely. A programmer may decide to construct a program without invoking a template and rely on distributed data structure support from the LPB, or he may decide to add some new data structures to those provided by the template. The LPB will provide support for such actions.

The LPB also supports high-level operations and abstractions. These allow users to call on operations which are not part of the base language but can be implemented in terms of the base language. The high-level operations and abstractions typically use distributed data structures of the kind mentioned above.

The following sections describe the distributed data structures that the LPB supports. The data structures are presented in order of increasing complexity. This is followed by a section on high-level operations and abstractions and a section that describes how the system supports the distributed data structures and high-level operations.

3.1 Basic Tuples

The simplest distributed data structure supported by the LPB is a Linda tuple[1]. Tuples are ordered sets of data elements where the data elements can be of different types. We shall use the term tuple *category* to denote tuple signatures from which tuples are instantiated. In particular, the LPB provides a mechanism for users to define a tuple category by specifying a label and declaring the variables for the fields of the tuple category. Once a tuple category has been defined, Linda operations on tuples from that category are easily supported by the LPB.

Defining a tuple category involves selecting an option, specifying a label and declaring variables. By convention, the first field of a tuple is usually a string that serves to label the tuple category. The LPB adopts this convention and enforces it: a tuple category is always referred to by its label. To initialize a tuple category with a label, the user invokes any of the generic tuple operations for new tuples in the `Tuple Commands` menu. Hence, if we wish to **out** a new tuple, we select the `new-tuple-out` command in the menu and we are prompted for the tuple category label. Once we have typed in a tuple category label, an input window appears with instructions to define the variables for the fields of the category. We type in variable declarations in this window as we would anywhere in a normal Linda program, inserting comments if desired. When the declarations are complete, we click the `input done` button. The appropriate tuple operation is inserted into the code and the corresponding variable declarations are placed within the scope of the local function. More importantly, the tuple category is now defined and designated by a label. Hereafter, whenever we wish to manipulate tuples from this particular category, we click on the corresponding label in the `Tuples` menu and then select an action. The choice of actions appears in the `Tuple Commands` menu. Hence, to manipulate a tuple, we select a tuple category and then based on the actions allowed on this tuple category, we select which action to call. Choosing any one of the **in, out, rd** or **eval** operations in this menu will automatically insert the operation and corresponding variable declarations. When a tuple operation appears in the code, it is in red to distinguish it from other operations and free text.

As the LPB inserts tuple operations, it keeps track of their placement. Consequently, it can offer support for quick reference to all uses of a particular tuple category. Selecting the `highlight-refs` command will cause all references to the currently selected tuple category to be highlighted across all open code modules. In fact, we can change a tuple category definition and have this change propagate across all the modules automatically. Suppose we choose to add a field to a tuple or change the name of the variable for a field. We select the `tuple-modify` command and are presented with an input window with all the declarations for the selected tuple therein, just as they appeared when we originally defined the tuple category. We now simply change the

[1]Once again, we use the term "tuple" very loosely. Strictly speaking, we are referring to tuple signatures from which tuples will be generated at run-time

declarations as desired. All references to the tuple category are updated to reflect any new changes and if new variable declarations need to be made, the LPB will take care of this. The LPB always inserts the same variable declarations for a particular tuple, regardless of where in the program an operation on the tuple is inserted. Users can override this using the `change-field-specs` command which is described below.

By default, all the fields in an **out** are actuals, whereas all fields except the first (the label) are formals in an **in**. This convention was chosen for empirical reasons: an **in** will typically remove a tuple from tuple space and bind the values of the fields to local variables. Needless to say, users will not always want all the fields in an **in** to be formals — they may want any number of fields to be actuals. The LPB needs to adopt some kind of convention, however, and since an **in** will usually involve at least one formal, the LPB sets all fields to be formal and lets the user change whatever fields need to be changed to actuals. To override a fields setting, the user has to select the `change-field-specs` command in the `Tuple Commands` menu. A menu pops up that lists the variables of the tuple category. Clicking on a particular variable will pop another menu up which shows the *status* of the field for that variable (Figure 3.1). The status of a field is either `actual`, `formal` or `other`. The current status is highlighted and changing it merely involves clicking on the desired status. A user may also wish to freely enter text for a field instead of relying on the variable name that the LPB generates. To change the status to such a customized string, the user selects `other` and then types in a string to replace the default field.

3.1.1 Tricky Situations

Supporting basic tuple operations requires taking care of some potentially tricky situations. Treating tuple fields which hold arrays is one such case because there are multiple ways to express them. Another tricky scenario occurs when two declarations of the same variable get inserted into the same function. The latter can be caused by the LPB itself as we shall see. The LPB deals with both of these situations.

Dealing with Arrays

Dealing with arrays in Linda is a complex issue — there are multiple ways to **out** or **in** an array since users can treat only parts of an array if they so wish. This has direct implications for the LPB: the LPB has to make default assumptions on how to treat fields in tuples which are defined as arrays. The user is not stuck with the LPB's default assumptions — the assumptions can be overridden by the user.

In Linda, if we wish to **out** an array and then **in** it as a formal, we may do so in the following manner:

```
int a[20];
...
out(''array'', a);
```

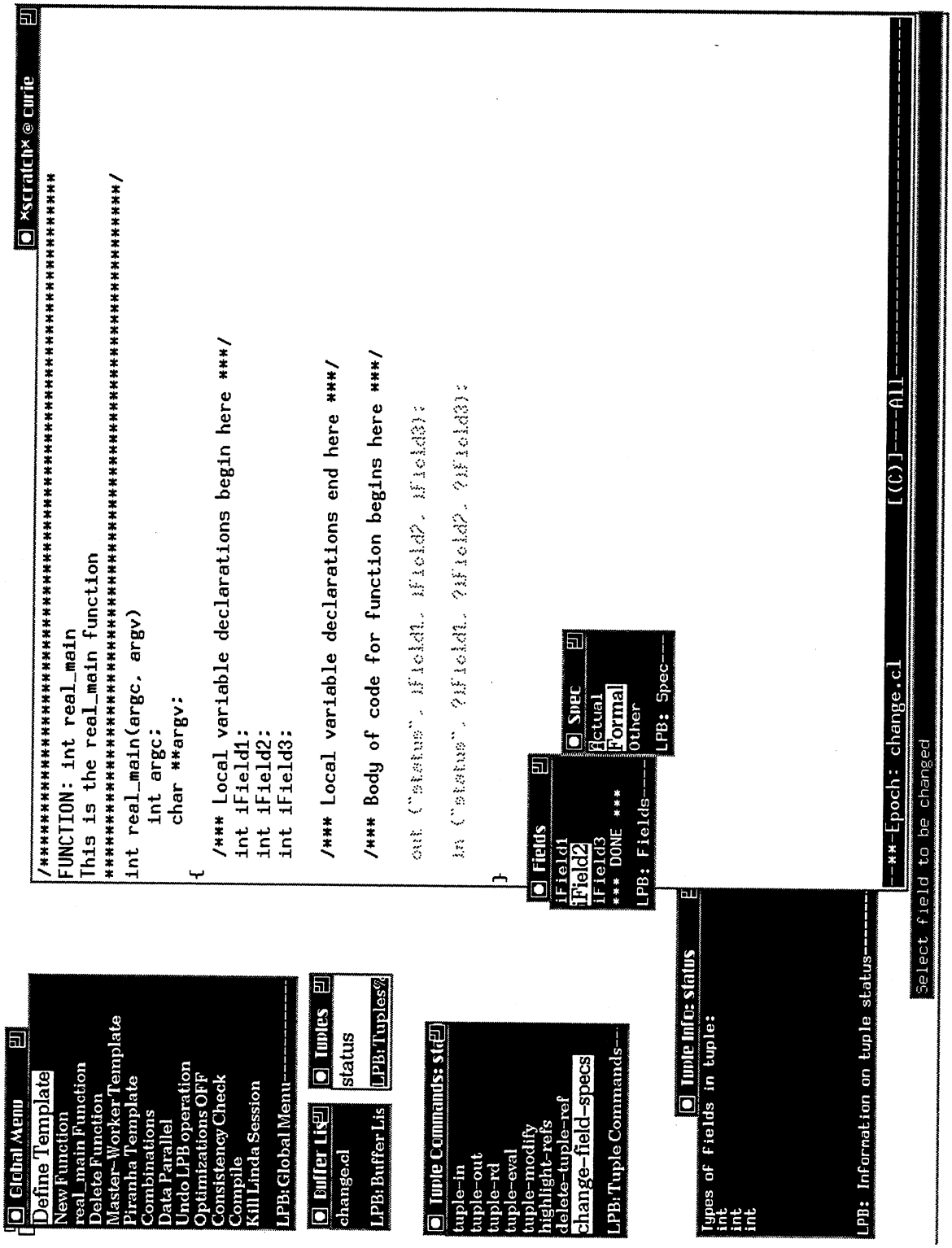

Figure 3.1: Changing the Field Status in a Tuple

```
...
in(''array'', ?a);
```

In this example, we are using a *fixed array field*, meaning that the length of the array is fixed. Fixing array length is quite restrictive. To fix array length, we assume that the entire array needs to be put out to tuple space. Furthermore, we need an array of the exact same dimension for the types of the fields to match. Since we need a general scheme for treating arrays such that their coding can be automated, the above approach is too restrictive. More interesting from our perspective is the varying field format. The following are all legal **out** operations in Linda:

```
out(''array'', a:20);
out(''array'', a:len_a);
out(''array'', a:);
```

The first example will output the first twenty elements of array *a* into tuple space. The second one will output *len_a* such elements, while the last example indicates a varying length field whose length is the declared length of the array. The last case is the most flexible case since any array with elements of the same type as *a* will match this field when used as a formal, regardless of the declared length of the new array. The corresponding **in** operation would be:

```
in(''array'', ?a:len_a)
```

where *len_a* must be an integer. The actual number of entries bound to *a* is returned in *len_a*.

When an array is declared as a field for a tuple category in the LPB, the tuple operations that the editor will construct will take the above into account. Thus, in the case of an **out**, the array name is taken and postfixed with a ":" so that it is put into tuple space as a varying field. In the case of an **in**, on the other hand, it is prefixed by a "?" to turn it into a formal, and postfixed with a ":len_variable-name" so that the number of entries actually read in can be returned (where "variable-name" is whatever the name of the variable is). Consequently, the new variable *len_variable-name* is automatically declared within the scope of the surrounding function.

Dealing with Variable Redeclarations

The other tricky scenario that can occur when automating tuple operation coding is variable redeclaration. It has been mentioned several times that variables necessary for a particular Linda operation are automatically declared within the scope of the surrounding function (pages 32, and 37). What if we were to insert two operations on the same tuple category within one function? Would the variables for the tuples get declared twice? The LPB will not redeclare variables if they have the same name and the same type. The code segment is parsed for this purpose. If, however, a variable

has the same name as one that has already been declared, but has a different type, a warning window pops up warning the user that a variable with a conflicting name has been declared within the local function.

3.1.2 The Tuple Information Window

When a tuple category has been defined, selecting it in the `Tuples` menu will not only change the commands menu to display the permissible commands for the tuple category, but will also change the `Tuple Info` window to show information on the tuple category. By default, this window will show information on the types of the fields in the tuples. For arrays, for example, it will show the type of the fields of the array, followed by the keyword "ARRAY" and a list of dimensions. This window is freely editable and the user can enter his own comments for this tuple category. These comments will always be associated with this particular tuple category and serve to remind the user of any relevant information when using tuples from this category.

3.2 Shared Variables

Shared variables are accessed by different processors. Implementing a shared variable in Linda typically involves putting it in tuple space. To modify a shared variable, a program will remove the tuple from tuple space, update the variable, and put the tuple back into tuple space. The LPB will ease the burden of coding for shared variables.

The LPB supports shared variables which are accessed by different processors. A shared variable is a special case of a generic tuple category. The difference between the two lies in the fact that shared variables conceptually have an evolving state associated with them. When a shared variable is initialized, it is set to some value and put out to tuple space where all processes can access them. Another process may choose to read the value of this shared variable (in which case it will execute a **rd** operation) or it may modify the value. Modifying a shared variable consists of removing the tuple from tuple space, binding the fields to some local variables, updating the values of fields, generating a new tuple with these updated values and putting the new tuple out to tuple space.

The LPB provides a number of constructs to manipulate shared variables. A user can initialize a shared variable by selecting the `init-shared-var` menu command. The familiar input window will appear, asking for variable declarations for the data which will be shared. When this has been completed, a new label appears in our `Tuples Menu`, namely the name of the first variable we declared in the input window, prefixed with the word "shared_". Clicking on this label will cause the `Tuple Commands` menu to change and display the commands allowable on a shared variable. This list is different from that of a generic tuple category. Instead of the standard **in, out, rd** and **eval** operations, we now have options to **init, in, rd**, and **modify** the shared variable. The **modify** option provides an input window which shows an **in** on the tuple followed

LPB option	Linda code
init	out("shared_*name*", field1, field2,...);

modify	in ("shared_*name*", ?field1, ?field2,...);
	field1 = expression1;
	field2 = expression2;
	...
	out ("shared_*name*", field1, field2,...);

rd	rd ("shared_*name*", ?field1, ?field2,...);
in	in ("shared_*name*", ?field1, ?field2,...);

Table 3.1: Shared Variable Options in the LPB and the Equivalent Linda Code

by an **out**. The user edits the fields of the **out** or inserts some assignment statement between the two operations to modify the state of the shared variable. Whatever is typed in this input window is what will appear in the program. Finally, as we are used to by now, any necessary variable declarations are automated . Table 3.1 shows the LPB shared variable options and the corresponding Linda code.

3.2.1 Specialized Shared Variables: Shared Counters

A shared counter is a specialized form of shared variable. In a shared variable, the fields could have any types. In the case of a shared counter, however, there is only one field and this holds an integer value. When initializing a counter, the user only needs to give it a label and an initial value. The label appears in the `Tuples` menu, prefixed by the word "counter_". The function of counters is simple: count either upwards or downwards. Consequently, the actions that can be carried out on counters through the `Tuple Commands` menu are correspondingly simple: **init, in, rd, increment** and **decrement** (in addition to the usual commands for highlighting references or deleting tuple operations).

3.2.2 Specialized Shared Variables: Counting Semaphores

Semaphores are a vital ingredient of a large class of systems programs. Linda tuples can be used as semaphores. All that is needed is a string within a tuple to identify the semaphore. An **in** on this corresponds to Dijkstra's **P** operation, and an **out** on it corresponds to a **V** operation [PS85]. Multiple **P** operations result in multiple **outs** of instances of the same tuple category. The LPB supports semaphores: the user selects

LPB option	Linda code
init	out("counter_*name*", ctr_*name*);
increment	in ("counter_*name*", ?ctr_*name*); out("counter_*name*", ctr_*name* + 1);
decrement	in ("counter_*name*", ?ctr_*name*); out("counter_*name*", ctr_*name* - 1);
rd	rd ("counter_*name*", ?ctr_*name*);
in	in ("counter_*name*", ?ctr_*name*);

Table 3.2: Counter Options in the LPB and the Equivalent Linda Code

the menu command and provides a label; the LPB then restricts the actions menu to performing an **in** or **out** on the semaphore.

3.3 Distributed Queues

Distributed queues of various kinds are often required in parallel programs. They may have multiple sources, sinks, or both. The synchronization and handshaking necessary for coordination among the various sources and sinks can be achieved through distributed head and tail pointers stored in tuple space. The LPB provides a set of menu functions to create and manipulate queues. Upon selection of a `create-queue` command, a popup menu offers choices for queue models (Figure 3.2). Once a model has been selected, all the tuples necessary for maintenance of the queue are automatically generated and initialized. A user is now free to select menu commands to add to or remove from the queue as desired (Figure 3.3). All tuple operations, declarations, and support code are automatically inserted in the appropriate places.

The different queue models are: single source with single sink, single source with multiple sinks, multiple sources with single sink, and multiple sources with multiple sinks. The single source with single sink model is similar to the sequential model. One specific process adds entries to the tail of the queue; a process (possibly the same one) removes entries from the head of the queue. When we move to multiple sinks, there are more consumers of queue items, i.e. multiple processes will remove entries from the head. Conversely, we can have multiple sources adding entries to the tail of the queue with only a single process consuming entries at the head. Finally, we may have multiple sources as in the former case, but also have multiple processes consuming the

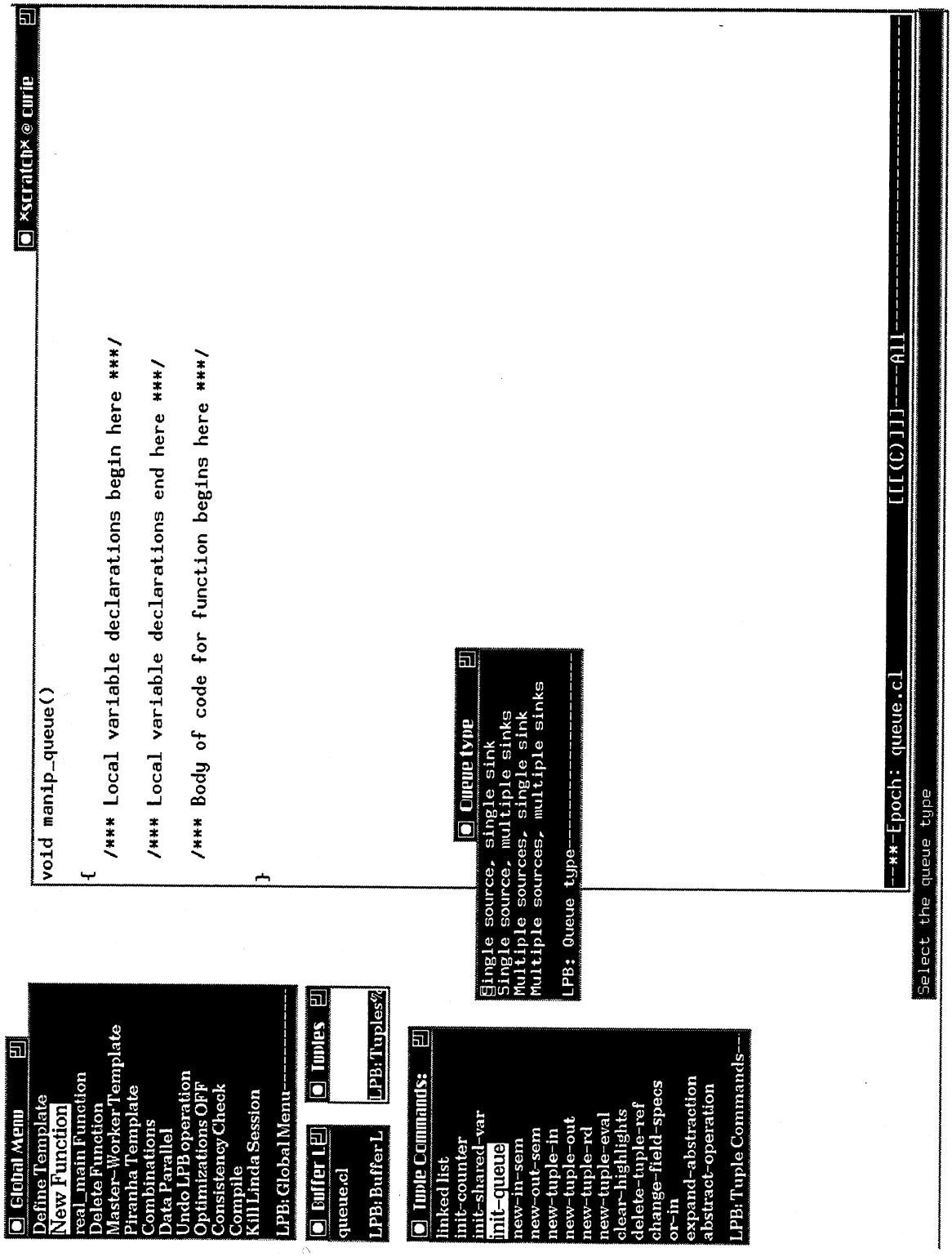

Figure 3.2: Choices of Queue Models

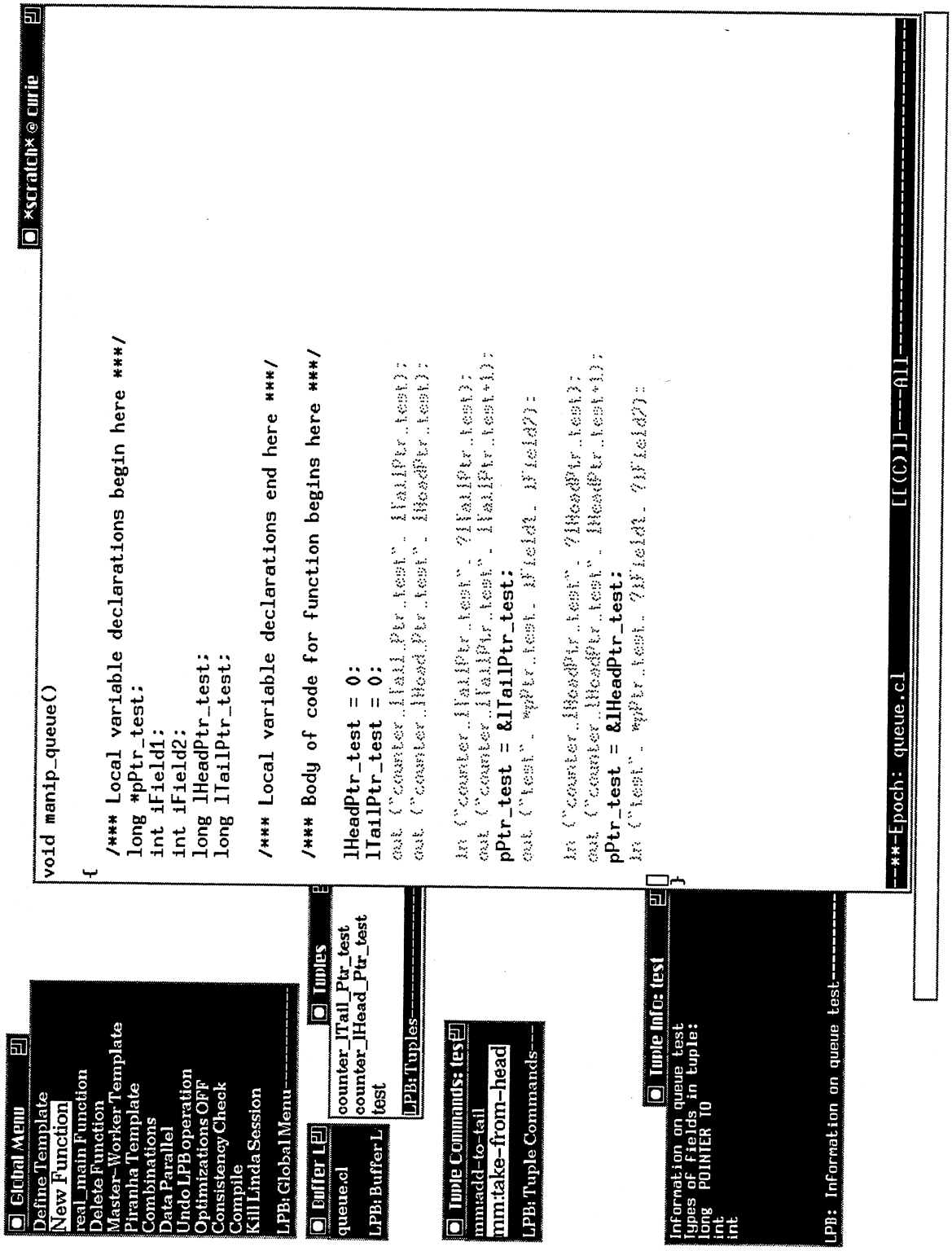

Figure 3.3: Queue Manipulation Operations

entries at the head.

To implement the queue operations, the LPB generates a tuple category to hold queue elements, and possibly one for a head pointer or tail pointer depending on the queue model chosen. Tail and head pointer tuple categories need to be generated if these need to be shared by more than one processor. Tables 3.3, 3.4, 3.5, and 3.6 show the different queue models, the LPB options for manipulating these queues, and the Linda operation sequences into which these options translate.

LPB option	Linda code
initialize	lHeadPtr_*name* = 0; lTailPtr_*name* = 0;
add-to-tail	pPtr_*name* = &lTailPtr_*name*; out ("*name*", *pPtr_*name*, field1, field2, ...); *pPtr_*name*++;
take-from-head	pPtr_*name* = &lHeadPtr_*name*; in ("*name*", *pPtr_*name*, ?field1, ?field2, ...); *pPtr_*name*++;

Table 3.3: LPB Options and Linda Equivalents for Single Source, Single Sink Queues

3.4 High-Level Program Constructs and Abstractions

The LPB supports high-level operations which are implemented in the base language. These operations are different from full program templates that direct control flow. High-level operations are there for users who may wish to use operations which are not part of the base language, i.e. they can take advantage of the functionality of high-level operations and treat them as if they are part of the base language. The abstraction feature of the LPB supports high-level operations and is a useful tool in constructing and viewing programs. It supports top-down programming by presenting a high-level view of program structure that can be expanded downwards at will, but also abstracted back up again to a conceptually more appealing higher-level format. This allows the programmer to concentrate on hierarchical program construction at a high level, and to deal with "blocks" of code represented by abstractions.

The LPB supports high-level operations which are not supported by the base language, but can be implemented by some sequence of operations in the base language.

LPB option	Linda code

initialize	lHeadPtr_*name* = 0;
	out ("counter_lHead_Ptr_*name*", lHeadPtr_*name*);
	lTailPtr_*name* = 0;

add-to-tail	pPtr_*name* = &lTailPtr_*name*; out ("*name*", *pPtr_*name*, field1, field2, ...); *pPtr_*name*++;
take-from-head	in ("counter_lHeadPtr_*name*", ?lHeadPtr_*name*); out ("counter_lHeadPtr_*name*", lHeadPtr_*name*+1); pPtr_*name* = &lHeadPtr_*name*; in ("*name*", *pPtr_*name*, ?field1, ?field2,...);

Table 3.4: LPB Options and Linda Equivalents for Single Source, Multiple Sink Queues

LPB option	Linda code

initialize	lHeadPtr_*name* = 0;
	lTailPtr_*name* = 0;
	out ("counter_lTailPtr_*name*", lTailPtr_*name*);

add-to-tail	in ("counter_lTailPtr_*name*", ?lTailPtr_*name*); out ("counter_lTailPtr_*name*", lTailPtr_*name*+1); pPtr_*name* = &lTailPtr_*name*; out ("*name*", *pPtr_*name*, field1, field2, ...);
take-from-tail	pPtr_*name* = &lHeadPtr_*name*; in ("*name*", ?*pPtr_*name*, ?field1, ?field2, ...); *pPtr_*name*++;

Table 3.5: LPB Options and Linda Equivalents for Multiple Source, Single Sink Queues

LPB option	Linda code

initialize	lHeadPtr_*name* = 0;
	lTailPtr_*name* = 0;
	out ("counter_lTailPtr_*name*", lTailPtr_*name*);
	out ("counter_lHeadPtr_*name*", lHeadPtr_*name*);

add-to-tail	in ("counter_lTailPtr_*name*", ?lTailPtr_*name*); out ("counter_lTailPtr_*name*", lTailPtr_*name*+1); pPtr_*name* = &lTailPtr_*name*; out ("*name*", *pPtr_*name*, field1, field2, ...);
take-from-tail	in ("counter_lHeadPtr_*name*", ?lHeadPtr_*name*); out ("counter_lHeadPtr_*name*", lHeadPtr_*name*+1); pPtr_*name* = &lHeadPtr_*name*; in ("*name*", *pPtr_*name*, ?field1, ?field2, ...);

Table 3.6: LPB Options and Linda Equivalents for Multiple Source, Multiple Sink Queues

In the context of the LPB, these high-level operations appear as abstractions which can be expanded into their equivalent base language representation. An expanded abstraction can be abstracted back again (buttons, on the other hand, can only be expanded). This higher level representation is more concise and easier to understand.

A similar approach has been explored in the past. The basic idea may seem similar to the Cedar [SZBH86] approach in its Tioga structured text-editor, but there is a key distinction. Tioga treats documents in a tree-structured manner where each node is a paragraph or a statement. This hierarchical node structure allows detail to be concealed in the interest of a conceptually higher-level view, much as in the LPB. A related feature appears again in editors that provide outline modes, allowing for a hierarchical perspective on text. Typically, these allow labels to be attached to text segments and then allow the document to be viewed at a label level. Any of these labels can be expanded into the text segment it represents. Note, however, that the abstraction feature of the LPB involves considerably more than mere in-place expansion. Expanding an abstraction involves actual code generation which may affect code spread across several modules. This code is automatically generated, and it is not a mere in-place insertion of a text segment the user has previously typed in.

As examples, we present the **or-in** construct and the shared linked list operations.

3.4.1 The *or-in* Construct

Linda programmers occasionally encounter situations where they would like to **in** any one of a number of possible tuples. There is no explicit construct in Linda which allows them to express this. As we shall see, explicitly coding this in Linda is nontrivial. It has been debated whether an operation should be added to Linda which would allow users to choose to **in** one of several tuples. This would require changing the Linda kernel to incorporate a new function and hardwire it into the language. This would be a drastic step with numerous implications on the elegance and simplicity of Linda. The undesirability of this is explored in Chapter 6. An alternative solution is to use the abstraction feature in the LPB. We can create an abstraction for the **or-in** operation such that it appears to be a Linda operation.

The LPB implements the **or-in** function as a higher-level operation. When the file is saved, the abstraction expands into the equivalent Linda code which is then compiled. The abstraction can also be expanded and abstracted again for viewing purposes. If the user selects the menu option for an **or-in**, a menu pops up with a list of the tuple categories that are known to the database. The user selects those tuple categories from which tuples can be **in**'ed for the **or-in**. The **or-in** appears to be a regular program construct, but the relevant lines in the code are highlighted (Figure 3.4).

Expanding the **or-in** abstraction will show the user how it is implemented in Linda code. Clicking anywhere on the highlighted portion selects that higher-level operation. Picking the `expand abstraction` button in the `Tuple Commands` menu causes the abstraction to expand. If the cursor is placed on the main section of the **or-in** expansion,

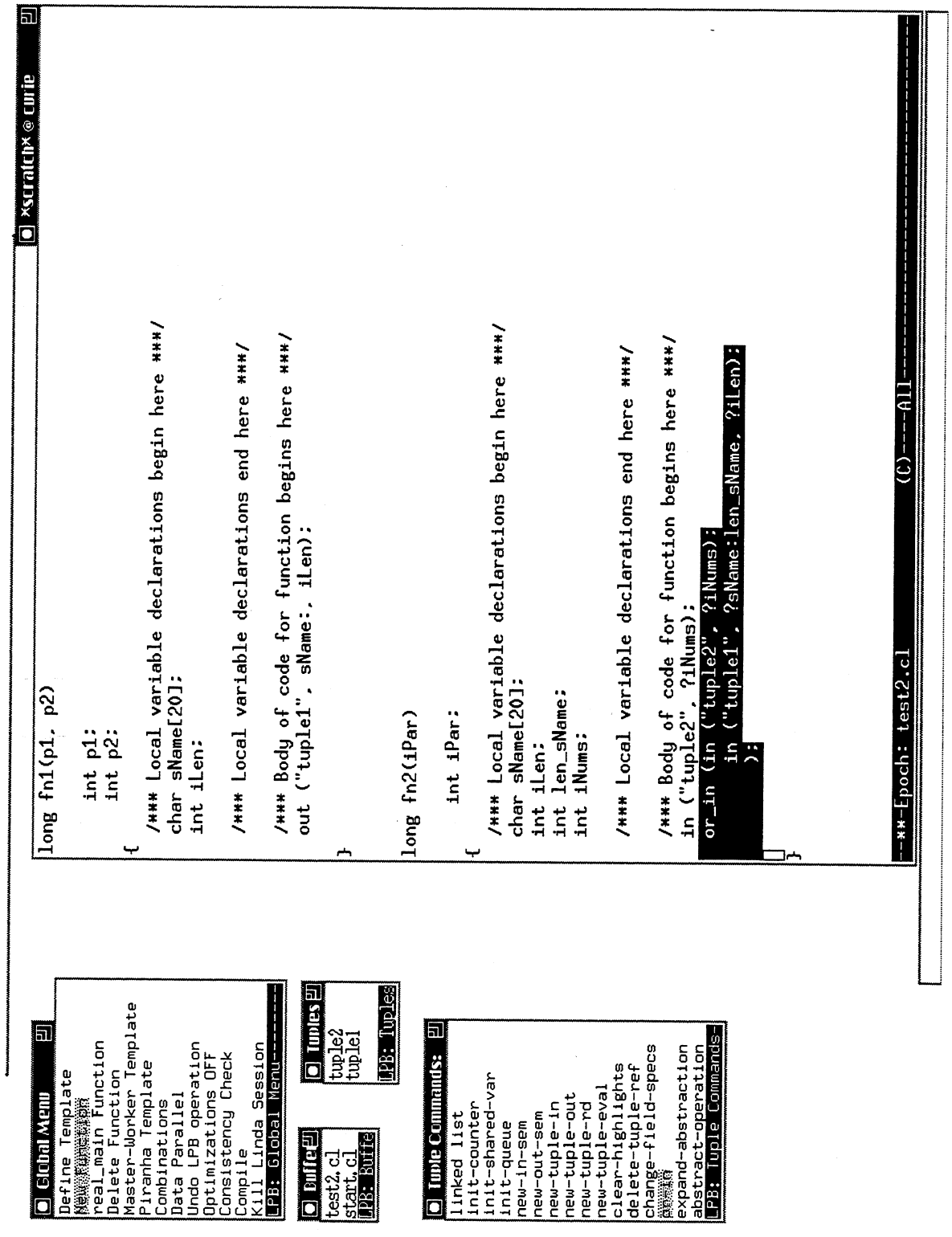

Figure 3.4: The **or-in** Abstraction

and the abstraction menu item is selected, all the expansion details disappear, and the abstraction reappears, making the **or-in** look very much as if it is a part of the language.

