The CRISP Programming Language System

An Historical Overview

Jeffrey A. Barnett

Albuquerque Retired Citizens

September 20, 2009

2 Crunching Lisp

- Timeframe is early 1970's so resources are strictly limited.
- Needed efficient Lisp-like tools to support speech understanding research.
- Systems would be developed by group of people with different talents, interest, and knowledge.
- Big part of research was to determine system structure.
- Decision made to invent and implement Crunching Lisp, aka CRISP, a task undertaken by Doug Pintar and Jeff Barnett.

3 Historical Context

- Lisp ancestors: Lisp 2, IBM 360 Lisp 1.5
- (D)ARPA Speech Understanding Research (SUR) Program
- IBM 370/145 with 8MB virtual 1MB real memory
- IBM VM available as OS

Truth in Talk Warning

Everything described in this talk was implemented and used to develop speech understanding systems; Exception, the language itself was not. The final few slides will make clear what this means.

4 My Observations/Mantra

Language Luxury

If development environment does not provide control and data structures well beyond the hackers' needs, the entire architecture invention process will be consumed (badly) inventing cliches that should be provided by the programming language; the application tends to be ignored.

Working and Playing Well Together

Programming language systems that support multiple programmers must provide constructs that might not be necessary for one or a very few hackers. Concerns are divided into yours, mine, and ours–support tools must support this separation. One-liners are irrelevant.

5 Rest of This Talk

- **Namespaces**
- Data Structures
- Control Structures and Spaghetti
- Types and Incremental Compiles
- Storage Management
- Odds and Ends
- What Actually Got Implemented!

6 Namespaces

Not Quite a Definition

Named objects are in different namespaces if they can have the same name without conflict; They are in the same namespace if they must have different names. Namespaces are important to keep developers out of each other's hair. N.B. Namspaces are typically defined by reference contexts as well as where names are defined.

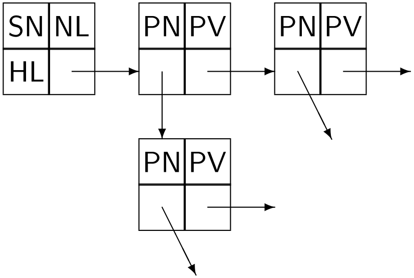
- Global objects: variables, functions, macros, types, etc.
- Property names
- Fields in a ntuple type
- Local variables viz a viz shadowing rules

7 Global Object Names

- Each global name had a symbol first and last name, e.g., cons$crisp or mark$syntax.
- Symbols used where a name was expected were converted to a name using guidance provided by the programmer:
    - Guidance was in the form a default last name and an ordered list of last names.
    - A symbol used in definition context got default last name.
    - A symbol, without local definition, used in reference context tried last names in order seeking a definition (use default if necessary).
- Compiler would visit all files seeking definitions before conversions.
- The current package system is usually cleaner but not always.

8 Properties on Symbols

- A symbol had a property list
- But it was really a tree
- Property "names" where ordered lists of symbols
- Typically one of those names matched the name of a global name pool



9 Rest of This Talk

- Namespaces
- **Data Structures**
- Control Structures and Spaghetti
- Types and Incremental Compiles
- Storage Management
- Odds and Ends
- What Actually Got Implemented!

10 Data Structures

- $\mathrm{node}_1, \ldots, \mathrm{node}_8$ made by $\mathrm{cons}_1, \ldots, \mathrm{cons}_8$
- Stack groups aka processes (discussed later)
- Structures (ntuples) named fields constant repeats
- Arrays (pointers and flat)
- Names (link, last, binding)
- Symbols (print name, prop list, name list, hash link)
- Numbers (small ints, int, real, complex) boxed and not
- Byte and half ints in structures and arrays
- Spaces (where various sorts of conses create things)

11 Rest of This Talk

- Namespaces
- Data Structures
- **Control Structures and Spaghetti**
- Types and Incremental Compiles
- Storage Management
- Odds and Ends
- What Actually Got Implemented!

