An evaluation of Major Lisp Compilers

阿部正佳
数理システム知識工学部

2009 年 10 月 8 日

概 要

Lisp 処理系はその性質上数値計算を苦手としてきた。しかし近年の処理系はコンパイラに適切な指示を与え、またコード中に型宣言を加えることにより Fortran や C といった古典的なコンパイラと比較しても遜色ないコードが出せるようになりつつあると言われている。しかし両者の生成コードの間には、実行効率の点で今だに無視出来ない開きがあるのも事実である。本レポートでは 3 つの主要な Common Lisp 処理系である ACL (Allegro Common Lisp 8.1), SBCL (Steel Bank Common Lisp 1.0.23), LispWorks (6.0) を中心に、その無視出来ない開きの原因を調査した。

目 次

1 はじめに

どのような観点で Lisp コンパイラを調査するのかを明確にするために。トラディショナルな最適化コンパイラのコード生成系で行われる処理について簡単にまとめておく。

1. 基本ブロックとフローグラフ

   プログラムの命令列を、最後に一つの分岐命令が置かれた形のブロックの列に区切ることが出来る。このとき各ブロックは基本ブロックと呼ばれる。そして、プログラムはこの各基本ブロックからなるノードと、基本ブロック間の分岐命令に対応した有効辺に対応するエッジからなる有効グラフと見なせる。このグラフをフローグラフと呼ぶ。

2. データフロー解析

   典型的な最適化コンパイラはプログラムをまずフローグラフに変換し、コード生成の各処理をこのフローグラフ上の変換として実装する。より良いコード生成を行うためには、各処理においてさまざまな静的解析が必要となる。そのような静的解析を総称的にデータフロー解析と呼ぶ。フローグラフはそのような解析に適した形式の一つである。

3. コード最適化

   プログラムの基本ブロックはターゲットマシンと独立な、中間コードと呼ばれる命令の列からなる。中間コードを、プログラムの意味を変えることなく、より良いコードに変換することをコード最適化と呼ぶ。後に、いずれの Lisp コンパイラが、どのようなコード最適化を行っているかを、以下の非常に基本的なコード最適化について調査する。

   (a) 定数畳み込み
   コンパイル時に計算可能な部分式を計算し定数に置き換える

   (b) 定数伝播
   値が定数である変数の出現をその定数で置き換える

   (c) 共通部分式削除
   同じ値に評価される複数の式の計算を一回で済ませる

   (d) ループ不変式巻上げ
   ループ中でいつも同じ値に評価される式をループ外に追いだす

   定数畳み込みのように、ひとつの基本ブロック内で行われる最適化はローカル最適化と呼ばれ、フローグラフ全体に対して行われる最適化はグローバル最適化と呼ばれる。定数伝播は基本ブロック内に限れば (ローカル定数伝播) 容易であるが、フローグラフ全体に対して行う (グローバル定数伝播) 場合、ある変数がある場所で常に同一の定数値を持つことを検出するためのデータフロー解析が必要になる。共通部分式削除についても、同様にローカル版とグローバル版があり、グローバル版ではまた別なデータフロー解析が必要になる。

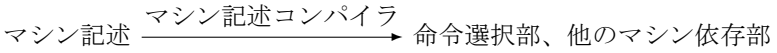
4. 命令選択

   コード最適化が行われたフローグラフに対して、命令選択はコンパイラの中間言語で表現されている演算の列を、ターゲットマシンに実際に存在する命令列で置き換える処理であり、コード生成系で最も重要な処理のひとつである。

   個々のターゲットマシンに応じて命令選択部を人手でコーディングするのではなく、ターゲットマシンの記述からコード生成系を自動生成する仕組みを備えたコンパイラをリターゲッタブルコ

ンパイラと呼ぶ。例えば GCC は最も広く使われている強力なリターゲッタブル C コンパイラのひとつである。

マシン記述は実際にはターゲットマシンの命令記述の他に、レジスタ構成や使用法についての定義、整数変数のビット幅等、ターゲットマシンに依存するさまざまな情報を含む。それらは、コード生成系の自動生成系を含む変換系 (マシン記述コンパイラ) によって、命令選択部及びターゲットマシン依存部に変換され、マシン独立部とリンクされてターゲットマシン用のコンパイラが作られる。



命令選択部は複雑なプログラムなので、人手での記述は面倒でありバグを招きやすい。X86 のようなクセのある CISC マシンの命令を巧みに利用するコード選択部の場合は特にそうである。一方で、現在のリターゲッタブルコンパイラ技術は成熟しており、ほぼ理論的に最適な命令選択部をマシン記述から自動生成することが可能となっている。特に、ダイナミックプログラミングによるコード生成系を自動生成するアプローチは現在では支流である。

5. レジスタアロケーション

   コンパイラの最終段階で、中間言語の変数 (仮想レジスタ) に実レジスタ (ターゲットマシンに実在するレジスタ) を割り当てる処理であり、命令選択部と並び、コンパイラの最も重要な部分のひとつである。よって、さまざまな手法が研究されてきたが、現在ではグラフカラーリングアロケータと呼ばれる手法が支流である。

   この手法ではまずフローグラフ全体に対して生存区間解析というデータフロー解析を行い、レジスタ割り当てのための制約条件を表す無向グラフを作成する。すると、レジスタアロケーションの問題は、色 (実レジスタ) をノード (仮想レジスタ) に、隣あうノードの色が異なるという条件下で、割り当てるという問題に変換される。そして最後にグラフ理論的な手法を用いて、このカラーリング問題を解くというのがグラフカラーリングアロケータである。

   干渉グラフの表現にメモリーを大量に消費するし、データフロー解析の計算コストも高いが、非常に優れたアロケーションが行えるので、最適化コンパイラでは現在標準的な手法である。

2 Lisp の事情

ここでは数値計算が苦手とされる Lisp の事情について説明する。ある C プログラマが書いてみた整数を加算するだけの簡単なプログラムの生成コードを見てみよう。

```
(defun int-add-1 (x y)
  (+ x y))
```

