Common Lisp as an Embedded Extension Language

A large part of HP PE/SolidDesigner's user interface is written in Common Lisp. Common Lisp is also used as a user-accessible extension language.

by Jens Kilian and Heinz-Peter Arndt

HP's PE/ME10 and PE/ME30 CAD systems contain an extension language based on the macro expansion paradigm. The user's input (commands and data) is separated into single tokens, each of which denotes a command, function, variable, macro name, number, string, operator, or other syntactic element. Commands, functions, and arithmetical expressions are evaluated by the language interpreter. Each macro name is associated with a macro definition, which is another token sequence (either predefined by the system or defined by the user). When the language interpreter encounters a macro name, it substitutes the corresponding token sequence (this process is called *expanding* the macro) and continues with the first token of the expansion.

Macro expansion languages are easy to implement and have been used in many applications where one would hardly expect to find an embedded language. For example, the $\TeX$ typesetting system contains a macro interpreter.

The HP PE/ME10 and PE/ME30 macro language includes powerful control constructs (such as IF/THEN/ELSE and LOOP/EXIT_IF/END_LOOP), local variables, and a mechanism for passing parameters to a macro when it is being expanded. These constructs make it possible to solve general programming problems. Because the HP PE/ME10 and PE/ME30 macro language is interpreted, programs can be developed in an interactive fashion and modifications can immediately be tried out. However, the resulting program is slower than a program written in a compiled language like C. HP PE/ME10 and PE/ME30 macros can be compiled to an intermediate form which executes faster than the pure interpreted version, but which is still slower than an equivalent C program.

One disadvantage of the HP PE/ME10 and PE/ME30 macro language is that it is nonstandard. No other application uses the same language, and programs written in it have to be ported when the user switches to another CAD system.

Common Lisp

Common Lisp was chosen as an extension language for HP PE/SolidDesigner because it is nonproprietary and widely used.

Surprising as it may be, Lisp is the second oldest high-level programming language still in common use. The only older one is FORTRAN. Lisp is to researchers in artificial intelligence what FORTRAN is to scientists and engineers.

Lisp was invented by John McCarthy in 1956 during the Dartmouth Summer Research Project on Artificial Intelligence. The first commonly used dialect was Lisp 1.5, but unlike FORTRAN (or any other imperative language) Lisp is so easy to modify and extend that over time it acquired countless different dialects. For a long time, most Lisp systems belonged to one of two major families, Interlisp and MacLisp, but still differed in details. In 1981, discussions about a common Lisp language were begun. The goal was to define a core language to be used as a base for future Lisp systems. In 1984, the release of Common Lisp: The Language[1] provided a first reference for the new language. An ANSI Technical Committee (X3J13) began to work on a formal standardization in 1985 and delivered a draft standard for Common Lisp in April 1992. This draft standard includes object-oriented programming features (the Common Lisp Object System, or CLOS). For a more detailed account on the evolution of Lisp, see McCarthy[2] and Steele and Gabriel.[3]

HCL, the implementation of Common Lisp used in HP PE/SolidDesigner, is derived from Austin Kyoto Common Lisp (itself descended from Kyoto Common Lisp). It corresponds to the version of the language described in reference 1, but already incorporates some of the extensions from reference 4 and the draft standard.

Applications of Extension Languages

Adding extension languages to large application programs has become a standard practice. It provides many advantages, some of which may be not as obvious as others. For the normal user of a system, an embedded programming language makes it possible to automate repetitive or tedious tasks. An inexperienced user can set it up as a simple record/playback mechanism, while "power users" can use it to create additional functionality. If the extension language has ties to the application's user interface, user-defined functionality can be integrated as if it were part of the original application.

If the application provides an API for adding extensions on a lower level, the extension language can itself be extended. This enables makers of value-added software to integrate their products seamlessly into the main application. As an example, the HP PE/SheetAdvisor application has been implemented within HP PE/ME30, offering a user interface consistent with the rest of the program.

As a final step, portions of the application can themselves be implemented in the embedded language. An example would be the popular GNU Emacs text editor, a large part of which is written in its embedded Lisp dialect.

A large part of HP PE/SolidDesigner, too, is written in its own extension language—about 30 percent at the time of writing. Most of this 30 percent is in HP PE/SolidDesigner's user interface.

