Program Builders as Alternatives to High-Level Languages

Shakil Ahmed and David Gelernter

YALEU/DCS/RR-887
November 1991

Program Builders as Alternatives to High-Level Languages*

Shakil Ahmed and David Gelernter
Department of Computer Science
Yale University
P.O. Box 2158, Yale Station
New Haven, CT 06520-2158
U.S.A.
Phone: (203) 432-1278
Internet: ahmed@cs.yale.edu, gelernter@cs.yale.edu
Bitnet: ahmed@yalecs.bitnet, gelernter@yalecs.bitnet

November 19, 1991

Abstract

"Program builders" are text editors augumented with language and programming-methodology specific functionality; they build programs in a conventional source language incrementally, under the user's control. Programs built in this fashion are eventually compiled by the source language compiler in the ordinary way. Generally speaking, program builders allow a "transparent" rather than (as conventionally) an "opaque" higher-level language to be superimposed on a lower-level language— "transparent" in the sense that the lower-language program remains visible to the user. We discuss the relative merits of program builders versus conventional higher-level languages in the context of an implemented case study: The Linda Program Builder. The LPB offers many of the advantages of very-high-level languages and others as well: (1) It can provide a more flexible and graphical user environment. (2) It can be customized more easily — each site gets high-level operations that it needs, and is not burdened with ones it does not need. (3) It is a good environment in which to try out new constructs. (4) In the LPB, high-level constructs augment rather than supercede highly flexible and general low-level constructs.

*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.

1 Introduction

"Program builders" are text editors augumented with language and programming-methodology specific functionality; they build programs in a conventional source language incrementally, under the user's control. Programs built in this fashion are eventually compiled by the source language compiler in the ordinary way. The Cornell Program Synthesizer [RT89a] is a classic example, although the systems that are the main focus here have broader and somewhat different goals. Generally speaking, program builders allow a "transparent" rather than (as conventionally) an "opaque" higher-level language to be superimposed on a lower-level language— "transparent" in the sense that the lower-language program remains visible to the user. We discuss the relative merits of program builders versus conventional higher-level language in the context of an implemented case study: The Linda Program Builder.

Linda is a coordination language that has been described at length. It provides a logically-shared associative object memory inhabited by passive and active "tuples." [CG89] is a representative paper.

2 Program Builders

High or very high level languages are the research community's traditional response to the growth in knowledge about programming methodology. A programming method has been developed, for example, in which programs maintain no mutable state objects and rely heavily on functions, including higher-order ones; this style can be sustained in conventional languages, but functional languages are designed to provide maximum support for the method, to (in a sense) enshrine it. Logic programming languages represent a related response to programming methods that center on theorem proving. The failed type-safe "verifiable" neo-Pascals of the 70's (for example Euclid [PHL+83]) represented an attempt to capture a method based on rigorous application of Hoare-type verification, strong typing and so on. There are a number of other examples.

An alternative approach to capturing a programming method is to design a *program builder* rather than a new programming language. The program builder is a kind of text editor, extended with certain language-recognition features; it generates program text following the user's instructions; the program text is eventually passed to an ordinary compiler.

A program builder can enforce a methodology in more or less the same way as a programming language, provided the underlying language is powerful enough to support the methodology. For example, if we need a version of Pascal in which goto's and free type unions are not allowed, predicates describing loop invariants are required and so on, we can write a program builder that generates Pascal programs following these constraints. C++ was implemented originally as a source-to-source translator, but might also have been implemented as a program builder, creating C source code incrementally as the programmer developed his object-oriented application.

A program builder may be disadvantageous in terms of efficiency; a full-blown high-level language can in principle lead to more efficient code generation. Further, program builders require that the programmer

remain fully conversant in the base language; they superimpose a *transparent* (not an opaque) overlay on the base language.

But the program-builder approach has advantages as well, in terms of the way in which high-level languages are supported and the kinds of constructs that are provided.

A *language* must be fairly static: neither implementors nor programmers can tolerate frequent or radical changes. *A program builder can be far more flexible.* It can be updated almost at whim, without any change to the base language specification or compiler. A language must be fairly broad-based: highly specialized languages rarely attract adequate applications and support communities. *A program builder can be highly specialized* or customized; any site can support the features that are important to its particular application mix without foisting a similar burden (in terms of more complicated language manuals, source programs and implementations) on other sites.