ACL Lisp コンパイラは以下のように C プログラマが卒倒しそうなコードを生成する (他の処理系でも同様)。

```
 0: 55       pushl    ebp
 1: 8b ec    movl     ebp,esp
 3: 83 ec 28 subl     esp,$40
 6: 89 75 fc movl     [ebp-4],esi
 9: 89 5d e4 movl     [ebp-28],ebx
12: 39 a3 be 00 cmpl  [ebx+190],esp
    00 00
18: 76 03     jbe     23
```

```
 20: ff 57 43 call     *[edi+67]  ; SYS::TRAP-STACK-OVFL
 23: 83 f9 02 cmpl     ecx,$2
 26: 74 03     jz      31
 28: ff 57 8b call     *[edi-117]         ; SYS::TRAP-WNAERR
 31: 80 7f cb 00 cmpb  [edi-53],$0        ; SYS::C_INTERRUPT-PENDING
 35: 74 03     jz      40
 37: ff 57 87 call     *[edi-121]         ; SYS::TRAP-SIGNAL-HIT
 40: 8b d8     movl    ebx,eax
 42: 0b da     orl     ebx,edx
 44: f6 c3 03 testb    bl,$3
 47: 75 0e     jnz     63
 49: 8b d8     movl    ebx,eax
 51: 03 da     addl    ebx,edx
 53: 70 08     jo      63
 55: 8b c3     movl    eax,ebx
 57: f8         clc
 58: c9         leave
 59: 8b 75 fc movl     esi,[ebp-4]
 62: c3         ret
 63: 8b 5f 8f movl     ebx,[edi-113]      ; EXCL::+_2OP
 66: ff 57 27 call     *[edi+39]  ; SYS::TRAMP-TWO
 69: eb f3     jmp     58
 71: 90         nop
```

ここに、SYS::TRAP-STACK-OVFL はスタックオーバーフローチェック関数、SYS::C_INTERRUPT-PENDING は割り込みチェック関数、EXCL::+_2OP は加算関数である。

2.1 インタクティブ環境

優れたインタラクティブ環境は Lisp の大きな特徴である。コンパイルされたコードの実行でも、その恩恵をさずかるためにコンパイラはその生成コード中にもスタックオーバーフローチェックのコードや、割り込み監視用のコードを挿入せざるをえない。しかし完成したプログラムからはそういったコードは除外し、代わりに最も最適したコードを生成してほしい。このためには以下のような宣言を行えばよい。

```
(defun int-add-2 (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (+ x y))
```

チェック関数の呼び出しは消えたが、生成コードは以下のように、まだまだ長い。

```
 0: 8b d8    movl     ebx,eax
 2: 0b da    orl      ebx,edx
 4: f6 c3 03 testb    bl,$3
 7: 75 0d    jnz      22
 9: 8b d8    movl     ebx,eax
11: 03 da    addl     ebx,edx
13: 70 07    jo       22
15: 8b c3    movl     eax,ebx
17: f8         clc
18: 8b 75 fc movl     esi,[ebp-4]
21: c3         ret
22: 8b 5f 8f movl     ebx,[edi-113]      ; EXCL::+_2OP
25: ff 67 27 jmp      *[edi+39]  ; SYS::TRAMP-TWO
```

ここで Location 11 の addl のみが C プログラマの期待する生成コードであろう。

2.2 ポリモルフィズム

Lisp は動的な型システムの言語であり、変数の型は動的に変化しうる (ポリモルフィズム)。組み込み関数の多くは、その実行時に coercion (引数の暗黙的な型変換、アドホックポリモルフィズムとも呼ばれる) を行う。例えば `(+ x y)` というフォームでは実行時の `x` と `y` の型に応じて整数、有理数、不動小数点数、あるいは複素数の加算が行われる。Lisp で

このようなポリモルフィズムを実現するため、Lisp 変数はオブジェクトそのものではなく、オブジェクトのアドレスを格納したポインタとして実装される。C 言語では単に二つの整数を加算すればよい所を、Lisp ではポインターをたどり、そこにおかれているオブジェクトから数値を取り出し、計算し、さらにその数値をオブジェクトとし、そのポインタ値を演算結果としなければならない。この計算を行うのが `EXCL::+_2OP` である。

この酷い状況を改善するために、Lisp では fixnum 型と呼ばれる小さい整数に関してはポインタを介さず直接変数に格納するという工夫をしている。変数の下位 2 ビットが 00 ならば、それはオブジェクトへのポインタではなく 4 倍された整数値と見なすのである。Lisp 変数の下位 2 ビットはこのようにタグとして使われる。fixnum のタグ表現は ACL, SBCL, LispWorks とも同じである。

先の `int-add-2` の生成コード Location 2: では二つの引数 (eax と edx に入ってくる) のタグを調べ、両者ともタグ表現された fixnum なら直接 `addl` により加算を行い、そうでなければ万能の `EXCL::+_2OP` にゆだねる、というコードになっている。

先の C プログラマーは複素数の加算など期待していない。このために、引数はつねに fixnum であると仮定してコンパイルしてもよい、という指示を与えることが出来る。

```
(defun int-add-1 (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x y))
  (+ x y))
```

これにより、C プログラマが期待していた以下のようなコードが生成される (出入り口のコードは略)。

SBCL と LispWorks でも同様の宣言をサポートしているが、これらの処理系では fixnum と型宣言された変数の加算でも、結果が fixnum に収まらないと仮定するので、上記の最後は `(the fixnum (+ x y))` と書かなければならない。以後の調査においては、簡潔さのために ACL の仮定に合わせるとする。したがって、 fixnum 演算を含むプログラムは、SBCL 及び LispWorks でテストする場合は、実際にはそれらを `(the fixnum ...)` で囲んだプログラムを使用している。

```
 0: 03 c2    addl     eax,edx
```

ただし、C プログラマは fixnum が C でいう int ではないことに注意しなければならない。タグ表現のために 2 bit 損をしているので、実際には 30bit の int となるからである。また、加算や乗算はタグ表現 (4 倍されていること) を無視しても問題ないが、割り算等タグ表現をケアするための余分なコードが生成されてしまう場合もある。

3 コード最適化の調査

ここでは Lisp コンパイラが、どのようなコード最適化を行っているかを調べる。

3.1 定数畳み込み

まず簡単な定数畳み込みから始める。これは例えば以下のようなプログラムの生成コードを調べればよい。これは + の引数が定数のみの場合であり、畳み込みというよりは単なるコンパイル時の定数計算の例である。

```
(defun test-const-fold-1 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (+ 1 2 3))
```

これに対する ACL の生成コードは以下のようであり、明らかに定数計算を行っている。返却値レジスタ tt eax に 6 ではなく 24 がセットされているのは、前述のタグ表現への変換が行われているからである。

実際のコンパイルコードには関数の入り口や出口処理、定数テーブルのセットアップ、フレーム上に割り当てられた変数領域の確保、等のコードが生成されているが、それらは省略して本質的な部分のみを引用する。

```
0: b8 18 00 00 movl  eax,$24      ; 6
```

SBCL も同様に定数計算を行っている。SBCL は `edx` を返却値レジスタとして使用している。

```
0AE0E03A:       BA18000000       MOV EDX, 24
```

LispWorks についても同様である。返却値レジスタは ACL と同様 `eax` である。

```
0:      B818000000       move  eax, 18
```