Lisp in HP PE/SolidDesigner

Fig. 1 shows the major components of HP PE/SolidDesigner. The Lisp subsystem is at the very core, together with the Frame (operating system interface) and DSM (data structure manager, see article, page 51) modules. All other components including Frame and DSM are embedded into the Lisp subsystem. This indicates that each component provides an interface through which its operations can be accessed by Lisp programs.

The introduction of new functionality into HP PE/SolidDesigner is usually done in the following steps:

- Implement new data structures and operations in C++
- Add Lisp primitives (C++ functions callable from Lisp) for accessing the new operations
- Add action routines to implement new user-visible commands, using the Lisp interface to carry out the actual operations
- Add menus, dialog boxes, or other graphical user interface objects to access the new commands.

As long as the Lisp interface—the primitive functions—is agreed to in advance, this process can be parallelized. A user interface specialist can work on the action routines and menus, calling dummy versions of the interface functions.

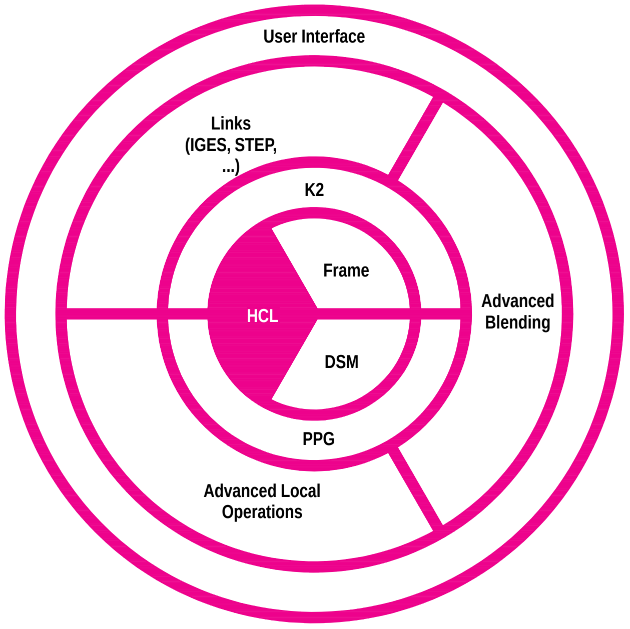

Fig. 1. HP PE/SolidDesigner system architecture. HCL is the Common Lisp subsystem. All components including Frame (operating system interface) and DSM (data structure manager) have interfaces to Lisp. K2 is the solid modeling kernel. PPG is the planar profile generator.

The article on page 14 describes, from a user interface developer's perspective, how action routines are written and how menus and dialogs are created. The mechanisms used there are not part of the Common Lisp standard but are extensions provided by the HCL dialect.

Action Routines

Action routines implement the commands that a user types or issues via user interface elements to HP PE/SolidDesigner. Commands are identified by their names, which are Lisp symbols evaluated in a special manner (similar to the SYMBOL-MACROLET facility in the Common Lisp Object System). Each action routine is actually an interpreter for a small language, similar in syntax to the command language used in HP PE/ME10 and PE/ME30. Like HP PE/ME10 and PE/ME30 commands, action routines can be described by their syntax diagrams. Fig. 2 contains the syntax diagram for a simplified version of HP PE/SolidDesigner's exit command. Below the syntax diagram is a state transition graph which shows how the command will be processed.

The definition of an action routine corresponds closely to its syntax diagram. The defining Lisp expression, when evaluated, generates a normal Lisp function that will traverse the transition graph of the state machine when the action routine is run. For example, the following is an action routine corresponding to the syntax diagram of Fig. 2:

```
(defaction simple_exit

 (flag)      ; local variable

  (; state descriptions

   (start nil
        "Terminate PE/SolidDesigner?"
        nil

        (:yes      (setq flag t)   answer-yes end)
        (:no       (setq flag nil) answer-no  end)
        (otherwise (display_error "Enter either :YES or :NO.")  nil start))

   (end (do-it)
      nil
      nil))

  (; local functions

   (do-it ()
        (when flag
          (quit)))))
```

As can be seen in this example, an action routine can have local variables and functions. Local variables serve to carry information from state to state. Local functions can reduce the amount of code present in the state descriptions, enhancing readability.

