プログラミング言語論2014 # 1 —
コンピュータとプログラミング言語

久野　靖 *

2014.4.10

0　はじめに

本科目「プログラミング言語論」の目的は、プログラミング言語とは何か、プログラミング言語において何が重要か、そのためにどのような工夫がなされているか、プログラミング言語がどのように実現されているのか、などのことがらに見通しを持って頂けるようになることです。これによって、情報科学/工学を専門として来なかった方でも、プログラミング言語に関する概観が持て、必要ならコードを眺めたり読解できるようになることを期待します。

進め方としては、毎回テーマを決めて説明した後、例題をもとに実習をして頂くようにします。さまざまな言語が出て来るので実習は大変と思うかも知れませんが、ゼロから書くというより動かす体験を持ってもらうことを中心にしますので。各回の内容は、おおむね次の順番で計画していますが、進み方によって変更するかも知れません。

- # 1 プログラミング言語の位置づけ、低水準/高水準、手続き呼び出し
- # 2 構文と意味、抽象構文木、インタプリタ/トランスレータ
- # 3 記号処理と Lisp、動的な言語、マクロ、環境
- # 4 関数型、参照透明性、高階性
- # 5 (検討中)

評価方法ですが、各回に出席していただいて議論や演習に参加していただくことと、最終レポートとを併せたもので評価します。最終レポートは、各回の資料に掲載されている演習問題から自分で関心を持ったものを 1 つ (5 回分全部から 1 つ) 選び、実施内容をレポートとして報告してもらうものです。では、これから 5 回よろしくお願いします。

1　コンピュータの仕組みと低水準言語

1.1　コンピュータとプログラム

皆様は「コンピュータとは何ですか?」という問いにどのように回答されますか。「○○ができる」「○○に使う」のような説明ではなく「定義」するとしたらという意味ですが。

今日のコンピュータが扱う情報は、すべてディジタル情報であり「0 と 1 の並び」つまり**ビット列** (bit sequence) で表現されています。たとえば数値「5」であれば「101、「3」であれば「11」のように **2 進法** (binary system) で表現できます。そして、足し算という計算をするなら、この 2 つのビット列を加工した結果として「1000」が返され、これが「8」を表す、というふうにするわけです。以

*経営システム科学専攻

上は簡単な例ですが、画像でも音楽でも、同様にして (非常に長い) ビット列として表現し、それを加工することで (拡大とか音色の補正とか) さまざまな処理を行います。

では、コンピュータの中ではどのようにして「ビット列の加工」を行っているのでしょうか。数の足し算のように基本的なものについては、そのための回路が組み込まれています。しかし、コンピュータに行わせたいような「ビット列の加工」のバリエーションすべてをあらかじめ電子回路として組み込んでおくことは明らかに不可能です。

そこで、コンピュータでは特定の計算を電子回路に組み込む代わりに、電子回路ではごく基本的なビット列の加工だけを用意しておき、それらを後で自由に組み合わせることによってさまざまな加工を行います。しかし、各種の加工を行う回路を「自由に組み合わせる」には、そういう配線を行う必要があるのでは、と思いますね? そこが実は重要なポイントで、現代のコンピュータでは配線を行う代わりに「どう組み合わせるか」を「命令として与える」ことで自由な加工を実現しています。ここがまさに、コンピュータを作り出した人たちの偉大なアイデアなのです。具体的には、次のようにしています (図 1)。

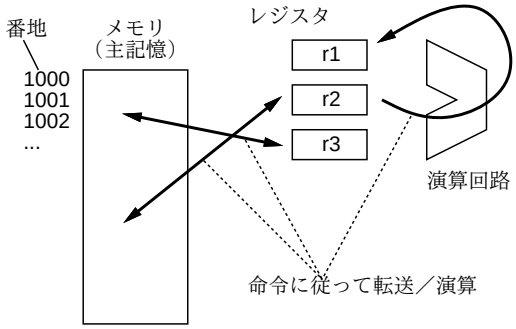


図 1: コンピュータと命令

- すべてのデータは**メモリ** (memory) ないし**主記憶**に格納する。メモリには**番地** (address) がついていて、番地を指定してデータを格納したり、取り出して来たりできる。
- CPU はデータをメモリから取り出し、レジスタと呼ばれる記憶場所を利用しながら、さまざまな加工を行い、結果をまたメモリに戻す。
- データの移動や加工はすべて**命令** (instruction) で指定する。

実際には 1 つの命令では簡単な動作 1 つしかできないので、命令を並べてそれを順番に実行していくことで、より込み入った動作を行わせます。この、命令を並べたものが**プログラム** (program) なのです。プログラムもまたメモリに格納されており、データの一種として扱えます。この方式を**プログラム内蔵** (stored program) 方式、ないし考案した科学者の名前から**ノイマン型** (Neuman architecture) などと呼びます。

1.2 小さなコンピュータのシミュレータ

本物の CPU の命令は複雑なので、まず、簡単化した (架空の)「小さなコンピュータ」を想定して、その命令を動かしてみましょう。この小さなコンピュータは JavaScript で実現されていますが (図 2)、そのことは置いておいて、とりあえず使ってみます。[1]

ここで「プログラム」と記された欄に、1 行に 1 つずつ、命令を書いて行きます。たとえば、次の 7 行のコードを打ち込んでから「実行」してみましょう (**コード** (code) というのは、「プログラムないしその断片」を表す一般的な用語です)。

[1]場所は内部: `http://w3in/~kuno/ecs14/sim.html`,
外部: `http://www2.gssm.otsuka.tsukuba.ac.jp/staff/kuno/ecs14/sim.html` となっています。