一般に、定数畳み込みは代数的な性質を駆使して式を変形し、可能な限り定数部分を「引き寄せ」て簡略化する最適化であり、決して容易なものではない。例えば GCC の定数畳み込み最適化のソース行数は 1 万行を超えている。簡単な例だが GCC では以下のような畳み込みを行える。

```
(1+n)+(n+2)*2  →  3*n + 5
```

プログラマが注意深くコーディングすれば良いと思われるかもしれない。しかしこのような冗長な式は実は他の最適化の式変形の結果として非常によく現れるものであり、複雑な最適化の最後あるいは中間段階のコードの対して、定数畳み込みは高頻度で使用される。そこで畳み込みが成功するかしないかで、後の最適化が成功するかどうかにかかわって来るのである。

すなわち、コード最適化を極めているコンパイラは、かならず優れた定数畳み込みを行っているといえる。(論理的に対偶で表現すれば: 「定数畳み込みを行っていないコンパイラは、高度なコード最適化を行っていない」と言える) 以下は非常に簡単な定数畳み込みのテストプログラムであり、式は単に 0 (fixnum) に畳み込める。

```
(defun test-const-fold-2 (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (+ 1 n -1 (- n)))
```

これに対する ACL の生成コードは以下のとおりであり、明らかに畳み込みは行われていない。

```
 2: 83 c3 04 addl     ebx,$4
 5: 83 c3 fc addl     ebx,$-4
 8: 33 d2     xorl    edx,edx
10: 2b d0     subl    edx,eax
12: 8b c3     movl    eax,ebx
14: 03 c2     addl    eax,edx
```

先に触れたように、SBCL と LispWorks は fixnum 型宣言された値の演算ポリシーが ACL と異なる。`(+ 1 n -1 (- n))` の部分式を `(the fixnum ...)` で囲むとそれらのコンパイラの最適化を妨げる可能性があるので、テストではこのままのコードを使用した。しかし、いずれにせよ SBCL も LispWorks もこの畳み込みは出来ていない。以下が SBCL のコードであり、明らかに畳み込みは行われていない。

```
0A950DDD:       8D4201          LEA EAX, [EDX+1]
      E0:       83C0FF          ADD EAX, -1
      E3:       F7DA            NEG EDX
      E5:       01D0            ADD EAX, EDX
      E7:       6BD004          IMUL EDX, EAX, 4

     ... snip ...
```

そして以下が LispWorks の生成コードである。同様に畳み込みは行われていない。

```
      0:      55              push  ebp
      1:      89E5            move  ebp, esp
      3:      50              push  eax
      4:      6A04            pushb 4
      6:      B502            moveb ch, 2
      8:      8B45FC          move  eax, [ebp-4]
     11:      FF15BCF90320    call  [2003F9BC]       ; SYSTEM:+$FIXNUM

      ... snip ...
```

3.2 定数伝播

次に定数伝播について調べるために、以下の簡単なプログラムの生成コードを見てみる。定数伝播を行っていれば、単に 3 を返すコードが生成されるはずである。

```
(defun test-const-propagation-1 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((a 1)
        (b 2))
    (+ a b)))
```

これに対する ACL の生成コードは以下のとおりである。

```
  0: bb 04 00 00 movl        ebx,$4       ; 1
     00
  5: ba 08 00 00 movl        edx,$8       ; 2
     00
 10: 8b c3      movl eax,ebx
 12: 03 c2      addl eax,edx
```

明らかに定数伝播は行われていない。Lisp のマクロは非常に強力である。定数伝播と定数畳み込みの連鎖によりマクロコードが部分的に簡略化される可能性は大きい。数値解析にかぎらず、マクロを安心して使うためにもこのような最適化処理はぜひとも必要である。

SBCL と LispWorks では上記のような簡単な場合、つまり定数が let バインドされた変数がそのまま現れる場所にかぎり、定数伝播を行っている。以下は SBCL の例であるが、LispWorks でも同様である。

```
BA0C000000       MOV EDX, 12
```

ただし、let バインドされた変数に対する代入があれば、その変数に対する定数伝播をあきらめてしまうようである。明らかに定数伝播可能な以下のような例に対して、

```
(defun test-const-propagation-2 ()
  (declare (optimize (speed 3) (safety 0) (debug 0)))
  (let ((a 1)
        (b 2))
    (declare (type fixnum a b))
    (setq a (the fixnum (+ a b)))
    a))
```

SBCL は以下のように実行時に加算を行なってしまう。

```
0AADC782:       B804000000       MOV EAX, 4
      87:       83C008           ADD EAX, 8
```

LispWorks も同様である。

```
      0:      83C004           add   eax, 4
      3:      83C00C           add   eax, C
```

関数型言語とはいえ、Lisp では代入もそれなりに使用される。マクロを安心して使うためにも、このあたりはもう少し改善してほしい所である。

3.3 共通部分式削除

複雑な共通部分式がプログラムに明示的に現れることはまれである。それは、複雑だが同一の式、あるいは同一の値に評価される式を繰り返すことは可読性、メンテナンス性欠けるプログラムとなるからである。しかし、`i+1` 程度の式が数回現れる程度のことはよくある。

より重要なのはプログラマがケア出来ない隠れた共通部分式がある。典型的には配列のインデックス計算がそれである。`A[i]+B[i]` という配列参照においてすら、`i` と配列要素のサイズとの乗算が 2 回現れる。よって配列を多用するような数値計算系プログラムでは共通部分式の削除は必須である。以下の明らかな共通部分式を含むプログラムで、その生成コードを調べてみる。

```
(defun test-common-subexpression-elimination (x y)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x y))
  (+ (- x y) (- x y) (- x y)))
```

これに対する ACL の生成コードは以下のとおりである。

```
12: 8b d8       movl ebx,eax   ; eax=x
14: 2b da       subl ebx,edx   ; edx=y  <<
16: 89 45 dc    movl [ebp-36],eax
19: 29 55 dc    subl [ebp-36],edx       <<
22: 03 5d dc    addl ebx,[ebp-36]
25: 8b c8       movl ecx,eax
27: 2b ca       subl ecx,edx            <<
29: 8b d1       movl edx,ecx
31: 8b c3       movl eax,ebx
33: 03 c2       addl eax,edx
```

明らかに共通部分式である 3 回の引き算が 3 回繰り返されている。また、ここではスピルも発生しており、レジスタ以外のワーク `[ebg-36]` が使用されている。共通部分式の処理をしていれば、以下のような簡潔なコード生成が可能になる。

```
subl  eax,edx
movl  edx,eax
addl  edx,eax
addl  edx,eax
movl  eax,edx
```

SBCL と LispWorks でも同様に、共通部分式削除の最適化はされず、減算が 3 回繰り返される。SBCL と LispWorks のコードは Appendix 参照のこと)。

