Developing Correct Compilers

A LISP Experiment

Jeffrey A. Barnett

Albuquerque Retired Citizens

January 24, 2009

2 Compiler Correctness

- Debugging programs is hard enough without the added complications of a defective compiler.
- Even suspicion of compiler bugs is sufficient to derail application development.

- Methods used to develop and debug a LISP 1.5 compiler in the late 1960's.
  - Developed time was approximately 6 months.
  - Included Runtime (GC, library, IO, etc.) and Assembler.
- Only two compiler bugs found in over a decade–actually the same bug twice.
- The methods included:
  1. Subset compiles correctly $\Rightarrow$ the whole language does too, and
  2. Explicitly use of context in compiler algorithms.

4 Miscellaneous Bullets

What is a compiler?

A compiler is a computer program that processes a formal specification written in a formal language and BINDS some aspects of the specification's meanings.

Why should you care about this talk?

I used to say that a competent programmer should/must write a compiler every few years in order to keep his journeyman status current. This idea was based on the fact that compilers usually represented the ultimate necessity for clean organization and, thus, it was a great way to revisit and practice the first principle of system organization: *structure über alles*.

- Emphasis on space saving: Reducing swap cost and GC hit much more important than saving an instruction execution here and there.
- Thus, there was a derived emphasis on branch optimization.
- Many simple functions such as car and cdr were NOT open coded.

My Point of View

I believe that many LISP compiler writers of the era were, given memory and address limitations, misguided in pursuing fast code at the expense of space reductions. (This observation, right or wrong, was the origin of many CRISP ideas.)

6 Rest of the Talk

- Subset-Implies-Whole Technique
- Use of Compiler Contexts

7 Subset Implies Whole Example

- Let language consist of variables, and, or, and not.
- Let L_1 and L_2 be expressions in this language.
- We will say that $L_1 \equiv L_2$ when L_1 can be transformed to L_2 by any number of applications of

  Double Negation (not (not x)) $\leftrightarrow$ x
  DeMorgan (not (and a...z)) $\leftrightarrow$ (or (not a)...(not z)), etc.
  Nesting (or a...z) $\leftrightarrow$ (or a...i (or j...q) r...z), etc.
- Note, if $L_1 \equiv L_2$ then L_1 and L_2 contain the same number of variable references and the references occur to the same variables in the same order.
- Also note that "$\equiv$" is an equivalence relation.
- Unfortunately, (or x x) $\not\equiv$ x

- Use compiler algorithm $\mathcal{A}$ s.t.

  If $L_1 \equiv L_2$, then $\mathcal{A}(L_1)$ and $\mathcal{A}(L_2)$ are provably bit-for-bit identical codes!

- So what?

- Consider the sublanguage where
  - No or is a top-level expression in another or.
  - No and is a top-level expression in another and.
  - All not's are wrapped around variables.
- Every expression in the original language can be converted, using the three transformations mentioned above, to an equivalent expression in the sub language.
- Therefore, if every expression in the sublanguage compiles correctly, so does every expression in the original language!
  - Ergo, one only needs to check the compiler is correct for expressions in the sublanguage.

10 Compiler Impact

- It's much easier to invent a reasonable set of test cases for the compiler when only the sublanguage needs to be considered.
- Note, algorithm $\mathcal{A}$ did not do explicit double negation reduction, de-nesting of and's and or's, or DeMorgan transformations.
- The equivalent output property was a magical property of $\mathcal{A}$ first noted to me by one of the LISP 2 hackers.
- I, later, proved the magic was real and not smoke.

11 Rest of the Talk

- Subset-Implies-Whole Technique
- Use of Compiler Contexts

12 Compilation Context

Context: Informal Characterization

The context in which a form appears tells the compiler what to do with the result of the evaluation. In LISP 1.5 three major contexts are identifiable:

Statement Evaluate for (side) effect. If a value is produced, it may be discarded.
[Submode] A place to go after evaluation is known.

Expression A value is expected and needed.
[Submode] A place in the code is expected to receive that value.

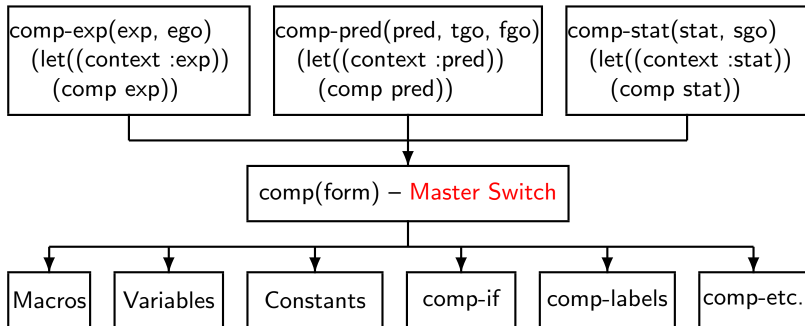
Predicate A value is expected and will determine the value of the program counter.