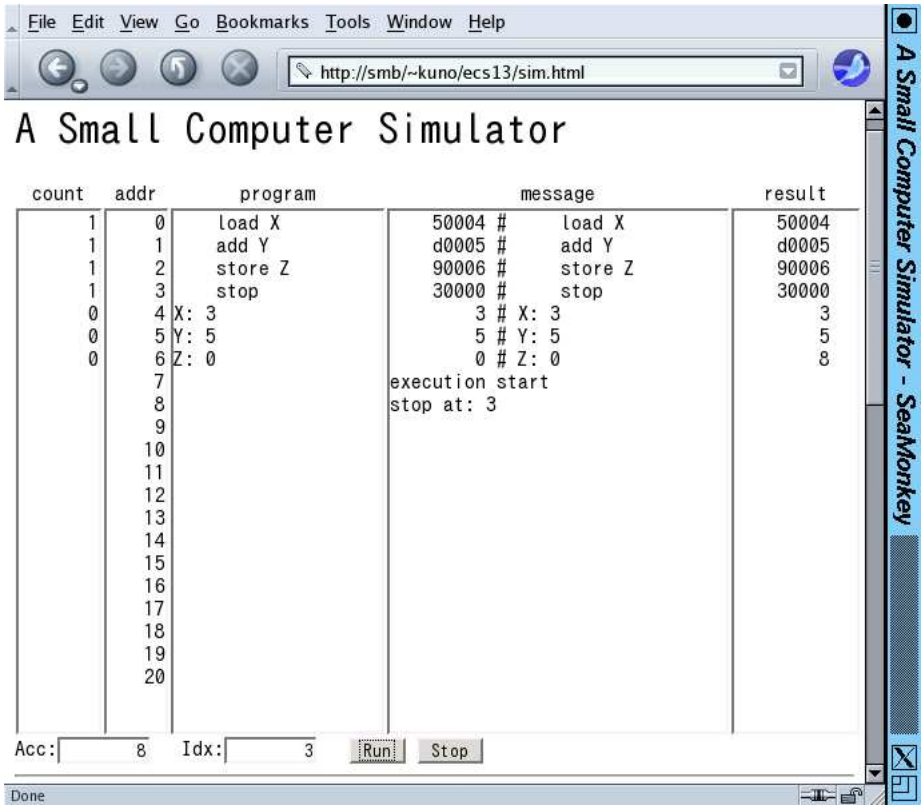


図 2: 「小さなコンピュータ」の画面

```
    load X
    add Y
    store Z
    stop
X: 3
Y: 5
Z: 0
```

ここで「load」は、数値をメモリの指定場所 (この場合は X という名前のついている番地) からアキュムレータ (Acc) というレジスタに取り出して来る命令、「add」は、メモリを指定場所 (この場合や Y という名前のついている番地) から取り出し、それを Acc の内容に足し込む命令、「store」は、Acc の内容をメモリの指定場所 (この場合は Z という名前のついている番地) に格納する命令です。これらの命令はいずれも「どこからどこへ」のように 2 つの場所を本来は指定する必要がありますが、その一方は Acc になっているので場所を 1 つ指定すれば済むのです。そして最後の「stop」は、その名前通りプログラムの実行を停止する命令です。

プログラムの後には、X、Y、Z という名前のついた場所を用意し、それぞれに 3、5、0 という値を入れておきます。そうすると、プログラムを実行した結果、X と Y の値を足した結果 (8 ですね) が Z の場所に格納され、確かに足し算ができています。

私たちが普段使っているコンピュータでは、画面やマウスなどを使ってデータをやりとりしますが、この「小さなコンピュータ」ではごく基本的な命令しか用意していないので、このようにメモリに直接データを用意して動かすようにしています。

このようにコンピュータが持つ命令やメモリ上の場所に名前をつけて表すプログラムの書き方を**アセンブリ言語** (assembly language) と呼びます。アセンブリ言語のプログラムは**アセンブラ** (assembler) と呼ばれるプログラムによってビット列、つまり 0 と 1 だけから成るプログラムに変換され、CPU はそれを実行します。この、CPU が直接実行する形のプログラムのことを**機械語** (machine language)

のプログラムと呼びます。[2]「小さなコンピュータ」では、実行開始時に機械語 (アセンブラの変換出力) を表示するようになっています。

上の例は「足し算」でしたから順番に命令を 4 つ実行すれば終わりでした。しかし実は、プログラムでは「計算した結果によって処理を切り替える」ということが可能であり、これによって複雑な処理が行えます。そのために、命令の中に「分岐 (ジャンプ) 命令」「条件分岐命令」があります。具体的には、通常の命令はその命令を実行し終わると「次の」命令に進むのに対し、分岐命令は番地を指定し、次はその番地の命令の実行に進むようにさせます。そして条件分岐命令は、「Acc が 0 でないならば」のように条件を指定して、その条件が成り立っている時だけ分岐します (成り立っていなければ、次の命令に進む)。

たとえば、今度は X と Y のうち「より大きい値を」求めてそれを Z に入れることを考えます。そのためには、X から Y を引いてみて、マイナスなら Y の方が大きいと分かります。この考えに基づいてプログラムを作ってみましょう。

```
      load X
      store Z
      sub Y
      ifp Skip
      load Y
      store Z
Skip: stop
X:  3
Y:  5
Z:  0
```

「sub」は引き算の命令です。このプログラムではまず、X を Acc に取り出し、とりえあず Z に入れます。次に、Acc の内容から Y を引き算します。ここで、もしプラスなら X の方が大きくて OK ですから、Skip という場所に「飛びます」(ifp は Acc の内容がプラスなら指定した場所に分岐する条件分岐命令です)。プラスでないなら、もう 1 回 Y の値を Acc に持って来て、Z に格納します。いずれにせよ Skip の所に合流して、そこでプログラムは止まります。このように、条件判断して自動的に処理を切り替えることで、コンピュータは複雑な処理が行えているのです。