3.4 ループ最適化

最後に基本的なループ最適化として、ループ不変式の移動について調べる。共通部分式と同様、ループ中で結果が不変である計算コストの高い式を書くプログラマーは少ないであろうが、やはり配列のインデックス計算やマクロ中の見えざるループ不変式は存在してしまう。この最適化は共通部分式とならび、特に数値計算関係では重要な最適化と言われる。以下はループ不変式 `(+ n 10)` を含むプログラムである。

```
(defun test-loop-invariant-hoisting (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (let ((s 0))
    (declare (type fixnum s))
    (dotimes (i n)
      (incf s (+ n 10)))
    s))
```

これに対する ACL の生成コードは以下のとおりである。

```
21: 89 45 d8     movl     [ebp-40],eax
24: 83 45 d8 28  addl     [ebp-40],$40 ; 10
28: 03 5d d8     addl     ebx,[ebp-40]
31: 83 c2 04     addl     edx,$4
34: 3b 55 dc     cmpl     edx,[ebp-36]
37: 7c ee        jl       21
```

明らかにループ不変式 `(+ n 10)` の計算がループ中で繰り返されており、ループ不変式移動は行われていない。SBCL と LispWorks でも同様に、ループ不変式移動は成されていなかった (SBCL と LispWorks のコードは Appendix 参照のこと)。

3.5 コード最適化のまとめ

対象とした Lisp コンパイラはコード最適化 (トラディショナルな意味での) を殆ど行っていないと思われる。ここで調査したものは非常に基本的なものではあるが、上記の結果からより高度なコード最適化を行っているとは考えにくい。

4 レジスターアロケーションの調査

最適化コンパイラというと、前述のコード最適化を極めているコンパイラのことと思われがちである。それは誤りではないが、コード最適化よりも重要なのが命令選択とレジスターアロケーションである。ここの性能が悪ければいくらコード最適化の努力をしても効果はない。

先のコード最適化の調査と異なり、コンパイラがどのような手法で命令選択とレジスタアロケーションを行っているかを生成コードのみから調べるのは困難である。ここでは主にコンパイラが仮想レジスタに対して割り当てることの出来る実レジスタの集合 (以下単に汎用レジスタと呼ぶ) を調べる。

一般に CPU のレジスタ全てをコンパイラが汎用レジスタとして使用出来るわけでなない。たとえばプログラムカウンタ、スタックポインタ、フレームポインタは汎用レジスタとして使用出来ない。他に、コンパイラが特定のレジスタを特定の目的に使う場合も汎用レジスタの数を制限してしまう。

X86 で 32 bit 汎用レジスタとして使える候補は、プログラムカウンタ、スタックポインタ、フレームポインタ、コンデションコードレジスタを除く `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` のわずか 6 つである。以下、レジスタプレッシャ(スピルの起こりそうな) の高いプログラムによりコンパイラの使用する汎用レジスタを調べてみる。

4.1 ACL の場合

以下のレジスタプレッシャの高いプログラムで、どのレジスタを汎用レジスタとして使っているかを調べてみる。

```
(defun test-regalloc-nreg-3 (x)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x))
  (let ((y1 (+ x 1))
        (y2 (+ x 2))
        (y3 (+ x 3)))
    (+ y1 y2 y3)))
```

ACL の出力は以下のようであり、完全にレジスタに乗った美しいコードが生成される。

```
 0: 8b d8       movl  ebx,eax
 2: 83 c3 04    addl  ebx,$4
 5: 8b d0       movl  edx,eax
 7: 83 c2 08    addl  edx,$8
10: 83 c0 0c    addl  eax,$12
13: 03 da       addl  ebx,edx
15: 03 c3       addl  eax,ebx
```

次にレジスタプレッシャーをひとつ上げてみる。

```
(defun test-regalloc-nreg-4 (x)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum x))
  (let ((y1 (+ x 1))
        (y2 (+ x 2))
        (y3 (+ x 3))
        (y4 (+ x 4)))
    (+ y1 y2 y3 y4)))
```

ACL はここですでに以下のようにスピルが発生してしまった。

```
12: 8b d8       movl  ebx,eax
14: 83 c3 04    addl  ebx,$4
17: 8b d0       movl  edx,eax
19: 83 c2 08    addl  edx,$8
22: 89 45 dc    movl  [ebp-36],eax
25: 83 45 dc 0c addl  [ebp-36],$12
29: 83 c0 10    addl  eax,$16
32: 03 da       addl  ebx,edx
34: 03 5d dc    addl  ebx,[ebp-36]
37: 03 c3       addl  eax,ebx
```

結局、ACL コンパイラは汎用レジスタとして eax, ebx, edx のみっつのみを使用しているようである。ecx の利用ポリシーは不明だが、短命なスクラッチレジスタやループのインデックス変数として使用されているようであり、少なくとも変数が割り当てられるレジスタではないようである。残りの esi と edi は定数及び組み込み関数のテーブルのベースを保持するために使用されている。例えば (cons (+ x 1) 1.23) といった式の生成コードには以下のようなコードが含まれる。

```
 0: 8b 56 12    movl  edx,[esi+18]    ; 1.23
 3: 8b 5f 8f    movl  ebx,[edi-113]   ; EXCL::+_2OP
                ..............
```

esi と edi はこの用途のみで利用されている。Lisp 処理系ではダイナミックなリンクを行う必要があり、そのへんの事情であることはわかる。幾つかのレジスタを特定の目的のために占有させてしまう、というデザインはレジスタの多い RISC マシンでは自然な選択であろうが、X86 のようなレジスタの乏しいマシンでは問題である。

たとえば SPARC では 8 個のグローバルレジスタ、8 個のローカルレジスタ、6 個の IO レジスタをもち、コーリング規約以外の部分では自由に使うことが出来る。以下はレジシスタプレッシャをさらに上げた `test-regalloc-nreg-4` に対する。 SPARC 上の ACL コンパイラの生成コードであるが、余裕で全ての変数がレジスタに乗っている。(Appendix も参考のこと)

```
 4: 98062004     add  %i0, #x4, %o4
 8: 96062008     add  %i0, #x8, %o3
12: 9406200c     add  %i0, #xc, %o2
16: 92062010     add  %i0, #x10, %o1
20: 90062014     add  %i0, #x14, %o0
24: a0062018     add  %i0, #x18, %l0
28: a206201c     add  %i0, #x1c, %l1
32: a4062020     add  %i0, #x20, %l2
36: a6062024     add  %i0, #x24, %l3
40: a8062028     add  %i0, #x28, %l4
44: 9803000b     add  %o4, %o3, %o4
48: 9803000a     add  %o4, %o2, %o4
52: 98030009     add  %o4, %o1, %o4
  ...
```