Further, a program builder can support *templates* more effectively than a conventional language. A template is a program stucture, an organizing framework for a particular kind of program. A conceptual point involving language design and a pragmatic one having to do with language support are both at issue here.

Conceptually, higher-level languages have generally offered high-level *operations*, not high-level program structures. They have offered unification, higher-order functions, complex relational queries—not high-level structures or frameworks for entire applications; we might say that most high-level languages are high-level in a "micro" versus a "macro" sense. Program templates do exist in some languages, but they tend to be generic and strictly syntactic: for example the program structures in Cobol or Pascal, or the class definition structure in Smalltalk. Pragmatically, this tendency may be related to the idea of a language as opposed to a program builder. A language is a class of character strings, and it's often unnatural to specify elaborate multi-part frameworks in such a setting. A program builder can lay down the framework itself, automatically, and guide the user through the filling-in process; hence templates become pragmatically more attractive.

In our domain, templates are particularly important. A great deal has been learned about building parallel applications using C-Linda, and a well-defined programming methodology exists [CG90]. Much of the methodology takes the form of templates. For example, many Linda programs are *master-worker* programs; a master process generates tasks and collects results, and a pool of worker processes performs tasks. Master and worker each have certain functions to perform, and in most cases they use one of a small range of possible strategies in carrying them out. Hence, the Linda Program Builder (LPB) [ACG91] supports a *master-worker template* that would have been (as we discuss) both conceptually and pragmatically inappropriate within Linda itself. *Piranha* programs are another example [BCG+91]: they are similar to master-worker programs, but the workers (called Piranhas) must be free to join and leave the computation dynamically, the master performs several specialized functions, and a "retreat" function must be supplied to clean things up when a Piranha leaves the computation. Again, a Piranha template is a natural part of the "language" supported by the LPB.

Program builders naturally support transparent overlays, which may be used hierarchically. We present two examples in the section below.

3 The LPB

The LPB is a menu-driven system built on top of Epoch[1] running under X-windows. It has many features that are relevant from a CASE perspective, but we concentrate on a few of the higher-level operations that illustrate the LPB as an alternative to conventional higher-level languages.

3.1 Templates

The user identifies a particular programming paradigm, and the LPB offers the appropriate template. An incremental approach is followed; buttons may be expanded into code segments. The programmer may choose to leave the framework whenever desired and return later if necessary — he has the full flexibility of the text editor available. Various stages in the expansion of the template will require further input, which is solicited through menus or input windows. Certain buttons may be dependent on others, requiring a specific order of expansion.

Users remain free to enter code or comments anywhere within the partially constructed program.

Figure 1 shows a segment out of a partially expanded master-worker template.