When HCL translates this action routine definition, it produces a Lisp function which, when run, traverses the state transition graph shown in Fig. 2b. If a state description contains a prompt string, as in the start state in the example, the translator automatically adds code for issuing the prompt and reading user input. Effectively, the translator converts the simple syntax diagram into the more detailed form.

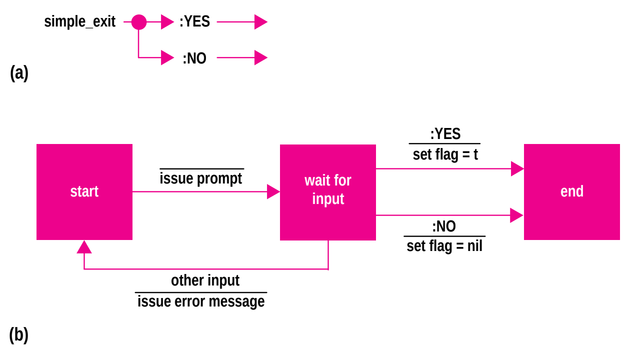

Fig. 2. (a) Simplified syntax of the exit command. (b) State transition diagram for the exit command.

For the example action routine, the translator produces a Lisp function definition much like the following:

```
;; Declarations of some external functions, for more efficient calling

(proclaim '(function get-parameter  (t t) t))
(proclaim '(function match-otherwise (t) t))
(proclaim '(function trigger-action-state-transition-event (t &optional t)
   t))

;; Transformed action routine

(defun simple_exit (&rest argument-list &aux input)

 (let (flag)              ;; local variable

   (labels ((do-it ()     ;; local function
              (when flag
                (quit))))

        (block nil
          (tagbody

           ;; label for state "start"
           1

           ;; prompting in state "start"
           (setq input (get-parameter argument-list "Terminate HP PE/
           SolidDesigner?"))

           ;; pattern matching in state "start"
           (cond ((equal input :yes)

                 (setq flag t)   ;; action taken
                 (trigger-action-state-transition-event  'answer-yes)

                 (go 0))        ;; transition to "end" state

                ((equal input :no)

                 (setq flag nil) ;; action taken
                 (trigger-action-state-transition-event  'answer-no)

                 (go 0))        ;; transition to "end" state

                ((match-otherwise input)

                 (display_error "Enter either :YES or :NO.")

                 (go 1)))       ;; transition to "start" state

           ;; label for state "end"
           0

          ;; initial action for state "end"
          (do-it)

             ;; exit from action routine
             (return))))))
```

Transitions in the state machine are transformed into goto statements within the function's body. The conditional construct cond represents decisions, like the three-way branch in state start. Before each state transition, the code can trigger an external event to enable graphical feedback in menus or dialogs.

The actual translation is somewhat more complicated because errors and other exceptional events must be taken into account. The translator also adds code to support debugging and profiling of an action routine. This code is stripped out when building a production version of HP PE/SolidDesigner.

Compiling Lisp Programs

It has often been said that Lisp is inherently slow and cannot be applied to application programming (one common joke is that the language's name is an acronym for "Large and Incredibly Slow Programs"). This is not true. Even very early versions of Lisp had compilers.[3] Lisp systems have even beaten FORTRAN running on the same machine in terms of numerical performance.

In HCL, the Lisp compiler takes a Common Lisp program and translates it into an intermediate C++ program, which is then compiled by the same C++ compiler that is used to translate the nonLisp components of HP PE/SolidDesigner. This approach has several advantages:

- The Lisp compiler can be kept small and simple (only 12,500 noncomment lines of code, less than 5% of the total amount of Lisp code)
- The Lisp compiler does not need to be retargeted when porting to a different machine architecture
- The Lisp compiler does not need to fully optimize the generated code; this task can be left to the C++ compiler
- The generated code is fully call and link compatible with the rest of the system
- The generated code can be converted to a shared library and dynamically loaded into a running HP PE/SolidDesigner.

The Lisp compiler is itself written in Lisp. Bootstrapping a new compiler version is easy because an interpreter is available.

The calling conventions for compiled Lisp functions are such that interpreted and compiled functions can transparently call each other. This allows keeping most of the Lisp code in compiled form, even when using the interpreter to develop new programs.