4.2 SBCL, LispWorks の場合

ACL ではスピルが発生した `test-regalloc-nreg-4` であるが、一方 SBCL と LispWorks ではスピルが起こらなかった (以下は SBCL のコード)。

```
0AF62E75:       8D4201          LEA EAX, [EDX+1]
      78:       8D4A02          LEA ECX, [EDX+2]
      7B:       8D5A03          LEA EBX, [EDX+3]
      7E:       8D7204          LEA ESI, [EDX+4]
      81:       8D1408          LEA EDX, [EAX+ECX]
      84:       01DA            ADD EDX, EBX
      86:       01F2            ADD EDX, ESI
      88:       C1E202          SHL EDX, 2
```

ACL と同様にプレッシャーを上げ調べてみると、SBCL と LispWorks は共に可能な `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi` の全てを汎用レジスタとして使っていることが分かった。この違いは SBCL と LispWorks のダイナミックリンカーに秘密があった。SBCL のダイナミックリンカーは定数や組み込み関数をメモリの特定番地に書き込む等の工夫により、ACL で必要とされる `esi` と `edi` のベースレジスタを不要にする努力をしているようである。ACL が定数と関数テープルを生成した先の例を SBCL に与えると、以下のようなコードが生成され、ベースレジスタは不要となっている。

```
MOV EDI, [#xB0F24F0] ; 1.23
CALL #x1000140       ; GENERIC-+
```

4.3 レジスタアロケーションのまとめ

ACL はおそらくレジスタ数が多いマシンに最初に実装されたと思われる。そのアーキテクチャに依存したダイナミックリンカーの構造を根本的に変え、`esi` と `edi` も汎用レジスタとして扱うようにしないかぎり、現在のたった 3 個の汎用レジスタではいくらレジスタアロケータのアルゴリズムを改良しようとも現在の多くのスピルは改善されないと思われる。

レジスターアロケーションのアルゴリズムについては、LispWorks のドキュメントにのみカラーリングアロケータを利用していると明記されており ACL と SBCL のアルゴリズムは不明である。

最後 SBCL の命令選択について簡単に触れておく。SBCL の生成コードでは以下のように `LEA` がしばしば使われる。

```
81:       8D1408          LEA EDX, [EAX+ECX]
```

この命令はアドレッシングモードの計算をうまく利用し、EDX = EAX + ECX を行っている。一方 ACL の生成コードでこのパターンを見たことはない。ACL の X86 に対する命令選択部は SBCL ほど努力はしていないと思われる。SBCL が上記のような自動的なリターゲッタブル命令選択系を実装しているのか、あるいは X86 個別に努力をしているだけなのかは不明である

5 浮動少数演算の調査

浮動小数点数は数値計算において極めて重要である。Lisp では整数 (fixnum) と異なり、浮動小数点数はタグ付けでポインタに埋め込むことが出来ず、全てボクシング表現せざるをえない。このことが、Lisp は数値計算が苦手とされてきた大きな理由のひとつである。

しかし、一つの関数内部ならボクシング表現なしに演算を行い、結果を返す場合のみボクシングすればよい。

ただし X86 では 80 ビットの浮動小数点演算のみがサポートされいるので、この最適化は double-float (64 bits) のみに適用される。しかも、8 個ある 80 ビット浮動小数点数レジスタ st(0) 〜 st(7) はは通常のレジスタと異なった風変わりなものであり、演算の第一オペランドはつねに st(0) (st と略記される) になければならず、演算により st は演算結果で、st(1) は演算前の st(2) で、st(2) は演算前の st(3) で、という風に置き換えられる、いわばスタック構造をしている。

よって、レジスターアロケーションも標準的な手法は使えない。しかし、ACL, SBCL, LispWorks ともにこのような最適化を行っている。以下は浮動少数点の加算プログラムである。

```
(defun test-float-add (a b)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type double-float a b))
  (+ a b))
```

の ACL の生成コードは以下のとおり。 (出口でのボクシング化コードは略)

```
 0: dd 42 f6 fldq [edx-10]
 3: dd da     fstp st(2)
 5: dd 40 f6 fldq [eax-10]
 8: dd db     fstp st(3)
10: d9 c1     fld st,st(1)
12: d8 c3     fadd st,st(3)
14: dd da     fstp st(2)
16: d9 c1     fld st,st(1)
18: dd d9     fstp st(1)
```

このように、関数本体ではボクシングされていない生の浮動小数点数がハードの命令により直接処理される綺麗なコードが生成される。SBCL でも同様の最適化により同様のコードが生成されている。

LispWorks ではこの最適化を有効にするためには、(declare (optimize ... , に (float 0) を追加する必要がある。以下が LispWorks の生成コードである。 ACL, SBCL と比較すると冗長である。

```
11:       8B7D08          move  edi, [ebp+8]
14:       DD4705          fldl  [edi+5]
17:       DD4005          fldl  [eax+5]
20:       DEC1            faddp st(1), st
22:       DD5DF8          fstpl [ebp-8]
25:       83EC08          sub   esp, 8
28:       8B75F8          move  esi, [ebp-8]
31:       8975F0          move  [ebp-10], esi
34:       8B75FC          move  esi, [ebp-4]
37:       8975F4          move  [ebp-C], esi
40:       8B7500          move  esi, [ebp]
```

```
43:      8975F8          move  [ebp-8], esi
46:      83ED08          sub   ebp, 8
49:      8B750C          move  esi, [ebp+C]
52:      897504          move  [ebp+4], esi
55:      DD45F8          fldl  [ebp-8]
58:      DD5D0C          fstpl [ebp+C]
```

6 ベンチマーク

ここでは実際に ACL, SBCL, LispWorks の生成コードの実行速度を比較する。

6.1 整数演算とスピル

以下のプルグラムにより整数演算 (fixnum) のみのテストを行う。型宣言によりボクシングはなく演算は全てインライン化されるので、事実上はレジスタアロケータの性能比較である。

```
(defun test-intop-spill (n)
  (declare (optimize (speed 3) (safety 0) (debug 0))
           (type fixnum n))
  (let ((sum 0))
    (declare (type fixnum sum))
    (dotimes (i n)
      (dotimes (j n)
        (dotimes (k n)
          (incf sum))))
    sum))
```