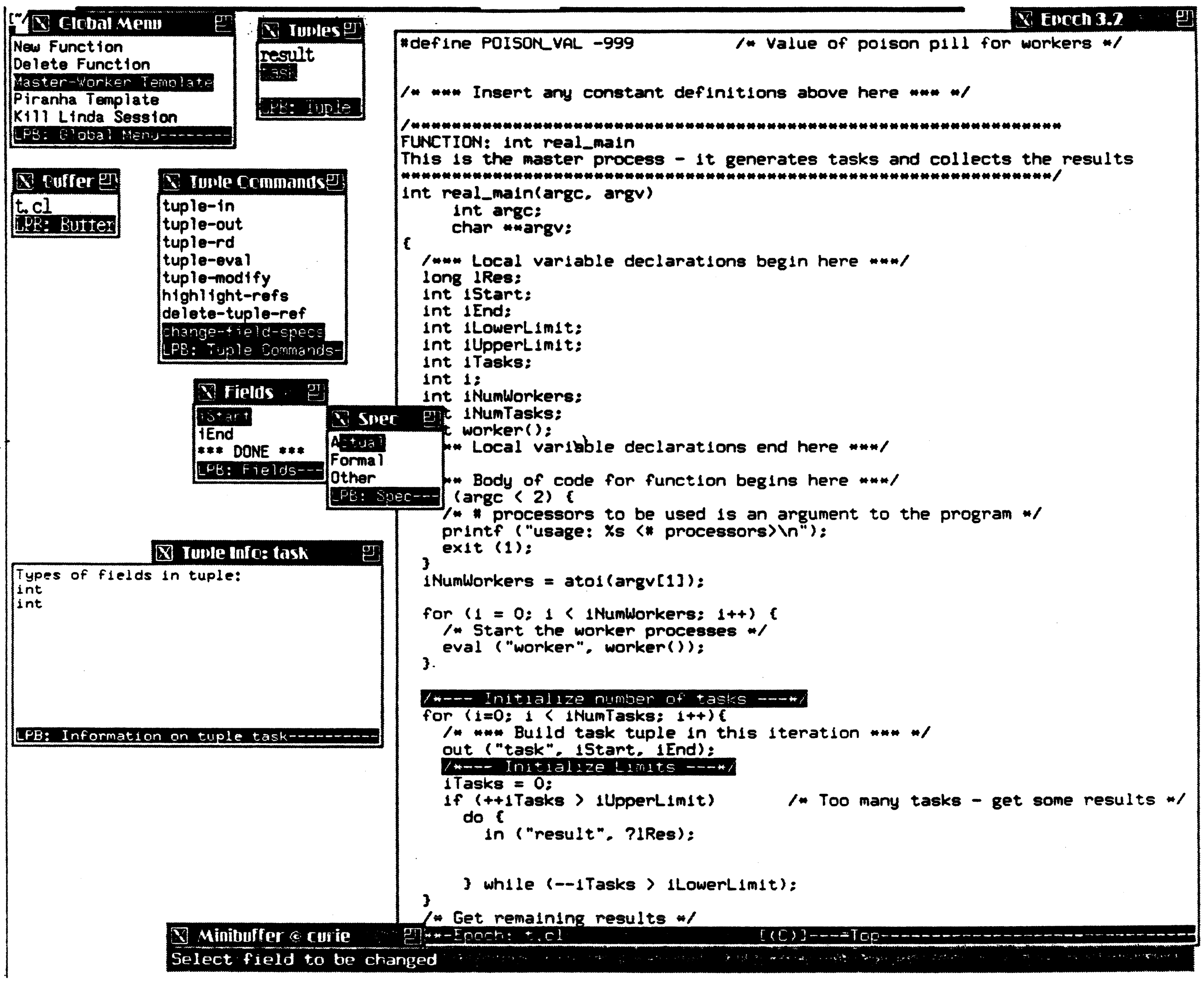
Templates may in principle be hierarchical. Consider a very simple, very-high-level template:

```
return maximum/apply f(...) to all [...]
```

This means "return the maximum/minimum of the results yielded by applying *f* to ...". It may generate a master-worker template in turn. Let's say this template is being used to generate a database search program of the sort described in [CG90]: return the best match between a target gene sequence and any sequence in a genetic database. This very-high-level template creates a master-worker template — guides the user, in other words, to the selection of this particular program structure. This master-worker template generates a C-Linda program in turn. When the user needs to tune and optimize his program, he returns to the master-worker template level (perhaps to substitute a watermarked for an "ordinary" task bag). The very-high-level template in turn can keep track of the class of master-worker templates to which it is relevant, and offer the user appropriate choices in future. Multi-level interactions may occur within the hierarchy.

The *process trellis* [Fac90][CG90] is another very-high-level template, for realtime data fusion; it too generates a master-worker template. This is the sort of structure that's far too specialized and hard-to-implement for a programming language; but it can be a "language feature" of sorts within a program builder.