The LPB implements the **or-in** construct by generating a new tuple category which is labeled "or-in". This category has one field for each tuple that is involved in the **or-in**. Each field is a boolean and we can think of the tuple as a vector of booleans. Whenever one of the tuples to be **or-in**'ed is put out into tuple space, an instance of the "or-in" tuple category is also put out with a `true` value in the field that corresponds to the tuple which was just put out. The rest of the fields have `null` values. Whenever a tuple is removed from tuple space, the LPB inserts an **in** on the "or-in" tuple with the corresponding field set to `true` and all other fields set to `null`.

What does this give us? By inserting these operations, every instance of a tuple which is involved in the **or-in** is flagged when put into tuple space, and the flag is removed before the tuple is removed from tuple space. When it comes to the actual **or-in**, the LPB replaces the abstraction with an **in** on the boolean vector. The fields are all formals in this case, i.e. we are looking for available flags. Carrying out the **or-in** now becomes a matter of looking through all the flags for the first one with a `true` value and doing an **in** on the corresponding tuple when such a flag is found. Needless to say, at least one of the flags must be `true` since no all-null boolean vectors are ever put out — a boolean vector only goes out when a tuple is put out and hence, by definition, the corresponding field must be `true`.

Figure 3.5 shows the expanded **or-in** abstraction. Note that the expansion causes all relevant references to tuples involved in the **or-in** to be preceded or followed by a corresponding operation on the boolean vector tuple. These changes will affect all open modules. The ability to affect code globally is a key distinction between abstractions in the context of the LPB and abstractions as we know them from other editors. There are numerous editors which allow text segments to be labeled by tags. Users can step up or down a hierarchy ladder by viewing tags as placeholders or having tags expand into text segments which they represent (e.g. the Tioga editor mentioned earlier). An LPB abstraction is clearly not merely an in-place expansion — it can have global effects on the code.

3.4.2 Shared Linked Lists

A shared linked list is a linked list which can be accessed by multiple processors. Such a list could be used to implement random-access queues, for example. The LPB provides support for shared linked lists — it offers options to create and manipulate shared linked lists.

The implementation for a shared linked list basically has two levels. The elements in the list have index values. The first level of the list is the index list, where each entry is represented by a tuple containing the index value and the next index value. The index value is the key into another tuple which actually stores the data element for

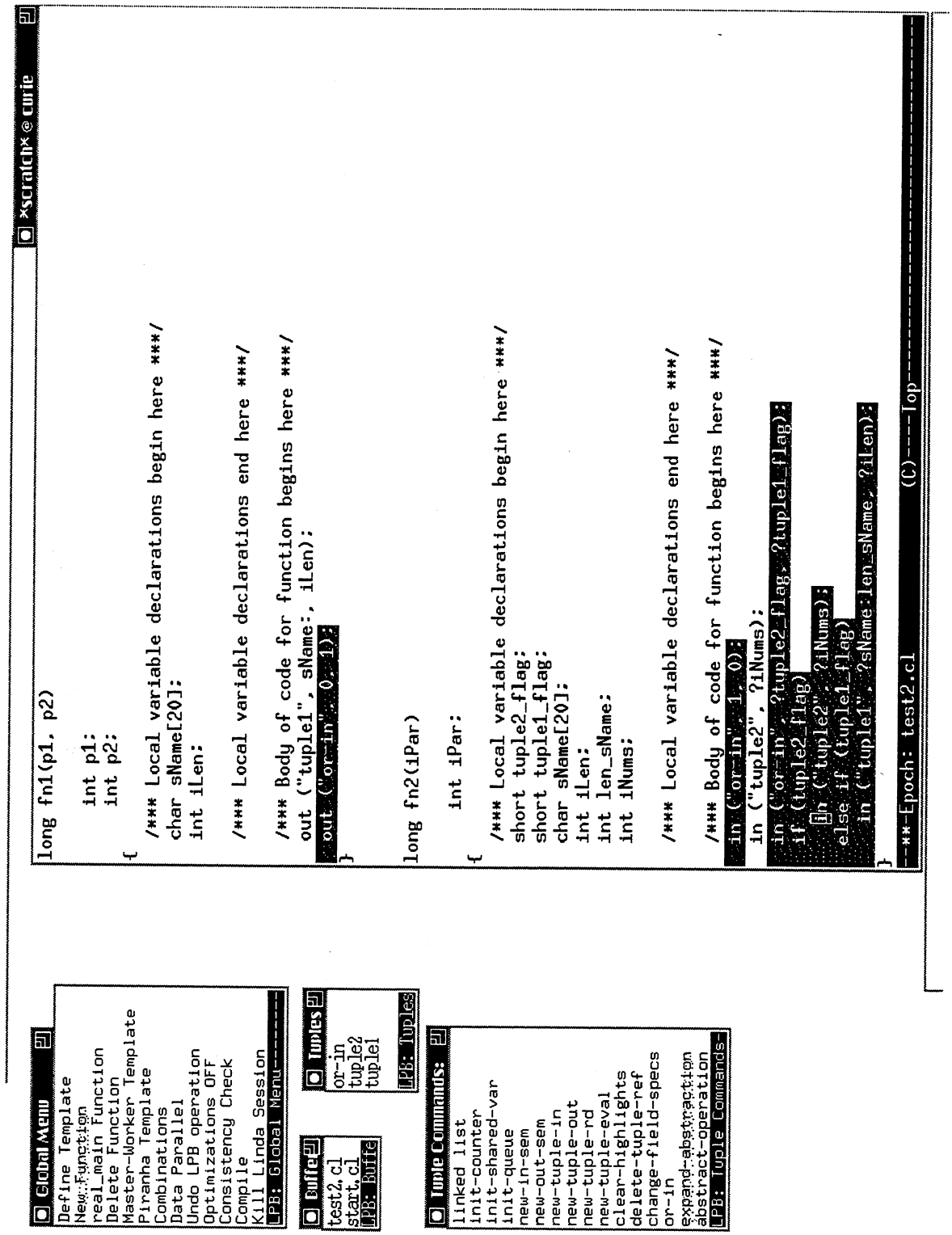

Figure 3.5: Expanded **or-in**

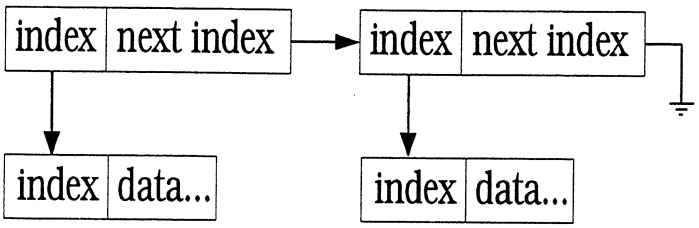

Figure 3.6: Linked List

that index. At the user level, the user simply selects an option from one of **initialize list**, **insert node** or **remove node**. The user actually views the code at this level of abstraction. If the user chooses to expand the abstractions, they expand into Linda code that manipulates the linked list. Hence, an **insert node** will expand into code that first acquires a unique identifier for the node and then creates a new node. To do this, we **in** the node after which the new one will be inserted, modify the appropriate fields to insert a new node, and **out** the new node index. We can then **out** the node with the data. Figures 3.7 and 3.8 show a linked list operation in pre- and post-expansion format.

To initialize a list, a user specifies a label and declares variables to hold the values for the nodes of the list. Thereafter, inserting a node requires identifying the list and indicating where on the list the new node should be placed. Each time a new node is generated, a new unique identifier has to be issued for this node. This is done by an ID-generating function which the LPB constructs and places into the module, within visibility of the user. The appropriate call to this function can be seen when the abstractions are expanded. Table 3.7 shows the LPB options for manipulating shared linked lists and the Linda code into which these options expand.

3.5 The Program Database

All the LPB's distributed data structures are maintained by a program-describing database. This database is the backbone of the LPB, maintaining all the information necessary to implement higher level operations and provide user support. This information is used to eliminate administrative memory-work and reduce keystrokes. For all tuple operations, for example, variable declarations and code insertion are automated. The database's information allows cross-module propagation of updates to tuple references when a tuple structure is modified. The database also supplies information to the compiler and visualizer.

The LPB continuously updates its program-describing database. Every tuple, function, abstraction, higher level operation, or other significant component of the program is entered into the database as it is used. The database keeps information on a tuple

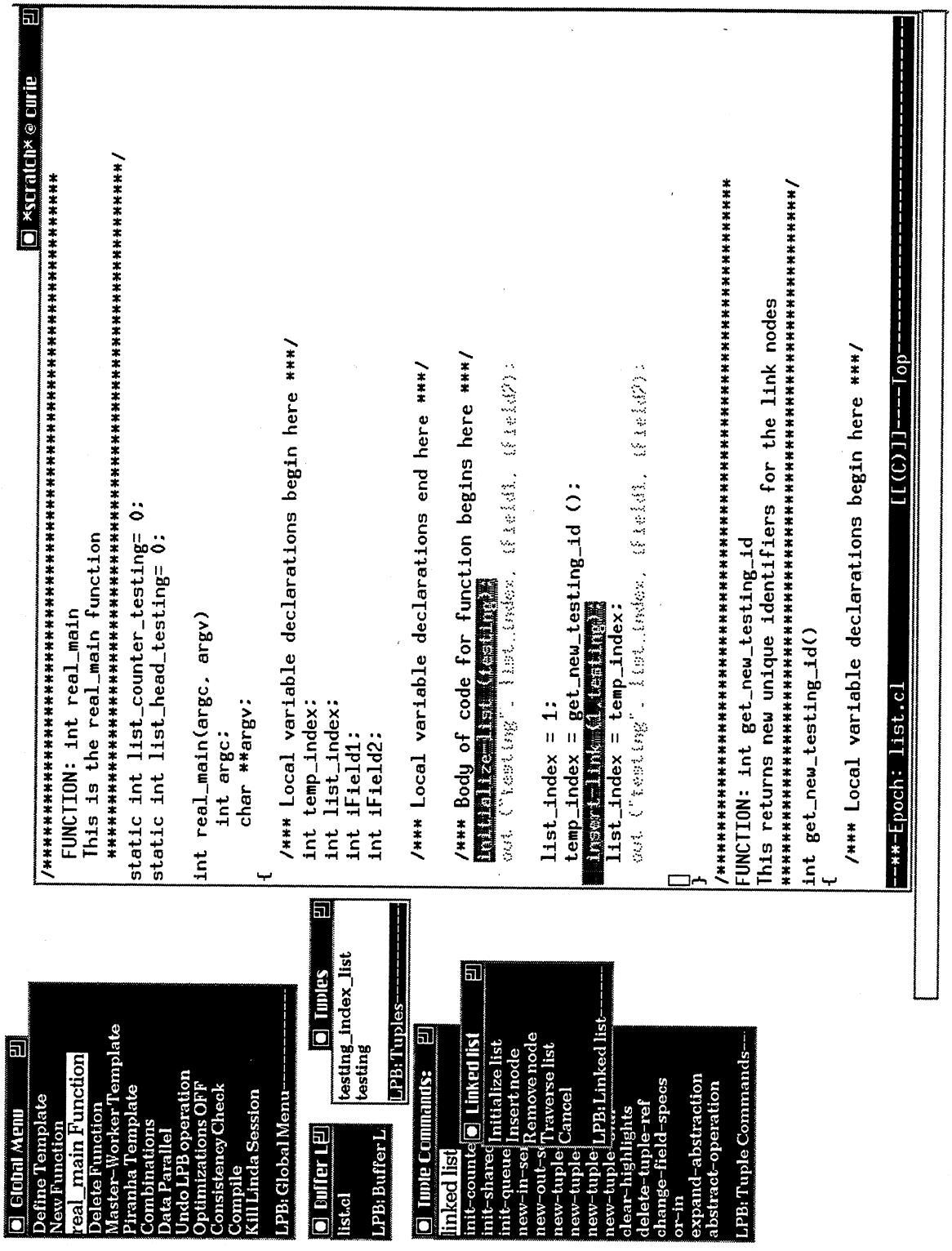

Figure 3.7: Linked List Operation Before Expansion

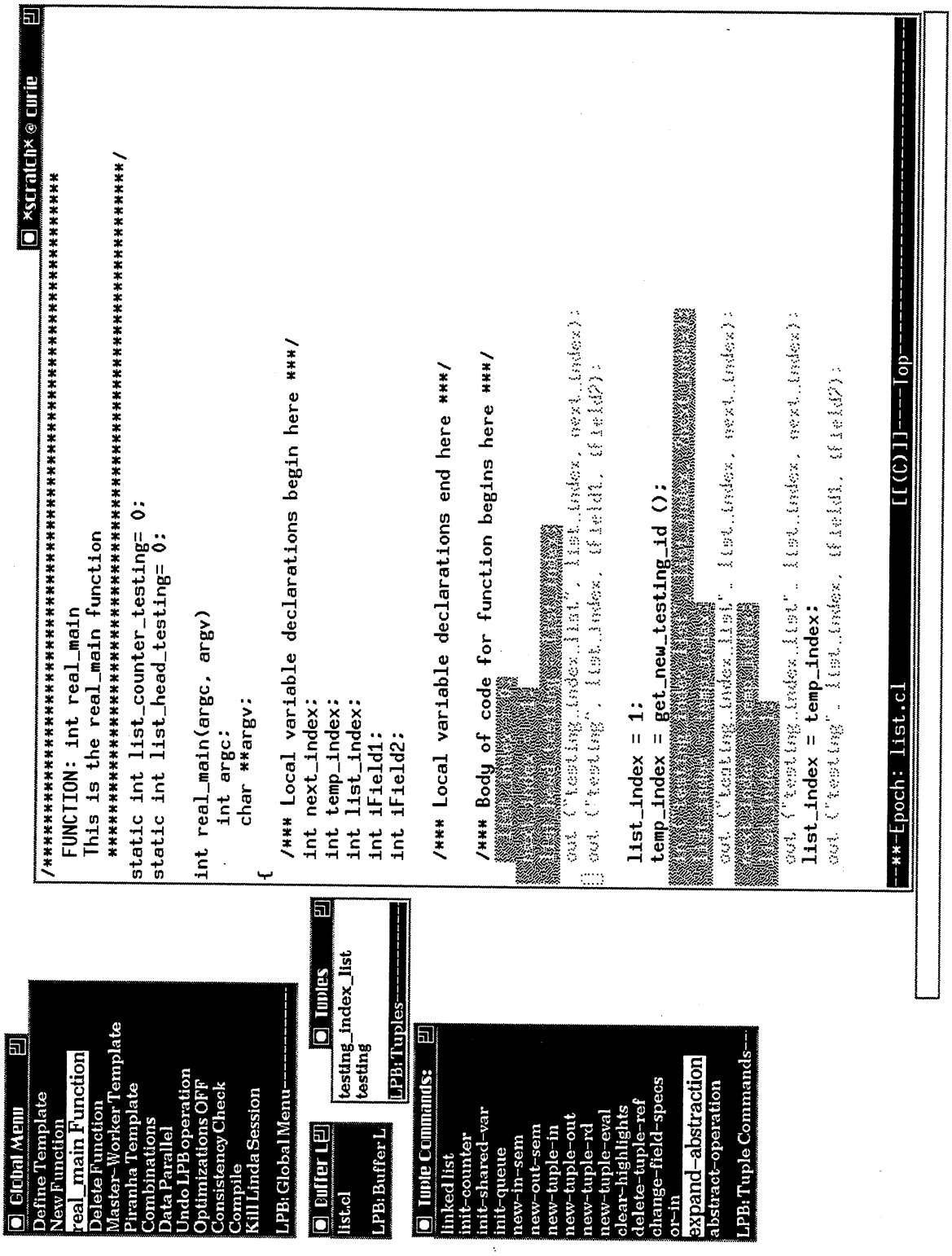

Figure 3.8: Linked List Operation After Expansion

LPB option	Linda code
initialize shared	list_index = 0;

linked list	next_index = -1;
	list_head_*name* = list_index;
	out ("*name*_index_list", list_index, next_index);
	out ("*name*", list_index, field1);

insert node	list_index = i;
	temp_index = get_new_*name*_id ();
	in ("*name*_index_list", list_index, ?next_index);
	list_index = temp_index;
	out ("*name*_index_list", list_index, next_index);
	next_index = temp_index;
	list_index = i;
	out ("*name*_index_list", list_index, next_index);
	list_index = temp_index;
	out ("*name*", list_index, field1);

remove node	list_index = i;
	if (list_index == list_head_*name*) {
	in ("*name*_index_list", list_index, ?next_index);
	list_head_*name* = next_index;
	}
	else {
	in ("*name*_index_list", list_index, ?next_index);
	temp_index = next_index;
	next_index = list_index;
	in ("*name*_index_list", ?list_index, next_index);
	next_index = temp_index;
	out ("*name*_index_list", list_index, next_index);
	list_index = i;
	}
	in ("*name*", list_index, ?field1);

Table 3.7: Shared Linked List Options and their Linda Equivalents

category's label, the variables used in its fields, the status of each of the fields, information on the nature of the tuple category and its use, and a record of all the places where references to the tuple category exist. The database also keeps track of which buttons have been expanded, and of the general state of the template.

The archive is global across a user's LPB sessions. It is saved together with the program files, and automatically loaded when a file is read in.

3.5.1 Database Implementation

Since the LPB itself is implemented in epoch and uses epoch-Lisp, the database consists of a series of lists. The main components to the database are the tuple table, the function table, and the expansions table. When the database is saved, the lists are put into functions and saved as function calls which can simply be loaded and executed to initialize the database.

The tuple table holds information on all tuples, including distributed data structures. Each entry holds information on the variables of the fields in the tuple category, the type of the tuple (i.e. what kind of distributed data structure it represents), a list of places where there are references to the tuple category, and some information on the nature of the tuple category. The list of references is itself a list which contains place references, operation types, as well as lists of each field and their respective statuses.

The function table keeps track of functions in the program. This includes noting the start and end points, return type of the function, and various pointers to different parts of the function. Figure 3.9 shows the basic data structure layout for the tuple and function tables. The following program is listed to demonstrate how the LPB maintains its database. The program tests counters, shared variables and shared linked lists. Some excerpts from the tuple table and function table follow the program listing.

```
/*****************************************************************
  FUNCTION: int real_main
  This is the real_main function
*****************************************************************/
int real_main(argc, argv)
     int argc;
     char **argv;
{
  /*** Local variable declarations begin here ***/
  int do_counter();
  int do_shared();
  int do_list();
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  eval (``process'', do_counter());
  eval (``process'', do_shared());
```

```
  eval (``process'', do_list());

}


int do_counter()

{
  /*** Local variable declarations begin here ***/
  long ctr;
  int i;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  ctr = 1;
  out (``counter_ctr'', ctr);

  for (i = 0; i < 15; i++) {
    in (``counter_ctr'', ?ctr);
    out (``counter_ctr'', ctr+1);

  }
}

static int list_counter_lst= 0;
static int list_head_lst= 0;

int do_list()

{
  /*** Local variable declarations begin here ***/
  int next_index;
  int temp_index;
  int list_index;
  int iMem;
  int i;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  iMem = 20;
  list_index = 0;
  next_index = -1;
  list_head_lst = list_index;
  out (``lst_index_list'', list_index, next_index);
  out (``lst'', list_index, iMem);

  for (i = 0; i < 10; i++) {
```

```
    list_index = i;
    temp_index = get_new_lst_id ();
    in (''lst_index_list'', list_index, ?next_index);
    list_index = temp_index;
    out (''lst_index_list'', list_index, next_index);
    next_index = temp_index;
    list_index = i;
    out (''lst_index_list'', list_index, next_index);
    list_index = temp_index;
    iMem += 5;
    out (''lst'', list_index, iMem);

  }


  for (i = 2; i < 3; i++) {
    list_index = i;
    if (list_index == list_head_lst) {
      in (''lst_index_list'', list_index, ?next_index);
      list_head_lst = next_index;
    }
    else {
      in (''lst_index_list'', list_index, ?next_index);
      temp_index = next_index;
      next_index = list_index;
      in (''lst_index_list'', ?list_index, next_index);
      next_index = temp_index;
      out (''lst_index_list'', list_index, next_index);
      list_index = i;
    }
    in (''lst'', list_index, ?iMem);

  }
}



int do_shared()

{
  /*** Local variable declarations begin here ***/
  int len_arr;
  int iIndx;
  char arr[7];
  int i;
  /*** Local variable declarations end here ***/
```

```
  /*** Body of code for function begins here ***/
  iIndx = 3;
  strcpy (arr, ``green'');
  out (``shared_iIndx'', iIndx, arr:);
  for (i = 0; i < 15; i++) {
    in (``shared_iIndx'', ?iIndx, ?arr:len_arr);
    out (``shared_iIndx'', iIndx+3, arr:);

  }
}

/*****************************************************************
FUNCTION: int get_new_lst_id
This returns new unique identifiers for the link nodes
*****************************************************************/
int get_new_lst_id()
{
  /*** Local variable declarations begin here ***/

  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  return ++list_counter_lst;

}
```

The above program generates a tuple table in the database. The following three tuple entries are taken out of the database as it is saved, to demonstrate how the information is stored. For each tuple, we have a tuple name, a list of variables, a parsed list of variables, a tuple type, a list of places and the information that will appear in the tuple information window. The list of places itself consists of a position number and file name for a reference to that tuple, the operation that is performed on that tuple, followed by a list of statuses of fields in that tuple operation.

```
(add-to-table
   '((``process''
      ``int do_counter();
''
      ((``int'' ``do_counter'' (function ``()'')))
      ``tuple''
      ((553 ``example.cl'' eval ((actual ``do_counter'')))
       (490 ``example.cl'' eval ((actual ``do_counter'')))
       (520 ``example.cl'' eval ((actual ``do_counter''))) )
       ``Types of fields in tuple:
int  FUNCTION, params:()
```

```
'')

     (``lst''
      ``int list_index;
int iMem;
''
     ((``int'' ``list_index'' (simple)) (``int'' ``iMem'' (simple)))
     ``list''
     ((2309 ``example.cl'' in ((formal ``list_index'')
                               (formal ``iMem'')))
      (1784 ``example.cl'' out ((actual ``list_index'')
                                (actual ``iMem'')))
      (1383 ``example.cl'' out ((actual ``list_index'')
                                (actual ``iMem'''))) )
     ``Information on list lst
Types of fields in tuple:
int
int
'')

     (``counter_ctr''
      ``long ctr;
''
     ((``long'' ``ctr'' (simple)))
     ``counter''
     ((863 ``example.cl'' inc-null ((actual ``ctr'')))
      (895 ``example.cl'' inc ((actual ``ctr'')))
      (801 ``example.cl'' out ((actual ``ctr''))) )
      ``This is a counter variable tuple.
Types of fields in tuple:
long
'')
```

The same program constructed the function table listed below. Each function entry lists the file the function resides in, the name of the function, the type of the function, the starting point for the text of the function, the ending point of the parameter declarations, the starting and ending points of the local variables, and the ending point for the function.

```
(add-to-function-table '(((``example.cl'' ``do_shared'' ``int'' )
  (2325 2347 )
  (2753 )
  (2400 2452 ))
```

```
((``example.cl'' ``do_list'' ``int'' )
 (914 996 )
 (2325 )
 (1049 1124 ))
((``example.cl'' ``real_main'' ``int'' )
 (194 253 )
 (569 )
 (306 362 ))
((``example.cl'' ``do_counter'' ``int'' )
 (569 592 )
 (914 )
 (645 666 ))
((``example.cl'' ``get_new_lst_id'' ``int'' )
 (2969 2990 )
 (3182 )
 (3043 3046 ))
))
```

Finally, the expansion table maintains the list of abstractions and expansions. The table includes locations, status (whether it is currently expanded or abstracted), and what functions to call for expansion or abstraction along with a list of arguments for them. The format is similar to the ones above.

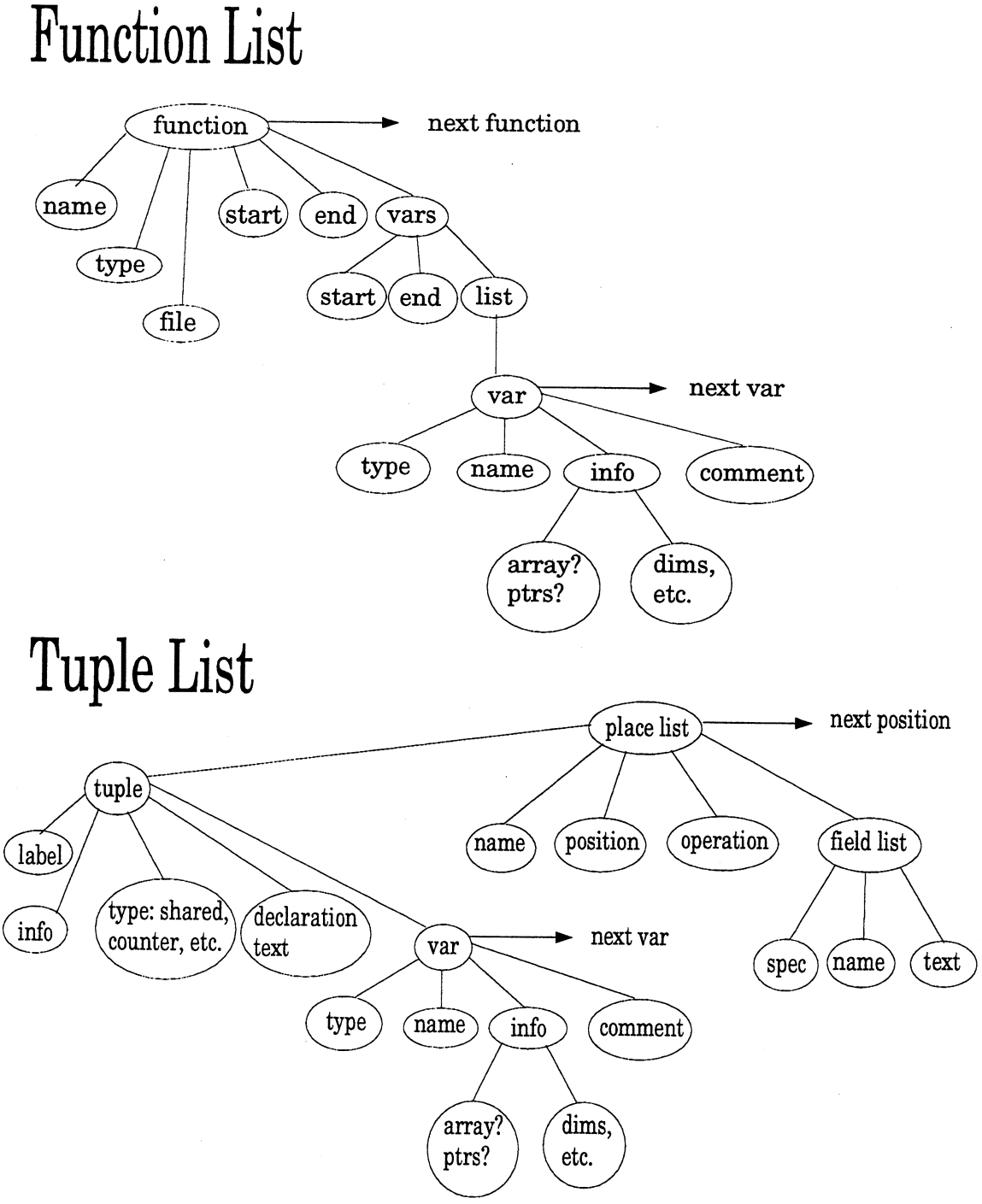

Figure 3.9: Tuple and Function Table Layouts

Chapter 4

Interaction With Other Tools

A major goal in designing the LPB was to build an integrated development environment. The LPB by itself is useful for the first phase of program construction, namely the coding. There are another three important phases: compiling, debugging and performance analysis. An integrated environment needs to support these other phases too. The LPB environment supports construction, compilation and debugging phases of a program, and although the infrastructure for performance analysis support exists, it was not implemented.

Users provide information to the LPB which the LPB organizes and passes on to other tools. During the program construction phase, the LPB gathers information which not only aids the coding, but is also extremely useful for supporting the compilation and debugging phases. By constructing a program, the programmer provides a fair amount of information to the LPB. By selecting a master-worker template, for example, the programmer tells the LPB that there will be a master process and some worker processes.

The information acquired by the LPB may also relate to data structures. For example, the LPB may be told that a shared linked list will be used and thereafter the LPB generates all operations for manipulating that list, i.e. the LPB has information on which operations scattered throughout the code are working towards the common goal of updating a shared data structure.

There are two very important ways in which this kind of information can be useful: (1) The compiler can optimize code based on this information, and (2) the debugger or visualizer can use the same information to better visualize the execution of a program.

The Linda compiler [Bjo90] and the visualizer (Tuplescope [BC90]) were modified to enhance performance based on semantic information passed from the LPB. Although the compiler has been modified to compile programs with LPB-supplied semantic information, portability of the code was not affected. As we shall see, a standard C-Linda program is always generated, and this can be passed to other users who do not have the LPB or optimizing tools of its environment.

To understand how the modified compiler and Tuplescope handle LPB-supplied information, an understanding of the original compiler is necessary. We shall thus first present a brief discussion on the original compiler and then describe how the compiler and Tuplescope were modified.

4.1 The Original Compiler

Over the years, a number of Linda compiler versions have been built. The compiler that was used for the LPB project was v2.4d[1] — we refer to this as the standard system. The complete details of the compiler implementation, while interesting, are not relevant to the LPB and they can be found in a number of standard Linda references ([Car87], [Bjo90], and [CG92]). We shall concentrate on those aspects of the compiler that were affected by the LPB information. Two key components of the compiler needed modifications to smoothly integrate it with the LPB: (1) The analyzer and (2) the run-time library.

4.1.1 The Analyzer

The analyzer's function is to streamline associative memory lookup and provide information to manage tuple space at runtime. It partitions tuples into different categories and decides on a series of runtime functions to call for implementing the tuple operations.

A *tuple pattern* is associated with the text of each tuple operation. The different fields in a tuple operation constitute the pattern. The argument list of an **out** or **eval** defines an *out-pattern*; the argument list of an **in** or **rd** define an *in-pattern*. At runtime, tuples instantiated from the patterns are manipulated by the tuple operations: An **out** or **eval** operation generates a tuple matching the pattern of its argument list and puts it out to tuple space; an **in** or **rd** operation builds an *anti-tuple* of the pattern specified in the argument list. The runtime system then tries to match a tuple to the anti-tuple. The fields that specify a pattern may be *actuals* or *formals*. Actuals are data values, while formals are place holders that are typically used with **ins** and **rds**. Tuple matching is associative and the easiest implementation would simply keep all tuples in a single, large collection. Unfortunately, this would mean searching through the entire collection for a match to an anti-tuple.

An exhaustive search can be avoided by doing some work at compile-time. The analyzer does something more sophisticated than an exhaustive search. Patterns can differ (and thus never match) in two ways: (1) The number of fields may differ, and (2) the types of the fields may differ. Hence, tuples generated from an out-pattern with four fields can never match anti-tuples generated from an in-pattern with three fields. Similarly, if there is a difference in type of a particular field between a tuple and

[1]The version number comes from the Linda compiler of Scientific Computing Associates, New Haven

an anti-tuple, the two will not match. Consequently, the analyzer can partition tuple space such that tuple patterns that can never match are not put in the same partition.

The analyzer scans the program for all tuple operations and creates partitions based on the tuple patterns. Tuples and anti-tuples instantiated from these patterns will be placed in the corresponding partitions at runtime. Consequently, when searching for a tuple to match an anti-tuple, the runtime system will immediately know which partition to search in and an exhaustive search of all tuples is no longer necessary.

The analyzer partitions further by applying a weaker version of the matching rules to each partition in turn. The analyzer assumes that any fields which depend on run-time data will match, i.e. fields that are non-constant actuals will match corresponding fields in the relevant tuple or anti-tuple. This follows from Linda semantics since the compiler cannot know *a priori* what the runtime values of the fields will be. Constant actuals that are equal trivially match. Following these rules, the analyzer constructs list of matchables for each pattern. When this stage is complete, the lists are recursively collapsed to form matchability chains. Each of these chains becomes a new partition.

Each partition is scanned by the analyzer to determine patterns of field usage. From the information it acquires at this stage, the analyzer selects a management paradigm for the partition and generates the appropriate calls to the runtime library routines that implement the tuple operations with the selected paradigm. Each of these library routines is known as a *handler*.

4.1.2 The Runtime Library

The analyzer selects one of five tuple management paradigms for each partition: hash, counting semaphore, queue, private hash (a variation on hash), and list (exhaustive search). The last is rarely needed.

At runtime, a tuple is packed into something called a proto-tuple-packet or *ptp*. A ptp contains a type signature and information about each field of the tuple. When a tuple is put out to tuple space, the responsible process builds the ptp structure and calls the handler routine that implements the tuple operation. The handler routine locks the appropriate shared data structures and transfers the information from the ptp structure into the shared space. When a tuple is removed from tuple space, the reverse of this happens.