12 Control Structures and Spaghetti

- Standard stuff:
  - Sequential: prog, etc.
  - Conditional: cond, if, select, case, typecase, etc.
  - Branch
  - Loop with asynchronous generators, FSM
  - Try
  - Pitch and catch
- Pseudo processes/coroutines (see next slide)

13 Stack Group AKA Process State

- A stack group contains the following,
- Two stacks:
    1. A stack of unboxed numbers
    2. A stack of pointers, special bindings, return points, catches
- Three pointers to other stack groups–two form trees
    1. The stack group that activated/started this one
    2. The context where special bindings are sought
    3. The place to look for a catch when unwinding
    4. NIL meant the global environment

Forestry Rules

There are two stack-group trees, context and handler. The fact that x is a parent of y víz a víz one type of pointer imposes NO restrictions on the relation of x and y víz a víz the other pointer type. All three pointers can be changed.

14 Stack Groups are Useful

- Example: A parser can fork the current process when a list of syntactic alternatives occur; each one inherits the parent's context (e.g., bindings) and a child can "improve" the parent in a way that will be shared.
- Think of the self-analytic power of eval-in(exp,SG).
- A stack group's context, starter, and/or handler parents can be changed individually or together as long as trees result.
- Groups can be used to mimic various sorts of closures.
- N.B. None of the modern closure disciplines are automatically provided; however, one could impose fairly straightforward naming and binding conventions to simulate them.

15 Coroutine Primitives

- $f(e_1, \ldots, e_n[, \mathrm{context}][, \mathrm{handler}][, \mathrm{activator}])$, where f is defined to be a process function as opposed to an ordinary function. Creates a new process with f as the top-level function and starts its execution in a new stack group.
- $\mathrm{resume}(h, \mathrm{exp})$, where h is a handle of a process. Starts h executing and provides value of exp to the resume point where h was suspended.
- set-x(h_1, h_2), where h_1 and h_2 are handles and x is context, handler, or activator. Sets the indicated parent of h_1 to h_2.
- A normal return from a top-level process function resumes the activator parent with the naturally produced value.
- eval-in as mentioned previously.
- $\mathrm{copy}(h)$ copies the handle and stack group.

16 Rest of This Talk

- Namespaces
- Data Structures
- Control Structures and Spaghetti
- **Types and Incremental Compiles**
- Storage Management
- Odds and Ends
- What Actually Got Implemented!

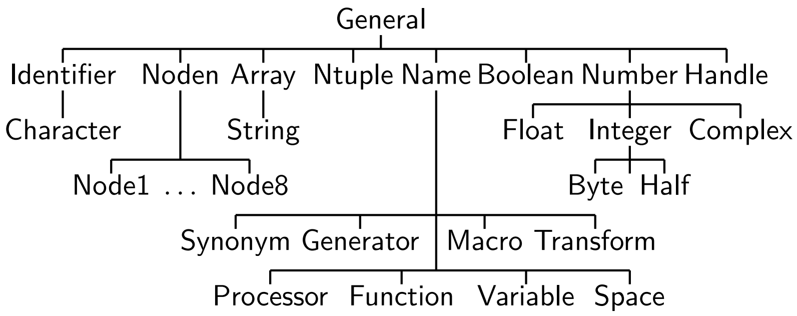
17 Types and Incremental Compiles

Basics

The system was strongly typed, supported incremental compiles, type definitions could be recursive, and numbers–with exact type specified–were unboxed.

- Basic types: general, cons nodes, integer, real, complex, number, symbol, character, name, handle, etc.
- Composite types:
  - Ntuples (with constant repeat counts)
  - Arrays with dimension and element type
  - Name and its subtypes:
    - Functions with argument/value types specified
    - Global variable with value type specified
    - A name was a shorthand, in type context, for its own type

18 Type Hierarchy



Notes

Need subspecification: Array, Ntuple, Processor, Function, Variable
May only appear as element type in array or ntuple: Byte, Half

19 Type Recursion

Example

Ntuple Example–A List of Alternating Integers and Floats

```
(def-ntuple foo1 (a integer)(b foo2))
(def-ntuple foo2 (a float)(b foo1))
```

Example

Functional Example–A function takes a symbol argument does something and returns a function to call next time:

```
(defun x$y (symbol) -> x$y)
```

Recall, name* can be used in type context to mean the type ascribed to that name; least-fixed point semantics intended.