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



```
Types of fields in tuple:
int
int

LPB: Information on tuple task---------
```

```
#define POISON_VAL -999          /* Value of poison pill for workers */


/* *** Insert any constant definitions above here *** */

/*****************************************************************
FUNCTION: int real_main
This is the master process - it generates tasks and collects the results
*****************************************************************/
int real_main(argc, argv)
     int argc;
     char **argv;
{
  /*** Local variable declarations begin here ***/
  long lRes;
  int iStart;
  int iEnd;
  int iLowerLimit;
  int iUpperLimit;
  int iTasks;
  int i;
  int iNumWorkers;
  t iNumTasks;
  t worker();
  ** Local variable declarations end here ***/

  ** Body of code for function begins here ***/
  (argc < 2) {
    /* # processors to be used is an argument to the program */
    printf ("usage: %s <# processors>\n");
    exit (1);
  }
  iNumWorkers = atoi(argv[1]);

  for (i = 0; i < iNumWorkers; i++) {
    /* Start the worker processes */
    eval ("worker", worker());
  }

  /*---- Initialize number of tasks ---*/
  for (i=0; i < iNumTasks; i++){
    /* *** Build task tuple in this iteration *** */
    out ("task", iStart, iEnd);
    /*---- Initialize Limits ---*/
    iTasks = 0;
    if (++iTasks > iUpperLimit)        /* Too many tasks - get some results */
      do {
        in ("result", ?lRes);


      } while (--iTasks > iLowerLimit);
  }
  /* Get remaining results */
**-Epoch: t.cl                  [(C)]---Top--------------------------------
Select field to be changed
```

Figure 1: Partially expanded master-worker template

3.2 Distributed data structures

Linda programs generally make use of distributed data structures such as distributed arrays, task bags, shared variables and so on. Many of these can be anticipated by the coordination-framework templates discussed above. Structures such as task bags, watermarked bags or distributed queues, to name a few, are often incorporated into the choices presented during the construction of a program through a template. But when the case arises where a programmer needs to specify a particular data structure outside of a specific template, the LPB will provide the necessary support.

For example, the LPB has menu options to support creating and manipulating shared variables and counters in tuple space. This includes special cases such as incrementing or decrementing a counter. Counting semaphores are supported in a similar manner. The labels of these variables appear in the tuple menu. Picking a tuple of one of these types will cause the commands menu to change accordingly, and the information window to display known information on the tuple.

Distributed queues of various kinds are required for many 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 in tuple space. The LPB provides a complete set of menu functions to create and manipulate queues. Upon selection of a create-queue command, a popup menu will offer choices on the various models available. 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. All tuple operations, declarations, and additional code will be automatically inserted at the appropriate places.

3.3 Other high-level operations

Certain operations ought to be supported, but not at the level of a full program template. The LPB supports high-level operations which can be expanded and then abstracted back again (buttons, on the other hand, can only be expanded).

The abstraction feature is a powerful tool that furnishes the user with a novel approach to viewing and constructing programs. For top-down program construction and stepwise refinement, it presents a high-level view of program structure that can not only be expanded downwards, but can then be abstracted back up again to a conceptually more appealing higher-level format for viewing purposes. This allows the programmer to concentrate on hierarchical program construction at a high level, and deal with "blocks" of code represented by abstractions which together form a larger program. (This is somewhat similar to the Cedar [SZBH86] approach in its Tioga structured text-editor. Tioga treats documents in a tree-structured manner where each node is a paragraph or a statement. This hierarchical node structure allows the concealing of detail to present a conceptually higher-level view, much as in the LPB.)

We give an example in Section 4.

3.4 The program database

Every tuple, function, abstraction, higher level function, and all other crucial components of the program are entered into a database as they are used. The database keeps information on a tuple's label, on the variables used in its fields, the status of each of the fields, information on the nature of the tuple and its use, and a record of all the places where references to the tuple exist.

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.

This database is the backbone of the LPB, maintaining all the information necessary to perform higher level operations and provide user support, eliminating the need for much memory-work and reducing keystrokes. The stored data supports automation that can prevent errors. It also allows information to be passed on to the compiler and visualizer. The compiler can make use of the information to further optimize code, and the visualizer can enhance its display by organizing its information better. These topics are beyond the scope of this paper.

4 The LPB as an alternative to higher-level languages

In assessing the LPB and comparing it to alternatives, the main question is: How do we satisfy the demands of programmers who want high level constructs for ease of programming, for maintainability, and for quick software development?

