Lisp によるリターゲッタブル コードジェネレータの実装

萩谷昌己
阿部正佳

東京大学情報理工学系研究科

発表内容

- KCl (Kyoto Common Lisp) の話
  - 80 年代
  - フリーで強力な Lisp 処理系の開発と普及（GCLへ）
- ゲーム開発ツールにおける Lisp
  - 90 年代前半
  - リターゲッタブル C コンパイラの KCl による実装
- Coins プロジェクトにおける Lisp
  - 2000 – 2005年
  - S-式ベースの中間言語（LIR)及びリターゲッタブルコードジェネレータの Lisp による実装

KCl (Kyoto Common Lisp) の話

ゲーム開発ツールにおける Lisp

当時の某ゲーム会社の状況

- 当時の開発環境
  - ゲームの実装はアセンブラ
  - Unix, C : ツール開発でぼちぼち使われていた
  - Lisp : 何それ？
- アセンブラ言語の限界
  - より高級で柔軟な言語
  - 許容できる範囲の実行効率
- さまざまな変換ツールの必要性
- ツール作成に向いた言語の必要性

ゲーム開発における Lisp（事例）

- メタルスレイダー グローリー（ファミコン）
  - AGL (Adventure Game Language)
    - 設計＝向井忠氏（現在（有）ガングー）
    - KCl で実装（約８千行）
    - コンパイラ、リンカ、ソースレベルデバッガ、他
- マザー２、シムシティー、他（スーファミ）
  - C コンパイラ
    - KCl で実装（約１万行）
    - リターゲッタブル(65816, V810 (Virtual Boy))
  - データ変換ツール群
    - マザー2で　30個　ほど必要になった