It is important to note that each tuple operation uses only one ptp structure. The handlers can only operate on one ptp structure at a time. As we shall see, this fact proved to be a hindrance to some potential optimizations of the new, LPB-optimized compiler.

4.2 The New, LPB-Optimized Compiler

Given certain sequences of operations, the standard pre-compiler may draw conclusions about the intended effects, but (like any compiler) it cannot in general infer the

user's intent in specifying particular sequences of operations. The LPB has superior information in this regard.

Since the program is being constructed through templates or other higher level conceptual frameworks, the LPB "knows" why the various operations are being used. For example, the LPB knows that certain pairs of tuple operations are associated with "counter" tuples which will always be updated in certain ways. Given a particular distributed data structure, the LPB knows which tuple operations need to be used to create and manipulate it.

This knowledge is valuable to a compiler. Given an understanding of what a series of Linda operations is intended to achieve, the compiler can transform the series into a semantically equivalent series of operations that are more efficient. In some cases the LPB can fuse these operations together and perform a smaller number of discrete operations over tuple space. We demonstrate this with three examples of increasing complexity.

4.2.1 Shared Counters

A shared counter is represented by a tuple in tuple space with an integer field that contains the value of the shared counter (Section 3.2.1). There is at most one instance of this tuple in tuple space at any time. An update consists of consuming the tuple, updating the value, and generating a tuple to replace the old one. Users who don't have the LPB, explicitly write pairs of **in/out** operations to update counters. The first (**in**) operation simply consumes the tuple that holds the counter. The second (**out**) operation introduces an updated tuple with the counter value incremented or decremented by one.

The LPB provides a shorthand mechanism for dealing with counters. With the LPB, the user defines a counter (using the relevant menu option), and thereafter selects an **increment** or **decrement** operation on this counter. The LPB then inserts the appropriate **in** and **out** commands into the code. While doing this, the LPB notes that the two operations are related to each other because they are both updating a counter. Furthermore, the LPB notes that there can only be one instance of a counter in tuple space at a time. Since the **in** and **out** commands have been inserted into the code, a standard compiler would understand this code as if a user had typed in these commands from scratch.

When the Linda pre-compiler parses and analyzes operations, it links certain handlers to particular tuple operations. For example, consider the following C-Linda code generated by the LPB:

```
/*****************************************************************
FUNCTION: int real_main
This is the real_main function
*****************************************************************/
```

```
int real_main(argc, argv)
     int argc;
     char **argv;
{
  /*** Local variable declarations begin here ***/
  long demo;

  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/

  demo = 1;
  out ("counter_demo", demo);          /* initialize counter */

  in ("counter_demo", ?demo);          /* increment counter */
  out ("counter_demo", demo+1);

  in ("counter_demo", ?demo);          /* decrement counter */
  out ("counter_demo", demo-1);

}
```

To generate this code, the user simply clicked on the `initialize counter` menu option, gave the counter a name, and then clicked on the `increment counter` and `decrement counter` options for that particular counter.

The original compiler (v2.4d) analyzes this code and places the tuple patterns into a single partition since all of the patterns potentially match. It then decides on a management scheme for this partition and replaces each of the tuple operations with calls to unique routines. These routines fill ptp structures and call the handlers for the chosen management scheme. The above tuple operations thus get replaced with calls to the following routines:

```
int real_main(argc, argv)
     int argc;
     char **argv;
{
  long demo;

  demo = 1;
  __lo_sdEWgH( demo);                          /* initialize counter */

  __lo_47IucU(&(  demo));                      /* increment counter */
  __lo_yPEjjL( demo+1);
```

```
  __lo_h5mGje(&(  demo));                          /* decrement counter */
  __lo_CJlHDs( demo-1);
}
```

The routines themselves are listed below:

```
int __lo_sdEWgH(arg1)                              /* replaces first out */
        long    arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_sdEWgH[1];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_sdEWgH[0].actual.il = arg1;             /* copy the actual */
  ptp.field_statics = psf___lo_sdEWgH;             /* fill ptp structure */
  ptp.field_vars = pvf___lo_sdEWgH;
  ptp.statics = &psd___lo_sdEWgH;
  out_queue(&ptp);                                 /* call queue handler */
}

static PTP_STATIC_FIELD_TYPE    psf___lo_47IucU[1]={
{1, 1, 6},
};

static PTP_STATICS_TYPE psd___lo_47IucU={1, 5, -1, 19, "count_ex.cl",
1, 22, 0, 2};

int __lo_47IucU(arg1)                              /* replaces first in */
        long    *arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_47IucU[1];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_47IucU[0].formal.il = arg1;             /* note ptr to formal */
  ptp.field_statics = psf___lo_47IucU;             /* fill ptp structure */
  ptp.field_vars = pvf___lo_47IucU;
  ptp.statics = &psd___lo_47IucU;
  in_queue(&ptp, 0, 1);                            /* call queue handler */
}

static PTP_STATIC_FIELD_TYPE    psf___lo_yPEjjL[1]={
```

```
{1, 0, 6},
};

static PTP_STATICS_TYPE psd___lo_yPEjjL={1, 5, -1, 20, "count_ex.cl",
1, 23, 0, 2};

int __lo_yPEjjL(arg1)                          /* replaces second out */
        long    arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_yPEjjL[1];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_yPEjjL[0].actual.il = arg1;         /* copy the actual */
  ptp.field_statics = psf___lo_yPEjjL;         /* fill ptp structure */
  ptp.field_vars = pvf___lo_yPEjjL;
  ptp.statics = &psd___lo_yPEjjL;
  out_queue(&ptp);                             /* call queue handler */
}

static PTP_STATIC_FIELD_TYPE    psf___lo_h5mGje[1]={
{1, 1, 6},
};

static PTP_STATICS_TYPE psd___lo_h5mGje={1, 5, -1, 22, "count_ex.cl",
1, 24, 0, 2};

int __lo_h5mGje(arg1)                          /* replaces second in */
        long    *arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_h5mGje[1];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_h5mGje[0].formal.il = arg1;         /* note ptr to formal */
  ptp.field_statics = psf___lo_h5mGje;         /* fill ptp structure */
  ptp.field_vars = pvf___lo_h5mGje;
  ptp.statics = &psd___lo_h5mGje;
  in_queue(&ptp, 0, 1);                        /* call queue handler */
}

static PTP_STATIC_FIELD_TYPE    psf___lo_CJlHDs[1]={
```

```
{1, 0, 6},
};

static PTP_STATICS_TYPE psd___lo_CJlHDs={1, 5, -1, 23, "count_ex.cl",
1, 25, 0, 2};

int __lo_CJlHDs(arg1)                          /* replaces last out */
        long    arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_CJlHDs[1];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_CJlHDs[0].actual.il = arg1;         /* copy the actual */
  ptp.field_statics = psf___lo_CJlHDs;         /* fill ptp structure */
  ptp.field_vars = pvf___lo_CJlHDs;
  ptp.statics = &psd___lo_CJlHDs;
  out_queue(&ptp);                             /* call queue handler */
}
```

The generated routines show that the analyzer chose the "queue" management scheme. The calls to the in_queue and out_queue routines are calls to the handlers that manage the queue. The outline for the code of these routines from the runtime library looks like this:

```
in_queue:

        lock queue
        while (queue empty)
                if nonblocking return 0

                add myself to waiting list for queue
                unlock queue
                go to sleep         ;;; see below for wake up call
                lock queue

        if rd
                copy data from first element on queue
                unlock
                return 1
        else
                remove first element
                unlock
```

```
                copy
                free element storage
                return 1


out_queue:

        allocate element storage
        lock queue
        add elements to tuple space
        wake up waiters
        unlock queue
```

From the compiler-generated code and the outlined runtime library routines, we can see that each increment or decrement operation involves two calls to library routines. Each of these routines (`in_queue` and `out_queue`) has to acquire a lock and later release the lock. Thus, every increment or decrement operation on a counter requires four lock acquisition and release operations.

The LPB-enhanced compiler treats things differently. Once the user has defined a counter, the LPB knows that apart from initialization, a counter update will always involve a pair of **in** and **out** operations. Every counter update is noted as two *special* tuple operations in a file. The compiler reads this LPB-generated information file and scans the list of "special" tuple operations. The compiler places these tuples into their own partitions and marks them with special handlers, tailored to meet their specific needs. These handlers are different from the ones that a conventional compiler would assign for these tuples. In particular, for counters, the LPB actually marks the **in** operation in an increment or decrement with special increment or decrement handlers. The second (**out**) operation, however, is marked as a "no op" and is thus ignored at runtime — the analyzer generates only a procedure stub for the operation and does not call any handler. The main function gets modified with calls to the proper runtime routines:

```
int real_main(argc, argv)
     int argc;
     char **argv;
{
  long demo;

  demo = 1;
  __lo_lx618b( demo);                    /* initialize counter */

  __lo_qZIeA9(&(  demo));                /* increment counter */
  __lo_3RXfcs( demo+1);
```

```
  __lo_To6ccf(&(  demo));                    /* decrement counter */
  __lo_qJDtcG( demo-1);
}
```

The routines themselves are:

```
static PTP_STATIC_FIELD_TYPE    psf___lo_lx618b[2]={
{0, 0, 1},
{1, 0, 6},
};

static PTP_STATICS_TYPE psd___lo_lx618b={1, 6, -1, 17, "count_ex.cl",
2, 25, 0, 0};

int __lo_lx618b(arg1)     /* replaces the first out, the initialization */
        long    arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_lx618b[2];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_lx618b[0].actual.b.data = "counter_demo";
      /* Normally, this data is suppressed and could be here too, but
         since the main part of the optimizations come from the
         handlers, we don't bother */
  pvf___lo_lx618b[0].actual.b.size = 13;
  pvf___lo_lx618b[1].actual.il = arg1;
  ptp.field_statics = psf___lo_lx618b;   /* fill ptp structure */
  ptp.field_vars = pvf___lo_lx618b;
  ptp.statics = &psd___lo_lx618b;
  out_counter(&ptp);                     /* call special counter handler */
  if (!status) return 0;

}

static PTP_STATIC_FIELD_TYPE    psf___lo_qZIeA9[2]={
{0, 0, 1},
{1, 1, 6},
};

static PTP_STATICS_TYPE psd___lo_qZIeA9={1, 6, -1, 19, "count_ex.cl",
2, 24, 0, 0};
```

```
int __lo_qZIeA9(arg1)    /* replaces the first in, i.e. the increment */
        long    *arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_qZIeA9[2];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_qZIeA9[0].actual.b.data = "counter_demo";
  pvf___lo_qZIeA9[0].actual.b.size = 13;
  pvf___lo_qZIeA9[1].formal.il = arg1;
  ptp.field_statics = psf___lo_qZIeA9;   /* fill ptp structure */
  ptp.field_vars = pvf___lo_qZIeA9;
  ptp.statics = &psd___lo_qZIeA9;
  inc_counter(&ptp);
      /* call special handler to increment counter in one step */
  if (!status) return 0;

}

static PTP_STATIC_FIELD_TYPE    psf___lo_3RXfcs[2]={
{0, 0, 1},
{1, 0, 6},
};

static PTP_STATICS_TYPE psd___lo_3RXfcs={1, 6, -1, 20, "count_ex.cl",
2, 23, 0, 0};

int __lo_3RXfcs(arg1)   /* This is just a stub with no call to any handler */
        long    arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_3RXfcs[2];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_3RXfcs[0].actual.b.data = "counter_demo";
  pvf___lo_3RXfcs[0].actual.b.size = 13;
  pvf___lo_3RXfcs[1].actual.il = arg1;
  ptp.field_statics = psf___lo_3RXfcs;
  ptp.field_vars = pvf___lo_3RXfcs;
  ptp.statics = &psd___lo_3RXfcs;
        /* Note that no handler is called */
```

```
  if (!status) return 0;

}

static PTP_STATIC_FIELD_TYPE    psf___lo_To6ccf[2]={
{0, 0, 1},
{1, 1, 6},
};

static PTP_STATICS_TYPE psd___lo_To6ccf={1, 6, -1, 22, "count_ex.cl",
2, 22, 0, 0};

int __lo_To6ccf(arg1)    /* replaces the second in, i.e. the decrement */
        long    *arg1;
{
  PTP_VAR_FIELD_TYPE    pvf___lo_To6ccf[2];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_To6ccf[0].actual.b.data = "counter_demo";
  pvf___lo_To6ccf[0].actual.b.size = 13;
  pvf___lo_To6ccf[1].formal.il = arg1;
  ptp.field_statics = psf___lo_To6ccf;   /* fill ptp structure */
  ptp.field_vars = pvf___lo_To6ccf;
  ptp.statics = &psd___lo_To6ccf;
  dec_counter(&ptp);
      /* call special handler to decrement counter in one step */
  if (!status) return 0;

}

static PTP_STATIC_FIELD_TYPE    psf___lo_qJDtcG[2]={
{0, 0, 1},
{1, 0, 6},
};

static PTP_STATICS_TYPE psd___lo_qJDtcG={1, 6, -1, 23, "count_ex.cl",
2, 21, 0, 0};

int __lo_qJDtcG(arg1)   /* This is just a stub with no call to any handler */
        long    arg1;
{
```

```
  PTP_VAR_FIELD_TYPE    pvf___lo_qJDtcG[2];
  PTP_TYPE      ptp;
  int           status;

  pvf___lo_qJDtcG[0].actual.b.data = "counter_demo";
  pvf___lo_qJDtcG[0].actual.b.size = 13;
  pvf___lo_qJDtcG[1].actual.il = arg1;
  ptp.field_statics = psf___lo_qJDtcG;
  ptp.field_vars = pvf___lo_qJDtcG;
  ptp.statics = &psd___lo_qJDtcG;
        /* Note that no handler is called */
  if (!status) return 0;

}
```

The code generated by the LPB-optimized compiler is clearly different from the code generated by the original compiler (v2.4d). The routines that replace the **out** calls in the counter updates are mere procedure stubs — they do not call any handlers. Each pair of **in** and **out** operations has become one **increment** or **decrement** operation. Hence, at the **in** stage, the appropriate handler is called and at the **out** stage, no handler is called.

The LPB-optimized compiler has chosen the "counter" tuple management scheme over the "queue" scheme. This follows from the directions the LPB gives to the compiler. There are three counter handlers called in this example: an `out_counter` routine that is called when the counter is initialized, an `inc_counter` routine that is called to increment the counter, and a `dec_counter` routine that is called to decrement the counter.

The three counter handlers are listed below:

```
int     out_counter(ptp_ptr)      /* handler for initializing counters */
register PTP_TYPE       *ptp_ptr;
{
  register COUNTER_TYPE *c_ptr;

  c_ptr = ptp_ptr->statics->set_id + linda_c_tab;
        /* Acquire ptr to shared structure to hold counter */

  _spinlock(&(c_ptr->lock));
  if (c_ptr->state == COUNTER_INIT) {
    fprintf(stderr, "misused counter: ");
        /* shouldn't have more than one instance of a counter tuple */
  }
```

```
  else {
    c_ptr->counter = ptp_ptr->field_vars[1].actual.il;
        /* Copy data from ptp structure to shared data space */
    c_ptr->state = COUNTER_INIT;
  }
  _spinunlock(&(c_ptr->lock));
}


int     inc_counter(ptp_ptr)      /* handler for incrementing counters */
register PTP_TYPE       *ptp_ptr;
{
  register COUNTER_TYPE *c_ptr;

  c_ptr = ptp_ptr->statics->set_id + linda_c_tab;
        /* Acquire ptr to shared structure to hold counter */

  _spinlock(&(c_ptr->lock));
  if (c_ptr->state == COUNTER_UNINIT) {
    fprintf(stderr, "misused counter: ");
        /* shouldn't be updating an uninitialized counter */
  }
  else {
    if (ptp_ptr->field_vars[1].formal.il) {
      *ptp_ptr->field_vars[1].formal.il = c_ptr->counter++;
        /* update counter value in place (rhs) and copy to local var */
    }
  }
  _spinunlock(&(c_ptr->lock));
}


int     dec_counter(ptp_ptr)      /* handler for decrementing counters */
register PTP_TYPE       *ptp_ptr;
{
  register COUNTER_TYPE *c_ptr;

  c_ptr = ptp_ptr->statics->set_id + linda_c_tab;
        /* Acquire ptr to shared structure to hold counter */

  _spinlock(&(c_ptr->lock));
  if (c_ptr->state == COUNTER_UNINIT) {
```

```
    fprintf(stderr, "misused counter: ");
        /* update counter value in place (rhs) and copy to local var */
  }
  else {
    *ptp_ptr->field_vars[1].formal.il = c_ptr->counter--;
        /* update counter value in place (rhs) and copy to local var */
  }
  _spinunlock(&(c_ptr->lock));
}
```

The increment or decrement handler locks the counter variable in tuple space, updates it *in place*, and then releases the lock. In contrast, if a compiler compiled the code with no LPB-supplied information, both the **in** and **out** operations would be carried out, each requiring a lock to access the tuple, followed by a release of the lock. Thus, the LPB-optimized complier saves two lock acquisition and release operations and avoids an additional tuple operation. Knowing enough to throw away the second operation entirely is very useful. As mentioned earlier (Section 4.1.2), the original compiler was built to deal with only one tuple operation data structure at a time. For this optimization, we are actually using two tuple operations at one time. Yet there is no way to access both simultaneously. Hence, throwing out the second operation is fruitful because we use the data from the first operation only. If we were to need data from the second operation too, we would need to modify the compiler to handle data from two or more tuple operations at one time, a change that proved to be necessary for other optimizations.

This optimization of counters shows impressive results. For a sequence of fifty thousand counter increments and fifty thousand counter decrements on a Sequent Symmetry with 16 processors, the v2.4d compiler took 7.54 seconds. The new LPB-optimized compiler took 2.98 seconds for the same sequence, better than a factor of two reduction. Since we not only eliminate an entire operation in this optimization, but reduce space and overhead constraints for even the single operation, this improvement by better than a factor of two is no surprise. Running the same test on a Sun Sparcstation 2 and Sparcstation 10 yielded almost identical factors of improvement.

Although the improvements were similar on the various machines, the reasons for the improvements need not be the same in all cases. In addition to eliminating the acquisition and release of two locks, the LPB-optimized compiler eliminates an entire operation and thus reduces the number of machine instructions executed. If the cost of acquiring and releasing a lock is high, it will dominate the time cost; if it is low, the time cost will be driven by the number of operations. To determine what the costs of acquiring and releasing locks are, a number of tests were run on a Sparc 10.

We know that the number of lock acquiring and releasing operations has been decreased by two. We also know that the C code for the LPB-optimized counter operations is shorter than the standard compiler-generated code. Unless a particular

Compiler	With Locks 200K incs/decs	Without Locks 200K incs/decs	Without Locks 2M incs/decs
standard (v2.4d)	144.40 s	1.26 s	12.61 s
LPB-optimized	80.53 s	0.47 s	4.68 s

Table 4.1: Timings for Counter Increments and Decrements with and without Locks

machine instruction or system call takes significantly longer than all others, the number of machine instructions executed is usually a good indicator of the time a code portion takes to execute.

For the standard (v2.4d) compiler, an increment or decrement operation is translated into an `in_queue` and `out_queue` operation. The total number of machine instructions for this sequence is 596. For the LPB-optimized compiler, an increment or decrement gets translated into the corresponding increment and decrement handlers. This only involves 234 machine instructions. Consequently, the timing improvements are in line with the number of instructions executed. Acquiring and releasing locks, however, can require very expensive system calls. To determine whether our timing improvements are coming from the reduced number of instructions or the reduced number of lock acquiring and releasing operations, some further tests were necessary.

For the purpose of the tests, the runtime library was modified slightly. A flag was introduced which is true when all locks are to be ignored. This flag can be dynamically set from within a program. This enables users to avoid acquiring and releasing locks. For a single threaded program, this does not lead to any contention problems. A test program was constructed to increment and decrement a counter and this was compiled with both the standard (v2.4d) compiler and the LPB-optimized compiler. The program was then run both with locks and without locks. The results are shown in table 4.1.

The results clearly show that acquiring and releasing locks eats up the bulk of time. The differences in time between the lockless and the normal versions are larger than a factor of 100. This definitely means that the improvements in performance of the LPB-optimized code over the standard (v2.4d) code are the result of eliminating some lock operations. Acquiring locks on the Sparc 10 is expensive since locks are implemented through the operating system using scheduling software. Consequently, lock acquisition time dominates the time required for counter operations.

For machines where lock operations are cheap, the machine operation count would play a more dominant role. This is the case with the Sequent Symmetry which has hardware support for lock acquisition. Since the machine operation count for the LPB-optimized code is less than half that of the normal code, however, one would expect similar factors of performance improvement even if the lock operation cost were very

low, and the results support this expectation.

4.2.2 Shared Variables

A similar optimization to that for counters was implemented for general shared variables (Section 3.2), although this case is slightly more complex. Updating a shared variable involves removing a tuple from tuple space, binding the fields to local variables, generating a new tuple to contain the updated values and putting this new tuple into tuple space. But what is happening conceptually? Conceptually, the shared variable in tuple space has an evolving state. Although different processes update the shared variable by removing a tuple from tuple space and replacing it with an updated tuple, the shared variable is always there conceptually. This is similar to the case of counters, except for a key distinction: There are no restrictions on the update operation. Consequently, it is much harder to migrate the operation to the Linda runtime kernel. Any of the variable fields in the second operation of the **in-out** pair could be function calls. Since we don't know *a priori* what such a function call will evaluate into, we cannot ignore the second operation as we did in the case of shared counters. For a shared counter, the update is either an increment or a decrement and since we know from the LPB exactly which of the two it is, we can update the value without evaluating the second operation. For general shared variables, we no longer have this luxury.

The original (v2.4d) compiler treats shared variable manipulations just as it treats counters. It selects the **in_queue** and **out_queue** handlers for the **in** and **out** operations associated with the shared variable manipulation. The LPB-optimized compiler replaces the calls to these two handlers with calls to two new routines.

The implementation of this optimization faced some problems that arose because the v2.4d compiler was not implemented with an LPB interface in mind. As mentioned in section 4.1.2, the compiler was designed to handle one data structure per tuple operation. For shared variable optimizations, we need access to two tuple operation data structures simultaneously. Hence, implementing this optimization involved some tricky coding which would be eliminated when a new compiler is designed from scratch with an LPB interface in mind. For a sequence of ten thousand shared variable updates on a Sparcstation 10, the v2.4d compiler needed 16.64 seconds, whereas the LPB-optimized one required 8.62 seconds. This improvement is again in line with expectations. A total elimination of the second operation would result in a reduction in time by at least a factor of 2. Since, however, the second operation could not be completely eliminated, our reduction is slightly less than a factor of two.

As in the case of counters, the main cause of the improved performance is the reduction in number of lock operations. This becomes obvious from running the same kinds of tests as was done for counters. The cost of lock operations is clearly shown by running the same tests as was done for counters. The results appear in table 4.2. The number of machine instructions for the standard (v2.4d) compiler-generated code

Compiler	With Locks 100K modifications	Without Locks 100K modifications	Without Locks 1M modifications
standard (v2.4d)	72.71 s	0.66 s	6.42 s
LPB-optimized	37.98 s	0.90 s	8.92 s

Table 4.2: Timings for Shared Variable Modifications with and without Locks

is 803. This includes the `in_queue` and `out_queue` operations as in the case of the counter. The number of instructions that are used for acquiring and releasing locks in this are 562. For the LPB-optimized code, however, the machine instruction count is 618. The number of instructions used for acquiring and releasing locks in this is 281, exactly half of the non-optimized case.

It is interesting to note that the "without locks" timings actually favor the non-optimized version. This may seem surprising initially, but if we examine the instruction count closely, it becomes clearer. Since 562 instructions in the non-optimized version are used for lock acquiring and releasing, only 241 instructions remain in the version run without locks. For the LPB-optimized version, 281 out of 618 instructions are used for lock acquiring and releasing. That leaves 327 instructions in the version without locks, i.e. this is a larger number than the non-optimized version without locks. Consequently, the optimized version takes longer to run without locks. Clearly, for this particular optimization, the LPB-optimization is only worthwhile if the lock acquisition and release costs are relatively high. For machines with cheaper lock operation costs, the benefits of the LPB-optimized code could diminish. In fact, for machines with lock acquire/release operations of negligible cost, the LPB-optimized code may actually take longer to run since the machine instruction count is higher.

Implementation details

Since the second operation of the **in-out** pair could not be eliminated, two new handlers were required. The LPB thus notes both operations in the update with special markers and they each get their own specialized handler. The first one locks the appropriate variable field and binds the relevant variable value. The lock is not released at the end of the operation since we know that the next operation will use the same space. Hence, when the second operation is called, the lock is already in place and we do not need to acquire it, proceeding immediately with evaluation instead. The result is placed into the locked structure, and the lock is released. We thus save two lock acquire/release operations which can be expensive at runtime. We also save on space since we do not have to generate the space for a new tuple to replace an old one — instead, we recycle the space from the old tuple.

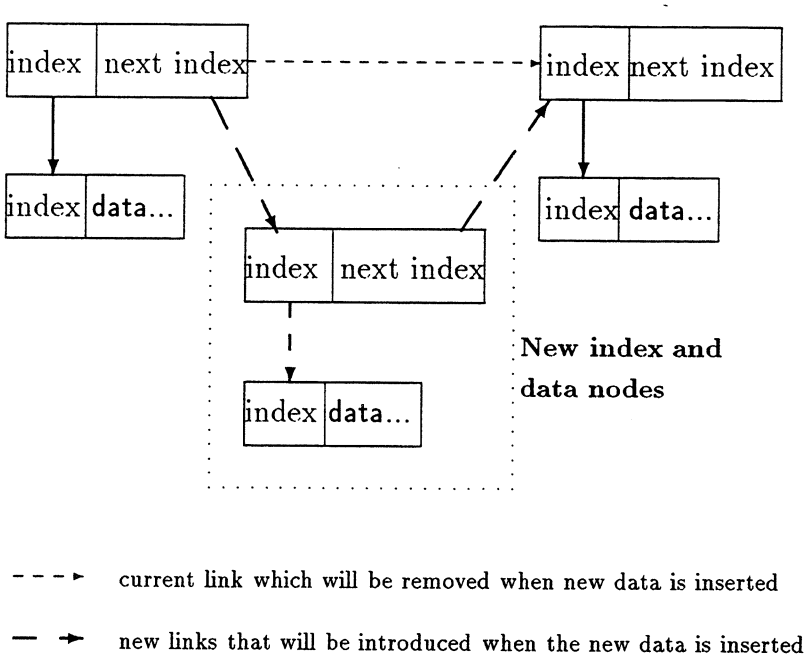

Figure 4.1: Inserting a Node in a Linked List

4.2.3 Shared Linked Lists

The optimizations for shared linked lists required some tricky modification of the compiler. A shared linked list has two levels (see Section 3.4.2, Figure 3.6). The first level is the index list, where each entry is represented by a tuple containing the index value and the next index value. The index value is the key into another tuple which actually stores the data element for that index.

Manipulating a shared linked list can be a tedious venture for a Linda programmer. Let us consider what happens when inserting a node into the list. With the LPB, the user can select an LPB option to insert a node into a list, and the appropriate abstraction is created. When expanded, there are several steps in the resultant code. Figure 4.1 shows the effect of the operations. First, the LPB inserts an **in** on the node index after which the new index node will be inserted. This node index is now modified such that its "next pointer" points to a new node index which has a unique identifier. The new node index, in turn, is modified such that its next pointer points to what the old, **in**'ed node index used to point to as the next node. The updated old index is now put **out** into tuple space and the new node index can also be put **out**. Finally, when all this has been done, the actual data node can be put **out** into tuple space. This whole exercise required four tuple operations.

Since the LPB maintains a list of all operations that manipulate linked lists, it can

pass this information to the compiler. The LPB-optimized compiler acts upon this information and fuses together many of the node index manipulation operations. In fact, all the node index updates are done in one large step after which the new data node is put out to tuple space.

When manipulating a very large list with ten thousand insert and delete node operations on a Sparcstation 10, the v2.4d compiler required 50.51 seconds. The LPB-optimized compiler ran the same sequence in 27.13 seconds. Once again, we expect the reduction in lock operations to cause the improvements in performance. Since the number of C operations is also reduced (and hence the number of machine instructions is likely to be lower), however, the improvements may hold even on machines where lock acquire/release operations are very quick.

Implementation details

The LPB-optimized compiler deals with the entire index node insertion or index node removal in one step. The problem is that we need to access multiple tuples simultaneously. We would like to update the old node tuple and insert the new one in one step. Hence we need access to both these tuples at the same time. Furthermore, we need to know two different variable values: the new index, and the old "next index." How can we accomplish this?

The trick is to "fool" the compiler into processing certain operations which are later discarded. We basically construct a dummy operation that the compiler knows how to ignore. We need a way to pass information on two tuples and some additional variable information to the compiler inside one data structure. To pass this information to the compiler, the LPB constructs a dummy **in** operation in whose fields it packs the necessary variable values. It then tags this operation as a special LPB "linked list" operation in the information file sent to the compiler. The analyzer proceeds to ignore the **in** aspect of this operation and only uses the appropriate data structure to gain access to the variables. At runtime, the compiler updates its own internally maintained linked list index with the appropriate values. Once this is done, the **out** of the actual node data can be done. This rather complex scheme proved to be rather fruitful as the numbers above show. Of course, if the compiler were redesigned such that it could better accommodate the information from the LPB, we would no longer have to resort to such tricks to achieve the optimizations (see Chapter 7). The data structures and code segments of the linked list handlers from the runtime library follow.

```
typedef struct list_table_entry {
  /* this is the building block for the internal linked list for
     node indices */
        int             iIndex;     /* index value for the node */
        int             iNext;      /* pointer to next index */
        int             iNextAbs;   /* absolute address of next index */
```

```
        SPIN_LOCK       lock;       /* lock for a particular node */
} LIST_TABLE_ENTRY;

#define LIST_TABLE_SIZE 1000

typedef struct list_entry {
  /* This table holds the internal linked list for node indices */
        LIST_TABLE_ENTRY table[LIST_TABLE_SIZE];
        int              iListHead; /* head of the list */
} LIST_TYPE;
```

The data structure is listed above and is fairly simple. In addition to keeping the index and next index values, the table also keeps the absolute address of the next index for quick access. The main runtime routines that manipulate the above data structures are listed below. We omit some of the administrative routines for the sake of clarity.

```
int init_linked_list(ptp_ptr)
    register PTP_TYPE       *ptp_ptr;
                           /* initializes table for new linked list */
{
  register LIST_TYPE    *lst_ptr;

  lst_ptr = ptp_ptr->statics->set_id + linda_lst_tab;
    /* obtain pointer to start of table */
  _spinlock(&(lst_ptr->table[0].lock)); /* lock first index element */
  lst_ptr->table[0].iIndex = 0;         /* first index is 0 */
  lst_ptr->table[0].iNext = -1;         /* initially no next index */
  lst_ptr->table[0].iNextAbs = -1;
  lst_ptr->iListHead = 0;               /* head is first */
  _spinunlock(&(lst_ptr->table[0].lock)); /* unlock first element */
}

int insert_link(ptp_ptr)
    register PTP_TYPE       *ptp_ptr;
                            /* insert a new node index into list */
{
  register LIST_TYPE    *lst_ptr;
  int                   iIndex, iAbs, iTemp, iTempNext;

  lst_ptr = ptp_ptr->statics->set_id + linda_lst_tab;
    /* obtain pointer to start of table */
  iIndex = find_list_elem (lst_ptr, ptp_ptr->field_vars[1].actual.il);
```

```
    /* find the absolute index of the element after which this one
       will be inserted -- note how the reference index is passed
       through the ptp structure */
  iAbs = find_next_space (lst_ptr, ptp_ptr->field_vars[2].actual.il);
    /* Find a storage place for the new node index */
  _spinlock(&(lst_ptr->table[iAbs].lock));   /* lock new node */
  _spinlock(&(lst_ptr->table[iIndex].lock)); /* lock node after which
                                                new one is placed */
  lst_ptr->table[iAbs].iIndex = ptp_ptr->field_vars[2].actual.il;
    /* copy the current index value to new index node */
  lst_ptr->table[iAbs].iNext = lst_ptr->table[iIndex].iNext;
    /* make next field of new index node point to next index node */
  lst_ptr->table[iAbs].iNextAbs = lst_ptr->table[iIndex].iNextAbs;
    /* copy the absolute index value of the next index node too */
  lst_ptr->table[iIndex].iNext = ptp_ptr->field_vars[2].actual.il;
    /* make previous index node point to current node */
  lst_ptr->table[iIndex].iNextAbs = iAbs;
    /* update its absolute address of the next index node too */
  if (iIndex == lst_ptr->iListHead) lst_ptr->iListHead = iAbs;
    /* update head pointer if it needs to be updated */
  _spinunlock(&(lst_ptr->table[iAbs].lock));  /* unlock new indx node */
  _spinunlock(&(lst_ptr->table[iIndex].lock));/* unlock prev indx node */
}

int remove_link(ptp_ptr)
    register PTP_TYPE       *ptp_ptr;
                            /* remove an index node from list */
{
  register LIST_TYPE *lst_ptr;
  int iIndex, iTemp;

  lst_ptr = ptp_ptr->statics->set_id + linda_lst_tab;
    /* obtain pointer to start of table */

  iIndex = find_list_elem (lst_ptr, ptp_ptr->field_vars[1].actual.il);
    /* find the absolute index of the element which is to
       be deleted -- note how the reference index is passed
       through the ptp structure */
  _spinlock(&(lst_ptr->table[iIndex].lock));   /* lock the index node */
  lst_ptr->table[iIndex].iIndex = -1;          /* delete it/free space */
  iTemp = find_list_prev (lst_ptr, ptp_ptr->field_vars[1].actual.il);
    /* find absolute index of previous index node */
```

```
  if (iTemp == -1) {
    lst_ptr->iListHead = lst_ptr->table[iIndex].iNextAbs;
        /* if the head is being deleted, update the head pointer */
  }
  else {
    _spinlock(&(lst_ptr->table[iTemp].lock));  /* lock prev indx node */
    lst_ptr->table[iTemp].iNext = lst_ptr->table[iIndex].iNext;
        /* make previous index node point to next index node */
    lst_ptr->table[iTemp].iNextAbs = lst_ptr->table[iIndex].iNextAbs;
        /* same for absolute address */
    _spinunlock(&(lst_ptr->table[iTemp].lock));
        /* unlock previous index node */
    _spinunlock(&(lst_ptr->table[iIndex].lock));
        /* unlock deleted index node */
  }
}
```

The routines above are fairly straightforward to follow. Their most important characteristic is the use of the ptp structure to reference variables. The ptp structure is constructed with data from the dummy **in** operations and this is how the routines access the variables that are packed into that dummy operation.