The conventional approaches are to add very-high-level features to existing languages, or to implement completely new languages. The LPB presents a clear alternative. In the LPB alternative, the user sees similar advantages in terms of strong support for high-level models and constructs. But he doesn't pay the traditional price in terms of a language that is highly complex (e.g. Ada [Bar80] or Common Lisp [Jr84]) or restrictive, inflexible or narrow in its range of applicability (terms in which logic and functional languages, for example, are often described). The program builder may be very high-level, special-purpose, idiosyncratic — but it may also be changed easily, customized or evolved readily — and simply bypassed when the very-high-level constructs it supports aren't the right ones.

An examination of two different examples in the context of the LPB will illustrate these points. Consider distributed queues. When the user wants to create a queue of tuples, he picks the appropriate menu option. This causes a new popup menu to appear which offers a choice of different queue models. Once a model has been identified, the user is prompted for some initialization data and the appropriate code is automatically created and inserted. Once the queue is initialized, further high-level operations for queue manipulation become available: adding to the tail or removing from the head of a queue. The LPB's support has two significant consequences. First, the task of removing the element from the queue is simplified, and the programmer specifies it at a "high" conceptual level. Second, the compiler can make use of this information about the programmer's intent to optimize the code generated ultimately from the C-Linda version of the program.

But — if the programmer knows that he is removing an element from the queue, and the compiler makes use of this information, would it not be easier simply to add these higher level features directly to the base language? Why not extend C-Linda to offer high-level queue operations, or throw Linda out and replace it with something "higher level"?

There are several reasons why not. Linda operations were meant to be simple and powerful. The simplicity of Linda as it stands gives users a flexible base from which to construct *any* desired operations. (Experienced programmers are well aware of the fact that a high-level language can never second-guess their requirements entirely. Adding new operations in addition to the old yields an over-complicated language; allowing new operations to supercede the old yields an insufficiently powerful one. The LPB consigns high level operations to the LPB level, general purpose operations to the language level. Note that complexity is nowhere near as damaging in software like the LPB as it is in the language itself. If some aspect of the LPB is of no use in some context, programmers don't have to learn it and *their implementations don't need to support it.* And Linda itself remains a simple *lingua franca* for the exchange of source code.

There is a second problem associated with higher-level languages of the kind described above. Languages should be fairly static. Neither programmers nor implementors can tolerate rapid or radical change. But programming *methodology* must evolve. In a young field like parallel programming, change can be rapid. Program builders such as the LPB provide a convenient alternative solution. In principle we would like to revise our language as methodologies become clear to us, without losing compatibility with the rest of the user community. At the same time, we don't want a galaxy of constructs that we do not need. If we produce mainly numerical scientific applications, the very-high-level constructs that are valuable to (say) graphics or database people may be of no use to us, and we don't want to bother with them: don't want them complicating our language manuals; don't want them complicating our compilers. Still, we want to *understand*, interface to, and execute graphics and database applications if need be. All this would appear to be impossible within the programming language framework, but easily achieved under the program builder approach.

The second LPB example will address another aspect of this argument. Evidence suggests that there are cases where a Linda programmer needs to **in** one of a selection of tuples. Any one of the selection would do. We refer to this as an **or-in**, i.e. the program will **in** one tuple *or* another tuple *or* another one, and so on. This is an operation which Linda itself does not provide, and yet it is used often enough to be worth a debate over whether it should be incorporated into the language. Adding it to the language would involve a significant programming effort, and modification to the various kernels. The LPB is a convenient forum in which to test this construct.

The LPB implements the **or-in** function as a higher-level operation. If the user selects the menu option for an **or-in**, a menu pops up with a list of the tuples that are known to the database. The user is free to select which of these tuples will constitute the **or-in**. What gets inserted into the code is the higher-level operation. It appears to be a regular program construct, but the relevant lines in the code are underlined (Figure 2). The underlining indicates that it is a higher-level operation. Expanding this abstraction will indicate to the user how this is implemented in Linda code. The **or-in** becomes an **in** of a bit vector to check which tuples may be available. This is followed by a conditional which checks which bit is on, and based on that, reads in the appropriate tuple. The bit vector has to be generated whenever one of the tuples of the **or-in** is used in an **out** or **eval**. Thus, the expansion causes all relevant references to those tuples in all open modules to be followed by a new **out** which puts out a bit vector with the bit corresponding to the **out**'ed tuple turned on (Figure 3). If the cursor is placed on the main section of the **or-in** expansion, 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.