演習 1-1 「小さなコンピュータ」で以下の処理から 1 つ以上 (できれば全部) 選び、それを実行するプログラムを 1 作成してください。

a. 5 つの値を A、B、C、D、E というラベルの番地に入れておき、その合計を Z というラベルの番地に入れて止まる。データ例: 1, 2, 3, 4, 5 → 結果 15 (16 進では F)。

b. A というラベルのついた番地に入れておいた数値を 5 倍し、結果を Z というラベルのついた番地に入れて止まる。データ例: 4 → 20 (16 進では 14)。

c. 3 つの値を A、B、C というラベルのついた番地にそれぞれ入れておき、その 3 つの最大値を求めて、Z というラベルのついた番地に入れて止まる。データ例: 3, 6, 2 → 結果 6。

いずれにおいても、プログラムがどのように動作するか説明すること。

1.3 ループのあるプログラム

すぐ上にある課題では、5 つの値の合計とか、5 倍とかなので、その数だけ足し算命令を使えば済んでいました。

[2]そして、アセンブリ言語と機械語を併せて**低水準言語** (low level language) と言います。低水準言語とは、CPU の種類が違えば違う書き方が必要となるような言語を意味します。

では、合計に戻って、もっと沢山の数を合計したければどうしましょうか? A、B、C…のように沢山変数を並べていてはプログラムが長くなって大変そうです。そこで、Acc のほかにもう 1 つ、インデックスレジスタ (Idx) というものが用意されています。そして、load 命令の拡張版 loadx では「指定した番地より Idx の値だけ先の場所」からデータを取り出すことができます。これを使って、並んだデータを順番に取り出し、合計して行けばよいのです。プログラムを見てみましょう。

```
       iload 0
Loop:  loadx Data
       ifz End
       add Sum
       store Sum
       iadd 1
       jump Loop
End:   stop
Sum:   0
Data:  1
       2
       5
       0
```

Data というラベルの後に数行ぶんのデータがありますが、これを順番に持って来て合計するわけです。

「iload」命令は Idx に指定した値 (この場合は 0) をロードします。次に、load 命令で Acc に値を持って来ますが、最初は Idx は 0 なので、ちょうど Data の場所の値が持って来られます。もしその値が 0 なら、これは終わりの印なので (合計を取りたいのに 0 をデータに入れる必要はないでしょうから)、End へ分岐します。そうでなければ、Acc の値と Sum を加え、その結果を Sum に入れます。続いて、「iadd」命令で Idx を 1 増やしてから、「jump」命令で無条件に Loop へ戻ります。すると、次は Data の次の場所から値が取り出せるわけです。これを繰り返して次々に値を足して行き、0 が現れたら End へ来て止まりますが、そのときには Sum には総計が入っています。

このプログラムの「肝」は、前方向への分岐命令を使って一群の命令列を「繰り返し」実行させることで、短いプログラムでも沢山の処理を行わせられる、というところにあります。これが、今日のコンピュータの重要な原理だと言えます。最後に、この「小さなコンピュータ」が持っている命令の一覧を掲載しておきます (表 1)。[3][4]

演習 1-2 「小さなコンピュータ」で以下の処理から 1 つ以上 (できれば全部) 選び、それを実行するプログラムを作成してください。

a. 正負とりまえぜて複数の整数を与えておき (0 が終わりの印)、それらの数値の「絶対値の合計」を求める。データ例:「1 -2 3 -4 5 0」→結果例: 「15」
b. 2 つの整数 (いずれも 0 以上) を与えておき、その 2 つの積 (掛けた結果) を求める。データ例:「3 5」→結果例:「15」
c. 複数の整数を与えておき (0 が終わりの印)、それらの数値の「最大」を求める。データ例:「-3 -5 -2 0」→結果例:「-2」

いずれにおいても、作成したプログラムに上記の入力例を与えた時、どの命令は何回実行されるか「理由を含めて説明」してください。

[3]条件分岐命令が沢山ありますが、その覚え方は次のようになります。ifz〜「if zero」、ifnz〜「if not zero」、ifp〜「if positive」、ifn〜「if negative」

[4]「コード」はその命令を表現するビット列です。すべて 2 つずつコードがありますが、大半の命令はどちらでも動作は同じです。値を取り出す命令 (add、sub、mul、load) のみ、「命令の後に指定した数値を取り出す」「命令の後に指定したメモリの場所から取り出す」の 2 通りに分かれています。

表 1: 「小さなコンピュータ」命令一覧

名前	コード	命令の動作
nop	00,01	何もしない
stop	02,03	プログラムの実行を停止
load	04,05	Acc に値を持って来る
loadx	06,07	〃(値/番地に Idx を足す)
store	08,09	Acc の値を格納する
storex	0a,0b	〃(番地に Idx を足す)
add	0c,0d	Acc に値を足す
sub	0e,0f	Acc から値を引く
iload	10,11	Idx に値を持って来る
iadd	12,13	Idx に値を足す
isub	14,15	Idx から値を引く
ifz	16,17	Acc ＝ 0 なら分岐
ifnz	18,19	Acc ≠ 0 なら分岐
ifp	1a,1b	Acc ＞ 0 なら分岐
ifn	1c,1d	Acc ＜ 0 なら分岐
jump	1e,1f	無条件に分岐
neg	20,21	Acc の符号を反転