13 Context Examples

```
(defun foo ()
  (let ()                     ; let
    (if p1 s1 s2)             ; if1
    (if p2 e1 e2)))           ; if2

EXP   EXP+  STAT  STAT+ PRED
let   e1    if1   s1    p1
if2         s2          p2
e2
```

14 Gross Compiler Organization



Special Variables

CONTEXT, EGO, SGO, TGO, FGO

15 Compiler Output Language

Compiler Output

The action of a compiler subroutine is to tack machine code instructions and labels onto a listing. A label is just a symbol or integer that will be used as a target of branch instructions.

Machine Code Instructions Used Below

L x v	Load register x with copy of value of variable v.
B ℓ	Branch to label ℓ.
BT x ℓ	Branch to label ℓ if register x $\neq$ nil.
BF x ℓ	Branch to label ℓ if register x $=$ nil.

16 Variable Compiler–Register Machine

```
(defun comp-variable (sym)
  (check-var-ref sym)
  (case context
    (:exp (attach '(L R ,sym))
          (and ego
               (attach '(B ,ego))))
    (:stat (and sgo
                (attach '(B ,sgo))))
    (:pred (attach '(L R ,sym))
           (and tgo
                (attach '(BT R ,tgo)))
           (and fgo
                (attach '(BF R ,fgo))))))
```

17 Compile Not Form

```
(defun comp-not (form)
  ;; FORM == (NOT x)
  (case context
    (:stat (comp-exp (cadr form) sgo))
    (:exp (comp `(IF ,(cadr form) NIL T)))
    (:pred (comp-pred (cadr form) fgo tgo))))
```

18 Compile IF Expression

```
form = (if p x y), ego = nil, labels a, b
comp-pred(p, nil, a)
comp-exp(x, b)
attach-label(a)
comp-exp(y, nil)
attach-label(b)

form = (if p x y), ego /= nil, labels a
comp-pred(p, nil, a)
comp-exp(x, ego)
attach-label(a)
comp-exp(y, ego)
```

19 Compile IF Statement

```
form = (if p x y), sgo = nil, labels a, b
comp-pred(p, nil, a)
comp-stat(x, b)
attach-label(a)
comp-stat(y, nil)
attach-label(b)

form = (if p x y), sgo /= nil, labels a
comp-pred(p, nil, a)
comp-stat(x, sgo)
attach-label(a)
comp-stat(y, sgo)
```

20 Compile IF Predicate

```
form = (if p x y), labels a, b
```

```
tgo, fgo /= nil            tgo=nil, fgo/=nil          tgo/=nil, fgo=nil
comp-pred(p,nil,a)         comp-pred(p,nil,a)         comp-pred(p,nil,a)
comp-pred(x,tgo,fgo)       comp-pred(p,b,fgo)         comp-pred(x,tgo,b)
attach-label(a)            attach-label(a)            attach-label(a)
comp-pred(y,tgo,fgo)       comp-pred(y,tgo,fgo)       comp-pred(y,tgo,fgo)
                           attach-label(b)            attach-label(b)
```

Contexts of Compilation Depend on Context

These three slides showing compilation of IF expressions, statements, and predicates indicate that the context of a form depends on the context of its parent. The subcontexts provide trivial chances for optimizations.

What's necessary to debug our compiler?

- First, debug the building blocks:
  - Variables
  - Constants
  - Function Calls
  - Macro Expander
- Second, deal with more sexy open-coded forms:
  - Try each form variation (e.g., IF with two or three arguments)
  - In each of the eight contexts.
- If it all works, there is a fair degree of assurance the compiler is good to go.

22 Comparison With Other Systematic Methods

- What other systematic methods?
- While there are many meta-tools to develop compiler syntax passes, there are hardly any that support code generation.
- The few I know about do not support systematic ways of doing checkout.
- The regression testing files are usually enormous and as much a source of pride as an alcoholic's beer belly!

In the interest of fairness

To be fair, most languages do not have the sort of consistent semantics and rules of interpretation that LISP does. The above methods simply exploit that consistency. Part of that consistency is expressed by order-of-evaluation constraints and the fact that you know, given the computational context, exactly what forms will and what forms will not evaluate.

- As mentioned in the introduction, a LISP compiler was developed using these techniques. It was found to be virtually error free in over a decade of service.
- The sublanguage-mirrors-the-whole concept arouse naturally through the recursive organization of the compiler. However, such transformations could be explicitly done if it would simplify debugging.
- Note, there where contexts not discussed above, e.g., binding and declaration. Of course they must be handled too.
- Most programming languages have similar sets of context and the compiler writer should investigate whether and how they might be exploited.
- Context optimizers, if available, are appropriate before graph-analysis optimizers take over.

24 Afterword (continued)

Life's Tough

Code, such as a compiler, that performs many different tasks is very hard to debug. Any approach or technique that suggests a systematic way to debug is worth gold.

LISP Code Often Resembles a Compiler

Many Lisp applications are organized by recursive descent strategies. If one can organize that recursion into a small number of contexts, debugging might be better structured. General use of a small number of special variables that control activity within the recursion is an indication that these methods might be applicable.