We have thus implemented a proposed language addition in the LPB, and enabled programmers to use and test it before it has actually been added to the language. This allows the operation to run through a trial period before the major task of adding it to the language is undertaken. Its usage patterns and usefulness need to be investigated before we commit ourselves to a language change, and the LPB is the testing medium.

There is one further advantage that the LPB offers over conventional languages with added features. The graphical user interface provides a level of user-friendliness that the base language cannot match. In combination with the abstraction facility, this becomes a powerful tool.

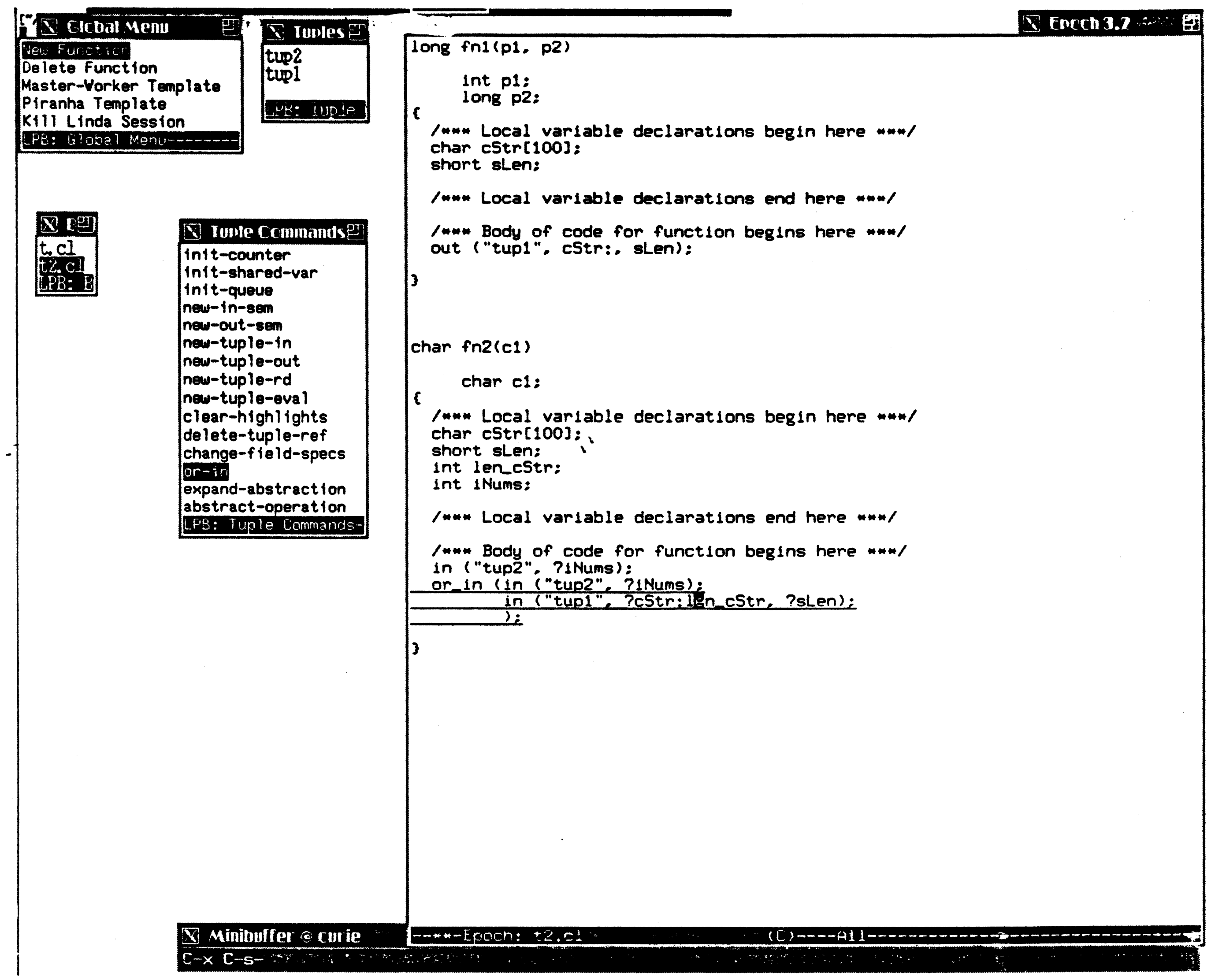
The disadvantage of the LPB in this context is efficiency. Wiring an operation into the base language allows the compiler to optimize its support. While the LPB does not provide that level of optimization, it does provide the compiler with semantic information that can lead to a different kind of optimization, but that is beyond the scope of this paper. In combination with the above-mentioned advantages, this amounts to a strong offsetting argument against the efficiency disadvantage.