Continuing the above example, here is the C++ code that the Lisp compiler produces for the simplified translated action routine (reformatted for better readability):

```
// Header file declaring standard Lisp data structures and functions
// (for example, LOBJP is the type of a generic pointer-to-Lisp-object)

#include <cmpinclude.h>

// Declarations for the compiled code (normally written to a separate file,
// included here for clarity)

static void L1(...);                    // Functions defined in this file
static void L2(LOBJP*);
static char *Cstart;                 // Data for communication with the Lisp
                                     // loader

static int Csize;
static LOBJP Cdata;
static LOBJP VV[14];                 // Run-time Lisp objects

static void LnkT13() ;               // Links to external Lisp functions
static void (*Lnk13)() = LnkT13;     // (see below for an explanation)
static void LnkT11() ;
static void (*Lnk11)() = LnkT11;
static LOBJP LnkTLI10(LOBJP ) ;
static LOBJP (*LnkLI10)(LOBJP ) = LnkTLI10;
static LOBJP LnkTLI9(int narg, ...) ;
static LOBJP (*LnkLI9)(int narg, ...) = LnkTLI9;
static LOBJP LnkTLI8(LOBJP , LOBJP ) ;
static LOBJP (*LnkLI8)(LOBJP , LOBJP ) = LnkTLI8;

// Initialization function, called immediately after the file is loaded

void example_initialize(char *start, int size, LOBJP data)
{
  // Reserve space on the Lisp stack

  register LOBJP* base=vs_top;
  register LOBJP* sup=base+0;
  vs_top=sup;
  vs_check;

  // Store data supplied by the loader, including Lisp objects
  // that were extracted from the original source code and that
  // will be needed at run-time (e.g., strings and symbols).

  Cstart=start;
  Csize=size;
  Cdata=data;
  set_VV_data(VV,14,data,start,size);

  // Link the compiled function “L1” to the Lisp symbol stored in VV[6],
  // which is ”SIMPLE_EXIT”.

 MFnew(VV[6],(void(*)())L1,data);//

  // Restore Lisp stack

  vs_top=vs_base_mod=base;
}

// Compiled function SIMPLE_EXIT

static void L1(...)
{
  register LOBJP*base=vs_base;       // Reserve space on the Lisp stack
  register LOBJP*sup=base+3;
  vs_check;

  { LOBJP V1;                        // Fetch ARGUMENT-LIST from the Lisp
                                     // stack
    vs_top[0]=Cnil;
    { LOBJP *p=vs_top;
      for(;p>vs_base;p--)p[-1]=MMcons(p[-1],p[0]);
    }
    V1=(base[0]);
    vs_top=sup;
    { LOBJP V2;                      // Set up variables INPUT and FLAG
      V2= Cnil;
      base[1]= Cnil;
    T3:;                             // Label “1” in TAGBODY
      V2= (*(LnkLI8))((V1),VV[0]);  // (GET-PARAMETER ARGUMENT-LIST “...”)
      if(!(equal((V2),VV[1]))){      // First clause of COND construct
        goto T8;
      }
     base[1]= Ct                     // (SETQ FLAG T)
      (void)((*(LnkLI9))(1,VV[2])); // (TRIGGER-...-EVENT ’ANSWER-YES)
      goto T4;                       // (GO 0)
    T8:;                             // Second clause of COND construct
      if(!(equal((V2),VV[3]))){
        goto T14;
      }
      base[1]= Cnil;                 // (SETQ FLAG NIL)
      (void)((*(LnkLI9))(1,VV[4]));  // (TRIGGER-...-EVENT ’ANSWER-NO)
      goto T4;                       // (GO 0)
    T14:;                            // Third clause of COND construct
      if(((*(LnkLI10))((V2)))==Cnil){
        goto T4;
      }
      base[2]= VV[5];                // (DISPLAY-ERROR “...”)
      vs_top=(vs_base=base+2)+1;
      (void) (*Lnk11)();
      vs_top=sup;
      goto T3;                       // (GO 1)
    T4:;                             // Label “0” in TAGBODY
      vs_base=vs_top;                // Call (DO-IT), passing a pointer to
      L2(base);                      // the lexical variables of SIMPLE_EXIT
      vs_top=sup;
      base[2]= Cnil;                 // Return from SIMPLE_EXIT
      vs_top=(vs_base=base+2)+1;
      return;
     }
   }
 }

// Compiled local function DO-IT

static void L2(LOBJP*base0)
{
  register LOBJP*base=vs_base;       // Reserve space on the Lisp stack
  register LOBJP*sup=base+1;
  vs_check;
  vs_top=sup;
  if((base0[1])==Cnil){              // Condition: lexical variable FLAG
    goto T26;
  }
  vs_base=vs_top;                    // (QUIT)
  (void) (*Lnk13)();
  return;
 T26:;
  base[0]= Cnil;                     // Return from DO-IT
  vs_top=(vs_base=base+0)+1;
  return;
}

// Links to external functions. These functions are called indirectly, via
// C++ function pointers.  At the first call, the corresponding compiled
// function is looked up and stored in the function pointer, thus avoid-
// ing the Lisp calling overhead on subsequent calls.

static void LnkT13 ()
{ // QUIT; called via normal Lisp calling conventions

  call_or_link(VV[13],(int *)&Lnk13);
}
```

```
static void LnkT11()
{ // DISPLAY-ERROR; called via normal Lisp calling conventions

  call_or_link(VV[11],(int *)&Lnk11);
}

static LOBJP LnkTLI10(LOBJP arg0)
{ // MATCH-OTHERWISE; declared to take exactly one parameter, which
  // can be passed without using the Lisp stack.

  return(LOBJP)call_fproc(VV[10],(int*)&LnkLI10,1,arg0);
}

static LOBJP LnkTLI9(int narg, ...)
{ // TRIGGER-ACTION-STATE-TRANSITION-EVENT; declared to take one
  // fixed and one optional parameter, which can be passed without using
  // the Lisp stack.

  va_list ap;
  va_start(ap, narg);
  LOBJP result=(LOBJP)call_vproc(VV[9],(int*)&LnkLI9,narg,ap);
  va_end(ap);
  return result;
}

static LOBJP LnkTLI8(LOBJP arg0, LOBJP arg1)
{   // GET-PARAMETER; declared to take exactly two parameters, which
    // can be passed without using the Lisp stack.

  return(LOBJP)call_fproc(VV[8],(int*)&LnkLI8,2,arg0,arg1);
}
```