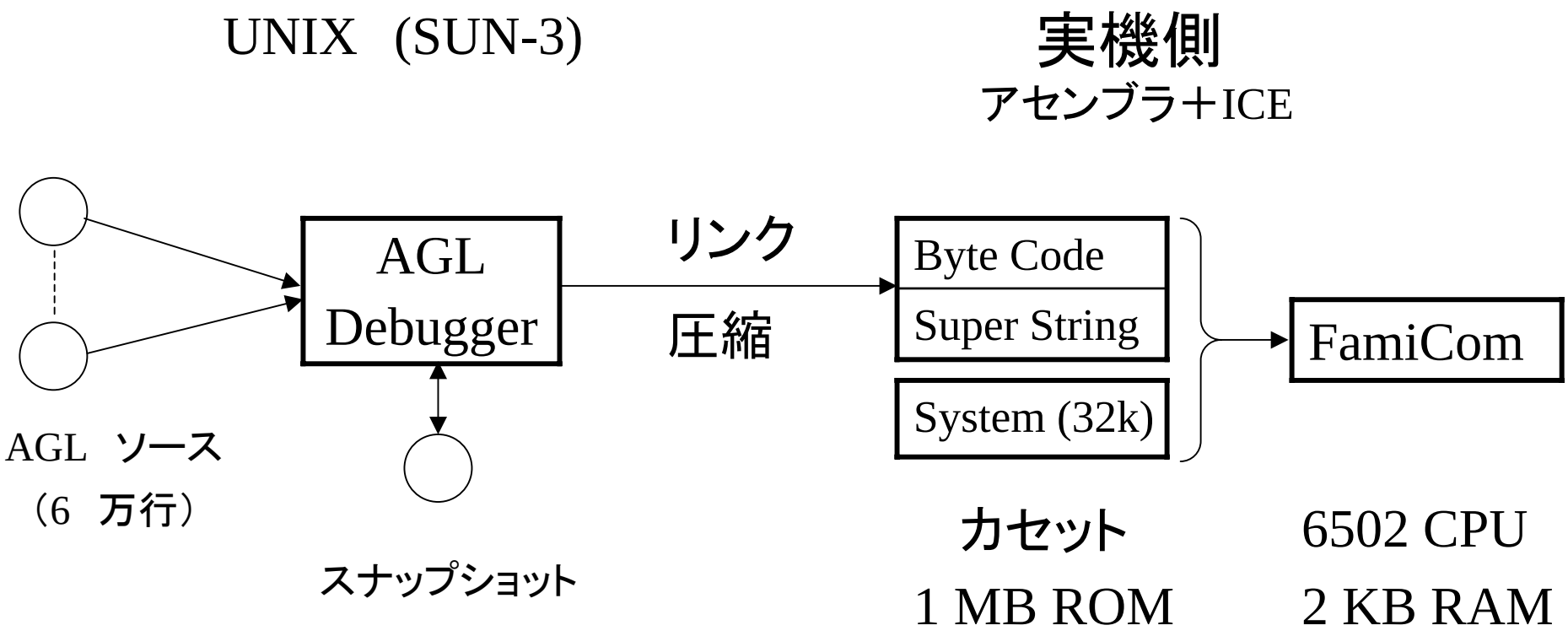
グローリーの開発（1988～1991）

- ファミコンで最大のソフト
  - グラフィックス、シナリオが評価された
  - 2000 年にスーパーファミコンへ移植された
- 開発ツールのほとんどを Lisp (KCl) で作成
- ゲームは全て AGL という言語で記述
  - 処理系実装より先に AGL で作業が進行していた
  - とにかく早く処理系が必要だった（Lisp の高い記述性）
- KCl で書かれた環境上でテスト、リコンパイル
  - インクリメンタルに修正可能 （Lisp のダイナミックリンク）
  - 仕様変更、処理系自体の修正も実行中に可能
  - 前日のデバッグの継続も容易 (Lisp のスナップショット）

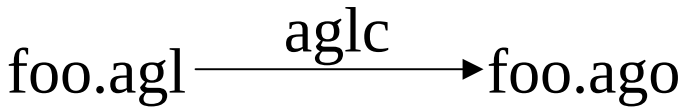
AGL の記述例

```
#scene s1_12
var     pointer1;
begin
 if (save!=255) {
 se_set(ANAUNSU); timer(1);
 mprint ("[振り向き][EL__F][A__A]:
            :「このたびは  ATFこう空を;:
            :[振り向き][A__B]:
            :ごりよう  いただきまして;:
            :まことに  ありがとうございます;:
            :・・・当シャトルは  15. 30分発;:
            :きどうターミナルステーション行きと;:
            :なっております・・・;:
            :りりくまで[EL__A]  もうしばらく  おまち下さい:
            :[振り向き]・・・」:
            :あずさ「[F__C]うき  うき・」[PAUSE][振り向き][A__A]");  }
 command;
  ['みる']{
    ['あずさ']{ ser(pointer1,FIX,b1_13_a,b1_13_2a); exit; }
    ['エリナ']{ bun(b1_13_b); flag_mem1 = 1; exit; }
    ['まど']{ bun(b1_13_c); exit; }
   pmenu; }
  ['はなす']{
    ['エリナ']{ bun (b1_13_d); flag_mem2=1; exit; }
    ['あずさ']{ bun (b1_13_e); flag_mem3=1; exit; }
   pmenu; }
  if(flag_mem1&&flag_mem2&&flag_mem3){
    ['さわる']{ ['エリナ']{ exit_menu; } pmenu; } }
```

グローリーの開発環境



オブジェクトフォーマットも S-式ベース



C コンパイラの開発

- まずスーパーファミコン（65816）用
  - リターゲッタブル（ツリーパターンマッチ）
  - レジスタアロケーション（カラーリング）
- プリプロセッサ以外は全て Lisp (KCl) で実装
- リターゲッタブルな構造
- マザー２　等のゲーム製作で使用された
- 次にバーチャルボーイ（V810）用
  - 素性の良い CPU （RISC）であり、65816 より容易だった
  - パイプラインスケジューラを追加
- C　コンパイラ販売
  - パソコンで動かすために、小さなリスプ（CLisp）を作り、KCl 版をリライト

Coins プロジェクトにおける Lisp

Coins プロジェクトの紹介

- Coins – Compiler INfraStructures プロジェクト
  - 中田、渡辺、佐々 (2000年4月 – 2005年3月)
  - 多言語(C, Fortran, Java, ...)、多機種（SPARC,X86,MIPS,...)
- Coins プロジェクトにおける仕事
  - 中間中間言語（LIR）の設計
  - リターゲッタブルなコード生成系の設計、実装
- 中間言語 LIR とその現状
  - 基本構造はそのままで、使用されている
- コードジェネレータ TMD の現状
  - SPARC, X86　のマシンを記述
  - 高速化のため、Java へリライト