5 Related work

The LPB's most important template-based structure editor predecessor is the Cornell Program Synthesizer [RT89b]. Unlike the synthesizer, however, the LPB does not enforce a rigid framework. Instead, the LPB captures methodologies and supports them, without imposing a strategy. What the LPB produces is source code, and the programmer is free to ignore or modify this as desired, a flexibility that is vitally necessary to any expert programmer.

Extensible parallel programming environments such as SIGMACS [SG91] generate a program database during compile time that can be used in 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 as well as syntactic information on the programs being developed. This information is used in guiding program development, for checking consistency, for documentation purposes, for providing optimizing information to the compiler [CG91], for benchmarking utilities to visualize performance in the spirit of [HE91], and for enhancing graphical monitoring.

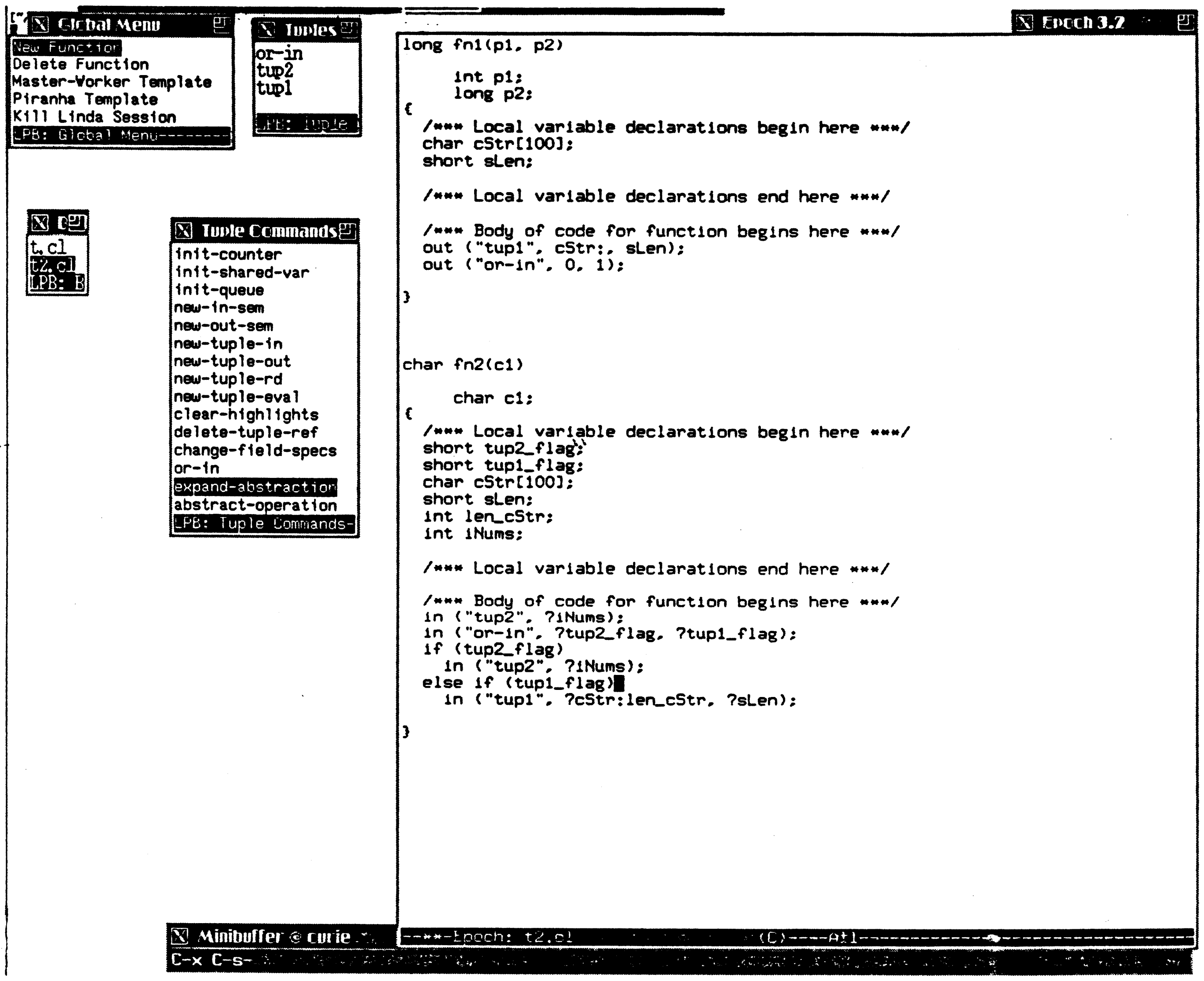
There is currently much research effort in visualizing the dynamic behavior of parallel programs. [KC91] is a good example. Since the LPB can convey semantic information to a graphical monitoring tool [BC90], programmers can visualize dynamic information at a higher abstraction level than would otherwise be possible.



```
long fn1(p1, p2)

     int p1;
     long p2;
{
  /*** Local variable declarations begin here ***/
  char cStr[100];
  short sLen;

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

  /*** Body of code for function begins here ***/
  out ("tup1", cStr:, sLen);

}


char fn2(c1)

     char c1;
{
  /*** Local variable declarations begin here ***/
  char cStr[100];
  short sLen;
  int len_cStr;
  int iNums;

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

  /*** Body of code for function begins here ***/
  in ("tup2", ?iNums);
  or_in (in ("tup2", ?iNums);
         in ("tup1", ?cStr:len_cStr, ?sLen);
         );

}
```

```
----Epoch: t2.cl              (C)----All--------------
C-x C-s-
```