1.4 本物の CPU の命令実行速度

今度は CPU の「速さ」について、少し調べてみましょう。「小さいコンピュータ」はソフトで実現しているため、命令の実行速度もごく低いものでした (1 秒間に 100 命令とか)。本物の CPU ではどれくらいの速さで命令を実行しているのでしょうか? 実際に測ってみましょう。まず、10G(百億) 回ループを周回するだけの AMD64 プログラム (アセンブリ言語) を用意しました。

```
.globl main
main: movl    $1000000000, %eax
loop: subl    $1, %eax
      jg     loop
      ret
```

「.globl main」というのは外部から参照できるラベルを指定するというアセンブリ言語の指示です。そして main というラベルから実行が始まります。最初の movl 命令では、レジスタ%rax に値 1G(十億) を設定します。2 番目の命令 subl では、そこから「1」を引きます。3 番目の命令 jg では、「引き算の結果が 0 より大きければ」loop へ分岐します。なので、subl と jg がともに 1G 回、繰り返し実行されます。最後の ret はこのプログラムを呼び出したところに戻り、プログラムを終了させます。

では、これを動かしてみましょう (以下の演習は Unix 上で行ってください)。まず、上のプログラムを「test.s」というファイルに打ち込みます (打ち込むのには Emacs などのエディタを使います)。なお、最後が「.s」で終わるのはアセンブリ言語のプログラムという意味になります。その後は、次のようにして実行させ、また時間を測ります (「%」は向こうから表示してくるコマンドプロンプトを表すので、打ち込まないように)。

```
% gcc test.s    ←機械語に変換 (アセンブラが動作)
% time ./a.out    ←時間計測しつつ実行
real    0m0.511s ←プログラムの実行実時間
user    0m0.504s ←プログラムの CPU 消費時間
```

```
sys     0m0.001s
%
```

およそ 0.5 秒で 2G(20 億) 命令を実行したわけですから、これを試したマシンの CPU の平均命令実行速度は 1 秒あたりおよそ 4G(40 億) 命令ということになります。また、1 命令あたりの平均所要実行時間は $\frac{1}{4\times10^9} = 25 \times 10^{-11}$ 秒ということになります。

ここまでで、コンピュータの原理や CPU の機能、ハードウェアの構成について説明してきましたが、結局コンピュータの何がこれほどのインパクトを世の中に与えているのでしょうか? いくつか挙げてみましょう。

- 汎用的なディジタル情報 — 世の中の多くの情報は、ディジタル表現することができ、その結果、コンピュータで扱うことができる。
- 汎用的な処理装置 — コンピュータは、プログラムを取り替えることで、ディジタル情報の「どのような」処理であっても、(その処理の方法が分かっている限り) 行うことができる。
- コンピュータの低価格化 — VLSI 製造技術の発達により、コンピュータはどんどん安価に作ることができるようになってきた。その結果、どこでも専用の機械や電子回路を組み立てて使うより、コンピュータを組み込んでソフトで処理を記述する方が安くて柔軟に処理できるようになった。
- コンピュータの高速化・大容量化 — ハードウェア技術の進化により、これまでは「扱えなかった」「計算できなかった」処理がどんどん実用的な時間でこなせるようになってきている。

このようにして、コンピュータ技術の発達が世の中に大きな変化をもたらしていることを指して、**ディジタル革命** (digital revolution) と呼ぶこともあります。

演習 1-3 gcc コマンドで、上の方法で時間計測を行いなさい。その上で、次の課題から 1 つ以上 (できれば全部) 解答しなさい。

a. 計測した CPU の、1 秒間あたりの平均命令実行数を示しなさい。また、1 命令あたりの平均実行所要時間を示しなさい。

b. 上の例では、2 つの命令とも同じ回数だけ実行していた。これを手直しした版 (たとえば、2 回引き算してから条件分岐する) についても同様に計測し、それぞれの命令の平均実行所要時間を求めよ。

c. 上の例では、アキュムレータ (`%eax`) を使って計算をしていた。代わりにメモリを使った版を示す。これについて計測をおこない、レジスタとメモリで `subq` 命令の実行速度がどれだけ違うか調べよ。

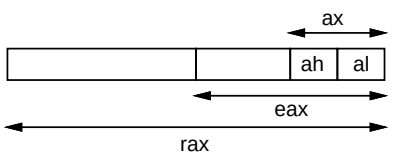
```
.globl main
main: movl    $1000000000, mem
loop: subl    $1, mem
      jg      loop
      ret
     .data
mem: .long  0
```

いずれも、計測した数値をまず示し、その上で計算式を示し、計算結果に基づいて論じること。

1.5 補足: AMD64 アーキテクチャの汎用レジスタ群

ここで少し補足として、先のコードに出てきた AMD64 アーキテクチャのレジスタ群について説明しておきます。「小さなコンピュータ」では Acc と Idx という 2 つのレジスタしか無かったのですが、

AMD64 ではデータを入れる目的に使えるレジスタは 64 ビットのものが 16 本あります。そしてこれらのうち 4 本は、レジスタの一部 (下 32 ビット、下 16 ビット、下 1 バイト、および下から 2 バイト目) を読み書きできます。これらの使い分けは図 3 にある別名を利用します。



8	8	16	32	64
ah	al	ax	eax	rax
ch	cl	cx	ecx	rcx
dh	dl	dx	edx	rcx
bh	bl	bx	ebx	rcx
		sp	esp	rsp
		bp	ebp	rbp
		si	esi	rsi
		di	edi	rdi
			r8d	r8
			○	○
			○	○
			○	○
			r15d	r15