コンパイラの中間言語

- 四つ組（三番地コード）
  - コンパイラの教科書
- アセンブラ言語
  - 特定機種専用のコンパイラ
- 基本演算のツリーのリスト
  - RTL (GCC, R.M. Stallman)
- 中級言語
  - C- - (FPL, Simon P.Jones)

中間言語 LIR の設計目標

- バックエンドで共通に使われる中間言語
  - コード最適化からコード生成まで単一の表現
- プログラミング言語として設計
  - 実装に依存したデータ構造ではなく
  - 実装に依存しないプログラミング言語
- 利用者にとって使いやすいもの
  - 抽象度、簡潔性のバランス

LIR の式

```
(SET I32 (MEM I32 (STATIC I32 “x”))
         (ADD I32 (REG I32 “y”)
                  (CONVSX I32 (REG I8 “z”))))

/* int x; register int y; register char z;
   x = y + z;   */
```

- S-式 ベースの構文
- 型付の演算をノードとする木構造

上位構造の導入

```
(MODULE “foo”
  (SYMTAB
    (“g”   STATIC I32     4 “data” XDEF)
    (“foo” STATIC I32     4 “text” XDEF)
    (“bar” STATIC UNKNOWN 4 “text” XREF))
  ;; int foo(int x, int y){
  ;;   g = bar(x,y+1);
  ;;   return g;}
  (FUNCTION “foo”
    (SYMTAB
      (“x” REG I32)
      (“y” REG I32))
    (PROLOUE (REG I32 “x”) (REG I32 “y”))
      (CALL (STATIC I32 “bar”)
            ((REG I32 “x”) (ADD I32 (REG I32 “y”)
                                    (INTCONST I32 1)))
            ((MEM I32 (STATIC I32 “g”))))
    (EPILOGUE (MEM I32 (STATIC I32 “g”)))))
```

上位構造の導入理由（実レジスタの問題）

```
x = getarg(0)
y = getarg(1)
y1 = y + 1
setarg(0,x)
setarg(1,y1)
g = call bar
return g
```

```
x = %r0
y = %r1
%r0 = x
%r1 = y + 1
%r0 = call bar
g = %r0
```

意味の定義（1）

component	syntax	semantics
L-program	**Lprog**	$\mathcal{P}$
L-module	**Lmod**	$\mathcal{M}$
L-association	**Lalist**	$\mathcal{A}$
L-data	**Ldata**	$\mathcal{D}$
L-function	**Lfunc**	$\mathcal{F}$
L-sequence	**Lseq**	$\mathcal{S}$
L-expression	**Lexp**	$\mathcal{E}$
L-type	**Ltype**	$\mathcal{T}$

意味の定義（2）

$$\mathcal{E} : \mathrm{Lexp} \to \mathrm{Env} \to \mathrm{Mem} \to (\mathrm{Mem} \times \mathrm{Bits})_{\perp}$$

$$\mathcal{F} : \mathrm{Lfunc} \to \mathrm{Env} \to [\mathrm{Bits}] \to \mathrm{Mem} \to (\mathrm{Mem} \times [\mathrm{Bits}])_{\perp}$$

$$\begin{aligned} \mathrm{Env} = \{ & \mathrm{reg} : \mathrm{RegEnv}, \\ & \mathrm{sta} : \mathrm{StaticEnv}, \\ & \mathrm{fra} : \mathrm{FrameEnv}, \\ & \mathrm{lab} : \mathrm{LabelEnv} \} \end{aligned}$$

$$\begin{aligned} \mathrm{Mem} = \{ & \mathrm{pc} : \mathrm{Location}, \\ & \mathrm{pm} : \mathrm{PMem}, \\ & \mathrm{rm} : \mathrm{RMem}, \\ & \mathrm{dm} : \mathrm{DMem}, \\ & \mathrm{tr} : \mathrm{Trace}, \\ & \mathrm{rs} : R \} \end{aligned}$$

$$\mathrm{Bits} = \bigcup_{n=0}^{\infty} \mathrm{Bit}\ n$$

$$\mathrm{Bit}\ n = \{0,1\}^n$$

意味の定義（3）

$$\mathcal{E}[\![(\mathbf{ADD}\ t\ \ x_1\ \ x_2)]\!]\,\rho\sigma = (\sigma_2, \mathrm{bz}\ w(\mathrm{zb}\ v_1 + \mathrm{zb}\ v_2))$$