ACL の生成コードは以下のとおり。前述のとおり、ACL ではたった 3 つの汎用レジスタしか使わないので、既にスピルが発生している。

```
 6: 89 75 fc  movl     [ebp-4],esi
 9: 89 5d e4  movl     [ebp-28],ebx
12: 33 db     xorl    ebx,ebx
14: 33 d2     xorl    edx,edx
16: 89 45 d8  movl     [ebp-40],eax        ; EXCL::LOCAL-1
19: 3b 55 d8  cmpl     edx,[ebp-40]        ; EXCL::LOCAL-1
22: 7c 08     jl      32
24: 8b c3     movl    eax,ebx
26: f8        clc
27: c9        leave
28: 8b 75 fc  movl     esi,[ebp-4]
31: c3        ret
32: 89 5d dc  movl     [ebp-36],ebx        ; EXCL::LOCAL-0
35: 33 db     xorl    ebx,ebx
37: 89 45 d4  movl     [ebp-44],eax        ; EXCL::LOCAL-2
40: 3b 5d d4  cmpl     ebx,[ebp-44]        ; EXCL::LOCAL-2
43: 7c 08     jl      53
45: 83 c2 04  addl     edx,$4
48: 8b 5d dc  movl     ebx,[ebp-36]        ; EXCL::LOCAL-0
51: eb de     jmp     19
53: 89 55 d0  movl     [ebp-48],edx        ; EXCL::LOCAL-3
56: 33 d2     xorl    edx,edx
58: 89 45 cc  movl     [ebp-52],eax        ; EXCL::LOCAL-4
61: eb 07     jmp     70
63: 83 45 dc 04 addl  [ebp-36],$4 ; EXCL::LOCAL-0
67: 83 c2 04  addl     edx,$4
70: 3b 55 cc  cmpl     edx,[ebp-52]        ; EXCL::LOCAL-4
73: 7c f4     jl      63
75: 83 c3 04  addl     ebx,$4
78: 8b 55 d0  movl     edx,[ebp-48]        ; EXCL::LOCAL-3
81: eb d5     jmp     40
```

SBCL の生成コードは以下のとおりであり、スピルが全くない、簡潔なコードが生成されている。

```
  0B0D20AA:       31C0            XOR EAX, EAX
        AC:       31FF            XOR EDI, EDI
        AE:       EB20            JMP L5
        B0: L0:   31C9            XOR ECX, ECX
        B2:       EB13            JMP L4
        B4: L1:   31DB            XOR EBX, EBX
        B6:       EB06            JMP L3
        B8: L2:   83C004          ADD EAX, 4
        BB:       83C304          ADD EBX, 4
        BE: L3:   8BF2            MOV ESI, EDX
        C0:       39F3            CMP EBX, ESI
        C2:       7CF4            JL L2
        C4:       83C104          ADD ECX, 4
        C7: L4:   8BDA            MOV EBX, EDX
        C9:       39D9            CMP ECX, EBX
        CB:       7CE7            JL L1
        CD:       83C704          ADD EDI, 4
        D0: L5:   39D7            CMP EDI, EDX
        D2:       7CDC            JL L0
        D4:       8BD0            MOV EDX, EAX
        D6:       8D65F8          LEA ESP, [EBP-8]
        D9:       F8              CLC
        DA:       8B6DFC          MOV EBP, [EBP-4]
        DD:       C20400          RET 4
```

LispWorks の生成コードは以下のとおりで、スピルがありやや冗長である。

```
      20:       33FF            xor   edi, edi
      22:       837DF400        cmp   [ebp-C], 0
      26:       7E3E            jle   L5
      28:       33DB            xor   ebx, ebx
      30:       8B75F4          move  esi, [ebp-C]
      33:       83EE04          sub   esi, 4
      36:       8975F8          move  [ebp-8], esi
L1:   39:       837DF400        cmp   [ebp-C], 0
      43:       7E32            jle   L6
      45:       33C9            xor   ecx, ecx
      47:       8B75F4          move  esi, [ebp-C]
      50:       83EE04          sub   esi, 4
      53:       8975FC          move  [ebp-4], esi
L2:   56:       837DF400        cmp   [ebp-C], 0
      60:       7E2D            jle   L7
      62:       33D2            xor   edx, edx
      64:       8B45F4          move  eax, [ebp-C]
      67:       83E804          sub   eax, 4
L3:   70:       83C704          add   edi, 4
      73:       3BD0            cmp   edx, eax
      75:       7D1E            jge   L7
      77:       83C204          add   edx, 4
      80:       8955F0          move  [ebp-10], edx
      83:       EBF1            jmp   L3
L4:   85:       E8A6A90700      call  20114962
L5:   90:       C9              leave
      91:       89F8            move  eax, edi
      93:       FD              std
      94:       C3              ret
L6:   95:       3B5DF8          cmp   ebx, [ebp-8]
      98:       7DF6            jge   L5
     100:       83C304          add   ebx, 4
     103:       89DA            move  edx, ebx
     105:       EBBC            jmp   L1
L7:  107:       89CA            move  edx, ecx
     109:       3B55FC          cmp   edx, [ebp-4]
     112:       7DED            jge   L6
     114:       83C104          add   ecx, 4
```

```
117:      89CA            move  edx, ecx
119:      EBBF            jmp   L2
```

同一マシン (x86 Linux) で n = 2000 における実行時間は以下のとおりとなり、レジスターに乗るか乗らないかは非常に重要であることが分かる。

SBCL	5.183 sec
LispWorks	7.577 sec
ACL	12.640 sec

6.2 浮動小数演算

以下はあるグラフ描画関数の最も重い部分であり、実際のアプリケーションから引用したものである。ここでは浮動小数点演算が支配的である。

```
(defstruct bmk-kk-position
  (x 0 :type fixnum)
  (y 0 :type fixnum))

(defmacro expt2 (x)
  (let ((y (gensym)))
    `(let ((,y ,x)) (* ,y ,y))))

(defun bmk-kk-Exm (nvec dist L m xm ym)
  (declare (optimize (speed 3) (safety 0) (debug 0) #+lispworks (float 0))
           (type (simple-array t (*)) nvec)
           (type (simple-array fixnum (* *)) dist)
           (type double-float L)
           (type fixnum m xm ym))
  (let ((sum 0.0d0))
    (declare (type double-float sum))
    (dotimes (i (length nvec))
      (unless (= i m)
        (let* ((p (aref nvec i))
               (xi (bmk-kk-position-x p))
               (yi (bmk-kk-position-y p)))
          (declare (type fixnum xi yi))
          (let ((ym-yi (float (- ym yi) 0.0d0))
                (xm-xi (float (- xm xi) 0.0d0)))
            (incf sum (/ (- xm-xi (/
                                   (* L (aref dist m i)
                                      xm-xi)
                                   (sqrt (the (double-float 0.0d0 *)
                                              (+ (expt2 xm-xi)
                                                 (expt2 ym-yi))))))
                         (expt2 (float (aref dist m i) 0.0d0))))))))
    sum))
```

nvec の要素数を 1000 とし、この関数を 10000 回呼び出した時間が以下である。

SBCL	0.400 sec
ACL	2.730 sec
LispWorks	9.440 sec

7 おわりに

いずれの処理系も、ターゲットマシンの実命令、実レジスタを利用するコード生成を行っており、コンパイラとしての基本的 (最低限の) 機能は満たしている。しかし、そのクオリティーについては、ここ