図 3: AMD64 の汎用レジスタ群

AMD64 ではその名前通り、アドレス (メモリ番地) が 64 ビットになっています。このため、メモリ番地を格納する目的でレジスタを使う場合は 64 ビットで使用します。具体的にはスタックポインタ (`%rsp`) とベースポインタ (`%rbp`) が常にそうなります (これらのレジスタの用途については後で述べます)。一方、整数データは 32 ビットが使われることが多いです。

2 プログラミング言語の基本機能

2.1 高水準言語の基本要素

前節まででは CPU が持つ命令語に対応した低水準言語を扱って来ました。しかし低水準言語によるプログラミングは非常に繁雑なため、今日ではより抽象化された言語記述を行う、**高水準言語** (high leve languages) の使用が一般的になっています。[5] ではここから、実際の高水準言語の話に入って行きましょう。C 言語で書かれた次のプログラムを見てみてください。

```
#include <stdio.h>

int main() {
  int x, y;
  printf("x> "); scanf("%d", &x);
  printf("y> "); scanf("%d", &y);
  while(x != y) {
    if(x > y) { x = x - y; }
    else      { y = y - x; }
  }
  printf("%d\n", x);
}
```

演習 1-4 このプログラムには、互いに異るどのような「概念」が登場しているでしょう? できるだけ多く列挙してみてください。

[5] 高水準言語の定義は、CPU の種別などに依存しない記述を行うプログラミング言語、ということになります。

多くの概念がありますが、そのうち最も基本的なのはコンピュータの要素や CPU の動作に直接対応するものだと言えます。具体的には次のものがあります。

- 変数 — コンピュータのメモリ上の場所に対応
- 変数参照 — メモリから CPU へのデータ転送 (ロード)
- 変数代入 — CPU からメモリへのデータ転送 (ストア)
- 演算 — CPU が備えている演算命令の実行に対応

これらは、CPU が持つ個々のデータ移動命令や演算命令に対応させられます。

ただし、上に挙げたものだけしか無かったとすると、CPU は順番に命令を実行するだけで、繰り返しや条件判断ができません。それらを追加しましょう。

- 比較演算 — CPU が備える比較命令の実行に対応
- while 文、if 文 — 条件分岐命令、分岐命令の配置や行き先ラベルの配置に対応

こちらについては、「小さなコンピュータ」でやったように、ラベルと条件分岐命令を組み合わせて枝分かれや繰り返しを構成することになります (図 4)。

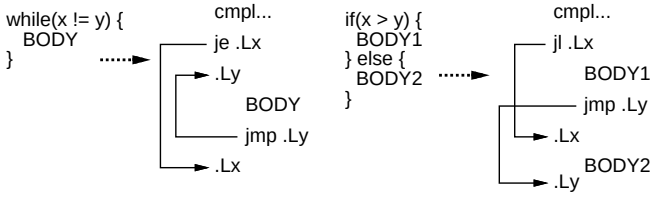


図 4: C の制御構造とアセンブリ言語コードの対応

演習 1-5 上の C プログラムを打ち込んで動作を確認した後、アセンブリ言語を表示させて内容を確認せよ。それが済んだら次のものを調べてみよ。

a. else の無い if 文、do-while 文、switch 文などの制御構造はアセンブリ言語でどのように変換されているか調べよ。
b. 配列の読み書きはアセンブリ言語ではどのように翻訳されるか調べよ。
c. その他、自分の興味のある C の言語要素について、それがどのように翻訳されるか調べよ。

2.2 手続き

制御構造と並んで重要な「CPU に直接はないけれどプログラミング言語によって人間が扱いやすい概念を提供してくれる」ものは何だか分かりますか? それは手続きですね。手続き (C では関数、Java ではメソッド) とはそもそも、何でしょう? プログラミング言語の側から見れば、次のようになるかと思います。

1. さまざまな箇所から名前を指定して呼び出すことで、手続き本体として書かれたコードを複数の箇所から利用できる。
2. 呼び出す際に複数の値 (実引数) を渡すと、その値が本体内から参照でき、これに応じて呼び出し箇所ごとに異るニーズに対応させられる。また返値を返すこともできる。

これがなぜ有難いかというと、「一連の動作に名前をつけて定義する」ことで「抽象化」が行えるから、といえます。抽象化とは、細部を見ずに全体を俯瞰して捉えるということですから。つまり、面倒な動作であっても一度実現して手続きとして定義してしまえば、以後はその名前を呼ぶだけで使える、ということに価値があるわけです。

演習 1-6 C 言語 (または別の好きな言語でよい) で、次の手続きを作って動かせ。それに基づいて、手続きの利点について検討しなさい。

a. 正の整数をパラメタとして受け取り、その整数が素数であるかどうかを調べる。
b. 正の整数をパラメタとして受け取り、2 以上その値以下の範囲ですべての素数を打ち出す。
c. 正の整数をパラメタとして受け取り、2 以上その値以下の範囲で「2 つの整数で一方が他方より 2 だけ大きい」ような組をすべて打ち出す。

2.3 手続きの実装

さて、では手続きはどのようにして実現されているのでしょうか。それは、CPU がある程度までこのような機能をサポートしてくれることを活用します。具体的には次の通り。

- 「call 行き先」命令を実行すると、この命令の次の命令の番地 (戻り番地) を保存してから、行き先として指定された番地にジャンプする。
- 「ret」命令を実行すると、保存してあった戻り番地を取り出して、そこへジャンプする。

この 2 つを図 5 のように組み合わせて使うことで、複数の箇所のどこからから呼び出された場合でも、呼び出した「次の」場所に戻って実行を続けることができるわけです。