Where

$$w = \#(t)$$

$$(\sigma_1, v_1) = \mathcal{E}[\![x_1]\!]\,\rho\sigma$$

$$(\sigma_2, v_2) = \mathcal{E}[\![x_1]\!]\,\rho\sigma_1$$

LIR との比較

- RTL
  - プログラミング言語として定義されていない
  - コード最適化の実装は複雑になる
    - 実レジスタが最初から出現
    - RTL 式は常に実在する命令を表現
- C- -
  - 対象は関数型言語
  - コード生成、レジスタアロケーションは対象外
  - リターゲッタビリティの実績が無い

LIR のまとめ

- バックエンドで共通に使える中間言語
- プログラミング言語として定義されている
- コード最適化フェーズで実レジスタは現れない
- 意味が厳密に定義されている

コード生成系 TMD の概要

- DP マッチングに基づく最適なコード選択
  - （木について）複数の可能なコード生成列から DP により最適な命令選択を行う
  - 現在の標準的手法
- 容易なマシン記述
  - 命令選択のみでなく、レジスタの制約等も記述可
  - ターゲットマシンの記述は Burg 系よりも自然

TMDにおける処理の流れ(1)

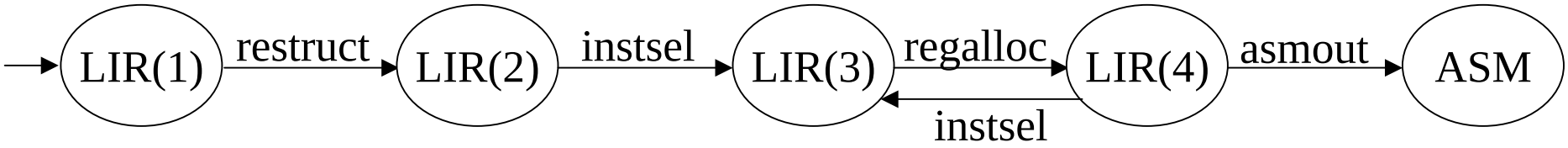
マシン記述

sparc.tmd
x86.tmd

生成されるプログラム

- restruct
  - ターゲット依存のコード変換
- instsel
  - 命令選択
- asmout
  - アセンブラ出力

TMDにおける処理の流れ(2)



- LIR(1)
  - ターゲットマシン独立の中間コード
- LIR(2)
  - コーリングシーケンス等の展開
- LIR(3)
  - ターゲットマシン命令の列
  - レジスタは仮想レジスタ
- LIR(4)
  - 実レジスタ割り当て済み

Burg 系コード生成系

- Twig(Aho,89)、Burg(Fraser,92)、LBurg(Fraser,95)
- アクション付の書き換え規則によるマシン記述
  - reg : reg + reg　{ add $2, $3, $1 , cost=1}
- 与えられたマシン記述から、コード生成系を生成
  - 効率的にパターンマッチを行い指定のアクションを実行するアクション付オートマトン
- DP によりトータルコスト最小のマッチングを行う
  - マッチング高速化手法の違ういくつかの実装がある
  - cost 指定が定数に限られるが、高速である、等