で調べたように大きな違いが認められる。この基本的な機能については、SBCL が最も優れておりついで ACL と LispWorks が続く。

ACL の最大の問題点はコンパイラが使用する汎用レジスタの使い方に無駄があることで、X86 では結果汎用レジスタとしてわずかみっつしか使用出来ず、ほとんどのコードでスピルが多発してしまう点にある。

これはレジスタアロケータの問題というよりシステム全体のレジスタの利用規約の問題である。現在支流である X86 系の CPU でより良いコードを生成するためには、ここを改善する必要がある。

いずれの処理系も、事実上コード最適化を一切行っていない。しかしながら Lisp はすでに汎用言語であり、そういう最適化を期待したいプログラムはごく普通に存在する。Common Lisp はレキシカルスコープが基本であり言語仕様も厳密に定められている。また Fortran の Common や Equivalence 宣言、あるいは C 言語の自由過ぎるポインタ機能といったものはなく、そういった言語で必要とされるような Aliase 解析 (見かけ上異なる変数だが実は同一メモリ番地に割り当てられてしまう変数を見極める) なども不要であり、ここで取り上げた程度の基本的なコード最適化を実装することに、大きな困難はないと思われる。

言い換えると、どの処理系もまだ非常に大きな性能改善が期待出来るといえる。Lisp の開発効率の高さ、開発環境の快適さについては今さら言及するまでもない。Lisp が汎用言語としての地位を確立し、広く使われるためにもここで述べたような最適化を行う高性能な Lisp コンパイラの出現を期待したい。

A セクション 3 への補足

ここではセクション 3 で省略した、共通部分式とループ最適化のテストプログラムに対する SBCL 及び LispWorks の生成コードを参考までに載せる。

A.1 SBCL

以下が `test-common-subexpression-elimination` の生成コードである。減算が 3 回繰り返されており、共通部分式削除は行われていない。

```
0AE9670A:       8BC2            MOV EAX, EDX
      0C:       29F8            SUB EAX, EDI <<
      0E:       8BC8            MOV ECX, EAX
      10:       8BC2            MOV EAX, EDX
      12:       29F8            SUB EAX, EDI <<
      14:       01C1            ADD ECX, EAX
      16:       29FA            SUB EDX, EDI <<
      18:       01D1            ADD ECX, EDX
      1A:       8BD1            MOV EDX, ECX
```

以下が `test-loop-invariant-hoisting` の生成コードである。ループ中で不変な `(+ n 10)` のループ外への移動は行われていない。

```
0AE75C82:       31C0            XOR EAX, EAX
      84:       31C9            XOR ECX, ECX
      86:       EB13            JMP L1
      88: L0:   8BDA            MOV EBX, EDX
      8A:       C1FB02          SAR EBX, 2
      8D:       83C30A          ADD EBX, 10  <<
      90:       C1F802          SAR EAX, 2
      93:       01D8            ADD EAX, EBX
      95:       C1E002          SHL EAX, 2
      98:       83C104          ADD ECX, 4
      9B: L1:   39D1            CMP ECX, EDX
      9D:       7CE9            JL L0
      9F:       8BD0            MOV EDX, EAX
```

A.2 LispWorks

以下が test-common-subexpression-elimination の生成コードである。減算が 3 回繰り返されており、共通部分式削除は行われていない。

```
       0:       8B7C2404          move  edi, [esp+4]
       4:       89FB              move  ebx, edi
       6:       2BD8              sub   ebx, eax       <<
       8:       89FA              move  edx, edi
      10:       2BD0              sub   edx, eax       <<
      12:       03DA              add   ebx, edx
      14:       2BF8              sub   edi, eax       <<
      16:       8D043B            lea   eax, [ebx+edi]
```

以下が test-loop-invariant-hoisting の生成コードである。ループ中で不変な (+ n 10) のループ外への移動は行われていない。

```
       0:       89C1              move  ecx, eax
       2:       33FF              xor   edi, edi
       4:       83F900            cmp   ecx, 0
       7:       7E21              jle   L2
       9:       55                push  ebp
      10:       89E5              move  ebp, esp
      12:       50                push  eax
      13:       33DB              xor   ebx, ebx
      15:       89CE              move  esi, ecx
      17:       83EE04            sub   esi, 4
      20:       8975FC            move  [ebp-4], esi
L1:   23:       8D4128            lea   eax, [ecx+28]  << 28H = 4 * 10
      26:       03C7              add   eax, edi
      28:       89C7              move  edi, eax
      30:       3B5DFC            cmp   ebx, [ebp-4]
      33:       7D0B              jge   L3
      35:       83C304            add   ebx, 4
      38:       89DA              move  edx, ebx
      40:       EBED              jmp   L1
L2:   42:       89F8              move  eax, edi
      44:       FD                std
      45:       C3                ret
L3:   46:       C9                leave
      47:       EBF9              jmp   L2
```

B 64 bits CPU

近年、X86-64 に代表される 64 bit CPU が普及しつつある。経験上、X86 で作成された ACL アプリケーションを X86-64 でリコンパイルするだけで大幅に高速化されることがあった。ここでは ACL の X86 及び X86-64 に対する生成コードを比較し、その理由のひとつを説明する。

もちろん、X86-64 ではレジスタのビット幅が 64 に拡張されているが、より重要なのはレジスタの個数が 16 個に増やされたことである。X86 でスピルした test-regalloc-nreg-4 であるが、X86-64 では以下のように全てがレジスタに割り当てられた。

```
  0: 4c 8b ef       movq       r13,rdi
  3: 49 83 c5 08    add        r13,$8
  7: 4c 8b e7       movq       r12,rdi
 10: 49 83 c4 10    add        r12,$16
 14: 4c 8b df       movq       r11,rdi
 17: 49 83 c3 18    add        r11,$24
 21: 48 83 c7 20    add        rdi,$32
 25: 4d 03 ec       addq       r13,r12
 28: 4d 03 eb       addq       r13,r11
 31: 49 03 fd       addq       rdi,r13
```