The following example demonstrates the manner in which the compiler is "fooled". There are three linked list operations in this code segment. The first one is a list initialization. The second one is an insertion operation in a loop and the third is a removal operation in a loop. The abstractions are italicized for clarity.

```
int do_list()
{
  /*** Local variable declarations begin here ***/
  int temp_index;
  int list_index;
  int iMem;
  int i;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  iMem = 20;
  list_index =0;

  initialize-list (index_list);   /* List initialization abstraction */
  out ("index_list", list_index, iMem);   /* actual node put out */

  for (i = 0; i < 10; i++) {  /* loop to insert new nodes */
```

```
    list_index = i;
    temp_index = get_new_index_list_id ();  /* get new unique id */

    insert-link (i,index_list);   /* insert index node after index node i */
    list_index = temp_index;
    out ("index_list", list_index, iMem);   /* actual node put out */

  }

  for (i = 2; i < 3; i++) {   /* loop to remove nodes */
    list_index = i;

    remove-link (i,index_list);   /* removes index node i */
    in ("index_list", list_index, ?iMem);   /* actual node removed */

  }
}
```

When the linked list abstractions are expanded, the abstractions are replaced with the Linda operations that appear below. Note that a new variable, next_index has been generated to implement the abstractions.

```
int do_list()

{
  /*** Local variable declarations begin here ***/
  int next_index;
  int temp_index;
  int list_index;
  int iMem;
  int i;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  iMem = 20;
  list_index = 0;
  next_index = -1;
  list_head_lst = list_index;   /* this is static variable that points
                                   to the head of the list */
  out ("lst_index_list", list_index, next_index);   /* index node */
\verb%  %{\bf   out ("lst", list_index, iMem);%   /* actual node being put out */

  for (i = 0; i < 10; i++) {
    list_index = i;
```

```
    temp_index = get_new_lst_id ();          /* get new unique id */
    in ("lst_index_list", list_index, ?next_index);
         /* see what the next node index after this one is -- where
            'this' refers to the node after which the new one
            is inserted */
    list_index = temp_index;
    out ("lst_index_list", list_index, next_index);
         /* make new node index point to old next one */
    next_index = temp_index;
    list_index = i;
    out ("lst_index_list", list_index, next_index);
         /* make old index point to new one as next one */
    list_index = temp_index;
    iMem += 5;
\verb%  %{\bf     out ("lst", list_index, iMem);%  /* actual node being put out */

  }

  for (i = 2; i < 3; i++) {
    list_index = i;
    if (list_index == list_head_lst) {
         /* special case if node being removed is the head of list */
      in ("lst_index_list", list_index, ?next_index);
      list_head_lst = next_index;
    }
    else {
      in ("lst_index_list", list_index, ?next_index);
         /* remove the index node first */
      temp_index = next_index;
      next_index = list_index;
      in ("lst_index_list", ?list_index, next_index);
         /* remove previous node index so it can be updated */
      next_index = temp_index;
      out ("lst_index_list", list_index, next_index);
         /* make previous node index point to next one */
      list_index = i;
    }
\verb%  %{\bf     in ("lst", list_index, ?iMem);%
         /* remove the actual node */
  }
}
```

When the program is saved, dummy operations are generated so that the necessary optimization information can be passed to the compiler. The first abstraction is replaced with a dummy **in** operation which contains two variables, *list_index* and *temp_index*. The second linked list operation, the node insertion, is also replaced by a dummy **in** operation. In this case *list_index* holds the value of the old index, i.e. the index after which the new one will be inserted. The variable *temp_index* holds the value of the new index. Finally, the last linked list operation, the removal of a node, is replaced by a dummy in operation. The dummy operations are not indented and hence stand out in the code.

```
int do_list()
{
  /*** Local variable declarations begin here ***/
  int next_index;
  int temp_index;
  int list_index;
  int iMem;
  int i;
  /*** Local variable declarations end here ***/

  /*** Body of code for function begins here ***/
  iMem = 20;
  list_index = 0;
in ("lst_index_list", list_index, temp_index);
         /* first dummy operation */
  out ("lst", list_index, iMem);

  for (i = 0; i < 10; i++) {
    list_index = i;
    temp_index = get_new_lst_id ();
in ("lst_index_list", list_index, temp_index);
         /* second dummy operation */
    list_index = temp_index;
    iMem += 5;
    out ("lst", list_index, iMem);
  }


  for (i = 2; i < 3; i++) {
    list_index = i;
in ("lst_index_list", list_index, temp_index);
         /* third dummy operation */
```

```
    in ("lst", list_index, ?iMem);
  }
}
```

4.3 Program Visualization

Sequential debuggers allow users to examine a program as it executes. The user can get a good overview by examining where execution of the program stands, what values are in the registers and what is currently in memory. For asynchronous parallel programs, the overview is not as clear. Potentially, each of hundreds of processes could be in their own stages of computation, each with its own memory and registers. Trying to follow the execution of each of the parallel processes is virtually impossible. Since debugging is very important to program development, it is necessary to develop a system that can help debug parallel programs easily. In particular, we need a system that can present the status of a parallel program in a comprehensible manner to users.

The Tuplescope visualizer [BC90] is a graphical monitoring tool that was designed as a possible answer to that question. It presents a dynamic image of an executing Linda program. Tuples are shown on the screen and their movement to and from tuple space is displayed as the program executes. Users can adjust speed of execution and observe which processes manipulate different tuples. A common error in parallel programming, namely deadlock, can be detected quite easily by observing tuple movement. Clicking on a tuple displays its contents, clicking on a process icon shows the process's last tuple operation. A user can also "step" through a program, not in the sequential sense of the word, but "step" through by stepping to the next tuple operation in the parallel program. A different click calls up a "traditional" sequential debugger.

While Tuplescope has helped Linda programmers debug their code, information from the LPB can enable Tuplescope to provide more useful information to programmers. Tuplescope was modified to use the same information from the LPB that the LPB-optimized compiler also uses. A discussion of the LPB-based Tuplescope features follows.

The LPB passes enough information to Tuplescope to allow a better organization of the display. In particular, Tuplescope presents distributed data structures in formats of their own. Consider a shared counter, for example. With conventional Tuplescope, a counter looks like every other tuple category: it has its own partition and every time the counter is is put into tuple space, a bubble appears in the partition window, indicating the presence of a tuple. When a counter is updated, the bubble first disappears when the process removes the tuple to be updated, and then reappears when the process puts the tuple back into tuple space. Conceptually, however, the user thinks of a counter as an evolving value which is always in tuple space until permanently removed. By using information from the LPB, Tuplescope can bring its display conceptually closer to the user's level.

The LPB tells Tuplescope which tuple partitions are reserved for counters and which operations relate to counter updates, and Tuplescope creates special windows for counters. A counter has its own window with a displayed value that changes as the counter is updated, instead of disappearing and reappearing whenever it is updated. Counter windows get a particular background color to distinguish them from other shared variables. The value display area is highlighted in red when the counter is uninitialized; it turns green when the counter has a value.

The situation for shared variables is very similar. Instead of having bubbles disappear and reappear, shared variables get their own representation. Each shared variable gets a window which shows the actual contents of the variable. This can be done without worrying too much about space constraints on the screen because a shared variable by definition is a single tuple, i.e. its tuple partition will never have more than one tuple in it. Of course, if the contents of the shared variable are large, such as a large array, the window may extend beyond the screen and users will have to move it around to see particular portions of data. As in the case of counters, shared variable windows have their own background colors and the values are shown in green or red depending on whether the variable has been initialized or not.

Queues and linked lists also benefit from enhanced visualization in the modified Tuplescope. Rather than show many identical bubbles of arbitrary order in a window representing the elements of a queue, the modified Tuplescope acts on information from the LPB and gives queues their own representation. Queues appear in windows of their own in a list format. This list shows ordering among the different elements and the contents of each element can also be seen clearly. The head and the tail of the list are at the top and bottom of the list. Since the list could be arbitrarily large, a scroll bar allows the user to scroll the list.

Linked lists are very similar to queues in nature, with the difference that an element can be inserted at any point in the list. When this happens, a new element simply appears in the correct place on the visible list.

Figure 4.2 shows a counter, a shared variable and a linked list as the original Tuplescope would have displayed them. Figure 4.3 shows how the modified Tuplescope displays the same data.

In the long run, the various templates the LPB offers could each have unique representations in Tuplescope. Piranha programs, for example, tend to follow a particular pattern which is related to general master-worker programs. The LPB is aware of what typically constitutes a Piranha program and which process is the "feeder" or "piranha" process. With this information, Tuplescope could display different icons for the different kinds of processes.

The current implementation of Tuplescope was not designed with the LPB in mind. We need an open interface which would allow the LPB to talk directly to Tuplescope. The problem is similar to the problems we faced when passing LPB information to the compiler. If Tuplescope and the Linda compiler were redesigned with more open communications interfaces, that would open numerous doors for further interaction

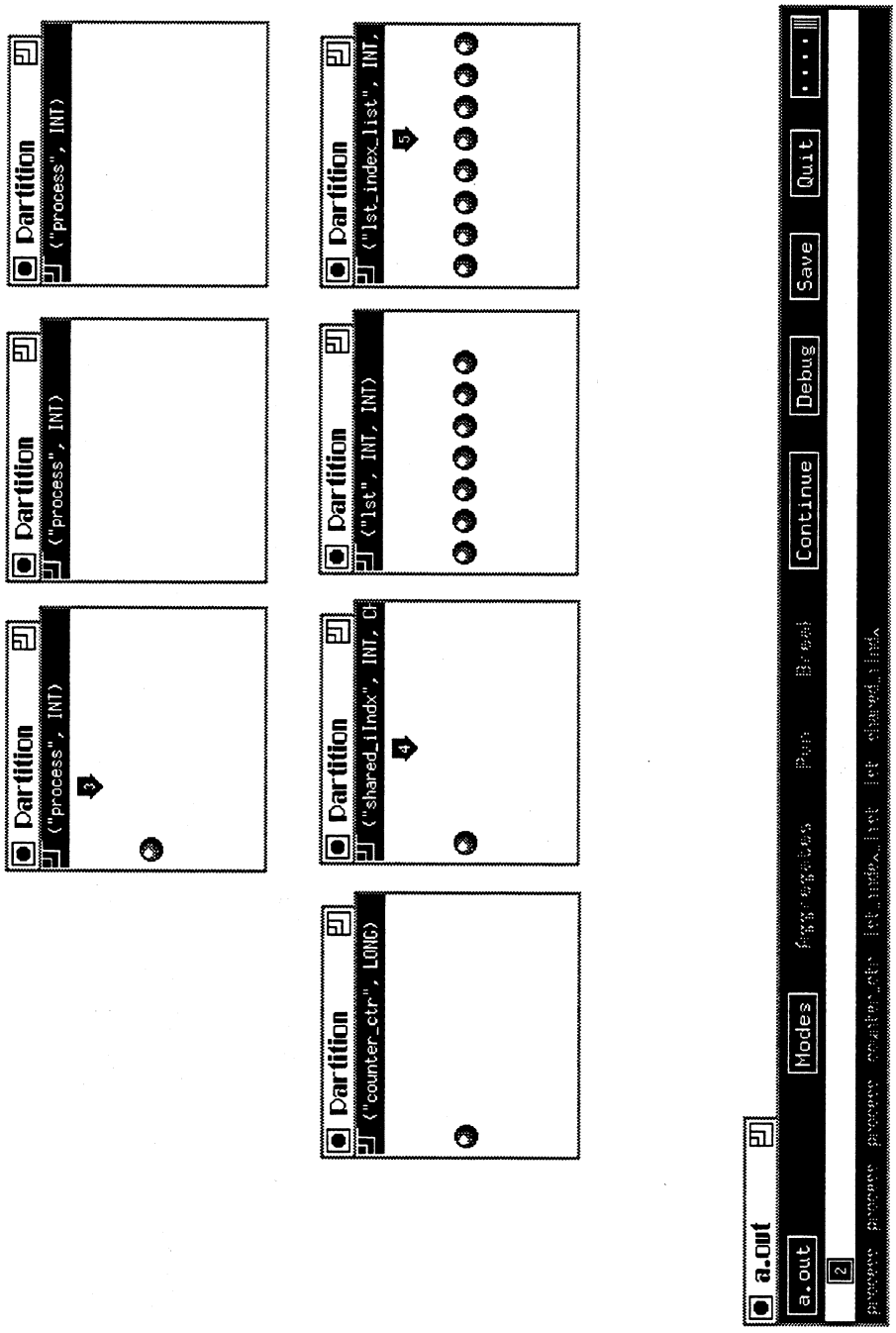

Figure 4.2: Original Tuplescope

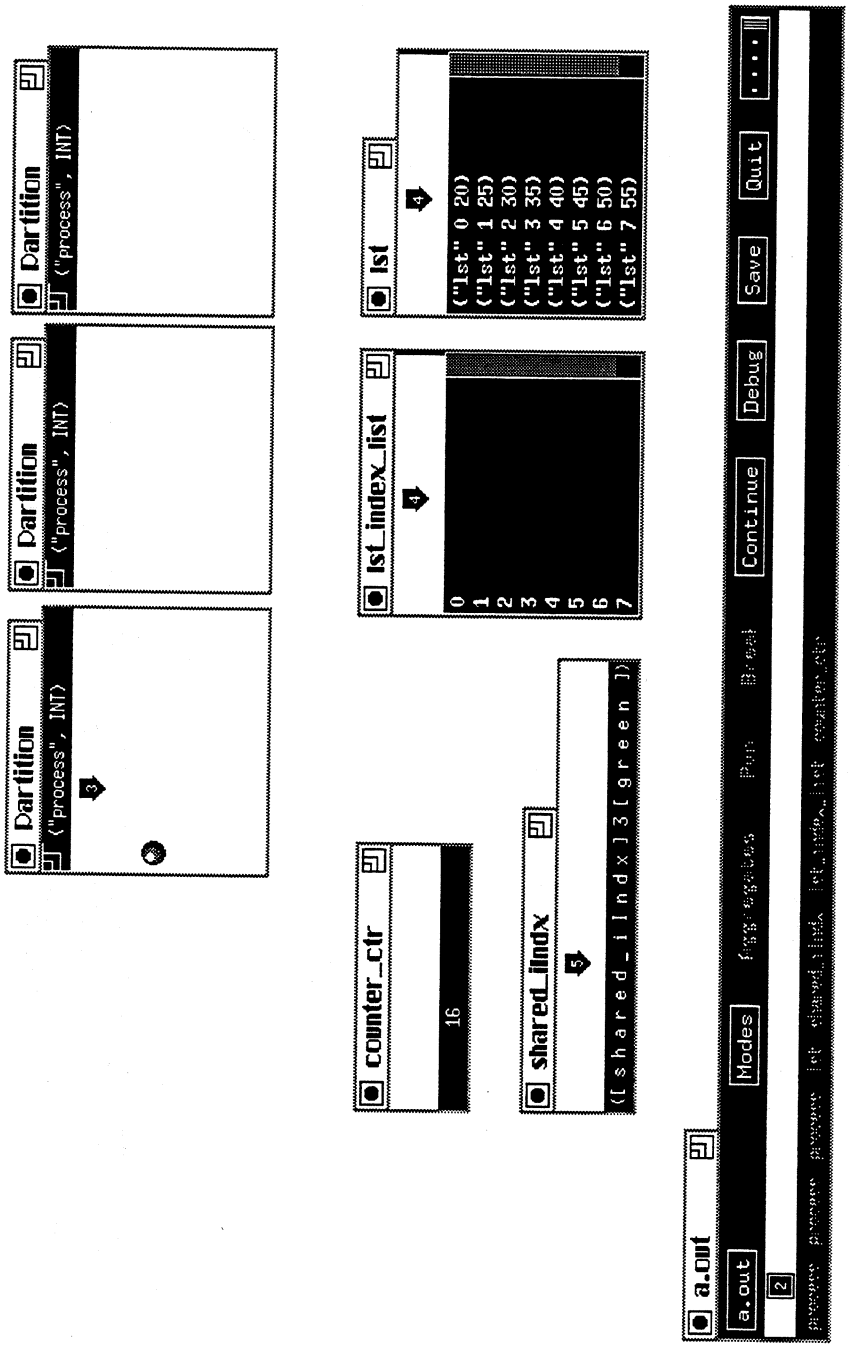

Figure 4.3: Modified Tuplescope

with the LPB. In particular, an interface language which allows the LPB to pass values and graphic object descriptions directly to Tuplescope would be extremely useful. The ultimate goal is to bring the visualization of the program to a level that is conceptually as close as possible to the level of the user's thoughts.

4.4 Interfacing to the Tools

The modifications to the compiler and Tuplescope to use LPB-information clearly benefit the user. The question of how the LPB passes the semantic information to the other tools still needs to be answered. More importantly, how does the LPB accomplish this without sacrificing portability of the resultant code? Figure 4.4 shows a diagram outlining LPB communication with the other tools. In a standard Linda environment (with no LPB), the user has a conventional editor with which he creates standard Linda program files. He then uses the standard compiler (v2.4d) to compile with or without Tuplescope.

The LPB environment has a flag for optimizations. The `optimizations on` flag can be toggled between "on" and "off" modes. Even with the flag in its "off" state, there are certain optimizations which the LPB carries out by default. The counter and shared variable optimizations fall in this class.

The LPB always generates a standard Linda file with a ".cl" extension. This file can be taken by any Linda user and compiled with a standard compiler. The LPB also generates a semantic information file with a ".lpbsem" extension for which the LPB-optimized compiler looks. If such a file is found, the LPB-optimized compiler uses it in conjunction with the standard ".cl" file to optimize the code. Since a standard Linda compiler does not look for the ".lpbsem" file, it simple compiles the standard ".cl" file, and thus, portability is not a problem.

If the optimizations flag is "on", some additional optimizations are enabled. This includes the shared linked list optimization (section 3.4.2). For such optimizations, the standard ".cl" file is still generated, as is the ".lpbsem" semantic information file. In this case, as in the previous cases, the ".cl" file contains the Linda expansions of the linked list abstractions. In addition to the ".cl" file, however, a third file, with a ".opt.cl" extension is created containing the dummy **in** operations designed to fool the compiler (section 3.4.2). To go with this file, a ".opt.lpbsem" file is generated which marks the dummy **in** operations as not being genuine **in** operations. When compiling the ".opt.cl" file, the LPB-optimized compiler searches for the ".opt.lpbsem" file. It reads the list of marked operations and is thus prepared for the dummy **in** operations and knows how to extract the necessary information from them.

Finally, the LPB also maintains a file with a ".lpbtup" extension. This contains the necessary information for the program database (section 3.5).

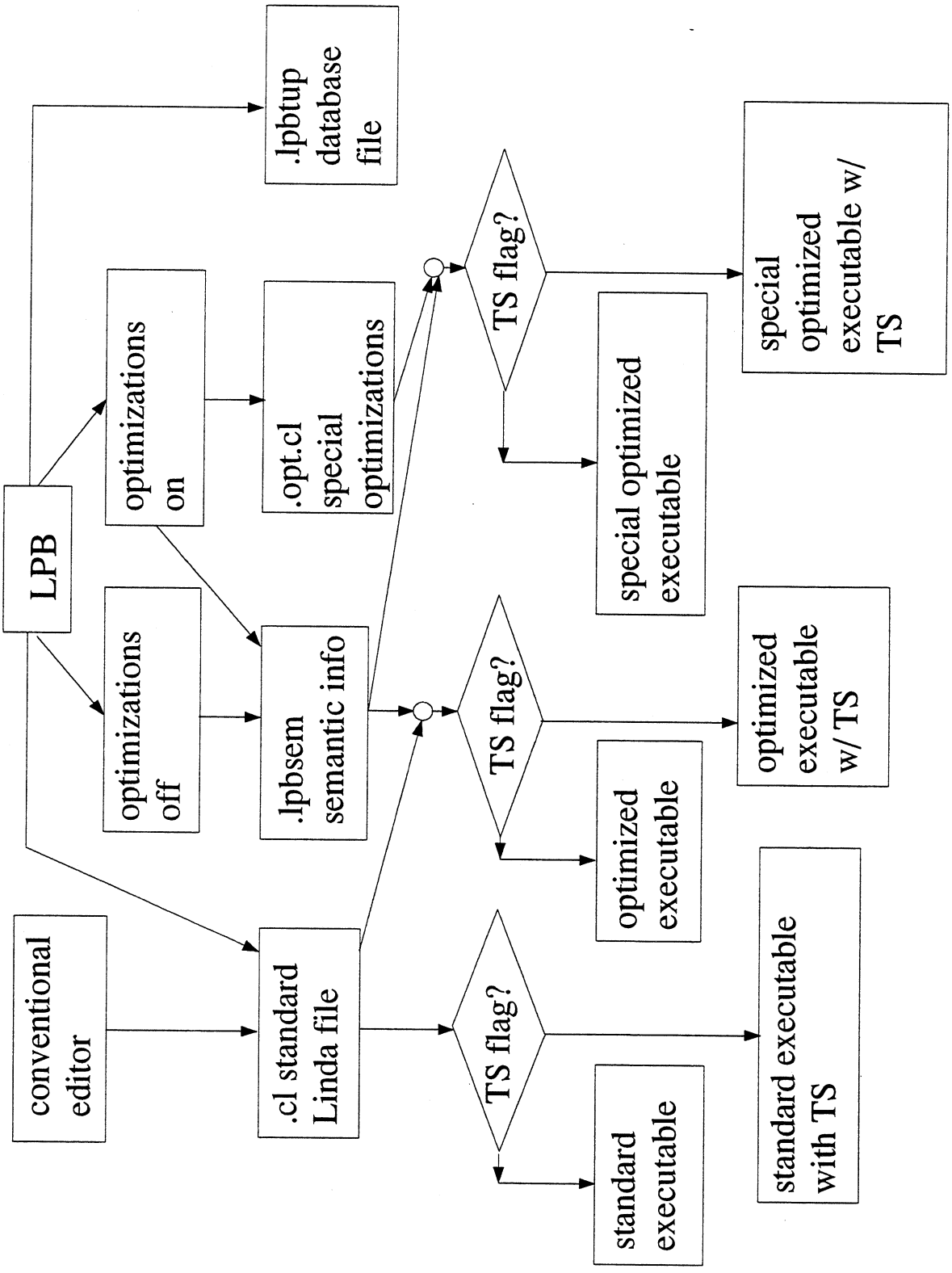

Figure 4.4: Interfacing to Other Tools

Chapter 5

Constructing Templates with the LPB

We have seen how templates provide coordination frameworks to users and guide them through program construction. The LPB comes equipped with some templates for parallel program construction using C-Linda. Programming methodologies can change, however, and thus it may become necessary to develop new templates. In fact, different user communities may employ varying methodologies and have different needs. Consequently, it would be good for users to be able to capture their programming experience for others and to be able to adjust templates with changing needs. How do we provide such extensibility and flexibility in the LPB framework?

To answer this question, the LPB allows users to build their own templates. There are two key issues in developing a template: (1) designing the template, and (2) building the template. Designing a good template involves analyzing usage patterns and determining what program structures will best suit users; building a template involves implementing the structures within the LPB framework.

The LPB presents a graphical interface to the user so that he can build his own templates. The interface appears in the form of a template-building template. With a design in mind, a template developer can use this interface to build his template. An intuitive manner to do this is to construct templates in a similar way to how they are used. Instead of simply clicking on existing buttons however, we first define the buttons. In particular, we define a button, click on it, and then define what it expands into. To understand the nature of the template-building template, a detailed example will be helpful.

In the following sections, we shall investigate the issues involved in designing templates and then incrementally construct a simple template to demonstrate how the template-building tool is used.

5.1 Designing Templates

The goal behind templates is to package programming experience and pass it to other users who can then use the templates to construct similar programs. The first step in designing templates is to determine what constitutes a programming pattern and to determine which patterns are worth encapsulating as templates. Programming patterns can be empirically determined by evaluating current needs. Deciding which patterns are worth encapsulating as templates, however, requires evaluating future needs and trends. To some extent, the latter is a question of economics: a template is only worth building if it will be used and if it reduces the time and effort of constructing and maintaining programs.

Template developers must carefully evaluate the needs of their programming community and make an economical decision on what should be built. A numbercrunching community will want a set of templates that encompasses a wider variety of numerical applications; database programmers will want templates that reflect the data structures and search algorithms that they most often use. To a large extent, empirical evaluation of usage and needs is an interactive process between the template designer and the programmers in the community.

Once a particular pattern emerges as a template candidate, three key issues determine how the template will be designed: (1) program structure, (2) data structures, and (3) graphical interface. Each of these categories requires some careful planning.

The first step in designing a template for a programming pattern is to determine what the structure of the program will look like. Because of the hierarchical nature of templates, a top-down approach is the most natural way to organize program structures. The hierarchical steps that go into constructing the program need to be abstracted out and organized. The number of subroutines in a program, the order in which they have to be defined, the number of processes within the program — the template designer needs to make note of all of these issues. He may decide to have one button for each subroutine (e.g. one for the master subroutine and one for the worker subroutine), or he may decide to have a button yield more buttons, or he may decide upon a completely different scheme. Ultimately, his goal is to establish a clear logical progression that will be easy to follow for users of the template. Each step should be clear in what it is to accomplish and should yield smaller, more refined steps.

Data structures are the other major component of program templates. The template designer has to guide users in choosing the right data structures, in filling them with the appropriate data, and in accessing and updating them at the right times. The designer has to identify the necessary data for a particular programming pattern. There may be a variety of acceptable data structures, in which case the choices have to be presented to the template user. The template designer must then decide how this data is to be entered by the user. Depending on the nature of the data, the user may be required to enter an entire data structure explicitly, or to declare some variables within an established data structure, or perhaps even to initialize data within a well-defined

data structure.

The program structure and data structures aspects of template design are far from being mutually exclusive. In fact, the two are typically closely related. Data structures may get allocated in some routines, initialized in others, and accessed in yet others. Determining when and where these stages happen is very much a program structure issue. It may not make sense to expand some routines until certain data structures have been defined. If there are a choice of data structures, the program structure may change depending on what data structure the user chooses. It is the template designer's duty to establish how the data structures fit into the program structure.

When the program structure and data structures have been established, the template designer selects interfaces for these structures. In designing the interfaces, the designer must keep in mind the ease of entering data, the guidance the system provides, and the ordering of expansions. The LPB provides numerous input interfaces which can easily be called from the template-building template. The following section will demonstrate how this is done.

The ease of entering data is something with which the template designer may need to experiment to arrive at an optimum solution. In some situations, it is best to require users to select a menu option. Other scenarios may require answering a simple yes/no question or providing explicit code. Ultimately, depending on the nature of the information required and the familiarity which users exhibit with particular interfaces, the template-designer has to make a carefully weighted decision.

Guidance during program construction is vitally necessary for users. The template designer should make buttons easy to understand and provide sufficient help windows to keep the users informed at all stages during program construction, especially when input is required. If buttons need to be expanded in specific orders, the appropriate error messages should ensure that the user understands what the ordering is and, more importantly, understand why that is so.

Clearly, there are several issues which are important to template design. None of them are black and white issues with clear-cut right or wrong approaches. Each of the issues requires careful investigation and evaluation by the template designer before a final decision is reached. Ultimately, template building is an evolving process which can only improve with more feedback from users.

5.2 An Example of Building a Template: Constructing a Master-Worker Template

To demonstrate how to use the template-building template, we shall incrementally construct a master-worker template like the one used in Chapter 2.

Suppose the master-worker template were not implemented and we wanted to implement one for future users of the system. As we know, a master-worker program consists of a master routine and a worker routine. The master generates a number of

worker routines, generates a number of task descriptors, waits for the results to these tasks, and then compiles the final result from these task results. The workers each execute the worker routine: look for a task descriptor, secure it, work on the task, generate the result for the task and then look for the next task descriptor.

To construct our master-worker template, we begin by clicking on `Define Template` in our `Global Menu` (Figure 5.1). The LPB asks us for a template name, so we choose to call the template "master". Since we have started defining a template, the usual menus are no longer relevant, and hence we are left with a `Template` menu and the `Buffer Menu`.

The edit window shows the clean slate on which we will construct our template. We define buttons for both the master routine and the worker routine. To do this, we select the `Insert Button` option in the `Template` menu and we are asked for the text of the button label. The first button will represent the master routine, so we specify `Master Routine` as the button text (Figure 5.2) This will cause the LPB to insert a button with yellow text on a green background at the current cursor position in our edit window. We also need to create a button for the worker routine and so we move the cursor down a few lines and repeat the procedure with `Worker Routine` as the text for the new button. We now have two green and yellow buttons in our window. The colors are significant: a green and yellow button indicates that the expansion for the button has not yet been defined, i.e. we have a placeholder with no definition for what it holds. Once we have defined what a button expands into, the colors change to red and yellow. This choice of color isn't arbitrary — red and yellow are the colors that buttons have within ready-to-use templates, i.e. these are the button colors that template users normally see.

When expanding a template, we know that clicking on a button expands that button into code. Since the template-building template is, by definition, also a template, we should expect clicking on one of its buttons to yield code. The expansions have not yet been defined, however, so no code is produced. Instead, we are asked to define the expansion by expressing the contents of the expansion. Hence, to define a button expansion, we click on the button. This causes an input window to appear titled "Button: *button label*", where "*button label*" is the text of the button label. In this example, if we click on the `Master Routine` button, we are presented with an input window titled "Button: Master Routine" (Figure 5.3). Eventually, when the ready-to-use template is expanded by a user, the `Master Routine` button will expand into whatever is entered into the input window. If we type in pure code, this code will replace the button when the button is clicked on during program construction. We may, however, also want to insert tuple commands into the code or to specify input points in the code where user input will be required. These actions are supported by the menus of the template-building template. The system prompts the user for any necessary parameters.