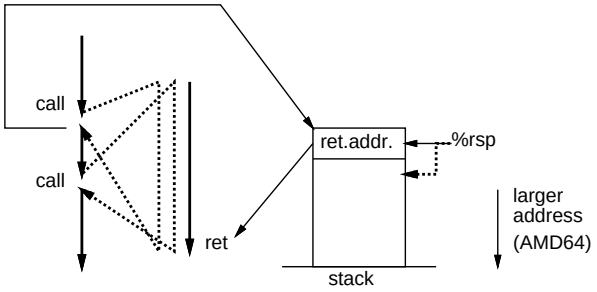


図 5: call 命令と ret 命令

しかし、戻り番地はどこに「保存」するのでしょうか。それは先に出てきたスタックポインタの周辺を使います。call 命令は、まずこのスタックポインタを 8 バイトずらして (減らして) から、その場所に戻り番地を格納します。一方、ret 命令は戻り番地を取り出し、スタックポインタを 4 バイト戻し (増やし) ます。スタックポインタが指すあたりを「スタック領域」と言います。

さらに、AMD64 ではベースポインタと呼ばれるレジスタが戻り番地格納位置のすぐ上を指していて、この 2 つのレジスタの「間」が 1 つの手続きが扱う領域の範囲 (**スタックフレーム**) となっています (図 6)。[6]

呼び出された手続きの入口では、まず call 命令が積んだ戻り番地のすぐ上にこれまでのベースポインタを積み、ベースポインタはスタックポインタのコピーとします。これによって、ベースポインタの指す先は次々に「連鎖」になり、必要なときに前の手続きのフレームをたどれる状態になります。その後、スタックポインタを必要なだけ「ずらす」ことにより、自分のフレームを確保します。手続きの出口では、スタックポインタを逆に「ずらす」ことで復元し、保存してあった旧ベースポインタを復元し、ret 命令で元の場所に戻ります。

以上をまとめると、手続きとはおおむね次のようなコードになります。

[6] スタック領域は具体的にどこなのかという疑問があると思いますが、OS ごとに適切な場所を用意した状態で実行が開始されるようになっているので、どの番地ということを自分で意識する必要はまずありません。

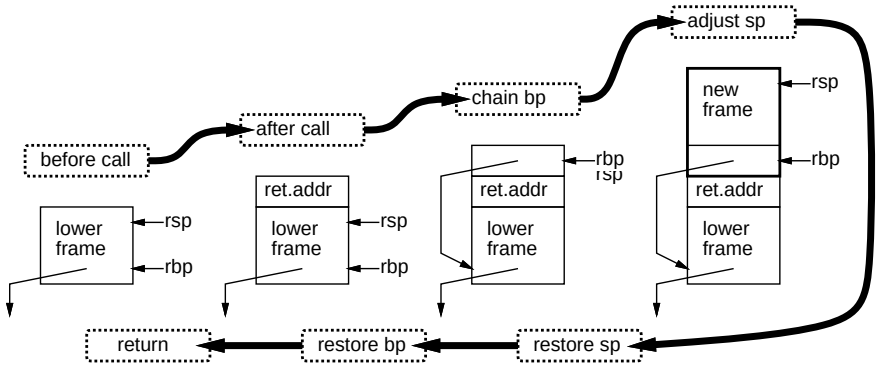


図 6: call/ret とスタックの調整

```
sub: pushq  %rbp          ←%rbp を保存
     movq   %rsp, %rbp    ←%rbp を%rsp と同じ値に
     subq   $定数, %rsp   ←%rsp をずらす
     (…手続き本体のコード…)
     addq   $定数, %rsp   ←%rsp を元の値に
     popq   %rbp          ←%rbp を復元
     ret                  ←戻る
```

なのですが、ベースポインタの保存/復元は、自分から他の手続きを呼ばないような場合には省略することができます。さらに、ローカル変数を扱わないときはスタックの調整も省略できます。

2.4 パラメタと返値

さてでは、引数 (パラメタ) や返値はどうやって渡すのでしょうか。IA32(32 ビットのインテルアーキテクチャ) ではレジスタが少なかったため、すべてメモリで渡していましたが、AMD64 では次のようにレジスタを使って渡します (これより個数が多い場合はしかたないのでスタック上にそのための領域を取りますがそれについては省略)。

```
%rdi / %edi   パラメタ 1
%rsi / %esi   パラメタ 2
%rdx / %edx   パラメタ 3
%rcx / %ecx   パラメタ 4
%rax / %eax   返値
```

たとえば、2 つの数値を受け取り、その和を返す関数 addasm() をアセンブリ言語でできるだけ書いてみます。

```
   .globl addasm
addasm:
  movq %rdi,%rax
  addq %rsi,%rax
  ret
```

第 1 パラメタを%rax(戻り値レジスタ) にコピーし、それに第 2 パラメタを加算しています。[7] 簡単のためスタックポインタ/ベースポインタの調整は省略しています。main は次の通り。

[7] ここでは 64 ビット命令でやっていますが、32 ビット演算でも別に構いません。

```
#include <stdio.h>
int addasm();
main() { printf("%d\n", addasm(5, 3)); }
```

これを実行すると当然「8」が出力されます。

演習 1-7 次のような (C 言語から呼べる) 手続きを作り、それがきちんと呼べていることも確認してみなさい。

a. 「右ローテートシフト命令」(rorl)、「左ローテートシフト命令」(roll) は C 言語では直接実現しにくいが場合によっては便利なビット操作である。これらの命令を呼んで結果を返すような手続きを作れ。

b. スタックポインタの値をそのまま (整数へのポインタとして) 返すような手続きを作れ。それを用いて、整数のローカル変数をループで順番に取り出して出力するような例題を書いてみなさい。

c. b. の手続きを用いて、戻り番地の値を 16 進法で表示させてみなさい。複数箇所から呼んだ場合、戻り番地にどのような違いがあるか検討してみなさい。

3 制御スタックと setjmp/longjmp

3.1 asm 指示による命令の挿入

ここまでは、アセンブリ言語のコードを別のファイルにしていましたが、実は gcc などでは **asm 指示** asm directive を使うことで、C 言語コードの真ん中にアセンブリ言語コードを挿入できます。例を見てみましょう。

```
#include <stdio.h>
sub1(int x) {
  printf("sub1. %d\n", x);
//asm("jmp tmp");
}
sub2(int y) {
//asm("tmp:");
  printf("sub2. %d\n", y);
}
main() {
  sub1(99);
  printf("main.\n");
  sub2(88);
  return 0;
}
```