Burg 系コード生成系の問題点

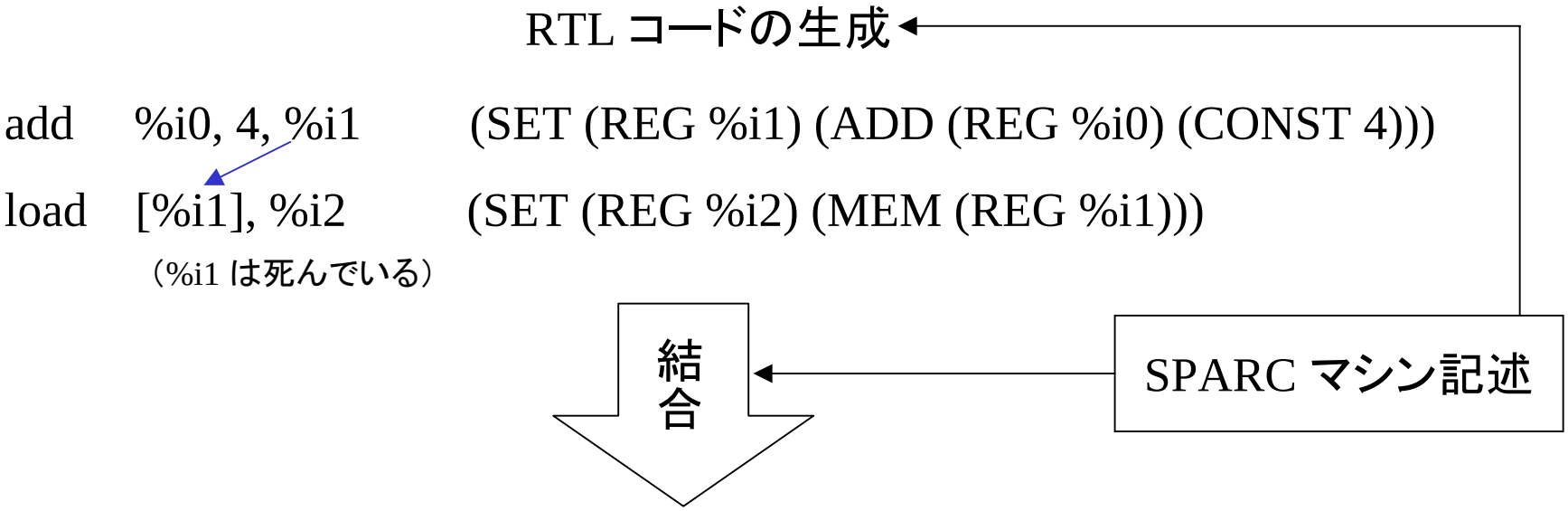
```
mem : *reg
gen : reg
gen : mem
reg : gen + gen {add $1, $2, $0}
void : gen = gen + gen {add $2, $3, $1}
```

- 同一の命令が異なる定義で表れ得る
  - 異なる「型付け」が必要となるため
  - VAX　のようなマシンの記述では冗長で誤りを招く
  - 命令の記述というよりは、コード生成系の記述
- レジスタ制約などの条件はサポートしない
  - 中間コードや、レジスタアロケータからの独立性

GCC 系コード生成系

- GCC(Stallman)、Zephyr Project(Davidson)
- マシン記述は命令の記述
  - 各命令につき、その記述は一つですむ。
- レジスタ制約の記述が可能
- ポストインクリメント等のサポート
- アドレッシングモードは固定されている
- マッチングは最適ではない
  - Burg 系とは異なる GCC 系独自のもの

GCC系の命令選択



load [%i0,4], %i2 (SET (REG %i2) (MEM (ADD (REG %i0) (CONST 4))))

高々隣接 3 命令までの、このような結合を収束するまで繰り返す

この手法により、一つの命令の記述は（基本的に）一つで済むが、最適でない:

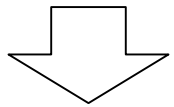
Davidson and Fraser, *Register Allocation and Exhaustive Peephole Optimization*

GCC 系の命令の記述から Burg 系のルールへ

mem : *reg
gen : reg
gen : mem

アドレッシングモードの書き換え規則は Burg 系の記述、命令の書き換え規則は GCC 系の（1対1の）命令記述から生成

```
(defcode add (SET gen (ADD gen gen))
  (asm (add $1, $2, $0)))
```



```
reg : gen + gen {add $1, $2, $0}
void : gen = gen + gen {add $2, $3, $1}
```

コード生成系 TMD の特徴

- 記述は命令の定義
  - 一つの命令について、その記述は一つで済む
- DP による最適な命令選択
  - 内部的に Burg 流の書き換え規則に変換
- レジスタ制約の記述が可能
  - 中間コード、レジスタアロケータとのインタフェースを固定
- 使用レジスタの最小化
  - Seth-Ullman アルゴリズムの組み込み
- 他
  - ミニインタプリタ （湯淺氏の Java 組み込み用 Scheme)
  - 簡単なマクロ機能