***Footnote**

Things were so arranged that all type definition loops pass through at least one name object.

20 Subtype Relation

Definition

We will write $t_1 \subset t_2$, where t_1 and t_2 are types, to denote t_1 is a subtype of t_2. That means that every object of type t_1 is also of type t_2. The idea is that anywhere one can deal with any object of type t_2, an object of type t_1 will be handled properly.

The Crucial Question

Let $t_1 = (\mathrm{func}(a_1) \rightarrow v_2)$ and $t_2 = (\mathrm{func}(a_2) \rightarrow v_2)$. When is $t_1 \subset t_2$?

The Answer

$t_1 \subset t_2$ iff $a_2 \subset a_1 \wedge v_1 \subset v_2$. These were the criteria used to determine whether functional objects needed to be shadowed.

21 Incremental Compilation

Conditions of Contest

Since the system was strongly typed, your first compile had to define enough stuff so that there were no references to undefined types. Typical strategy was to use dummy definitions then use incremental compile to replace dummies with real definitions.

- Two types of files: 1) pseudo directory–list of files, 2) code.
- Compiler made two passes over all files in input tree:
  1. Grab all top-level definitions and process them.
  2. Compile code with all definitions available.
- First pass dealt with the intricacies of recursive definitions and decided, for existing definitions whether current definition could be abandoned (or kept and shadowed).
- Both new and old definitions might need to coexist to provide type safety.

22 Rest of This Talk

- Namespaces
- Data Structures
- Control Structures and Spaghetti
- Types and Incremental Compiles
- Storage Management
- **Storage Management**
- Odds and Ends
- What Actually Got Implemented

23 Storage Management

- There were multiple sorts of spaces:
  - One for each cons_i, where $1 \leq i \leq 8$
  - Ntuple
  - Array
  - Stack Group
  - etc.
- A space was made up of dynamically allocated regions that were not necessarily contiguous.
- One space of each type was selected by binding to an appropriate variable, e.g., default-cons2$crisp.
- Conses went into the selected space by default.
- Each space definition specified several policy functions that advised object creation and the GC.

24 Space Policy Functions

Method to Accumulate Storage Management Knowledge

Policy alternatives allow experiments with memory management tactics as well as application of well-know schemes.

Cons	The cons function determined object allocation policy. Smart cons, from erasure, next available.
Initial Mark	Mark any objects in space that are kept by policy, e.g., a symbol with a property list.
Mark	Marks object determined to be a keeper and causes strong pointers to be chased.
Prune	Abandon unnecessary objects (x out weak links).
Plan	Determine where each object in space will reside after GC–fold space if appropriate.
Update	Update pointers from objects in space.
Move	Relocate objects in space.
Fixup	Clear bookkeeping junk.

25 Storage Management Experiments

- Was smart cons really smart? It didn't seem so.
- Was copy collect the cat's meow? It might have been but our limits on address and well as memory made real experiments inconclusive.
- Were erasure lists the wave of the future as well as the past? Seemed to make no sense in VM systems.
- Should array/ntuple space regions be allocated to max size or to something more reasonable and usual? No conclusive results.
- What are the tradeoffs between larger name space blocks and availability of GP registers for computing? Not tried.
- What is the impact of quicker storage release to OS? Extraordinary!

26 Rest of This Talk

- Namespaces
- Data Structures
- Control Structures and Spaghetti
- Types and Incremental Compiles
- Storage Management
- **Odds and Ends**
- What Actually Got Implemented!