ここでは asm 指示はコメントアウトされています。このまま実行したところ、結果は (当然) 次のようになります。

```
sub1. 99
main.
sub2. 88
```

では、コメントを外したらどうなるでしょう? (間) 次の通り。

```
sub1. 99
sub2. 99
main.
sub2. 88
```

当たりましたか? 「gcc -S」でアセンブリ言語を出力して見ればなぜそうなるかはよく分かると思います。

演習 1-8 上の例を動かして確認しなさい。また、後半の結果がなぜそうなるのか説明を書きなさい。

3.2 大域脱出

隣のサブルーチンにジャンプするとかの冗談はさておき:-)、手続きの基本原則は「入れ子構造」です。つまり、A → B → C → D の順に手続きを呼んだら、必ず逆順に戻って来ます。これは、「一連の手順を抽象化する」という本来の目的から見れば当然のことなのですが、場合によってはちょっと不便なこともあります。

具体的には、手続きを呼んだ先で何らかのエラーがあり、処理を中止して戻って来る場合がそうです。手続き呼び出しの深い連鎖の中でそのようなことをしようとすると、全部の呼び出しにおいて「呼び出し先でエラーがあったかどうかをチェックして、エラーだったら以後の処理を中止して呼び出し元にもエラーを返す」必要がありますが、それを手で統一的にコーディングするのはなかなか面倒です。

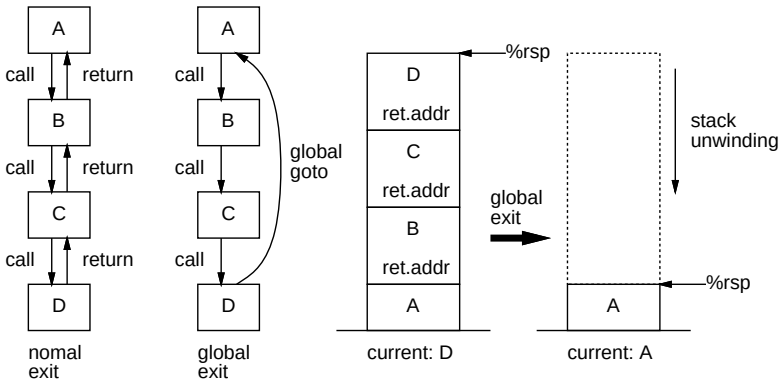


図 7: 大域脱出とスタックの巻き戻し

この面倒を避けるやり方として、エラーの出た箇所 (D) から途中を飛ばして一気に元の場所 (A) に戻ってくることが考えられます。これを「大域脱出」と呼びます (図 7 左)。このようなものの必要性が認められたことから、C++や Java など最近の言語では例外機構が備わっているわけですが、これを C でどうやるか考えてみてください。

もちろん、単純に D の中から A の中にジャンプするのでは駄目です (なぜか?)。きちんと、レジスタ群 (とりわけスタックポインタ) を A が動いているときの状況に戻さないといけません (図 7 右)。これを、積まれているスタックを積まれていない状態に戻すことから、「スタックの巻き戻し」と呼びます。ですが、実際にはあまり難しく考える必要はなくて、(1)A の中で現在のレジスタ群を保存し、(2) 大域脱出のときに保存してあったレジスタ群を戻せば、元に戻ります。

C の標準ライブラリでは、これらを実現するために、(1) は setjmp、(2) は longjmp という名前の関数を呼びます。実際に見てみましょう。

```
#include <stdio.h>
#include <setjmp.h>
```

```
jmp_buf buf;
sub2() {
  printf("sub2.\n");
  longjmp(buf, 1);
}
sub1() {
  printf("sub1 entry.\n");
  sub2();
  printf("sub1 exit.\n");
}
main() {
  printf("main entry.\n");
  if(setjmp(buf) == 0) {
    sub1();
  }
  printf("main exit.\n");
}
```

ここで、setjmp.h で定義されているデータ構造 jmp_buf がレジスタ類を保存するための場所です。そして setjmp でこの場所にレジスタを保存し、手続きを何段か呼び出し、奥の方で longjmp によりこの保存してあった環境に戻ると…どこに戻りましょうか?