This example illustrates several important properties of compiled Lisp code. First, the C++ code still has to access Lisp data present in the original program; for example, it has to attach a compiled function to a Lisp symbol naming that function. Second, parameter passing for Lisp functions is usually done via a separate stack, but the overhead for this can be avoided by declaring external functions. In a similar way (not shown here), the overhead of using Lisp data structures for arithmetic can be avoided by introducing type declarations (which are not compulsory as in C++). Third, some Lisp constructs (e.g., lexical nesting of function definitions) have no direct C++ equivalent.

Compiling a Lisp program can have quite a dramatic impact on its performance. HP PE/SolidDesigner takes about one half to two minutes to start on an HP 9000 Series 700 workstation. If all the Lisp files are loaded in uncompiled form, start time increases to between one half and one hour.

Conclusion

A large part of HP PE/SolidDesigner is written in Common Lisp. To the developers, this approach offered a very flexible, interactive mode of programming. The finished programs can be compiled to eliminate the speed penalty for end users.

Common Lisp is also used as a user-accessible extension language for HP PE/SolidDesigner. It is a standardized, open programming language, not a proprietary one as in HP PE/ME10 and PE/ME30, and the developers of HP PE/SolidDesigner believe that this will prove to be an immense advantage.

References

1. G.L. Steele, Jr., S.E. Fahlman, R.P. Gabriel, D.A. Moon, and D.L. Weinreb, *Common Lisp: The Language*, Digital Press, 1984.
2. J. McCarthy, "History of LISP," in R.L. Wexelblat, ed., *History of Programming Languages*, ACM Monograph Series, Academic Press, 1981. (Final published version of the *Proceedings of the ACM SIGPLAN History of Programming Languages Conference*, Los Angeles, California, June 1978.)
3. G.L. Steele, Jr. and R.P. Gabriel, "The Evolution of Lisp," *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference*, Cambridge, Massachusetts, April 1993. pp. 231-270.
4. G.L. Steele, Jr., S.E. Fahlman, R.P. Gabriel, D.A. Moon, D.L. Weinreb, D.G. Bobrow, L.G. DeMichiel, S.E. Keene, G.Kiczales, C. Perdue, K.M. Pitman, R.C. Waters, and J.L. White, *Common Lisp: The Language, Second Edition*, Digital Press, 1990.