27 Odds and Ends–Linguistic

A few things that didn't fit into any of the above categories:

- Characters/symbols given left/right precedence as well as class attributes. Facilitate building a powerful infix reader front end to the language system. (Most prefix herein is for you).
- A version of own/static variables.
- Availability of many compile-time facilities:
    - Macro–as you know it today
    - Synonym–symbol macro
    - Transform–defsubst (without defun)
    - Generator–part of the compiler for special forms, e.g., IF
    - Global name as type
- System-level primitives borrowed from Lisp 2:
    - $\mathrm{core}(\mathrm{int})$
    - $\mathrm{bits}(\mathrm{int}, \mathrm{int}, \mathrm{memref})$
    - $\mathrm{drive}(\mathrm{exp}, \mathrm{type})$
    - $\mathrm{callit}(\mathrm{exp}, \mathrm{type})$

28 Odds and Ends–Implementation

- The executing process/coroutine always accessed global variable values through top-level binding cells (shallow binding).
- Non-executing process/coroutine maintained deep bindings.
- The GC process always ran with the global environment as its parents.
- The IBM 360 Lisp 1.5 system was used to build the bootstrap mechanism; its output was a core image.
- CRISP, from early on, was able to recompile itself and build core images.
- We had a large investment in Lisp 1.5 software so we just loaded Lisp in a CRISP heap and managed it, pretty much, like a coroutine.

29 Rest of This Talk

- Namespaces
- Data Structures
- Control Structures and Spaghetti
- Types and Incremental Compiles
- Odds and Ends
- Storage Management
- **What Actually Got Implemented!**

30 What Actually Got Implemented

CRISP was Developed in Parallel with Speech Research

Our primary goal was speech research; CRISP was merely a supporting technology. After the two of us spent the better part of a half year designing and starting CRISP implementation we got tired–lack of sleep. We needed the system-building capabilities but not necessarily the language.

The Certa Perfect Sleeper

The approach we decide on was to implement the world's most capable assembler, CRISP Assemble Program (CAP). We had already built all the supporting libraries: storage management, type mechanisms, file processor, input/output, coroutine support, arithmetic, GC, etc. All we had to do was make it possible to access the hard stuff in a way that wasn't error prone.

31 Call and Branching Example

```
(defun ext (a b c)
  (if (eq a b) (cons a c)
      (cons a (cons b c))))

(defCAP ((a symbol) (b symbol) (c node2))
  (L R5 a)
  (C R5 b)
  (BE (CALL cons2
            (ST R5 PUSH)
            (L R5 C))
      (RET))
  (CALL cons2
        (ST R5 PUSH)
        (CALL cons2
              (MV b PUSH)
              (L R5 C))))
```

32 Process Start Example

```
(defCAP spawn ((arg general) (continue handle))
  (CALL RESUME
        (L R5 continue)
        (ST R5 push)
        (START proc-fcn
               (L R5 arg)
               (ST R5 push)
               (L R5 self$crisp)
               (ST R5 push)
               (ST R5 push)
               (ST R5 push))))
```

Start a process by calling proc-fcn, with the specified argument, pass the value given by our restarter to continuation, then return the value given by our next restarter.

33 Binding Example

```
(defCAP boom ((n integer))
  (L R5 n)
  (CR R5 R0)
  (BLE (LR R5 R0) (RET))
  (BLOCK
    (CALL boom
          (DECF R5)
          (ST R5 PUSHN))
    (ST R5 PUSH)
    (ST R0 PUSH)
    (BIND ((x node2) (y general global))
    (CALL cons2
          (L R5 x)
          (ST R5 PUSH))))
```

```
(defun boom (n)
  (when (> n 0)
    (let((x (boom (1- n)))
         y)
      (dec (special y))
      (cons x x))))
```

34 Miscellaneous CAP Examples

```
(L R5 (S_B_C R4))   %%S an ntuple

(typecase
  (type1 $instructions)
  (type2 $instructions)
  ...)

(catch (reasons) $instructions)

(throw (reasons) $instructions)

(try $($instructions))
```

$ indicates 0 or more repetitions:

35 Endnotes

- We built speech systems in CRISP and were able to comfortably experiment with most aspects of system architecture/structure.
- The use of a machine language was little hindrance since
  1. In the old days, assemblers were meant for people.
  2. Machine order codes weren't so damn kinky.
- Many of the more exotic data and control mechanisms were used early on, then simplified when we better understood what we were doing.
- CRISP was archived at Princeton(?) home for retired languages.
- I searched for and found Doug Pintar while preparing this talk. He's alive and well in Denver, Colorado!