Since every Linda program has a `real_main` function, we decide to make our master routine the `real_main` function. We thus enter this function in the input window for

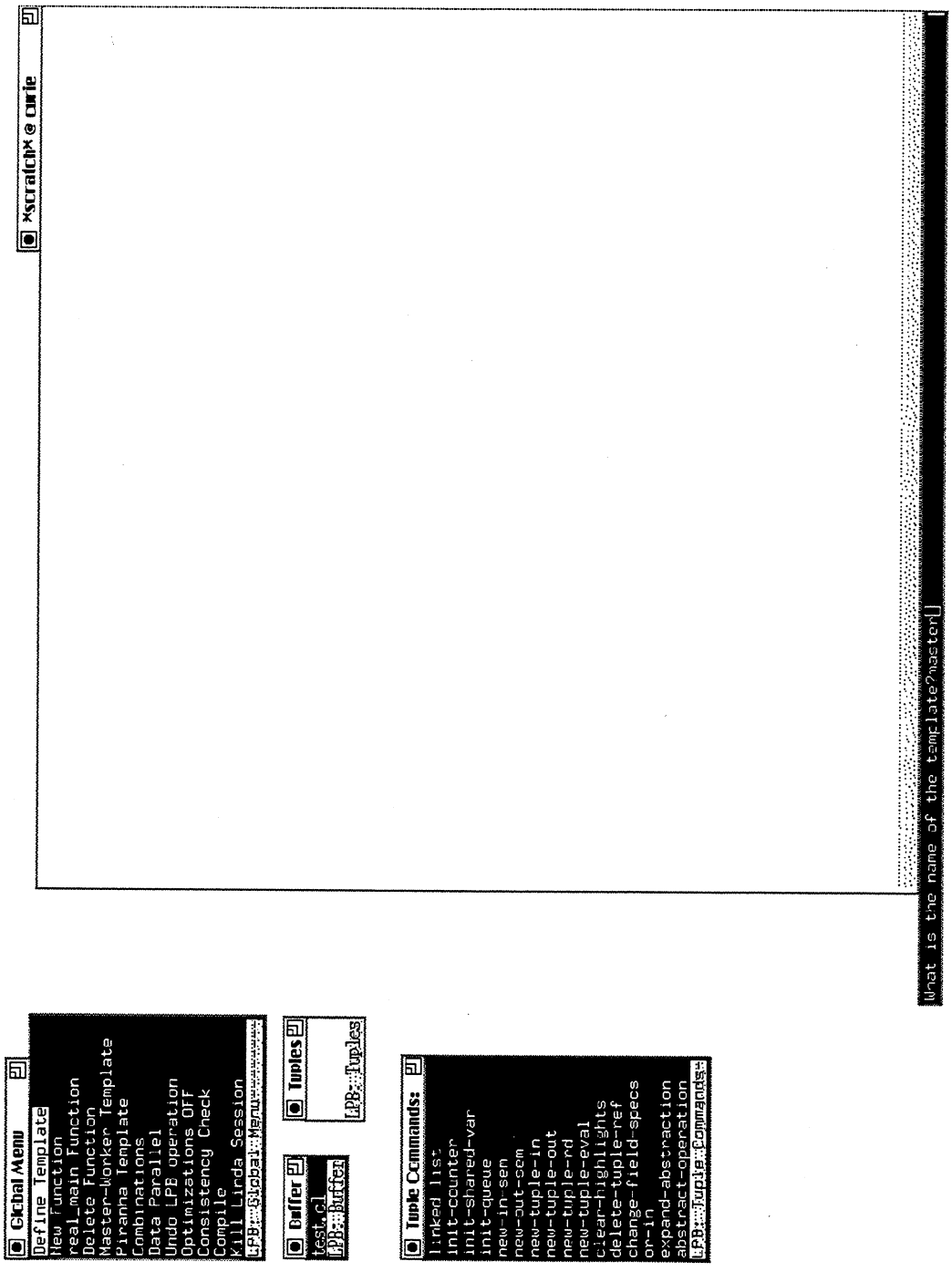

Figure 5.1: Constructing a Master-Worker Template: Starting Out

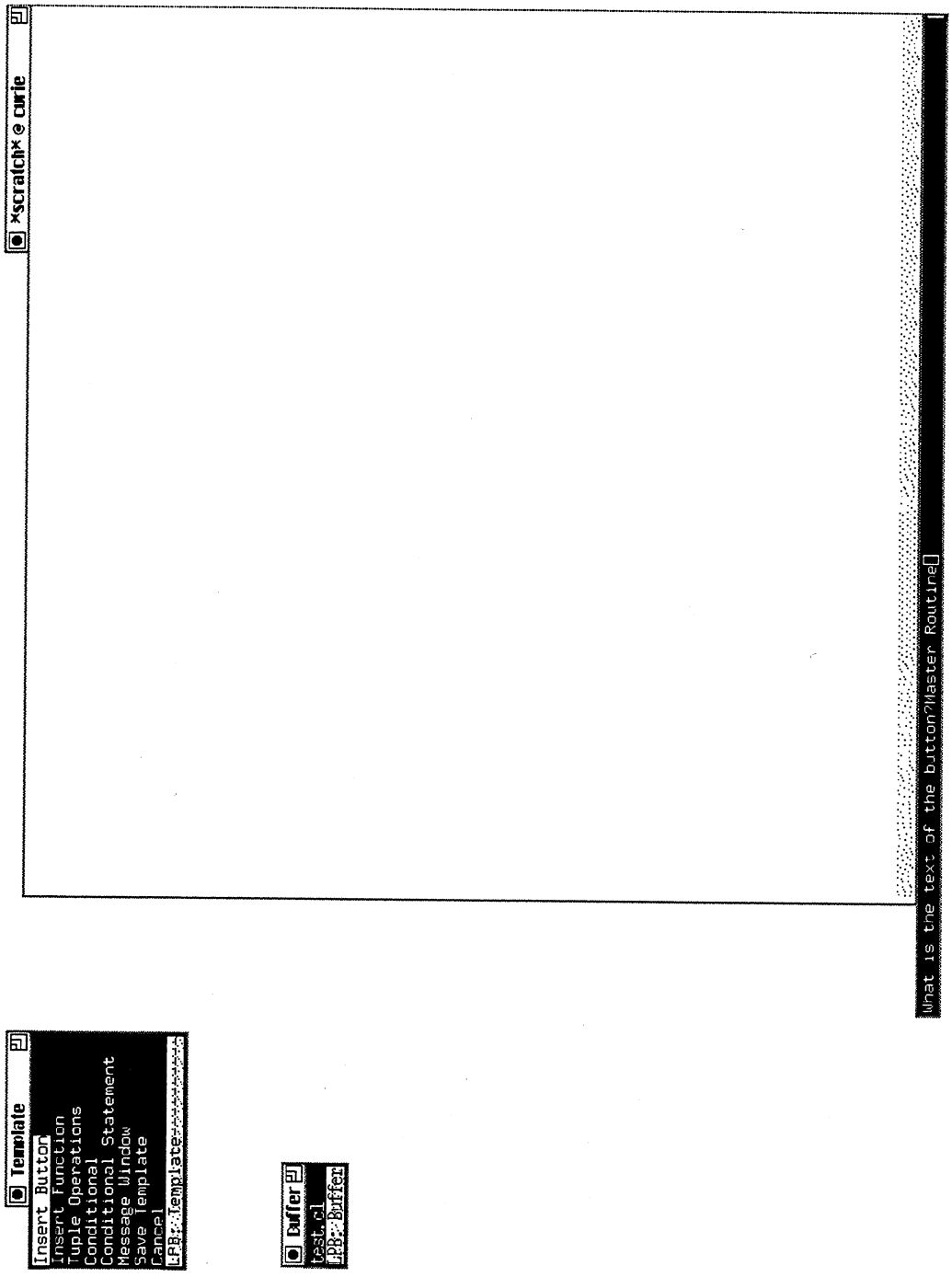

Figure 5.2: Constructing a Master-Worker Template: Inserting a Button

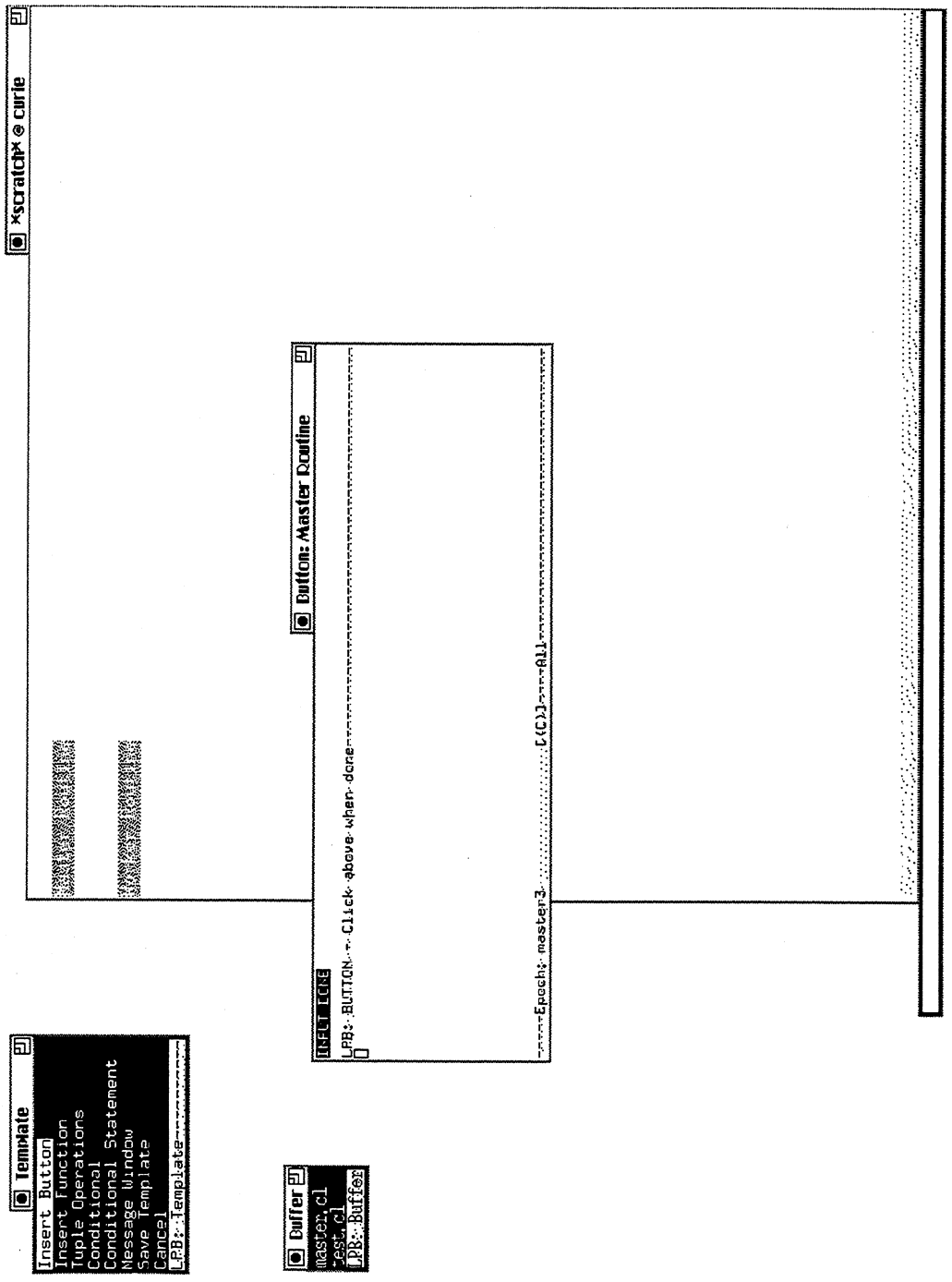

Figure 5.3: Constructing a Master-Worker Template: Defining a Button Expansion

the master routine expansion. To do this, we select the `Insert Function` option in our `Template` menu and type in "real_main" when prompted for the function name. We also type in "int" as the type of the function. The LPB now pops up another input window titled, "int real_main" in which it asks the user to declare the arguments to the `real_main` function. We declare the usual command line arguments (`argc` and `argv`) and click on our `INPUT DONE` button (Figure 5.4).

The above actions cause the LPB to insert a skeleton of the `real_main` function. To fill in the skeleton, we start by typing in the initial code that checks the command line arguments for the number of workers. We are now ready to prepare the loop which generates the actual workers. We specify the loop boundaries and insert the **eval** operations. To do the latter, we click on the `Tuple Operations` option in our `Templates` menu and a popup menu appears. The popup menu lists the various types of tuple operations we can insert (Figure 5.5). We select the `tuple eval` option. The LPB asks for a label of a tuple to **eval** and we type in "worker". The inserted **eval** operation appears as a highlighted placeholder in the input window. The color of the highlight indicates that the text represents a placeholder.

The placeholders denote something that will happen when the template is actually used. In the above case, the placeholder denotes the stage in the code where the appropriate **eval** operation will be inserted into the code when the template is expanded.

With this behind us, we proceed to define two new buttons within the input window. These are essentially buttons within a button expansion — i.e. when the button for the master routine is expanded, it will yield code with these new buttons in it. The new buttons are for generating the task descriptors and for gathering the results. The familiar green and yellow labels appear. To define how these buttons expand, we click on them. We start with the `output task` button (Figure 5.6).

Suppose that in addition to the actual button expansion, we would like a help window to pop up when the user clicks on the `output tasks` button during program construction. We can accomplish this by selecting the `Message Window` option in our `Template` menu. This will cause green delimiters to be inserted into the current input window, marked "Message window:" and "End message window". Any text that appears between these two delimiters will appear in a message window when the user reaches this point during template expansion.

With the help window text specified, we proceed to develop the task descriptor loop. Much as with the **eval** loop, we create the loop skeleton and then insert a tuple **out** operation by selecting the appropriate option. We label this tuple "task" and the highlighted placeholder appears in our window. During template expansion, the user will be asked at this point in the code to declare the variables for the fields of this "task" tuple. Essentially, the placeholder designates a standard **tuple-out** operation on the tuple "task" that will be carried out at its point in the template. If the tuple has already been defined, the user will not be prompted to declare the variables.

Suppose we were to insert another `tuple-out` call on "task" somewhere in the

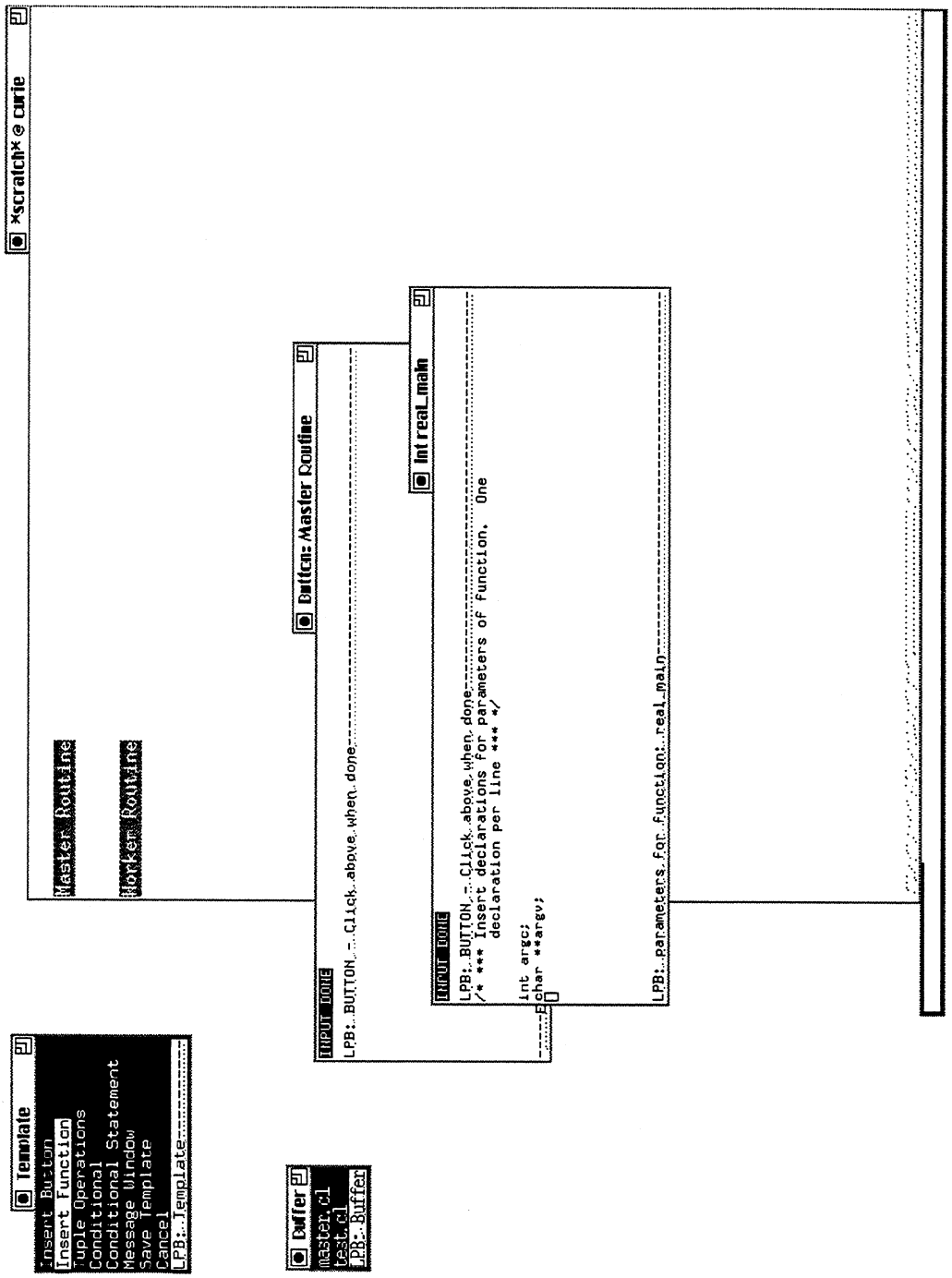

Figure 5.4: Constructing a Master-Worker Template: Declaring Arguments

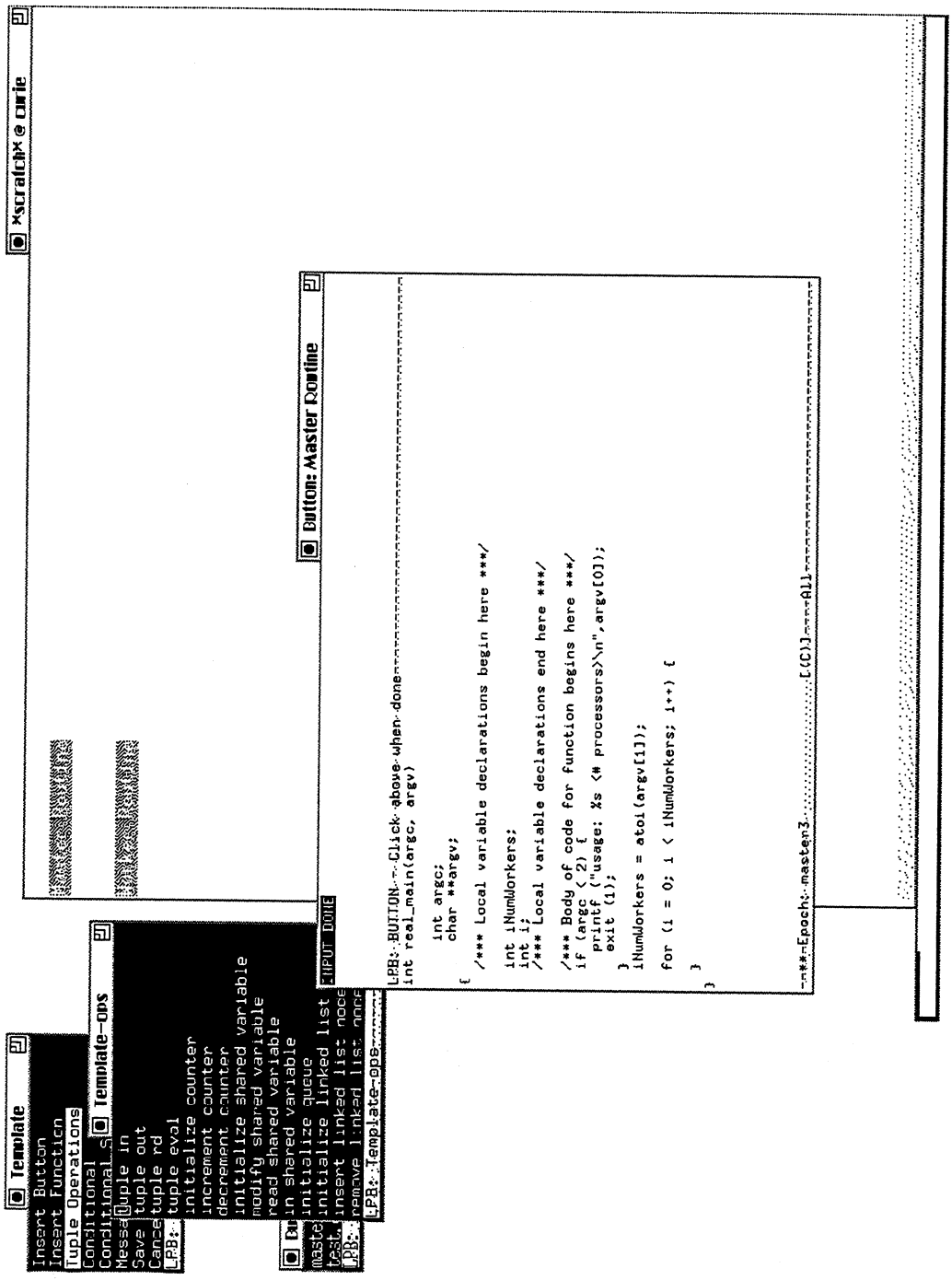

Figure 5.5: Constructing a Master-Worker Template: Inserting Tuple Operations

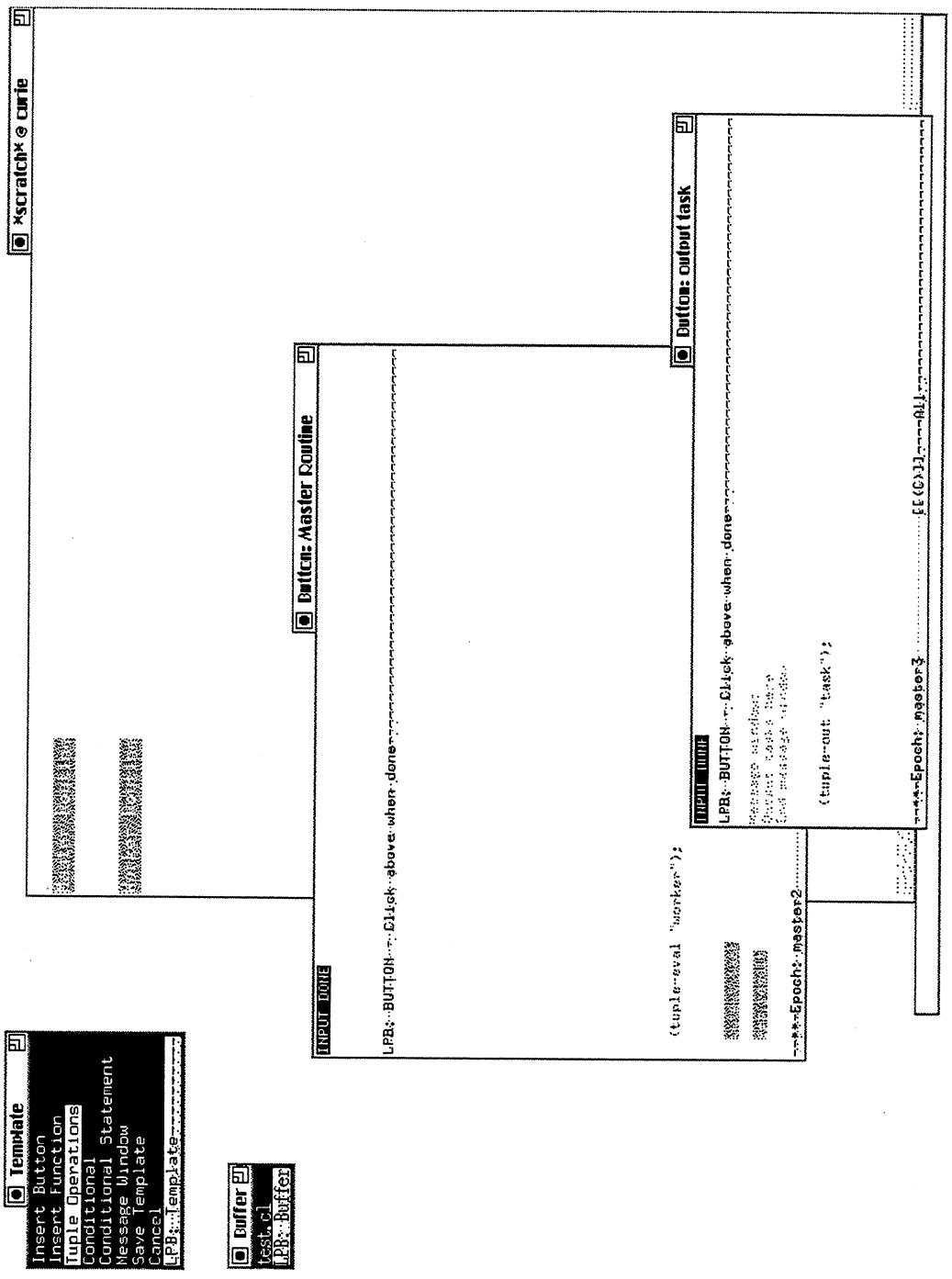

Figure 5.6: Constructing a Master-Worker Template: The Task Loop

template design. The user is not required to redeclare the variables. Instead, before the tuple operation placeholders expand, the LPB checks the database for existing definitions on the tuple label. If the tuple has already been defined, the operation is inserted into the code and the appropriate variable declarations are made.

When we finish defining the `output task` button, we click on the `INPUT DONE` button and our `output task` button turns red and yellow, indicating that it has been defined. We now concentrate on the `get result` button and essentially repeat the same pattern we used during the `output task` button expansion (Figure 5.7).

Eventually, our master routine is completed and we concentrate on constructing the worker routine skeleton. This involves inserting a function, writing some skeleton code, and inserting some buttons, much as in the master routine case (Figure 5.8). Finally, when the worker routine button expansion has been defined completely, we are done.

As we have seen, building templates is simply a matter of invoking a template itself. The approach for building a template is clearly similar in nature to using a template to construct a program. This fact should make it easier for users to capture their programming experience in the form of templates which can be passed to other users. Essentially, the template-building template is a means by which the LPB provides extensibility and flexibility within its framework.

5.3 Problems with the Template-building Template

There are some limitations to the template-building template which can make it difficult to construct complicated templates. The template-building template is a very high-level tool which does not give the builder a great degree of control over the finer details of construction.

Consider the case of inserting a tuple operation. To insert a tuple operation in a template, the template-builder selects the appropriate menu operation, defines a label for the tuple and the LPB inserts a placeholder. When the template is expanded by a user, the user is prompted for variable declarations for the tuple if the tuple has not yet been defined. The problem with this approach is that the template-builder has no direct access to what the user declares during template expansion time.

Suppose a user will declare an array of integers or a single integer while expanding the template. The template-builder wants the template to behave in different ways depending on the declared variables. Unfortunately, the template-building template does not allow the builder to express this kind of condition. He has no access to what the user types in during template expansion. To access this data, the builder has to query the tuple table in the program database. This can only be done through the LPB library routines. While the set of library routines is fairly extensive, they require a certain degree of familiarity before they can be used. Using the routines also requires knowledge of epoch lisp.

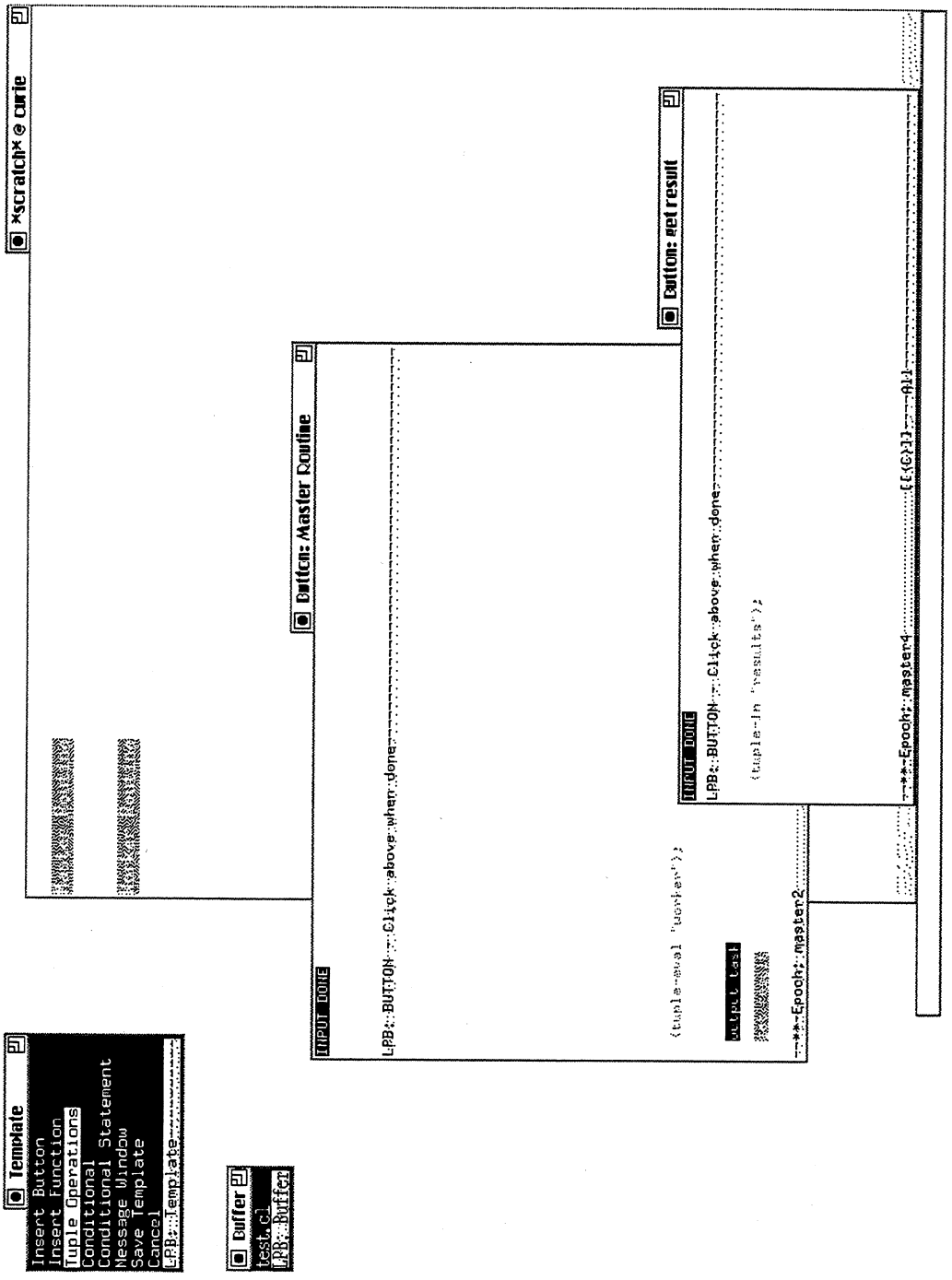

Figure 5.7: Constructing a Master-Worker Template: The Result Loop

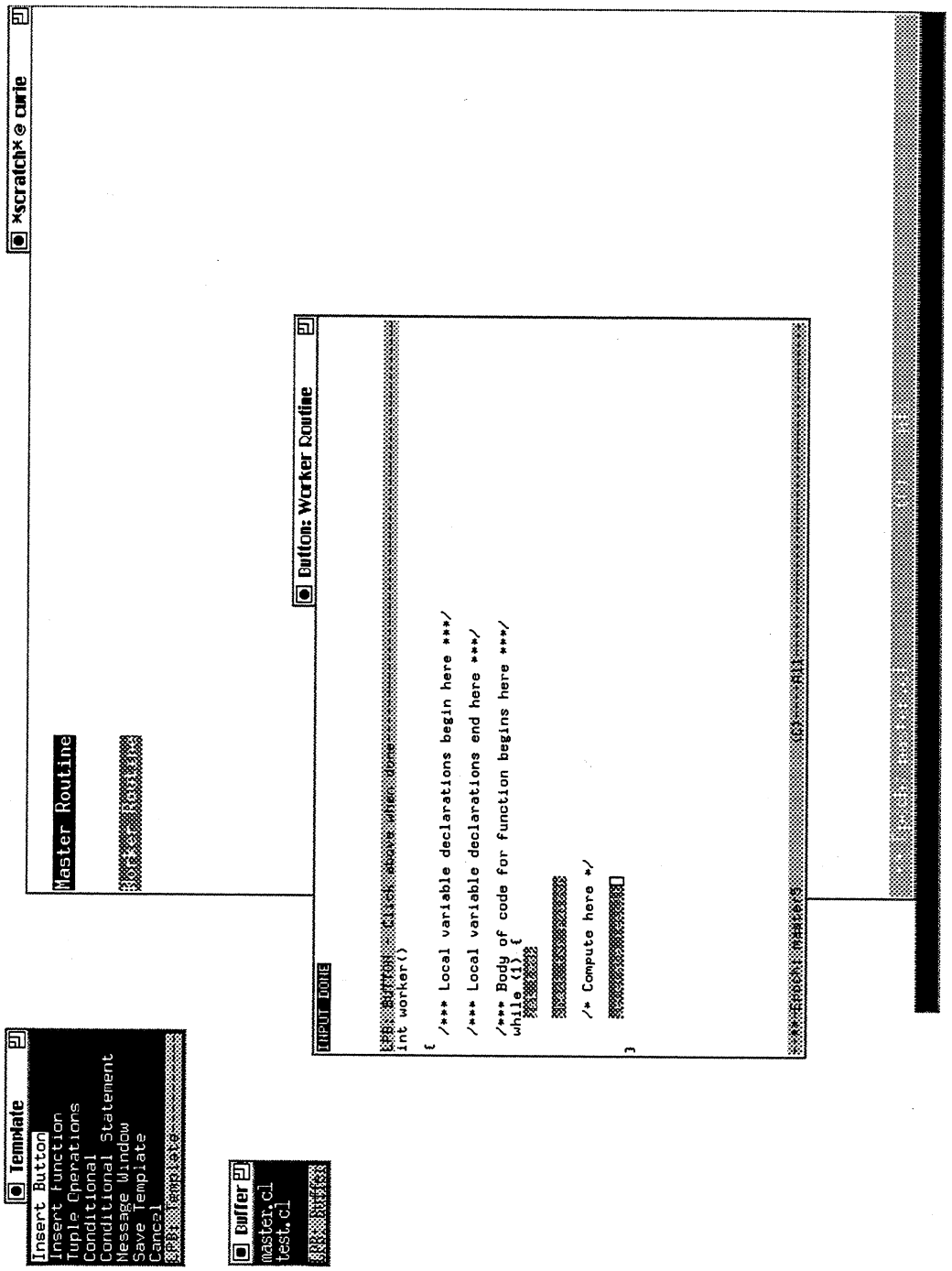

Figure 5.8: Constructing a Master-Worker Template: The Worker Routine

Alternatively, the template-building template could be made more sophisticated. Conceivably, we could add constructs to the template-building template which give the builder abilities to query the program database at any time and construct conditionals based on these queries. If the demand for such constructs grows, this would definitely be a worthwhile development.