Figure 2: The or-in abstraction



```
long fn1(p1, p2)

     int p1;
     long p2;
{
  /*** Local variable declarations begin here ***/
  char cStr[100];
  short sLen;

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

  /*** Body of code for function begins here ***/
  out ("tup1", cStr:, sLen);
  out ("or-in", 0, 1);

}



char fn2(c1)

     char c1;
{
  /*** Local variable declarations begin here ***/
  short tup2_flag;
  short tup1_flag;
  char cStr[100];
  short sLen;
  int len_cStr;
  int iNums;

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

  /*** Body of code for function begins here ***/
  in ("tup2", ?iNums);
  in ("or-in", ?tup2_flag, ?tup1_flag);
  if (tup2_flag)
    in ("tup2", ?iNums);
  else if (tup1_flag)
    in ("tup1", ?cStr:len_cStr, ?sLen);

}
```

Figure 3: Expanded **or-in**

6 Conclusions

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

References

[ACG91] Shakil Ahmed, Nicholas Carriero, and David Gelernter. The Linda Program Builder. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990) (invited paper)*. Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[Bar80] J.G.P. Barnes. An Overview of Ada. *Software Practice and Experience*, pages 851–887, 10 1980.

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

[BCG$^+$91] R. Bjornson, N. Carriero, D. Gelernter, D. Kaminsky, T. Mattson, and A. Sherman. Experience With Linda. Research Report 866, Yale University Department of Computer Science, 1991.

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

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. The MIT Press, 1990.

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

[Fac90] Michael Factor. *The Process Trellis Software Architecture for Parallel, Real-time Monitors*. PhD thesis, Yale University, New Haven, Connecticut, 1990. Department of Computer Science. In progress.

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

[Jr84] Guy Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.

[KC91] James A. Kohl and Thomas 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.

[PHL$^+$83] G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London. Notes on the Design of Euclid. In *Programming Languages: A Grand Tour*. Computer Science Press, 1983.

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

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

[SG91] Bruce Shei and Dennis 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.

[SZBH86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, pages 419–490, October 1986.