同様のプログラムでスピルが起こる限界まで調べてみると、X86-64 では ACL コンパイラは 9 個の汎用レジスタを利用していることが分かった。X86 ではスピルが多かった `test-intop-spill` であるが、X86-64 では以下のような簡潔なコードが生成された。

```
  0: 45 33 ed       xorl       r13d,r13
  3: 45 33 e4       xorl       r12d,r12
  6: 4c 8b df       movq       r11,rdi
  9: 4d 3b e3       cmpq       r12,r11
 12: 7c 0a          jl 24
 14: 49 8b fd       movq       rdi,r13
 17: f8             clc
 18: 4c 8b 74 24 10 movq       r14,[rsp+16]
 23: c3             ret
 24: 45 33 c9       xorl       r9d,r9
 27: 4c 8b c7       movq       r8,rdi
 30: 4d 3b c8       cmpq       r9,r8
 33: 7c 06          jl 41
 35: 49 83 c4 08    add        r12,$8
 39: eb e0          jmp        9
 41: 33 c9          xorl       ecx,ecx
 43: 48 8b d7       movq       rdx,rdi
 46: eb 08          jmp        56
 48: 49 83 c5 08    add        r13,$8
 52: 48 83 c1 08    add        rcx,$8
 56: 48 3b ca       cmpq       rcx,rdx
 59: 7c f3          jl 48
 61: 49 83 c1 08    add        r9,$8
 65: eb db          jmp        30
```

`n = 2000` における実行時間の比較は以下のとおり。

```
        X32         21.344 sec
        X32-64      4.992 sec
```

同一マシンでありながら、スピルのないコードにより約 4 倍高速化されている。

C SMP LispWorks

近年、マルチコア CPU が普及しつつある。マルチコア対応の Lisp 処理系では、一つの処理を Lisp の複数スレッドに分散させ、それらを複数プロセッサで処理することにより、スループットの大きな改善が期待出来る。

ここでは、そのような処理系のひとつ LispWorks 6.0 を 8 CPU のマルチコアマシン Xeon(R) 2GHz で評価した結果を参考までに載せる。

```
(defun parallel-naive-pi (&optional (nproc 1) (ntimes 1))
  ;; Calculates pi using NPROC processes with initial function naive-pi.
  (let ((pis (make-list (* nproc ntimes) :initial-element 0)))
    (format t "Nproc=~d RealTime:" nproc)
    (dotimes (i ntimes)
      (let ((lock (#+allegro   mp:make-process-lock
                   #+lispworks mp:make-lock))
            (nrests (list nproc))
            (time (get-internal-real-time)))
        ;; Create NPROC processes.
        (loop
            for i from 1 to nproc
            for c on pis
            do (mp:process-run-function (format nil "~d" i)
                 ;; Each process records its (actually the same) result
                 ;; to the list, then increments the semaphore and exit.
                 #+lispworks nil
```

```
                    #'(lambda (c)
                        (setf (car c) (naive-pi))
                        (#+allegro   mp:with-process-lock
                         #+lispworks mp:with-lock
                          (lock)
                          (decf (car nrests))))
                    c))
            ;; Wait for the processes to complete.
            (mp:process-wait "wait" #'(lambda (x) (zerop (car x))) nrests)
            (format t " ~s" (/ (float (- (get-internal-real-time) time))
                               internal-time-units-per-second))))
        (format t "~%")
        ;; Returns the mean of results from the processes.
        (/ (apply #'+ pis) (* nproc ntimes))))

   (defun naive-pi (&optional (n  #xffffffff))
     ;; Calculates pi using the Gregory series of N (<= #xffffffff) terms.
     (declare (optimize (speed 3) (safety 0) (debug 0) #+lispworks (float 0))
              (type fixnum n))
     (* 4.0d0 (loop
                 for i fixnum from 1 to n
                 for k fixnum = 1 then (+ k 2)
                 for s of-type double-float = 1.0d0 then (- s)
                 sum (/ s (float k 0.0d0)) of-type double-float)))
```

parallel-naive-pi は比較的重い数値演算 naive-pi を nproc 個のスレッドで走らせ、全計算が終了した時点で実行時間を表示することを、ntimes 回試行するものである。試行する毎に結果のばらつきがあるため、以下では nproc = 1 .. 24, で各 4 回試行を行った結果が以下である。

```
CL-USER 3 > (loop for n from 1 to 24 do (parallel-naive-pi n 4))
Nproc=1 RealTime: 10.282 10.234 10.281 10.235
Nproc=2 RealTime: 10.282 10.281 10.281 10.297
Nproc=3 RealTime: 10.281 10.281 10.282 10.297
Nproc=4 RealTime: 10.281 10.281 10.281 10.297
Nproc=5 RealTime: 10.281 20.563 20.562 20.532
Nproc=6 RealTime: 10.281 10.281 20.578 20.563
Nproc=7 RealTime: 10.281 10.281 20.485 20.515
Nproc=8 RealTime: 10.453 20.532 20.515 30.813
Nproc=9 RealTime: 20.5 20.5 20.078 20.516
Nproc=10 RealTime: 20.109 20.578 20.313 20.343
Nproc=11 RealTime: 20.563 20.531 30.578 20.516
Nproc=12 RealTime: 30.5 30.937 30.453 30.954
Nproc=13 RealTime: 30.593 30.844 20.313 30.796
Nproc=14 RealTime: 30.594 30.844 30.562 30.75
Nproc=15 RealTime: 30.61 30.844 40.687 30.828
Nproc=16 RealTime: 40.813 30.984 21.672 41.187
Nproc=17 RealTime: 41.079 40.859 30.969 31.218
Nproc=18 RealTime: 31.235 41.015 41.094 41.328
Nproc=19 RealTime: 41.125 41.141 41.281 41.406
Nproc=20 RealTime: 37.485 41.39 35.922 41.141
Nproc=21 RealTime: 38.594 41.25 50.89 41.125
Nproc=22 RealTime: 36.391 51.75 38.953 51.516
Nproc=23 RealTime: 51.375 51.531 51.391 39.5
Nproc=24 RealTime: 51.421 33.016 51.375 33.031
```

試行によりなぜかばらつきがあるが、nproc が CPU の個数である 8 以下では理想的な負荷分散が行われており、さらに 9 以上でもゆるやかに負荷が増えていくことが確認出来た。

ACL でもネーティブスレッドを利用出来るが、残念ながら現在はマルチコアに対応していない。 以下は同一条件下における ACL の生成コードの実行結果である。

```
CG-USER(4): (loop for n from 1 to 10 do (parallel-naive-pi n 4))
Nproc=1  RealTime:  9.078  9.062  9.063  9.062
Nproc=2  RealTime: 18.047 18.063 18.046 18.094
Nproc=3  RealTime: 27.078 27.11  27.156 27.187
```

```
Nproc=4  RealTime: 36.157 36.14  36.094 36.188
Nproc=5  RealTime: 45.14  45.219 45.25  45.188
Nproc=6  RealTime: 54.203 54.234 54.141 54.156
Nproc=7  RealTime: 63.406 63.172 63.297 63.219
Nproc=8  RealTime: 72.39  72.266 72.313 72.312
Nproc=9  RealTime: 81.313 81.25  81.406 81.266
Nproc=10 RealTime: 90.297 90.344 90.343 90.391
```

明らかに 8 つのプロセッサへの負荷分散がなされていない。なるべく早い対応を期待したい。