Chapter 6

Extending a Base Language with a Program Builder

The methodologies and higher level operations supported by the LPB address an issue that is wider than CASE for parallelism. LPB-like program builders present an attractive alternative to new programming languages.

Chapter 5 touched upon the issue of extensibility and adaptability in the LPB framework. The template-building template allows us to capture programming experience to present as a guidance mechanism to other users. In addition, however, there are other powerful extension features which the LPB provides. Consider the **or-in** example from section 3.4.1. Using the abstraction mechanism of the LPB, we were able to implement a proposed language addition without adding it to the base language itself. This characteristic is important because it allows us to keep the base language relatively static in nature — i.e. we can add constructs on a need-to-use basis without changing the language or imposing these new constructs on those who don't need them. This ability to customize language appearance according to individual needs without changing the language, has significant potential: it represents an alternative to creating new programming languages when programming needs change. Traditionally, when users have asked for special-purpose constructs, a common solution has been to extend languages or develop new languages to meet the needs. The program-builder approach presents an attractive alternative. Users have also asked for frameworks that allow them to capture programming experience for later use by other programmers. As we have seen (section 5), the LPB also addresses this requirement. We present a deeper examination of the issues below.

6.1 Preprocessors as an Alternative to New Programming Languages

The demands of programmers for higher-level tools and special purpose constructs ultimately lead to one question: How do we capture programming experience and provide special-purpose conceptual and practical aids to a programmer in a framework that can adapt with changing methodologies? If we consider this question independently of what we know about the LPB, there are four potential answers. We could (1) rely on the base language, (2) build libraries of subroutines, (3) offer syntactic support, possibly with the addition of (4) semantic support.

6.1.1 Relying on the base language

Relying on the base language is not particularly helpful. The need for additional constructs arises when a base language is not expressive enough or powerful enough for certain kinds of actions to be quickly and easily expressed. When the need for special-purpose constructs arises, the natural tendency is to add these constructs to the base language. In principle, this approach will work, but it presents some serious drawbacks. What is useful or desirable today may no longer be so tomorrow. We may end up making our language arbitrarily large and complex, only to never use the features that were added along the way. Worse, every time new needs develop, constructs will continue to be added to the language and hence to the compiler. Needless to say, this gives the language a continuously evolving nature that would drive both implementors and users alike insane. Languages need to be fairly static in nature for users to familiarize themselves with the languages and develop any sense of expertise therein.

6.1.2 Building libraries of subroutines

Building libraries of subroutines eliminates the problems of (1), but doesn't quite fulfill our needs either. For our purposes it is important that special-purpose constructs look to the programmer as if they were language features: syntax must be clean, and the compiler must be capable in principle of profiting from the programmer's decision to use these particular constructs. While individual subroutines could be optimized, it would be difficult to optimize based on the relationship between different subroutine calls. Constructs such as the **or-in** would be extremely difficult to implement with libraries of subroutines because such constructs make global changes. Further, we intend for our language extensions to incorporate an interactive graphical environment.

6.1.3 Syntactic and semantic support

Syntactic and semantic support can be provided in two different ways. We might build a new language; alternatively we might build some sort of preprocessor, which

recognizes our new constructs and translates them into some base language. The key difference is ultimately one of degree and not kind, but a preprocessor generally targets a higher-level virtual machine than the compiler. Other things being equal, a higher-level virtual machine language is more human-readable and more portable than a lower-level one. A good C++ preprocessor, for example, will target C as a target language. The resultant code is generally more human-readable than the assembly language code which a C++ compiler would generate.

The LPB is a preprocessor that targets C-Linda as a virtual machine; C-Linda is a family of compilers that ultimately produces object code for different machines. The object code in the latter case is linked to a runtime communication library designed for a particular kind of interconnect and communication environment. C-Linda is a "higher-level virtual machine language" than the languages targeted by the C-Linda compilers themselves.

Hence when we have new language features to support, it is better in certain well-defined ways to implement them within the LPB than to design new languages. C-Linda is portable over many asynchronous parallel environments, and the advantages of that are obvious. Those advantages, combined with the fact that people can read C-Linda more easily than lower-level target code, mean that the preprocessor environment is far more *customizable* and *dynamic* than a new language would be. Sites A and B (or even users A and B) can freely customize their preprocessors to support only and exactly the sort of constructs they need. The object code their customized systems emit is portable: hence they are excused, in their customization labors, from low-level hacking; Just as important, no matter how much A's and B's preprocessors diverge, A can still run B's codes and vice versa. And no matter how much they diverge, A can still *read* B's codes more easily than assembler, and vice versa. They share the same relatively human-readable target code. Focusing the same arguments on a single site, the preprocessor can be "dynamic": language features can be added, changed or deleted without requiring low-level re-implementation, and without destroying the compatibility or readability of older-version codes. The base language remains the *lingua franca* in which to exchange code with other users from different sites who may have different customization needs.

These advantages all depend, of course, on (1) the preprocessor's producing codes that are acceptably efficient, and (2) the preprocessor's being user-extensible in fact and not just in principle. Assuming these conditions are met, the preprocessor beats the new language option decisively for our particular needs: we suspect that "customizability" will be important in parallel and distributed programming environments, and we know for sure that "dynamism" is desirable.

The disadvantage of program builders in this context is efficiency. Wiring an operation into the base language allows the compiler to optimize its support. While a program builder may not provide that level of optimization, it can provide the compiler with semantic information that can lead to a different kind of optimization, as we have seen with the LPB. It is important to note that adding customized, user-specified

optimizations to the compiler is different from changing a compiler to accommodate a new language. The user-specified optimizations are optimal and they are local to a specific site; they do not affect anyone else. In combination with the above mentioned advantages, this amounts to a strong offsetting argument against the efficiency disadvantage.

It is important to remember that a preprocessor can never be more powerful than the base language. Anything that a preprocessor can achieve can also be expressed in the base language. The preprocessor offers convenience and can offer fancy constructs such as the **or-in** construct of the LPB, but ultimately it all is transformed to the base language. Nevertheless, a preprocessor can still offer the advantages listed above.

The only remaining question is: what kind of preprocessor should we build?

6.2 Particular Characteristics of the LPB

The LPB is an odd beast in preprocessor-land. While the LPB can be thought of as a preprocessor, it transcends traditional preprocessors (such as cpp or m4). The particular characteristics of the LPB style are: (1) *Input*: the LPB defines an environment—an interactive graphical one—and not merely a language. (2) *Output*: the LPB emits not only target code, but hints or directives to the compiler and visualizer. (3) *Trajectory*: the set of language features supported by the LPB is user-extensible in ways that are different from other systems.

6.2.1 Input

When requiring input for a specialized operation, the LPB will interactively direct a user through a tree of options. In contrast, when using a conventional preprocessor, a user would have to make a subroutine call which is specific enough to represent the entire path down the tree. A conventional preprocessor can offer many such routine calls with lengthy argument lists to create numerous powerful options, but this would be hard to master and this scheme cannot provide *active* support during the construction phase. The LPB provides active support such as guiding users through incremental program templates or menu-driven queue model selections with automated code insertion. There is an incremental methodology in the script of a program builder which a conventional preprocessor cannot duplicate.

6.2.2 Output

At the output stage, the interactive nature of the LPB and the continuous acquisition of semantic information enable it to pass useful information to other tools in the environment. In fact, the LPB could become a "smart" assistant with the addition of an expert database system that is able to identify and call up either (a) templates that are likely to be useful for the type of program under construction, or (b) fragments of

complete, real applications that are related in significant ways to the program under construction, and are thus potentially useful guides. The Linda group at Yale has developed an expert database system with the right capabilities in principle [FG91]; retargeting it to the LPB is research for the near future.

6.2.3 Trajectory

The LPB language features are user-extensible in a number of ways. The template-building template is a mechanism to add new specialized operations and new frameworks for program construction. A user also has the option of implementing proposed language additions (e.g. the **or-in** function) which globally affect key code segments. The user can further specify how the new features will add to the LPB's semantic knowledge and provide directions on how much of this knowledge is passed on to the other tools. Hence, when adding a new customized operation, the user not only specifies how it transforms into the base *lingua franca*, but defines how it could be optimized and how it is to be visualized. This level of extensibility distinguishes the LPB from preprocessors (such as C++ in preprocessor guise [Str86]) that implement complete and self-contained languages. In this context, "extensibility" does not refer to name overloading or inheritance, although the word is often taken to mean that in the object-oriented community. Extensibility refers to extending language features by means of added constructs that a preprocessor implements.

This LPB's extensibility also distinguishes it from conventional macro preprocessors. Macros are in-place expansions — they cannot globally affect code segments. Furthermore, macros are not incremental in nature. An LPB construct can prompt the user for information and guide the user down a tree of options to arrive at some result. For a macro to simulate this behavior, a user would have to provide enough arguments to specify the entire path down the tree. There is clearly a difference in convenience.

Unfortunately, the current interface between the LPB and other tools requires some non-trivial coding by the user to specify the optimizations and visualizations. This problem can be overcome by changing the interface and modifying the other tools in the environment to accommodate this. Chapter 7 discusses the new interface and the necessary redesign of the other tools in the environment.

Chapter 7

Future Work — Some Preleminary Designs

One of the limiting factors of LPB performance is its format of interaction with other tools in the environment. This is not surprising since the other tools preceded the LPB and thus were not designed to interact with it. Given that the LPB carries useful semantic program information, it is desirable to change both the compiler and the visualizer to best use this information. We need an environment where a user can define an abstraction, specify how to transform it into a base language, clearly state how to optimize it, and easily create a visualization scheme for it.

From Chapter 4 we know that there are limitations in both the compiler and the visualizer which required tricks to circumvent. But the implications of this are stronger: the limitations actually restrict the expressivity of optimizations and enhancements.

The compiler is constructed such that it deals with only one tuple operation at a time. For each tuple operation, a corresponding data structure is accessed by the compiler. Unfortunately, this data structure is limited to one tuple operation and only one such structure is accessible to the runtime system for a given operation. There is no way for the runtime system to access two tuples simultaneously. Most of the optimizations that the LPB can suggest, however, involve fusing multiple tuple operations together. Implementing such optimizations may thus require accessing multiple operations simultaneously. This leads to a dilemma: the LPB is ready to provide suggestions on how to optimize code, but the compiler is unable to use the suggestions. To overcome this dilemma, the internals of the compiler need some major changes.

Tuplescope suffers from a similar problem. The current LPB-suggested visualization enhancements had to be hardcoded into the Tuplescope code. This required creating X-windows widgets and writing the code to manipulate them. To investigate feasibility and demonstrate usefulness, this approach may suffice, but for the purposes of the average user, it is clearly lacking since the average user is not familiar with the internals of Tuplescope code, nor should he be. We need a framework for users to quickly and

clearly build visualization schemes for operations and abstractions.

7.1 Redesigning the Compiler

The Linda compiler needs to be redesigned to accommodate semantic information from the LPB. In the process of redesigning, however, we must ensure portability, i.e. the compiler must still compile standard, non-LPB generated programs. This point may seem trivial since all conventional programs fall in a subset of LPB-generated programs, but it is not that simple. An LPB-generated program will generate **in/out** pairs and tell the compiler that the pair is a counter increment, for example. In absence of the information that the pair updates a counter, the compiler has to be able to treat it as a conventional pair of tuple operations. Redesigning the compiler requires clearly understanding the kind of information the LPB will provide. To do this, examination of the current interface is necessary.

Currently, the LPB-optimized compiler looks for a semantic information file that is associated with a particular program file. If the semantic information file is not found, the code is compiled as a standard C-Linda file. If the file is found, the information is read and parsed. The information is expressed in a simple language. In a sense, we can think of this as an interpreted language, i.e. the language specifies what optimization actions to take, and the LPB-optimized compiler interprets these instructions.

The semantic information file contains a list of operations that need to be optimized. Each operation is listed with its line number, a short description of the operation, and a list of related operations. The list of related operations tells the compiler which operations may need to be fused together. As we know, the interface between the analyzer and the runtime kernel is built such that only one tuple data structure is passed at a time. Consequently, fusing the operations together required some tricky coding using dummy operations.

As we have seen in Chapter 4, the LPB-optimized compiler attaches a handler to each operation. To determine how to optimize an operation, the analyzer goes through a list of conditions. Evaluating the conditional yields a handler associated with this particular kind of operation. This handler needs to be written by the developer of the optimizations and added to the compiler. This in itself would not be so bad if that were all that were required, but unfortunately the conditional needs to be changed for every optimization that is added, i.e. for every optimization we add, we have to introduce a new case in the conditional. This is a problem because it requires users to reach deep into the compiler kernel and change code they ideally shouldn't even see, let alone modify.

A possible way around this problem is to have the LPB specify the handler to call directly, letting the analyzer create calls to the handler without first evaluating a conditional. The optimization developer still needs to write the handler functions, but just puts these in a new file and attaches the object code of such a file to the

compiler object code. Instead of going through a conditional, the analyzer directly reads the handler label and introduces a call to this handler. This also means that the LPB needs to provide the arguments for the handler. Of course, in order to write the handler functions, the developer does need a public applications programming interface to the compiler. Given such interface specifications, the developer will be able to write routines that use compiler variables and functions without actually looking at the source code for the compiler.

There is one more problem the new compiler needs to address. Recall the example of the shared linked list (section 3.4.2). One of the problems in that example was the lack of communication of data, specifically, the compiler needed additional data to carry out the optimizations. The default interface is via the operation information. Our solution was to pack values into a dummy tuple operation and fool the compiler into temporarily thinking of it as a genuine operation. The complications of implementing that optimization can be avoided if the LPB directly fed data to the compiler, i.e. if there were a more direct mechanism to interface than via operation information.

The various requirements above lead to a new interface language definition (Figure 7.1 shows the grammar). This interface language is basically a set of instructions to the compiler which then interprets this information to optimize the program. Anything in boldface is a lexical token. Newlines are not implicit "or" operators — those are specified by the "|" character. The proposed language is quite similar to the language that is currently defined. In fact, there are only two differences — but they are important. Unlike the current interface language, the proposed one allows the LPB to specify handlers directly and list variables to pass to these handlers.

As we can see from the grammar of the interface language, the LPB organizes information by tuple categories. An `LPB-partition` consists of delimiters, a tuple category identifier and a list of `operations`. Each operation entry holds the line number and file name, an LPB-assigned label, the optimization handler label, the list of variables the handler will use, and a list of operations which relate to the current one. The latter list is used to fuse operations together or for any other situations which may require relating operations together.

The new interface language is only useful if the compiler can act on the incoming information. The newly designed compiler thus needs to call the handlers specified by the LPB. It also has to be designed in a manner which allows several tuple data structures to be accessed at once. In particular, for tuple operations in the related operations list, it may be necessary to access all the data structures simultaneously. In addition, the LPB may be passing a list of variables. The new compiler will have to find these variables in the symbol table or pass them as arguments and use the appropriate values in the computations within the handler.

With the new interface language and the new compiler in place, expressivity is considerably improved. The user can now define his own optimizations without digging into the compiler source code, and can have the LPB directly tell the compiler which handlers to call. The user can also invent schemes that relate numerous operations

LPB-info	→	LPB-partition LPB-info | ∈
LPB-partition	→	"begin LPB partition" LPB-partition-type operations "partition end"
LPB-partition-type	→	**counter** | **shared-var** | **linked-list** | **queue** | **identifier**
operations	→	file-name line-number LPB-label handler list-of-vars related-ops operations | ∈
list-of-vars	→	var-name list-of-vars | ∈
related-ops	→	file-name line-number op list-of-vars related-ops | ∈
file-name	→	**identifier**
line-number	→	**integer**
var-name	→	**identifier**
LPB-label	→	**LPB-init-counter** | **LPB-increment-counter** | **LPB-decrement-counter** | **LPB-init-shared-var** | **LPB-modify-shared-var** | **LPB-init-linked-list** | **LPB-insert-node** | **LPB- delete-node** | **LPB-init-queue** | **LPB-add-to-tail** | **identifier**
handler	→	**inc-counter** | **dec-counter** | **out-counter** | **in-counter** | **in-shared** |**out-shared** | **modify-shared** |**init-list** |**insert-node** | **delete-node** |**init-queue** |**add-to-tail** | **identifier**
op	→	**NOP** | **INC** | **DEC** | **identifier**

Figure 7.1: The Proposed Interface Language

together — he simply has the LPB mark the related operations and any necessary variables, and then has it write the information into the semantic information file. The semantic information file is then passed to the compiler. Of course, this system still requires the user to develop his own runtime handlers, but he can now do so without investigating the compiler source code.

7.1.1 Possible Further Improvements

While the above scheme is an improvement over the current implementation, further developments may be possible. Consider the manner in which optimizations are specified. For every optimization, the user still needs to write his own handler. Although this is better than having to change the compiler code, it still requires some degree of knowledge of the compiler's runtime system. Some degree of automization in generating optimization specifications would obviously make the system more attractive.

The LPB idea of capturing programming experience could possibly be extended to optimizations. Since we can capture programming experience and offer template menus for program construction, why not do the same for optimizations? This would allow us to categorize common patterns of optimizations and to offer these to users in menu format. Indeed, we could guide users through optimization design. For example, we could create an optimization class for simple fusing of operations for in-place updates to tuples with locks. The shared counter optimizations would fall into this class. The user would click on the appropriate menu option, provide some information on the data, and the handlers would be ready. Ultimately, this level of refinement is necessary if optimizations are to be easily specified by average users.

Alternatively, we could develop a meta-language for users to write their own optimizations in. The current interface between the LPB and the other tools is through an interface language. The idea is fairly simple: The compiler compiles a C-Linda program, but performs certain performance-enhancing actions based on hints that it receives. To decipher these hints, the tool interprets the instructions that are specified in the interface language. There are two distinct levels. A standard program file is generated which any compiler can compile. In addition, a list of hints is expressed in an interface language. Any tool that is equipped with an interpreter to interpret these hints can act upon these hints. Currently, the actions are directly specified by the user in a language that is distinct from the interface language. If, however, the interface language were extended such that the actions themselves could be expressed with the interface language, things may look simpler for users. In that case, all the communication as well as the optimization handlers would be expressed in this language. Even if we have a menu-driven interface to develop optimizations, the generated code could still be in this interface language. Thus, if we develop a comprehensive interface language, we have a solid basis on which to develop the environment.

7.2 Redesigning Tuplescope

The other tool that needs to be modified to interface better with the LPB is Tuplescope. The usefulness of designing visualizations based on LPB-supplied information has been demonstrated in Chapter 4. Unfortunately, the current state of the system requires users to develop such visualizations by changing the Tuplescope code. Implementing visualizations for shared linked lists and queues involved writing non-trivial code and creating the appropriate X-widgets and subroutines. To be useful, the ability to use information from the LPB to design new visualizations should not be limited to Tuplescope gurus. A user-friendly interface for user-designed visualizations is necessary.

The proposed solution is to add a visualization design tool to Tuplescope. Users will develop callback routines which are called by the runtime handlers. The Linda runtime system will call upon a handler to execute an operation. This handler will first check to see whether the Tuplescope flag is on. If the flag is on, a special Tuplescope callback routine is called first. The new user-friendly interface ultimately has to generate these callback routines.

The visualization tool is partially inspired by GUI builders such as the Microsoft Windows Software Development Kit [1]. The idea is to provide a graphical user interface that helps users design the windows and icons for visualization, and supports the actual code generation for the handler routines. One possible strategy would be to take an existing development kit such as the Builder Xcessory [Sol92] and build our tool on top of it. The main problem with this strategy is that these tools are not usually built to be integrated with other tools. Furthermore, the needs of our tool are very specific and closely tied to the LPB itself. Using a more sophisticated tool is overkill because it provides many features that distract from the actual goal. Figures 7.2 and 7.3 outline our proposed scheme.

A menu of tuple category labels is the heart of the scheme. This menu is very similar to the tuple category menu we are familiar with from the LPB. In addition to the category labels, however, it also has a `New` option which allows users to define new categories of visualization. Each category has its own visualization format. Click on a category label and the corresponding visualization window appears.

The category visualization window is titled, "Tuple Partition Window". The user designs his desired visualization format in this window. Once he has sized it to liking, he divides it into display zones. Each zone is a subwindow of the partition window. These subwindows have different formats: they could be process windows, text windows, graphics windows, or list windows. A list window, for example, holds an ordered set of text lines and has a scroll bar to scroll up or down this list. Graphics windows allow icons to be displayed at different points — Tuplescope windows displaying bubbles (see section 4.3) are such windows. To insert a particular subwindow into the partition

[1] Microsoft is a registered trademark of Microsoft Corporation

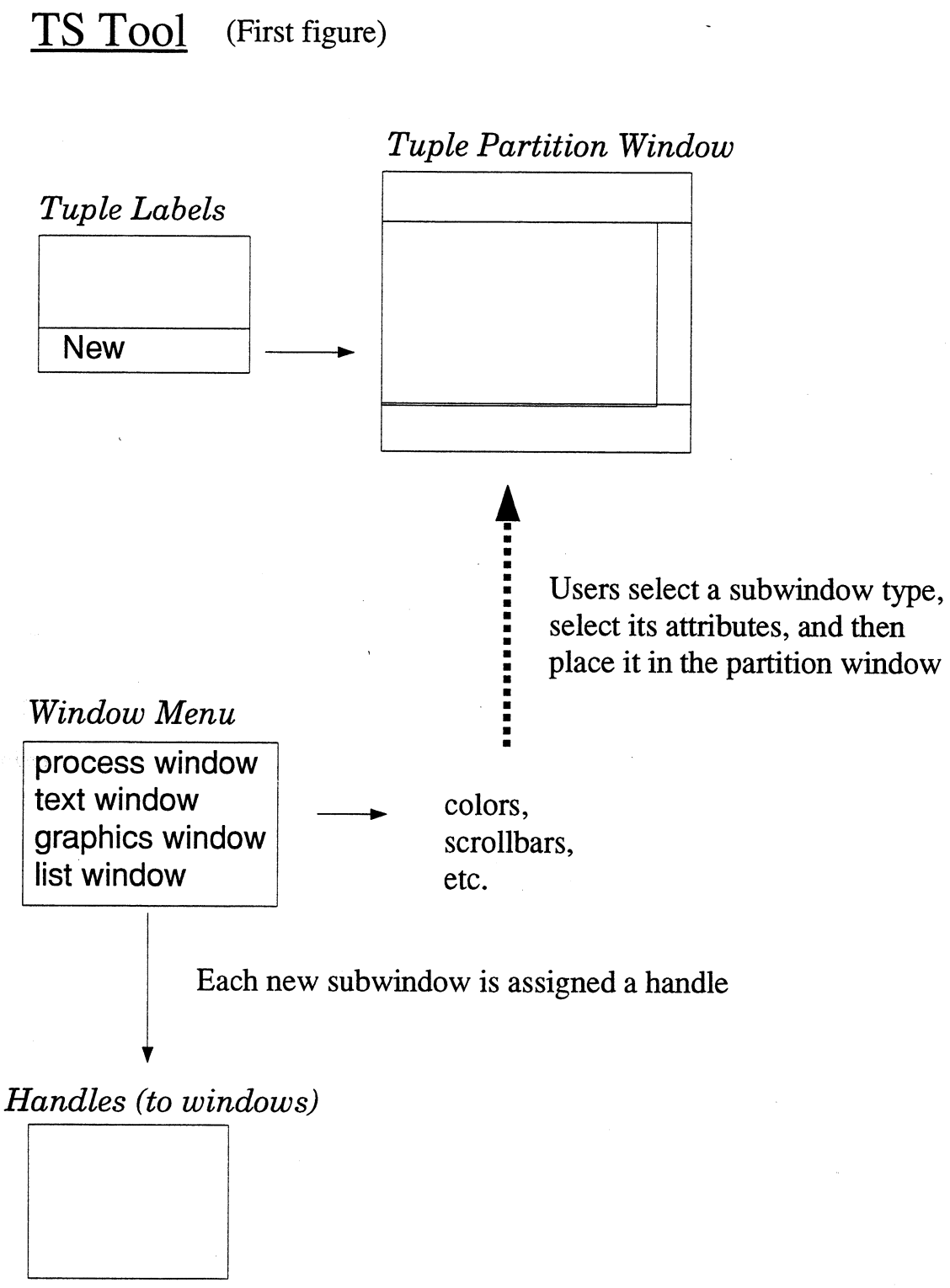

Figure 7.2: Proposed Tuplescope Visualization Design Tool (part 1)

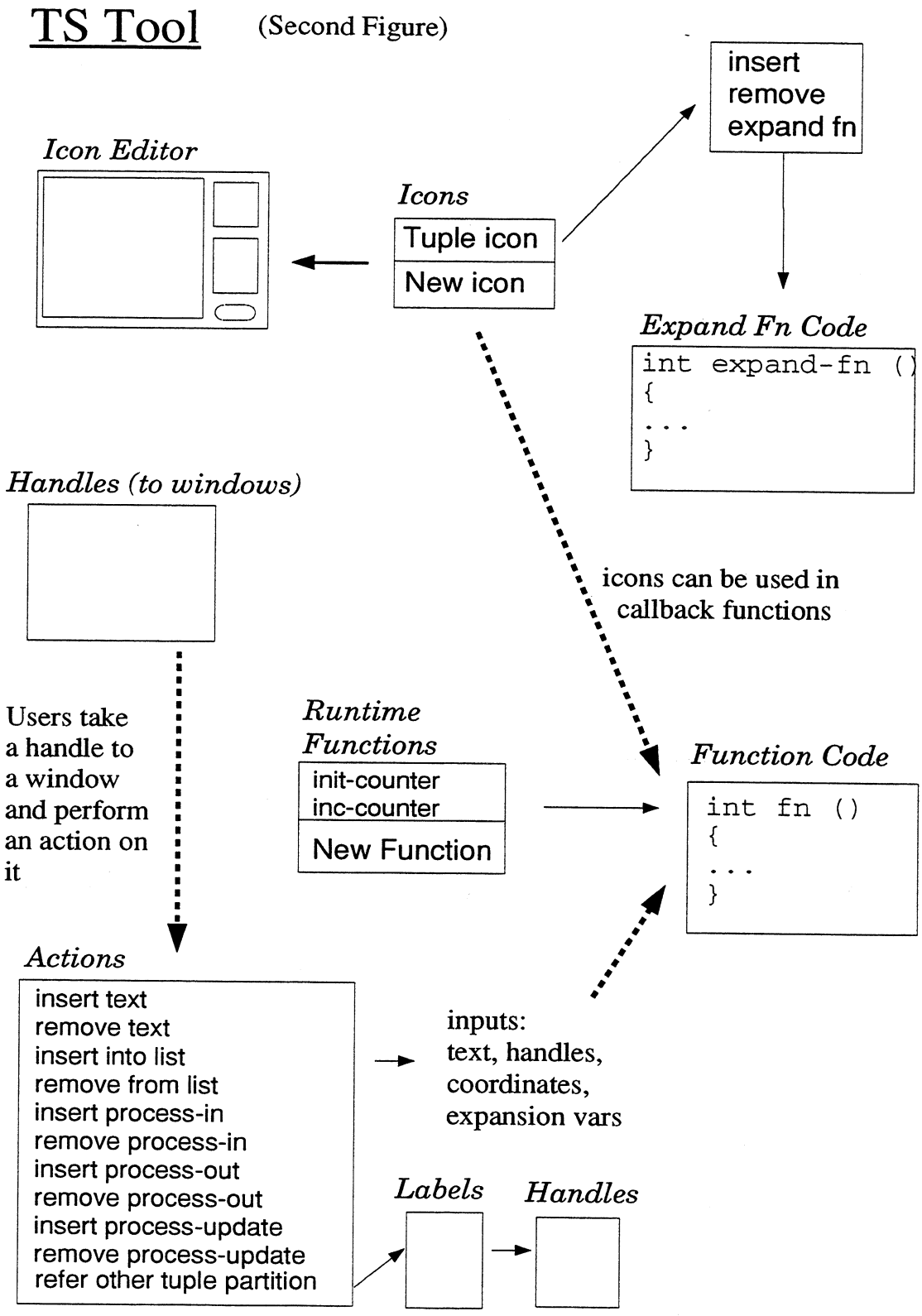

Figure 7.3: Proposed Tuplescope Visualization Design Tool (Part 2)

window, the user selects one of the subwindow types in the `Window Menu`. He will be prompted for some initialization data such as color, after which he has to size and place the resulting window within the partition window.

Every time a subwindow is created, a handle to this subwindow becomes available. This handle can be used in future to refer to the subwindow. Every subwindow handle is inserted into the `Handles` menu. Each time a different tuple category is selected, the partition window changes, and consequently, so do the handles into the subwindows of that partition window.

The `Runtime Functions` menu lists the handler functions which are called directly from the compiler. These functions direct display behavior. For every operation on a particular tuple category, one should define a corresponding routine for Tuplescope if one wishes that operation to affect the display. Clicking on a function name pops open an editable window for the function code. Although the user writes the code, he is not on his own — the tool will provide support as the following paragraphs will show. It is important to remember, however, that the correctness of the visualization is the responsibility of the implementor.

The goal in writing these Tuplescope functions is to define what happens to the various partition windows and their subwindows. We have direct access to pointers to the partition window and also can refer to all subwindow handles. For a particular function, we need to carry out certain actions within these subwindows. We may wish to insert or remove text, insert into or remove from a list, or insert or remove various icons. The permissible actions appear in an `Actions` menu. Hence, to specify an action, the user acquires a handle to the relevant subwindow from the `Handles` menu and then selects the action from the `Actions` menu. Depending on the action chosen, different inputs may be required from the user. In the initial design, there is no provision for adding the the `Actions` menu. This could, however, become necessary, at which point an `Add Action` option has to be added to the menu. This would call forth an input window allowing the user to define his own actions.

Potentially, the most important feature of the `Actions` menu is the ability to refer to other tuple categories. Clicking on the `refer other tuple partition` button lists the other tuple categories, and clicking on one of those, in turn, gives access to handles to all of the subwindows in that category. Given a handle to the subwindow of another category, the callback routine for the current category can affect both the current category window and the other category window. For abstractions (such as the **or-in**) which involve more than one tuple category, this is be particularly useful. In an **or-in**, a tuple is removed from one of several categories. Hence, we could have all the category windows affected by the **or-in** change color to display that a tuple is being chosen from one of those categories.

The icons that are inserted and removed from the windows need to be defined by the user. The `Icons` menu lists the icons that have already been defined. An icon editor allows new icons to be defined or existing ones to be modified. For each icon, we can also define an expansion function. This function is executed when the icon is

clicked on at run-time. When a tuple bubble is clicked on at run-time in the current implementation, for example, a window pops up listing the tuple data. The expansion function is provided for precisely such cases.

With this proposed Tuplescope tool, users could define their own visualizations more easily than if they were to do so from scratch. The approach is quite simple: draw the windows as they should appear, then write the handler functions controlling these windows and actions on them which are supported by menus. The result of all this is an environment which is more customizable by the average user than the current one.

7.3 Additional Components

The new interface language, together with the suggested modifications to the Linda compiler and Tuplescope, would make the environment easier to use and more customizable for users. While these are definitely avenues worth pursuing, there are additional components which could be added to the environment. We present ideas for two possible components that deserve further research. Neither of these ideas has been explored in any depth.

7.3.1 A Performance Monitor

A performance monitor could make use of information from the LPB. Performance tuning can be a vital component of parallel programming. Parallel programs are meant to run fast, and a little tuning of parameters can make a huge difference. A performance monitoring tool such as Paragraph [HE91] can play an active role in this. Paragraph is a graphical system that visualizes the performance and behavior of a program from trace information. Although it is geared towards message-passing models, it can also be used in conjunction with Linda, although not as meaningfully as it can with message-passing systems. Paragraph allows users to view processor utilization, communication traffic and other such performance data.

A performance monitoring tool that obtains information from a program builder should be able to offer greater benefits than Paragraph. Suppose we build a tool that uses information from the LPB. Depending on which template was used, a performance model would be selected and displayed with the appropriate graphical displays. Various performance metrics could be used. For example, in a database search program, we may wish to examine the total number of **ins** and **outs** in a particular category. In other examples, we may be interested in the total number of tuples in the program, the number of tuples accessed by the different processes, the number of counter updates from a process, the amount of time spent waiting for each tuple, or a wide variety of other data. The LPB could tell the performance monitor what data to display and how to display it.

In conjunction with the LPB, the compiler, and Tuplescope, the performance monitor would complete the environment, being useful in later stages of program development when performance tuning comes into play.

7.3.2 Interfacing to an Expert Database

One of the goals of the LPB was to provide a framework in which programming experience can be captured. This experience is then presented to the user in the form of templates. The identification of a particular template, however, is still up to the user, i.e. the user needs to identify a particular paradigm and then select the appropriate template. It would be useful if, given some description of a goal, the system were able to call up templates which may be useful for the type of program under construction. The system could also bring up fragments of existing programs which directly relate to different segments of a template.

Fertig has developed an expert database system which has some of the necessary capabilities [FG91]. Developing an interface to the LPB and using information from the LPB program-describing database could prove to be an interesting project for the future.

One can envision such a tool being a part of the LPB framework or vice versa, i.e. the LPB could be part of a larger expert framework. The LPB could provide certain types of information to the expert system, or the system in turn could run a program that analyzes the LPB-generated program and extracts relevant data out of it. The situation is similar to interactions between tools in other environments. Consider automatic or semi-automatic parallelizers. A program-builder could pass hints on how to parallelize to the parallelizing compiler which is part of the system. On the other hand, a parallelizing compiler could look at the program and extract its own parallelizing information out of the program.