マシン記述例
（レジスタ定義）

```
(def *reg-I32* ( (foreach @io (i o)
                  (foreach @n (0 1 2 3 4 5)
                    (REG I32 "%@io@n")))
                 (foreach @n (0 1 2 3 4 5 6 7)
                  (REG I32 "%l@n")) ))
(def *reg-I16* ( (foreach @io (i o)
                  (foreach @n (0 1 2 3 4 5)
                    (SUBREG I16 (REG I32 "%@io@n") 0)))
                 (foreach @n (0 1 2 3 4 5 6 7)
                  (SUBREG I16 (REG I32 "%l@n") 0))))
(def *reg-o0-I32* ( (REG I32 "%o0")))
(def *reg-o1-I32* ( (REG I32 "%o1")))
(def *reg-call-clobbers*
    ( (foreach @n (0 1 2 3 4 5)
       (REG I32 "%o@n"))
      (foreach @n (0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)
       (REG F64 "%f@n")) ))
```

マシン記述例
（アドレッシングモード定義）

```
(defrule reg (REG _ _) (asm (reg-asm-name $0)))
(defrule reg (SUBREG _ _ _) (asm (reg-asm-name $0)))
(defrule addr reg)
(defrule addr con13)
(defrule addr (ADD I32 reg reg) (asm `(add ,$1 ,$2)))
(defrule addr (ADD I32 reg con13) (asm `(add ,$1 ,$2)))
(defrule con (INTCONST _ _) (asm `(con ,(caddr $0))))
(defrule con sta)
(defrule sta (STATIC I32 _) (asm `(sta ,(caddr $0))))
(defrule con13 (INTCONST _ _)
  (cond (<= -4096 (caddr $0) 4095))
  (asm `(con ,(caddr $0))))
(defrule con5  (INTCONST _ _)
  (cond (<= 0 (caddr $0) 31))
  (asm `(con ,(caddr $0))))
```

マシン記述例
（命令定義１）

```
(foreach (@op @code) ((ADD add) (SUB sub) (BAND and) (BOR or) (BXOR xor))
  (defcode @code (SET I32 reg (@op I32 reg rc))
    (asm `(@code ,$1 ,$2 ,$0))
    (cost 1)))


(foreach (@n @l) ((2 1) (4 2) (8 3) (16 4) (32 5))
  ;; シフトによる乗算。
  (defcode mul-sll@l (SET I32 reg (MUL I32 reg (INTCONST I32 @n)))
    (asm `(sll ,$1 (con @l) ,$0))
    (cost 1)))
```

マシン記述例
（命令定義２）

```
(foreach (@op @lib) ((MUL .mul)
                     (DIVS .div) (DIVU .udiv)
                     (MODS .rem) (MODU .urem))
  ;; これらは古い SPARC では o* を壊すライブラリコールになる。
  ;; %o0 = %o0 `@lib` %o1
  (defcode lib@lib (SET I32 reg (@op I32 reg reg))
    (cond (eq $0 $1)
          (regset ($0 *reg-o0-I32*)
                  ($1 *reg-o0-I32*)
                  ($2 *reg-o1-I32*)))
    (asm '(call (sta "@lib")) '(nop))
    (clobber (REG I32 "%o1")
             (REG I32 "%o2")
             (REG I32 "%o3")
             (REG I32 "%o4")
             (REG I32 "%o5"))
    (cost 10)
```