There are clearly some interesting issues which deserve further research.

Chapter 8

Applying the Concepts to Another Environment

As remarked in Chapter 1, the ideas expressed by the LPB are not limited to Linda, indeed not even to parallel programming. Some of the concepts and lessons we have learned from the LPB can be just as well applied to other environments. An immediate concern is how to modify the LPB to accommodate different languages. What if we wanted a PVM program builder or if we wanted to use Fortan instead of C as a base language?

Modifying the LPB for different base languages is fairly straightforward. If Fortan is desired instead of C, interfaces for user-defined program functions would need to be changed and consequently also the data structures that hold function data. The main features, namely the templates and abstractions, would still be the same.

Although we can apply the template and abstraction features of the LPB to almost any language, there are some features we can add that are peculiar to a language. Consider PVM, for example. The LPB support for tuples clearly does not apply to PVM as it stands. But PVM users have to build data structures to send which need to be unmarshaled at the receiving end. We can add features to the LPB to build and unravel these data structures. In fact, instead of having tuple categories, we could use message categories from which users can instantiate messages without worrying about marshaling or unmarshaling data.

Potentially, we could add more sophisticated language-specific features to the LPB. PVM users need to specify destination node identifiers when sending messages. Keeping track of these identifiers can be a tedious task. For specialized cases, the LPB could construct stylized schemes where the node addresses are handled by the LPB. If the program uses a ring structure for communicating between nodes, for example, the destination node for messages from a specific node is always known. Consequently the LPB could automatically generate references to the destination nodes.

Capturing programming experience in the form of templates is an idea that could be

used for most programming languages. This also applies to abstractions. Essentially, the concept of a program builder which not only interacts with the user, but also interacts with a compiler, visualizer or other tools is flexible enough for many different environments.

Concretely, the LPB framework could be used to develop templates for other programming languages. Users would be able to develop programs quickly, and benefit from the programming experience of others. Some administrative duties can be handed to the program builder itself.

A potentially very interesting use of the framework would be to develop tutorial systems. A good teaching tool needs to guide users through program development. Carefully designed templates can accomplish precisely that. Initially, the templates can be made very rigid. As novices become more experienced, the templates can be relaxed enough to allow users the flexibility of introducing their own ideas.

The framework can also be used as a language extending mechanism (see Chapter 6) for most programming languages. Users can tailor environments according to their own needs without imposing their modifications upon others. The end products, namely the programs, are still portable. Thus, instead of ending up with numerous dialects of a programming language, we can have one base language and customized environments.

Finally, the LPB framework can serve as a means to introduce a new programming methodology without changing the base language. By using the template framework, abstractions and the underlying infrastructure of the LPB, a totally new methodology can be imposed without changing the language. The advantage of this is that the base language remains constant while methodologies evolve. To demonstrate how the ideas of the LPB can be used for precisely such a purpose, the LPB framework was used to prototype an object-oriented C environment.

8.1 A Template-based Object-oriented Methodology Layered on Top of C

The LPB framework was used to layer an object-oriented methodology on top of C. It is important to distinguish between layering a methodology on top of an existing language like C, and developing an object-oriented language (e.g. C++). In the former case, the user still programs in the base language (e.g. C), but the environment somehow ensures that he thinks in an object-oriented framework.

Adopting an object-oriented approach to programming does not require changing programming languages. The object-oriented methodology merely requires programmers to think in terms of data abstractions and to program in particular ways. Specifically, object-oriented methodology requires procedures to be associated with data. A *class* is a definition from which data will be instantiated at runtime. The data has certain *methods* or procedures associated with it. Each instantiation is referred to as an *object.* Objects essentially carry their own environments with them: their own data,

and their own procedures.

Each class has a public interface definition. Users who instantiate objects from a class can only access data or procedures that have been defined as public. All else is considered private and only procedures within the class can access private data or procedures.

One attractive feature of the object-oriented methodology is *name overloading.* Since objects carry their own environments, different objects can have procedures with the same name. We can thus have objects from different classes, each with its own `print` routine which tells the object how to display itself.

Finally, the concept of inheritance is a powerful feature that enables polymorphism. Classes can inherit properties from other classes. For example, a class for cars could inherit from a class for generic vehicles since cars share some basic properties with all vehicles.

All the above features are the necessary ingredients of an object-oriented methodology. The problem is to incorporate these ingredients into a framework that does not require a new language. The framework should be fairly strict in adhering to object-oriented principles. A problem with languages such as C++ has been that they do not insist that users follow the object-oriented methodology. Consequently, many programs developed in C++ have only elements of object-oriented design in them. Our object-oriented framework will adopt a more restrictive approach, allowing users to only write object-oriented programs.

We use the LPB environment to impose the object-oriented methodology. The user still programs in C, but he is guided in an object-oriented framework which forces him to think in this methodology. The end product is still in C. At first glance, templates and classes appear to have some things in common. Templates allow the reuse of program structure, much as classes allow reusing code. By combining templates with a framework for classes, we obtain a powerful tool for object-oriented programming.

The object-oriented methodology was implemented using the template framework, menus and abstractions. As long as the LPB templates are followed, the design enforces an object-oriented methodology. The user need not learn new constructs or use new syntax for writing code — the LPB oversees and enforces the object-oriented framework; the user writes his code. The user does, however, have to know how to navigate the LPB menus.

When a user starts the object-oriented environment within the LPB, a menu of classes appears. The menu lists all the defined classes and includes a `NEW CLASS` option. To view an existing class, the user clicks on its label; to define a new one, he clicks on the `NEW CLASS` option. Each class has its own representation.

Each class has some standard features. When a class is selected at the class defining stage, it is represented in the edit window. There are buttons for each feature. The `initialization code` button expands into a window in which the user writes the code that is executed when an object is instantiated from the class. The `public data` and `private data` buttons require the user to declare the variables in each of those

categories. Buttons for `public functions` and `private functions` are there for users to declare the functions from those categories or to access already defined functions. In particular, clicking on either of those two categories produces a list of defined functions as well as a `NEW FUNCTION` option.

In accordance with object-oriented concepts, classes can use both layering and inheritance. Hence, there is an `is a` button for inheritance and a `has a` button for layering. Clicking on the `is a` button produces a list of classes from which the current class can inherit. Clicking on one of these classes will cause the current class to inherit from the selected class. This in turn affects the public interface of the class.

Users can define a public interface for a class, and can use existing public interfaces to manipulate objects. To work on the public interface definition, the user simply declares the necessary public data and functions. Users of defined classes, however, do not have access to the definitions. In order to use a public interface, the user must access an object that is instantiated from a class. If he clicks on a class label, a list of objects instantiated from that class pops up in the `Instantiations` window. If he wishes to instantiate a new object, he can do so by clicking on the `INSTANTIATE` option in the `Actions` menu. Once a particular instantiation has been selected, the user clicks on the object label and menus appear that list the public data and functions which can be accessed for that particular object. To enforce access via the public interface, menu choices are limited to public data and procedures. In this manner, all private data and functions are protected — the user simply never has the option of accessing something private.

Objects may need to refer to themselves. The `self` button within the class template provides this ability. When a user writes a function within a class, a click on the `self` button will insert a reference to the object through which the the `self` method was invoked. Suppose a user instantiates an object `obj` from a class `A`. If a function `fn` in the class `A` refers to `self`, then the function `obj.fn` will have a reference to the object `obj` itself.

A simple example will demonstrate how the environment works.

8.1.1 Example: Building a Stack-based Calculator

Suppose we wish to build a class for a simple stack-based calculator that only operates on integers. We start by building a class for a stack of integers. We click on the `NEW CLASS` option in the `Classes` menu and type in `IntStack` when prompted for a name for the class. The `IntStack` class is now displayed in the edit window (Figure 8.1).

Our `IntStack` class will use an array of integers to implement the stack and an integer to hold the index of the top of the stack. Hence, we need to declare the appropriate variables as private variables and we need to initialize the top index to point to the bottom of the initially empty stack. We click on the `private data` button and declare our variables in the window that appears; we click on the `initialization code` button and type in the initialization for the top index (Figure 8.2). Any memory

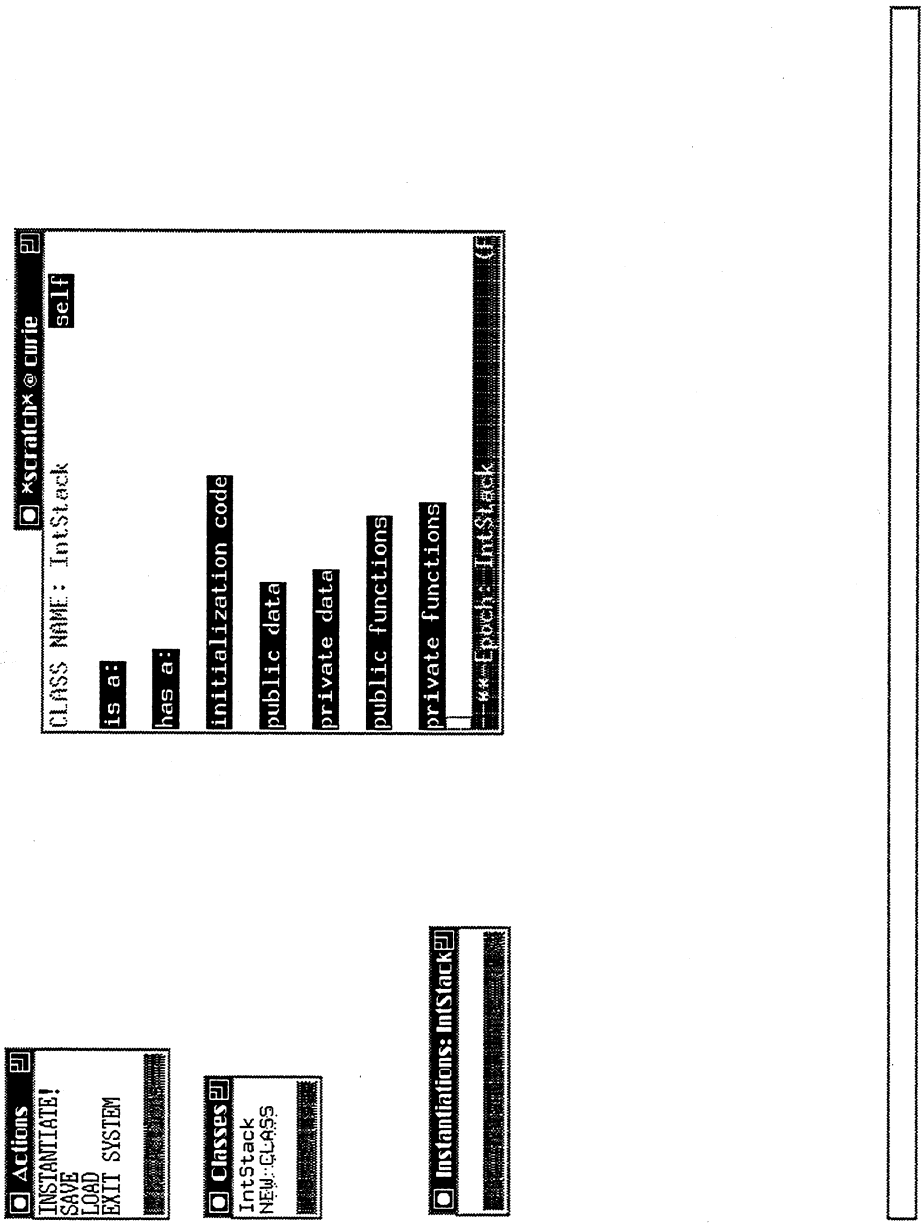

Figure 8.1: An Object-oriented Example: Defining the IntStack Class

allocations would have to be made in the initialization routine. Deallocation would be handled by the system by means of a sophisticated garbage collector which monitors the number of references to objects and memory. Although this garbage collector has not been implemented as part of our prototype, it is fairly straightforward to implement.

The `IntStack` class needs public functions to push integers on the stack or pop them off the stack. To define the `push` function, we click on the `public functions` button and select `NEW FUNCTION` in the menu that appears. We then type in the name and return type for the function. An input window for the new function pops up. This window initially holds the function skeleton. An additional input window appears, asking us to declare the parameters to the function. The `push` function requires a single argument: the integer to be pushed onto the stack. We thus declare a single integer in the parameter window (Figure 8.3).

Once the parameters are declared, the `push` function skeleton needs to be filled. We do this by directly editing the input window for the function. To push an integer on the stack, we simply copy the integer onto the top of the stack and shift the top of stack index by one (Figure 8.4). We define the `pop` function in a similar manner. The `Public Functions` menu now lists both of the newly defined functions and we are done defining the `IntStack` class.

Now that the `IntStack` class has been defined, we will use it to construct another class. We declare a new `calculator` class. For the sake of this example, the calculator is simple — it only operates on integers. The core of the calculator will be a stack of integers instantiated from the `IntStack` class. Since we will be using an object instantiated from the `IntStack` class, we click on the `has a` button to indicate layering. A menu pops up, listing the possible classes which the `calculator` class can use (Figure 8.5). We click on the `IntStack` option and now are ready to proceed.

Instantiating an object from a class is fairly straightforward. Suppose, for example, that we wish to instantiate an object of type `IntStack` in a function within the `calculator` class. We define our function and move the cursor to the point where the `IntStack` is to be instantiated. We then click on the `IntStack` label in our `Classes` menu and subsequently click on the `INSTANTIATE` button in the "Actions" menu (Figure 8.6). The system responds by asking us for a label to which to bind the instantiation. We type in "stack" and our instantiation is inserted into our code and highlighted to show that it is an instantiation of an object (Figure 8.7).

This framework thus allows us to use class layering which allows objects to access objects from other classes. This is useful, but no object-oriented framework is complete unless it deals with inheritance.

8.1.2 Inheritance

Inheritance allows an object of one class to inherit all public properties from another class and thus be treated as if it were part of that other class. In particular, if class D inherits from class A — i.e. class D is derived from class A — then every object of

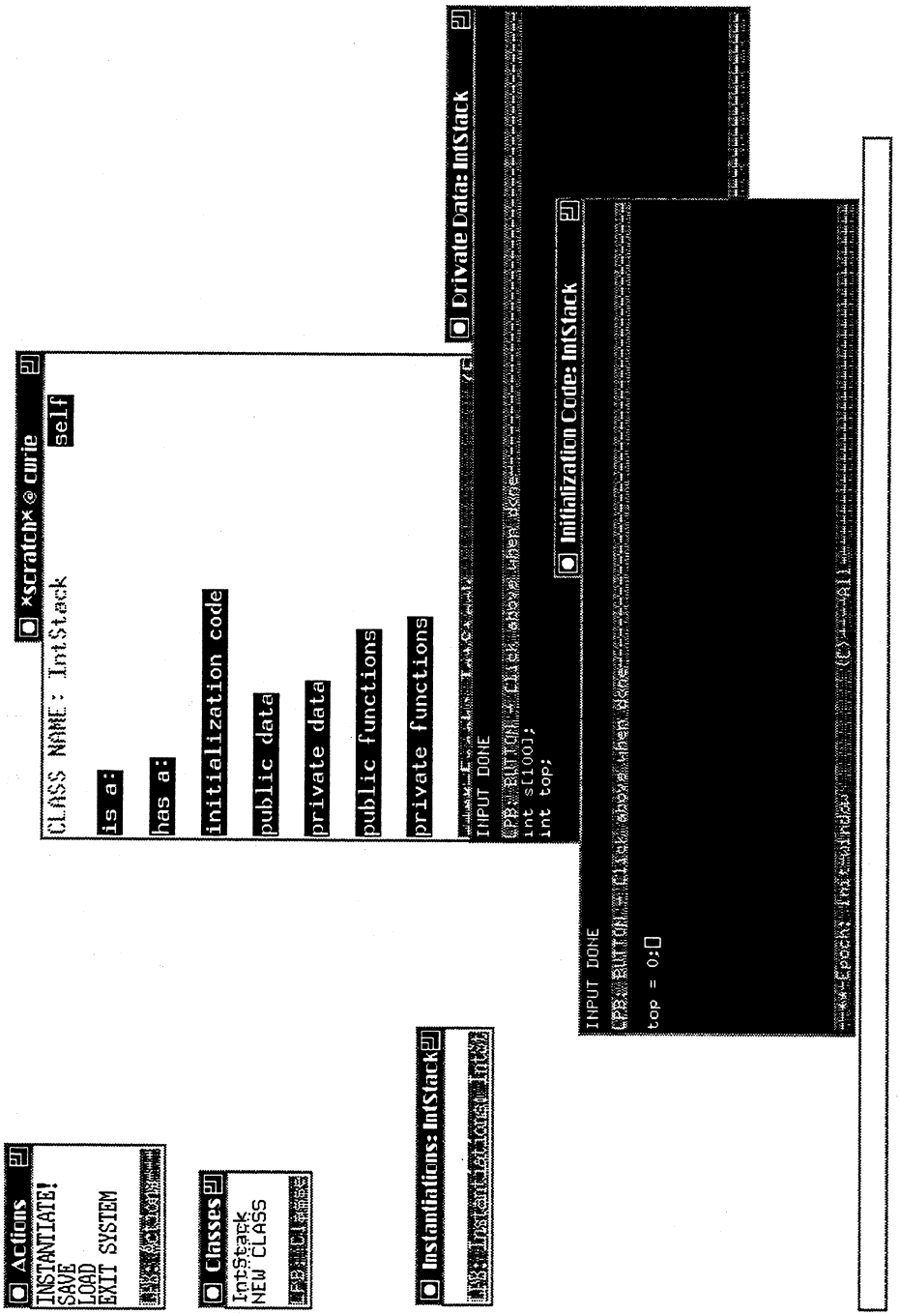

Figure 8.2: An Object-Oriented Example: Private Data and Initialization Code

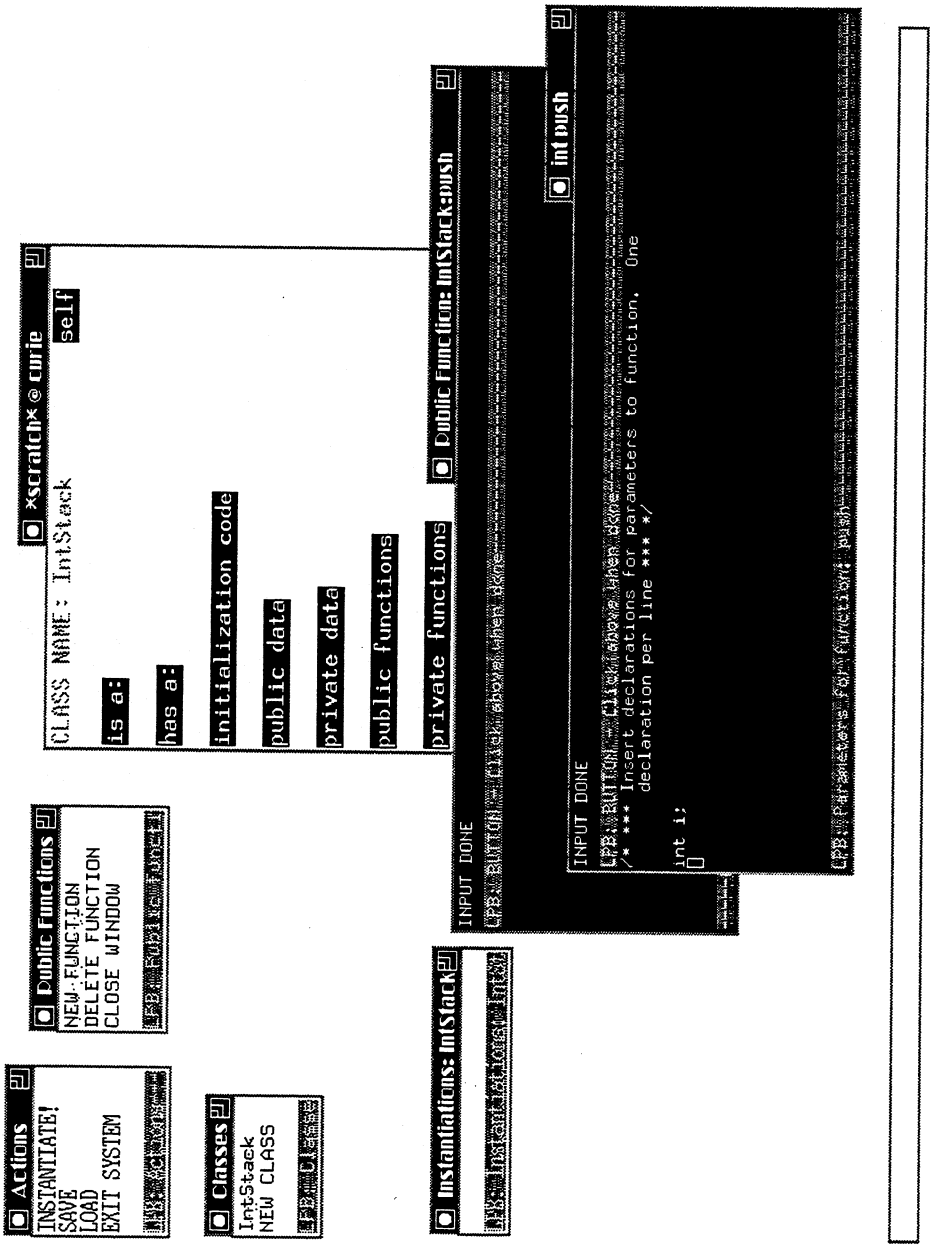

Figure 8.3: An Object-Oriented Example: Parameters to the Push Function

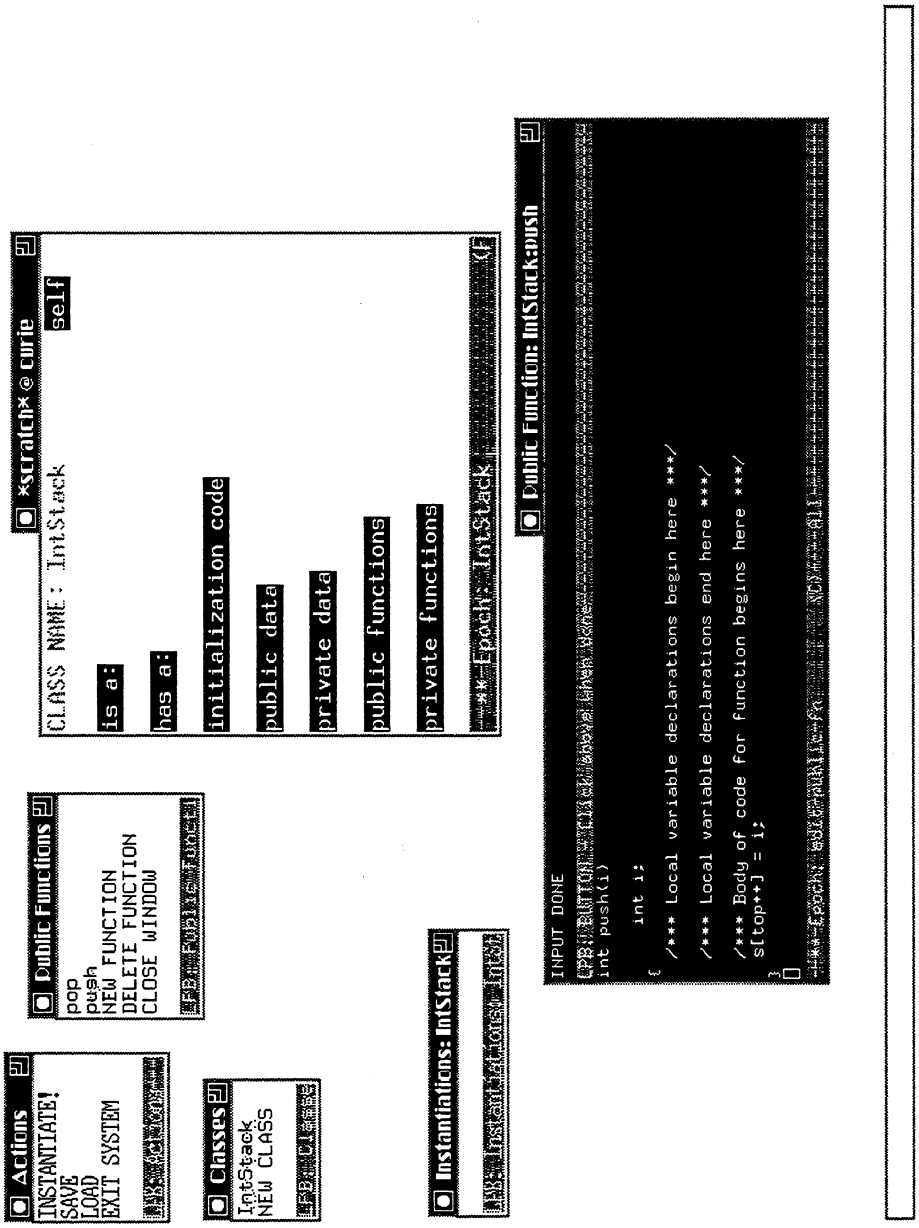

Figure 8.4: An Object-Oriented Example: Completing the Push Function

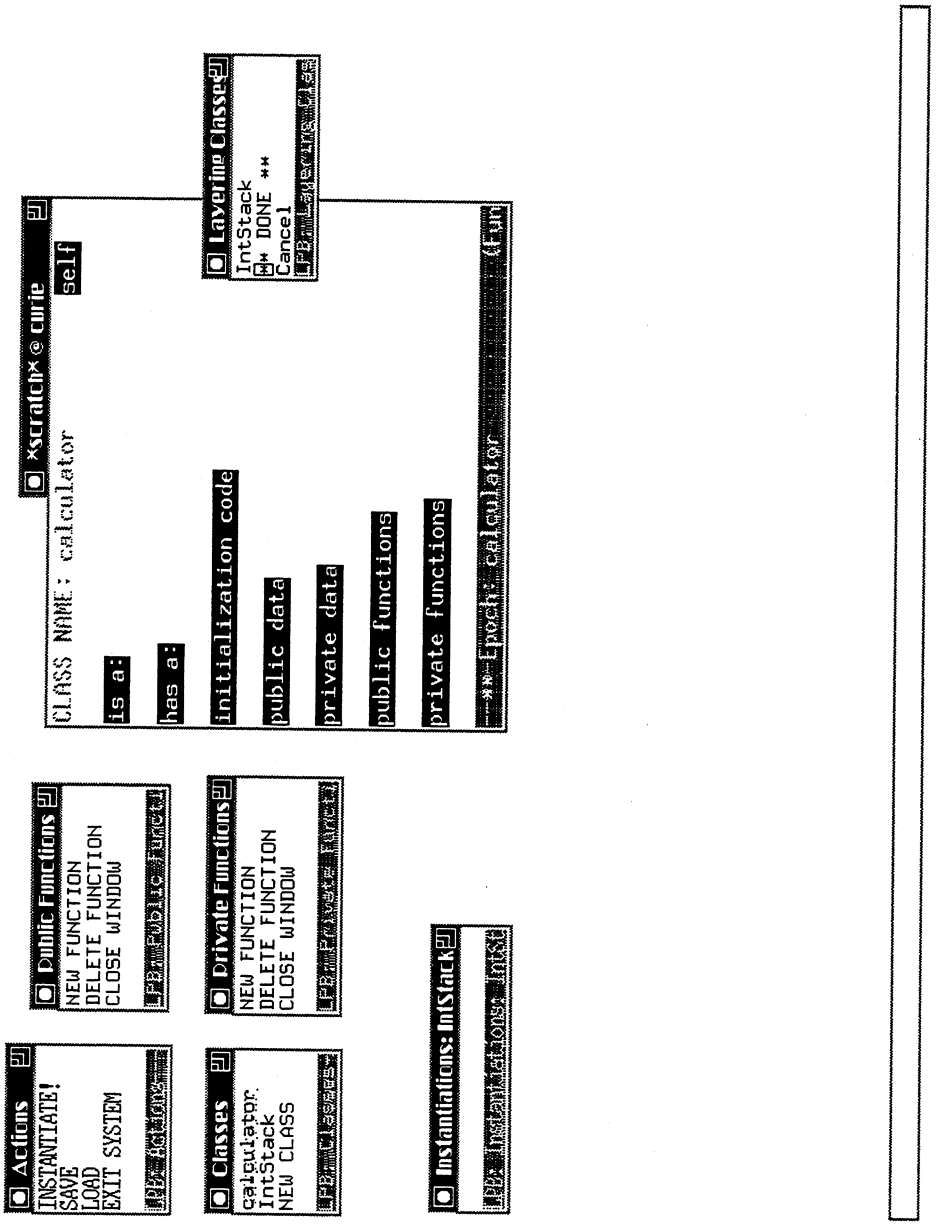

Figure 8.5: An Object-Oriented Example: Layering Classes

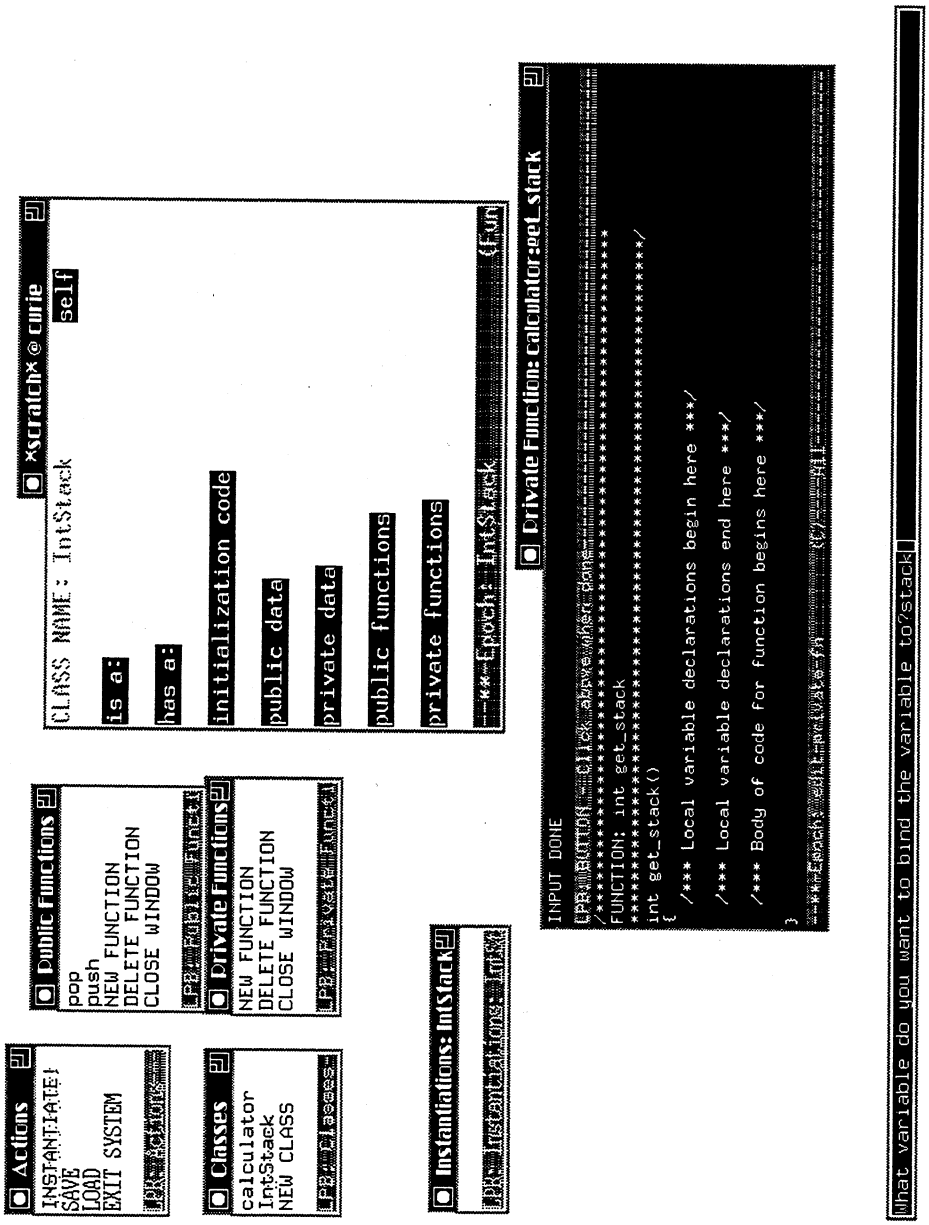

Figure 8.6: An Object-Oriented Example: Instantiating an Object

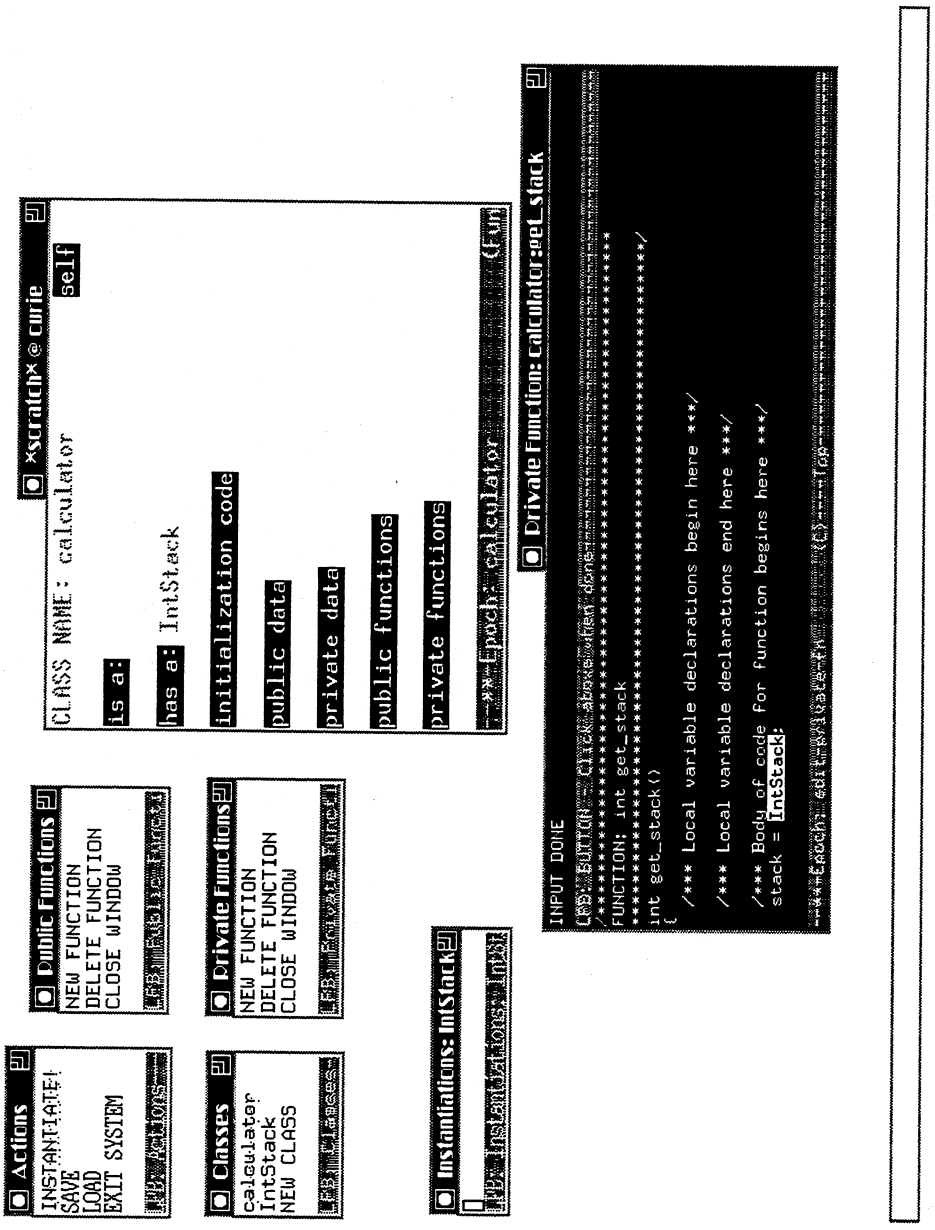

Figure 8.7: An Object-Oriented Example: An Object Instantiation

type D can be used as if it were an object of class A.

The benefits of inheritance are substantial. It allows common behavior of related classes to be abstracted to more general classes. It also allows classes to be specialized incrementally. The object-oriented framework constructed with the LPB provides the benefits of inheritance.

Consider the example of designing a spreadsheet using object-oriented design. We define a generic cell class which has properties that will be shared by more specialized classes. The generic cell class has three public functions: `display`, `whichColumn`, and `whichRow`. It also has some private data that is used by its functions (Figure 8.8). Now suppose we define a specialized cell class for equations. We call this cell `EquationCell` and declare a public function called `compute`. More importantly, we click on the `is a` button and declare that `EquationCell` inherits from (i.e. is derived from) class `Cell`. If we now instantiate an object from the `EquationCell` class and bind this instantiation to `eqn_cell_ptr`, this object appears in the `Instantiations` menu. We click on this object label, and two menus appear: one listing the public functions and one listing the public data. These are the functions we can use for this object. The compute function from the `EquationCell` class is listed as expected. In addition to this, however, the `display`, `whichColumn`, and `whichRow` functions are also listed in the public interface. This is a result of inheritance — the `EquationCell` has inherited all the public properties of the generic `Cell` (Figure 8.9).

8.1.3 Implementation

Prototyping the object-oriented framework using the LPB proved remarkably easy. The underlying infrastructure is provided by the LPB. The LPB provides support for creating templates, abstractions and menus, and provides input routines used to declare functions and parameters. The relevant data are always stored in the program-describing database.

Designing the object-oriented framework required creating the proper templates, abstractions and menus. It also required creating a new data structure to hold classes and their data and functions. The new data structure keeps track of class dependencies and wraps data structures for individual functions within a class data structure. This rather simple scheme was basically all that was necessary to impose the object-oriented methodology.

Unfortunately, creating the data structure to hold the classes required some knowledge of the LPB's implementation. There is currently no interface to design or use internal LPB data structures. Instead, an environment implementor has to use the library of epoch lisp routines that constitutes the basis of the LPB. While this is easy to do for someone familiar with the library, there is a learning curve to be mastered for people who are not. This ultimately leads to the question of whether the LPB is useful only to the end users or whether it can also be useful to the environment developers.

The benefits to environment developers may not be as significant as those to end

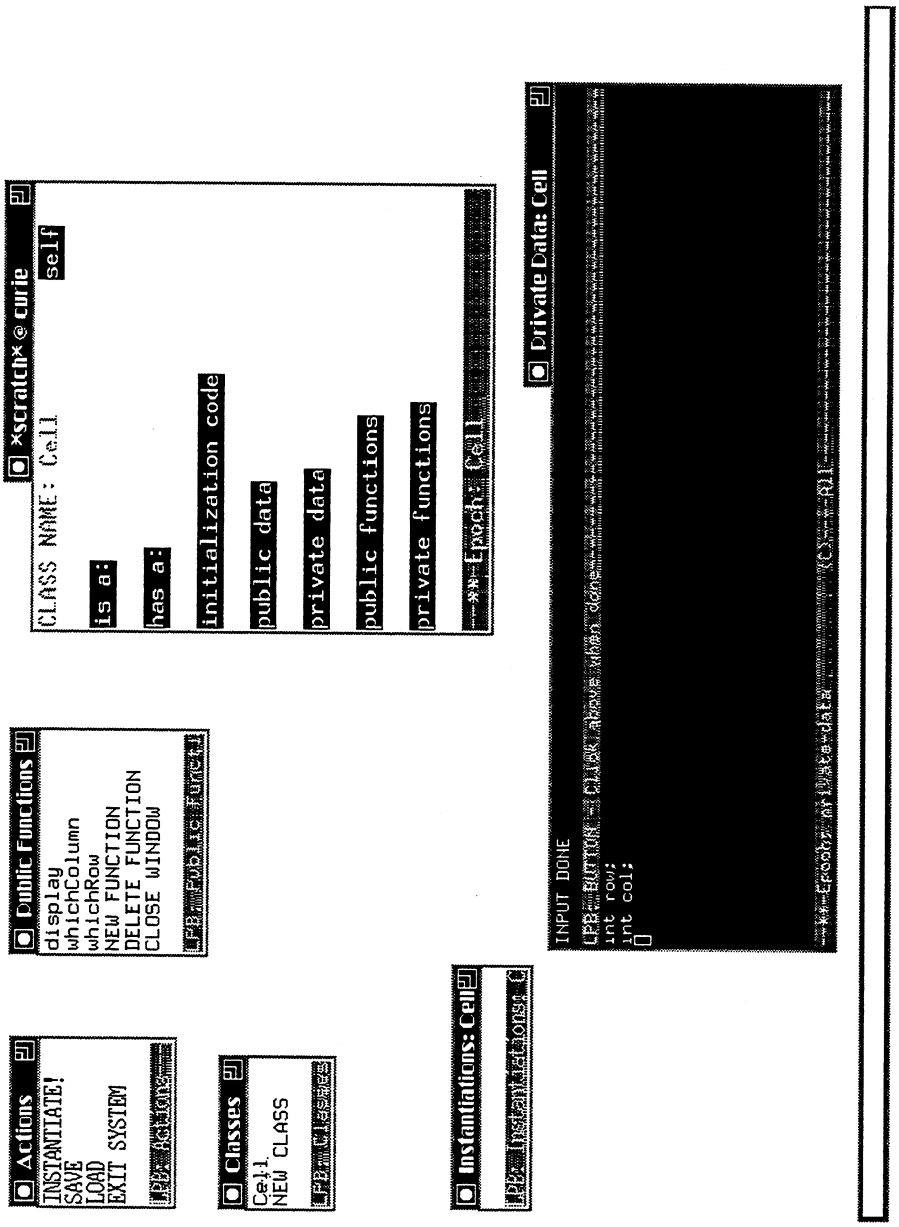

Figure 8.8: An Object-Oriented Example: Defining a Cell Class

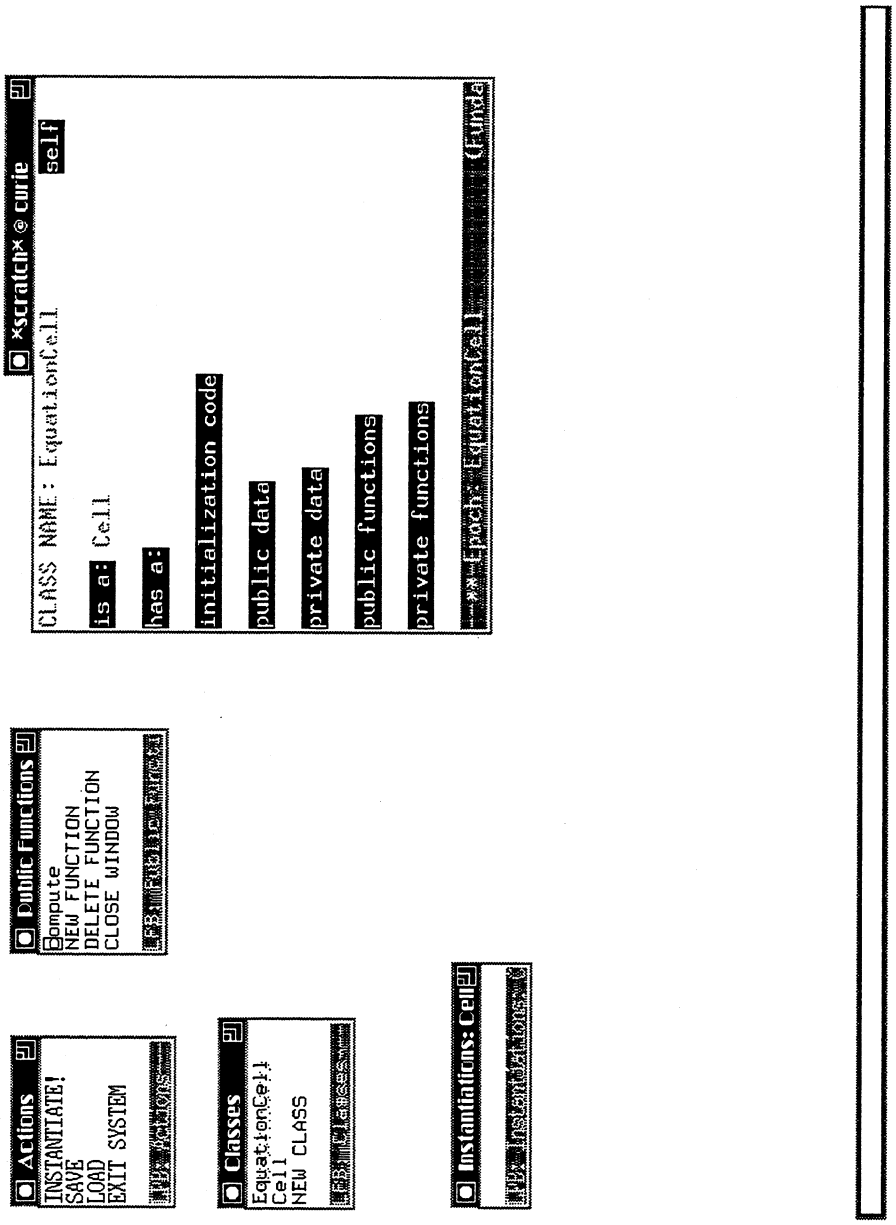

Figure 8.9: An Object-Oriented Example: Inheriting a Class

users, but they are still worthy of mention. The LPB routine library has a collection of routines that can be used quickly to develop customized environments. The object-oriented C example is a case in point. The developer did not need to build supporting routines from scratch. Instead, the library routines provided much of the needed support. The entire object-oriented C prototype was implemented in a day.

In the long run, a better interface is desirable. The template-building template is clearly insufficient to construct generic environments. In the object-oriented methodology case, for example, there is no simple syntactic pattern of use which represents layering or inheritance. Consequently, implementing the methodology required reaching beyond the template-building template and using some of the LPB library routines. There is potential here for future research. One could envision a meta language driven by a graphical interface, allowing users to design data structures and routines to manipulate them.

The primary beneficiary of LPB-developed environments is the end user. The user follows a new methodology, but sticks to a familiar base language. The most attractive feature of the object-oriented framework in the LPB is possibly the fact that the user still programs in C. Each function he writes is still in C and the whole object-oriented methodology is imposed by the interface. Templates can guide users in constructing programs, and the graphical, menu-driven interface gives users a good overview. Even though the end user is the primary beneficiary of LPB-developed environments, the LPB provides a good infrastucture upon which developers can build their environments quickly.

Two of the LPB's most attractive features are its support for debugging and optimizing. At first glance, neither seems particularly applicable to the object-oriented environment we have prototyped. Although no debugging or optimizing features were incorporated in the prototype, some come for free and others could be introduced relatively easily. For example, undefined methods cannot be dispatched because the interface only allows access to defined methods. Furthermore, the LPB could potentially detect circular class definitions since classes are defined by invoking menu functions which store classes internally in a table. It would be fairly simple to search the table for circular definitions. Optimizing the object-oriented code based on semantic information could follow if a particular runtime system is targetted. The optimizations would follow the same strategy that the LPB currently employs for C-Linda.

There are some interesting issues regarding program evolution that were not pursued in the object-oriented prototype. For example, if there are modifications to a method in a class, will all uses of that method be updated with the appropriate type-checking and error detection information? The LPB should be able to maintain enough information to provide that level of updating. In fact, the case is similar to that of tuple updates. Recall how tuples that have already been defined can be modified — a user may decide to change the number or nature of fields of the tuple. All the changes are propagated to all references to that tuple throughout the code. The LPB can do this automatically because it maintains information on where references to a

tuple occur. Similarly, since object usage is menu-driven in our prototype, it should be fairly straightforward to maintain information on all references to objects from specific classes and run checks on them whenever classes are updated in any way. Of course, there are some non-trivial questions that arise from this. For example, if multiple users are sharing an object library, modifying an object that is being used by a different user might require obtaining permission from that user. The environment developer has to make decisions on how much automatic updating to support.

The environment developer has to specify the transformation from the high level features of a methodology to some base language. In our example, the transformation is from our object-oriented methodology to C. It is easy to visualize transforming the user code into some subset of C++. Although this appears to be self-defeating in light of the earlier discussion on the desirability of keeping all code in C, there is something to be learned from running the transformation as a Gedanken experiment. The mapping from C++ to C is a solved problem — the early C++ preprocessors did precisely that. In the case of our object-oriented environment, however, code maps to a very small subset of C++, leaving out all the complications of the latter. This is because our environment does not support the numerous features that C++ offers. Instead, our environment supports only the basic necessities of object-oriented programming. The lesson to be learned from this is that there is a simple transformation from our object-oriented environment to C. This is because any programs generated with our environment map to a small subset of C++ and C++, in turn, has a straightforward mapping to C.

The relationship to C++ raises an interesting question. If we decided to build an LPB-environment on top of C++, what would it look like? C++ already offers inheritance as part of the base language. An object-oriented environment built on top of C++ could still look very similar to our prototype system on top of C. Some of the LPB environment's features will already be supported by the base language and hence won't be a novelty. But there are additional benefits. One of the main problems with C++ is that it allows users to write code that is not really object-oriented. There is nothing which enforces an object-oriented methodology. In that respect, the LPB environment can enforce the methodology simply by nature of its interface. Furthermore, because of the nature of information that the user gives to the LPB, the system could do error checking involving things such circular class definitions or illegal method dispatches. Finally, if new,refined object-oriented methodologies arise, they can be incorporated into the LPB framework.

The object-oriented framework we implemented on top of C was just a prototype, and thus only a flat transformation was implemented for demonstration purposes, i.e. inheritance was not handled by the transformation to C. Extending the environment to transform inheritance into C, however, should be a fairly straightforward exercise. Furthermore, the LPB object-oriented environment does not support parameterized types. Parameterized types would allow us to bind types at runtime. For example, we could defined a generic stack class and instantiate an IntStack from the generic stack

class at runtime.

The LPB object-oriented environment could be extended to support parameterized types. Menus would offer a generic type which can be used as a placeholder when defining a class. Internally, the LPB system would mark these placeholders as incomplete until type specifications have been provided. At the time of instantiation, the user is then prompted by the system to declare what type he would like to instantiate an object from that class with, i.e. the user is asked to fill the placeholder for that particular instantiation. When the LPB generates the C code, the type information is already in place.

Finally, there is the issue of passing classes to other users. The header file for a program is constructed in the form of an LPB file and contains information only on the public interface. The actual implementation of the public procedures and all private data and procedures are passed to other users as object code. Thus, users can instantiate objects from classes and use their public interfaces, but cannot access the private data or procedures.

What we have learned from the exercise is that the LPB framework can readily be extended to implement or prototype other environments. The ideas behind the LPB are not restricted to Linda or even to parallel processing, although some of its features support parallel programming directly. The object-oriented C environment example has demonstrated how the LPB can be used to adapt completely new programming methodologies and layer them on top of a base language.

Chapter 9

Related Work

The LPB relates to other work in various ways. The editor component of the LPB is most closely related to structure editors. The visualization modifications to Tuplescope relate to visual programming environments, data visualizers, and debuggers for parallel processing. Finally, there has been some recent work on parallel programming environments that could provide hints on where research in the area is headed.

9.1 Structure Editors

The LPB's most important template-based structure editor predecessor is the Cornell Program Synthesizer [RT89b]. The Cornell Program Synthesizer allows users to specify attribute grammars for languages and automatically constructs structure editors according to those grammars. The resulting structure editors are very rigid in enforcing structure, not allowing users to deviate from the structures at any stage. This is great for novices who need a strong guiding hand to produce syntactically correct programs, but can be restrictive on expert programmers who don not need to be led by the hand. Creative programmers generate ideas at whim and like to code them immediately. There are two ways in which a rigid framework can hinder creativity: (1) The additional overhead of going through fancy interfaces and following incremental guidance can slow users down, and (2) a specific framework that insists on a particular methodology can prevent users from creating methodologies of their own. Other structure editor systems that are similar to the Cornell Program Synthesizer include Aloe [FMM81] and the related Gandalf [HN86] environments, MENTOR [gea84], CENTAUR [BCD+89] and IPSEN [Lew89].

To overcome these problems and allow flexibility, the LPB allows users to bypass templates at any stage. Unlike the synthesizer, the LPB does not enforce a rigid framework. Instead, the LPB captures methodologies and supports them, without imposing a strategy. The LPB produces source code, and the programmer is free to ignore or modify this as desired. This flexibility is essential to any expert programmer.

The key idea is to serve as an aid to users only when the aid is wanted. Unsolicited help can be a nuisance if it hinders creative software development since it can prevent programmers from developing original solutions different from the suggested ones. Of course, all the flexibility comes at a price. Since we allow users to bypass templates, we could end up in potential inconsistency problems (see section 2.1).

KBEmacs [Wat85] was designed with a similar motivation to the LPB. Like the LPB, KBEmacs also used an Emacs-like environment and layered a higher-level programming environment on top. The guidance frameworks in KBEmacs are known as *cliches*. Cliches are program skeletons with placeholders that are filled in by users — a concept which is similar to that of LPB templates. Unlike LPB templates, however, cliches do not incrementally guide the user through option trees, nor do they provide frameworks to implement global-effect constructs such as the **or-in**. The KBEmacs emphasis is on formalizing guidance frameworks and did not investigate issues like the above. Criticisms of KBEmacs include its very slow speed and lack of a taxonomic library of cliches.

PSG [BS86] allows both structure and text editing, although the dynamic semantics of the language are defined in a functional language which is used to interpret the programs. In a sense, the approach is somewhat similar to the Cornell Program Synthesizer because it generates language-specific programming environments from a formal language definition. The language definition itself, however, has to be written in the functional language. This requires the user to be familiar with the functional language.

Odin [CO90] investigates the idea that tools should be centered around a persistent centralized store of software objects. This is only vaguely related to the LPB in the sense that the LPB uses a persistent store of tuple definitions from which users can generate tuple operations at any time. The LPB notion of "persistent" is somewhat more restrictive, however, because killing an LPB session also kills the persistent tuples. If a new session is started with the same files, the tuple store is reactivated. In the meantime, however, the "persistent" store disappears and if other unrelated files are opened, a new "persistent" store is created.

9.2 Parallel Programming Environments

An interesting approach is taken by Enterprise [SSLP93] and HeNCE [BDG+92], which allow users to express parallelism graphically. Users can thus specify communication and synchronization through a different medium from the conventional textual one. Both these systems use PVM and are closely tied to a message passing framework. When developing a program with one of these systems, the user draws the nodes representing the processes of his program, and then draws arcs to represent the communication between the processes. For programmers interested primarily in developing parallel software using message passing, Enterprise and HeNCE serve a useful purpose.

The program builder approach is more general. We can use a program builder to construct environments for other programming requirements (e.g. an object-oriented environment, Section 8.1). The basic infrastructure of a program builder is not tightly coupled to a particular framework and is thus more adaptable. One could, however, build a more restrictive interface on top of a program builder for a select community such as the programmers who prefer a message passing approach to parallel software. In fact, we can build a PVM environment with the LPB (Chapter 8) which provides similar benefits to HENCE by taking care of all administrative aspects of communication for the user.

Extensible parallel programming environments such as SIGMACS [SG91] generate a program database during compile time that can be used during later modifications to the program. The LPB, on the other hand, maintains a dynamic program-describing database that grows as the program is constructed. This allows the system to maintain semantic program information that is very difficult to infer at compile time. There are other systems that gather semantic information on a program. In Faust [Lev93], for example, the system tries to collect semantic information on a program in order to parallelize the program. At times, it may interactively ask the user for semantic information to help in its parallelization attempts. This is somewhat different from the LPB approach where the semantic information is implicitly provided by the user during the program construction stage. Systems like Faust probe deeper into the code to find very specific kinds of information, to find dependencies in loops. On the other hand, the LPB automatically infers semantic information at the program construction stage. This information is used to guide program development, to check consistency, to document, to provide optimizing information to the compiler [CG91], to enhance graphical monitoring, and potentially also to benchmark programs by visualizing performance in the spirit of Paragraph [HE91].

9.3 Visualization Tools

There is currently much research effort in visualizing the dynamic behavior of parallel programs. PARADISE [KC91] is a good example. Users construct models using a metalanguage. Similarly, DPOS [EK91] uses a metalanguage to define networks and provides a set of tools for visualizing and debugging. Users construct programs incrementally in this framework and make use of the metalanguage to specify program segments at high levels of abstraction. I-Pigs [Pon91] is an interactive graphical environment for concurrent programming, using a specially-designed graphical language, Pigsty. Pigsty is based on CSP and Pascal, and currently limited to single processes and one-dimensional arrays of processes.

The advantage of the LPB approach is that the user still programs in a familiar base language. Information is implicitly provided by the user to the LPB while constructing programs, and this information is automatically conveyed to the visualizer.

Tuplescope [BC90], the graphical monitoring tool, acts on this information and programmers can visualize dynamic information at a higher abstraction level than would otherwise be possible. Furthermore, Tuplescope visualizes program behavior at runtime, unlike systems such as Paragraph [HE91] which animate program behavior from trace information after a program has already been run.

As Tuplescope is expanded further to accommodate more detailed and better visualizations, it will become more interesting to investigate some of the work that has been done in *visual systems* (systems that allow users to employ non-textual aids in constructing programs). There has been much work in the general area of visual systems such as PECAN [Rei85], Garden [Rei86], Use.It [HZ76], PegaSys, [MH86], PICT/D [GT84], ThinkPad [RGR85], and PT [HA88].

Some commercial environments such as HP's Softbench, ParcPlace C++, Lucid's Energize, or DEC's FUSE [Fol92] use techniques similar to the ones in the systems mentioned above.

9.4 The Linda Program Builder

The LPB features which stand out are its use of graphics and hypertext to display and expand templates, the use of abstractions, and the ability to gather semantic information and pass it to other tools. In addition, the other tools enhance their performance using the semantic information. Finally, there is the general flexibility and adaptability of the program builder approach. The flexibility and adaptability has interesting implications: it suggests that the use of a program builder as a "dynamic" preprocessor alternative to new programming languages may allow users to customize their environments according to needs without confusing anyone else in the process (see Chapter 6).

Chapter 10

Conclusions

To understand the implications of the Linda Program Builder and program builders in general, we should first examine the accomplishments of the LPB.

10.1 Summary of Accomplishments

The Linda Program builder is a CASE (Computer Aided Software Engineering) tool with a broader mission than simply to serve as a CASE utility. It supports construction of parallel programs written in C-Linda, and also provides the infrastructure to develop coordination or guidance frameworks for other environments. LPB abstractions are used to build constructs which are transformed into a sequence of base language operations. This allows language extensions to be added without changing the language itself.

The backbone of the LPB is the program describing database. The LPB gathers semantic information on programs as they are written and puts the information in the database. When a user saves a program, the LPB generates a semantic information file which is later read by the compiler and the graphical monitoring tool. The compiler optimizes its code based on the information acquired from the semantic information file; the graphical monitoring tool enhances its display based on the same information.

The LPB's template-building template is a mechanism for constructing coordination frameworks. Programmers can construct templates capturing their experience by invoking this mechanism.

To overcome difficulties integrating with the current LPB environment, both the Linda compiler and visualizer need to be redesigned. If they are redesigned with an LPB interface in mind, performance will be improved on a number of fronts: The LPB will be able to provide more information to the other tools, and the tools will be able to enhance their performance based on this. Specifically, the compiler can optimize code and Tuplescope can visualize programs better. Consequently, the user also should see improved performance since his debugging time is reduced. It will be easier for users to

specify their own optimizations and visualizations. Preliminary work on the redesign of the tools has been done and a new interface language linking the LPB and the tools has been designed.

The concept of a program builder transcends the LPB. Its facilities for code generation and transformation, and for graphical and dynamic abstractions, suggest its use as an alternative to new programming languages. It allows environments to be customized according to needs without affecting portability of the code that is constructed.

Although the LPB in its current form caters to a particular programming community, it is not limited to any specific programming environment. The basic infrastructure and ideas can support very different environments. A prototype of an object-oriented C environment was constructed using the LPB framework to demonstrate how the LPB can be used to develop other environments.

10.2 Conclusions

A number of lessons have been learned from the LPB project. We have learned how to construct a tool that aids programmers in developing explicitly parallel code. The LPB supports basic tuple operations as well as higher-level functions, and offers complete templates and program structures. It offers a framework that can help capture programming methodologies and guide users through program development. As new methodologies emerge, they can be added to the LPB in the form of templates which are constructed via the the template-building template. The templates aid users without imposing themselves on the users. Templates can serve an educational role in teaching novices how to write particular kinds of programs. But they can also serve as an aid to expert programmers.

Further, the LPB's program-describing database can supply information to other tools in the environment, enabling optimization at compile-time, enhanced visualization at run-time, and eventually also performance monitoring for efficiency.

There are some interesting implications of the LPB. It is characteristic of a potentially significant trend in programming language design. It addresses the traditional conflict between keeping a language simple and simultaneously demanding that it be higher-level. The proposed solution is to combine a simple, general coordination language with a higher-level, domain-specific system that provides the power and higher-level abstractions that a programmer can selectively employ.

In sum, we can have our cake and eat it too. If we can capture the methods and idioms that skilled programmers rely on *without* complicating the language itself with a galaxy of high-level, special-purpose constructs, we have a solution to an important problem.

Bibliography

[ACG94] S.W. Ahmed, N.J. Carriero, and D.H. Gelernter. A Program Building Tool for Parallel Applications. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, May 1994.

[AG92] S.W. Ahmed and D.H. Gelernter. A CASE Environment for Parallel Programming. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering.* IEEE Computer Society Press, July 6-10 1992.

[BC90] P.A. Bercovitz and N.J. Carriero. TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs. Research Report 782, Yale University Department of Computer Science, April 1990.

[BCD+89] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGPLAN Notices 24, 2*, February 1989.

[BDG+92] A. Beguelin, J. Dongarra, G.A. Geist, R. Manchek, K. Moore, R. Wade, J. Plank, and V. Sunderam. HeNCE: A Users' Guide. User Guide, Oak Ridge National Laboratory, February 1992.

[Bjo90] R.D. Bjornson. *Linda on Distributed Memory Multi-Processors.* PhD thesis, Yale University Department of Computer Science, New Haven, Connecticut, 1990.

[BS86] R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, October 1986.

[Car87] N.J. Carriero. *Implementing Tuple Space Machines.* PhD thesis, Yale University Department of Computer Science, New Haven, Connecticut, 1987.

[CG89] N.J. Carriero and D.H. Gelernter. Linda in Context. *Communications of the ACM*, April 1989.

[CG91] N.J. Carriero and D.H. Gelernter. A Foundation for Advanced Compile-time Analysis of Linda Programs. Technical report, Yale University Department of Computer Science, 1991.

[CG92] N.J. Carriero and D.H. Gelernter. A Foundation for Advanced Compile-time Analysis of Linda Programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, number 589 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1992.

[CG94] N.J. Carriero and D.H. Gelernter. Case Studies in Asynchronous Data Parallelism. 1994.

[CO90] G. Clemm and L. Osterweil. A Mechanism for Environment Integration. *ACM Transactions on Programming Languages and Systems*, January 1990.

[EK91] J.D. Evans and R.R. Kessler. A Metalanguage and Programming Environment for Parallel Processing. *LISP and Symbolic Computation: An International Journal*, 1991.

[FG91] S. Fertig and D.H. Gelernter. A Software Architecture for Acquiring Knowledge from Cases. In *Proc. of the International Joint Conference on Artificial Intelligence*, August 1991.

[FMM81] P.H. Feiler and R. Medina-Mora. An Incremental Programming Environment. *IEEE Transactions on Software Engineering*, September 1981.

[Fol92] M.J. Foley. Can We Talk. *SunExpert*, January 1992.

[gea84] V. Donzeque gouge et al. Programming Environments based on Structured Editors: The Mentor Experience. In *Interactive Programming Environments, Barstow, Shrobe and Sandewall.* McGraw-Hill, 1984.

[GT84] E.P. Glinert and S.L. Tanimoto. Pict: An Interactive Graphical Programming Environment. *Computer*, November 1984.

[HA88] Hsia and Amber. Programming through Pictorial Transformations. In *Proceedings of the IEEE International Conference on Computer Languages 1988.* IEEE Computer Society Press, October 1988.

[HE91] M.T. Heath and J.A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, September 1991.

[HN86] A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, December 1986.

[HZ76] M. Hamilton and S. Zeldin. Higher Order Software - Methodology for Defining Software. *IEEE Transactions on Software Engineering*, March 1976.

[Kam94] D.L. Kaminsky. *Adaptive Parallelism with Piranha.* PhD thesis, Yale University Department of Computer Science, New Haven, Connecticut, 1994.

[KC91] J.A. Kohl and T.L. Casavant. Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems. In *Proceedings of the Fifth International Parallel Processing Symposium.* IEEE Computer Society Press, April 30 - May 2 1991.

[Lev93] J.M. Levesque. FORGE 90 and High Performance Fortran (HPF). In *Software for Parallel Computation.* Springer-Verlag, 1993.

[Lew89] C. Lewerentz. Extended Programming in the Large in a Software Development Environment. *ACM SIGPLAN Notices 24,2*, February 1989.

[MH86] M. Moriconi and D.F. Hare. The PegaSys System: Pictures as Formal Documentation of Large Programs. *ACM Transactions on Programming Languages and Systems*, October 1986.

[Pon91] M.-C. Pong. I-Pigs: an Interactive Graphical Environment for Concurrent Programming. *The Computer Journal*, August 1991.

[PS85] J. Peterson and A. Silberschatz. *Operating System Concepts.* Addison-Wesley, 1985.

[Rei85] S.P. Reiss. Pecan: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering*, March 1985.

[Rei86] S.P. Reiss. Garden Tools: Support for Graphical Programming. In *Advanced Programming Environments, Lecture Notes in Computer Science Nr.244.* Springer Verlag, 1986.

[RGR85] R.V. Rubin, E.J. Golin, and S.P. Reiss. ThinkPad: A Graphical System for Programming by Demonstration. *IEEE Software*, March 1985.

[RT89a] T. Reps and T. Teitelbaum. *The Synthesizer Generator : A System for Constructing Language-based Editors.* Springer-Verlag, 1989.

[RT89b] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, 1989.

[SG91] B. Shei and D. Gannon. SIGMACS A Programmable Programming Environment. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990).* Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[Sol92] Integrated Computer Solutions. *The Builder Xcessory User's Guide.* Integrated Computer Solutions, Inc., 1992.

[SSLP93] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. Research Report, Department of Computer Science, University of Alberta, 1993.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[SZBH86] D.C. Swinehart, P.T. Zellweger, R.J. Beach, and R.B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, October 1986.

[Wat85] R.C. Waters. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, November 1985.