実行例（命令選択）

a = *(_b+1) + c*4 - d*e

（a, c, d, e はレジスタ、_b　はグローバル）

中間コード（LIR）表現

```
(SET I32 (REG I32 "a")
    (SUB I32 (ADD I32 (MEM I32 (ADD I32 (STATIC I32 "_b")
                                        (INTCONST I32 1)))
                      (MUL I32 (REG I32 "c") (INTCONST I32 4)))
             (MUL I32 (REG I32 "d") (REG I32 "e"))))
```

```
SET I32
|  &nregs = 3
| * 16, stmt: (SET I32 ($ 0 reg) (SUB I32 ($ 1 reg) ($ 2 rc)))  ; sub.stmt
|
+--REG I32 "a"
|     &nregs = 1
|   * 0, reg: (REG ($ -1) ($ -1))  ; reg.1
|
+--SUB I32
   |
   +--ADD I32
   |  |  &nregs = 1
   |  | * 5, reg: (ADD I32 ($ 1 reg) ($ 2 rc))  ; add.reg
   |  |
   |  +--MEM I32
   |  |  |  &nregs = 0
   |  |  | * 3, reg: (MEM I32 ($ 1 addr))  ; load-I32.reg
   |  |  |
   |  |  +--ADD I32
   |  |     |  * 2, addr: (ADD I32 ($ 0 reg) ($ 1 con13))  ; addr.4
   |  |     |
   |  |     +--STATIC I32 "_b"
   |  |        |     &nregs = 0
   |  |        |   * 0, sta: (STATIC I32 ($ -1))  ; sta.1
   |  |        |   * 0, con: ($ 0 sta)  ; con.2
   |  |        |   * 2, reg: ($ 1 con)  ; load-con32-I32.reg
   |  |        |
   |  |        +--INTCONST I32 1
   |  |           * 0, con13: (INTCONST ($ -1) ($ -1))  ; con13.1
   |  |
   |  +--MUL I32
   |  |  |  &nregs = 1
   |  |  | * 1, reg: (MUL I32 ($ 1 reg) (INTCONST I32 4))  ; mul-sll2.reg
   |  |  | * 1, rc: ($ 0 reg)  ; rc.1
   |  |  |
   |  |  +--REG I32 "c"
   |  |  |     &nregs = 1
   |  |  |   * 0, reg: (REG ($ -1) ($ -1))  ; reg.1
   |  |  |
   |  |  +--INTCONST I32 4
   |  |
   |  +--MUL I32
   |     |  &nregs = 2
   |     | * 10, reg: (MUL I32 ($ 1 reg) ($ 2 reg))  ; lib.mul.reg
   |     | * 10, rc: ($ 0 reg)  ; rc.1
   |     |
   |     +--REG I32 "d"
   |     |     &nregs = 1
   |     |   * 0, reg: (REG ($ -1) ($ -1))  ; reg.1
   |     |
   |     +--REG I32 "e"
   |           &nregs = 1
   |         * 0, reg: (REG ($ -1) ($ -1))  ; reg.1
```

実行例（アセンブラ出力）

```
%o0 = d
%o1 = e
%o0 = %o0 * %o1
V@1 = %o0
V@6 = c
V@2 = V@6 * 4
V@3 = "_b"
V@4 = * (V@3 + 1)
V@5 = V@4 + V@2
V@7 = V@5 - V@1
a = V@7
```

asmout →

```
mov     d,%o0
mov     e,%o1
call    .mul
nop
mov     %o0,V@1
mov     c,V@6
sll     V@6,2,V@2
set     _b,V@3
ld      [V@3+1],V@4
add     V@4,V@2,V@5
sub     V@5,V@1,V@7
mov     V@7,a
```

比較

	Burg	GCC	TMD
最適な命令選択	○	×	○
Seth-Ullman 考慮	×	×	○
同一命令の定義は一箇所	×	○	○
アドレッシングモード定義可	○	×	○
レジスタクラスや制約の指定	×	○	○
ポストインクリメントモード等	×	○	×

	LCC	TMD
SPARC	1163	877
X86	998	650

COINS コンパイラの性能

	GCC-O2	COINS-O2
164.gzip	2.74 *	4.02
171.swim	2.67	2.50 *
175.vpr	3.57 *	5.82
181.mcf	0.741	0.738 *
197.parser	5.12 *	6.77
isort	179.92	108.09 *
ssort	184.55 *	247.39
heap	68.81 *	69.27
shell	63.57	63.47 *
queen	69.46	67.91 *

（*：勝ち）

コンパイラ記述言語としての Java

- フィールドを直接いじらないのがスタイル？
  - 一つのフィールドにつき setXX、getXX
  - データ構造が極めて理解しにくい
- 継承はあまり役に立たないのに…
  - 直和型の代用程度
- 標準のクラスを使うべし？
  - 使いにくい List
- マクロが無い
- 可変引数
- 中間コードクラスに大量のメソッド
- クラスのバイナリサイズが 64k
- 遅い
- ….

Lisper のための Coins

- Coins は GCC と同程度の性能に達している
- 中間言語 LIR は S-式ベースであり、言語としてきちんと定義されている
- 各最適化フェーズは LIR のプログラム変換として実装されており、中間の LIR も完結した言語として S-式で出力出来る
- Lisper は実装の詳細(Java)を知らなくても、Coins をベースに Lisp で新たな最適化を組み入れることが出来る！