なにしろ setjmp も手続きであり、その中で保存したということは、戻る位置もこの中になります。そしてそこから戻って来るということは、さっき setjmp を呼んだその同じ場所に戻って来るわけです。そしたら、タイムマシンで昔に戻って昔と同じ自分になったときみたいに、堂々めぐりになってしまいそうですが…そうならないために、setjmp は返値を使います。

つまり、最初に保存して戻った時は返値として 0 が返り、大域脱出で戻って来た時は 0 以外が返ります。なので、setjmp の返値を見て、0 だったら保存したところなのでそこから下請けの手続きを呼び、0 でなければ脱出してきたところなので下請けは呼ばずに終わる、というのが正しいわけです。上の例は確かにそうなっています。動かしてみましょう。

```
main entry.
sub1 entry.
sub2.
main exit.
```

確かに、あるはずの「sub1 exit.」がすっとばされて、sub2 からいきなり main に戻っています。ところで、今は setjmp の返値は 0 か否かだけ見ていましたが、「さまざまなエラー」の種別が知りたいかも知れません。そこで longjmp では第 2 引数として 0 以外の値を指定すると (0 だと 1 とみなされます)、その値が setjmp の値として返される、というふうにできています。

ところで、setjmp/longjmp はあくまでもこのように使うことを想定して作られていますが、とにかくレジスタを保存し、その状態に戻る、というのが機能ではあります。とすると、手続きを呼ばずに main の下の方で longjmp して上の方に戻る、みたいな使い方とか、sub2 の中で setjmp して main に戻って来てから longjmp するとか、いろいろ「正しくない」使い方もあり得ます。ただしそこで何が起きるかは十分把握していないと危険です。

演習 1-9 asm 文を使って main の中にラベル、sub2 の中にジャンプ命令を挿入し、sub2 から直接 main の中にジャンプさせてみなさい。まず何が起こるか予想し、その後で実行により確認しなさい。

演習 1-10 setjmp/longjmp の例を動かして確認しなさい。動いたら、次のような「正しくない」使い方を試してみなさい。いずれも何が起こるか予測してから試すこと。

a. longjmp で戻って来たあとまた sub1 を呼ぶことで無限ループを作ってみる。
b. longjmp を main の最後で実行して無限ループを作ってみる。
c. sub2 の中で setjmp して main に戻って来てから longjump することで無限ループを作ってみる。
d. その他、面白い「いたずら」を考えて実行してみる。

4 並行性とスレッド

4.1 スレッドの概念

(大域脱出を含めても) 制御構造や手続き呼び出しで組み立てられたプログラムはあくまでも「一筆描き」で実行され、その中の動作の順番はきっちりと決まっています。

しかし世の中には、複数の事柄を「並行して」進めたい、というニーズも沢山あります。たとえば、複数の物体が動くようすを実時間アニメーションするのだったら、各物体ごとにその物体の動きを計算するコードがあって、それらが並行して動くことで全体のプログラムが動作している、という設計が自然かも知れません。

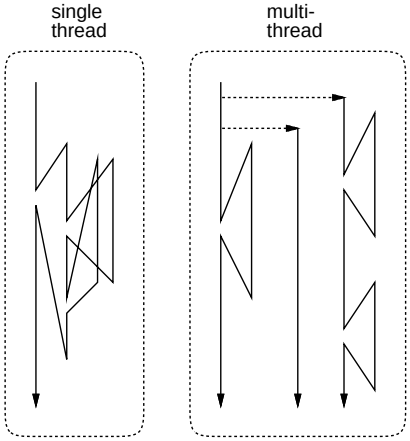


図 8: シングルスレッドとマルチスレッド

このような並行性をプログラミング言語でどう扱うか、というのは大きな問題で、さまざまな提案があるのですが、その話に行ってしまうと他の話題ができなくなるので、ここでは「通常の手続きが複数並行して実行する」というモデルに絞って取り上げます。つまり、それぞれの手続き実行が 1 つの「一筆描き」であって、それが N 本あるわけです。この 1 つの「一筆描き」のことをスレッド、1 つのプログラム中に複数のスレッドがあるようなものをマルチスレッドと呼びます (図 8 右)。そしてこれと対照的に、ここまでに出て来たような一筆描きが 1 つだけのものをシングルスレッドと呼びます (図 8 左)。

マルチスレッドはさらに、次の 2 種類に大別できます。

- ネィティブスレッド — 各スレッドを 1 つの CPU(ないし CPU コア) によって実行する。
- グリーンスレッド — 全てのスレッドは 1 つの CPU(ないし CPU コア) を適宜譲り合いえながら使用する。

ネィティブスレッドは本当に各スレッドが同時的に動作するので、全体として計算が高速になります。これを並列実行と呼びます。並列実行は、CPU の単一スレッド実行速度が (物理的限界から) 向上しなくなり、その一方で半導体技術の進歩により多数の回路が詰め込めて複数コアを持つ CPU チップが普及したことで、性能向上の手段として多くの期待が掛けられています。

その一方で、複数のスレッドが並列に実行されるということは、それらの間でのデータの受け渡しや処理の同期などに多くの注意が必要となります。さらに、たとえ正しく動いても、複数コアの能力を最大限駆使した高速なプログラムを作るのは高度な技術やチューニング作業が必要です。つまり簡単に言えば、並列プログラミングは (正しく動作するように作るという点でも、性能を発揮させるという点でも) 非常に難しい、ということです。

一方、グリーンスレッドの方は、並行した処理を並行した処理として記述すること、つまり記述しやすく理解しやすくすることが目的です。そして、実際に動作を行わせる CPU は 1 つしかないので、あるコードの部分が走っている最中に他のコードが走って干渉することは無いようにできます。具体的には、「切り替わっていいよ」と明示的に指定したところでだけ他のスレッドへの切り替わりが起きるようにします。[8]

4.2 グリーンスレッドの実装例

では、スレッドはどうやって実装すればいいのでしょうか。ネイティブスレッドだと OS の助けが必要でややこしくなるので、ここではグリーンスレッドを例にとって、簡単なスレッドライブラリもどきを実装してみることにします。

まず、スレッドはそれぞれが固有のスタックを持つ必要があります。というのは、各スレッドの一筆描きはその途上で自由に手続きを呼ぶわけですから、その制御にスタックが必要です。また、各スレッドの「固有の」データはローカル変数つまりスタックに置かれますから、その点でも各スレッドは固有のスタックを必要とします (これらの点はネイティブスレッドでも同じことです)。

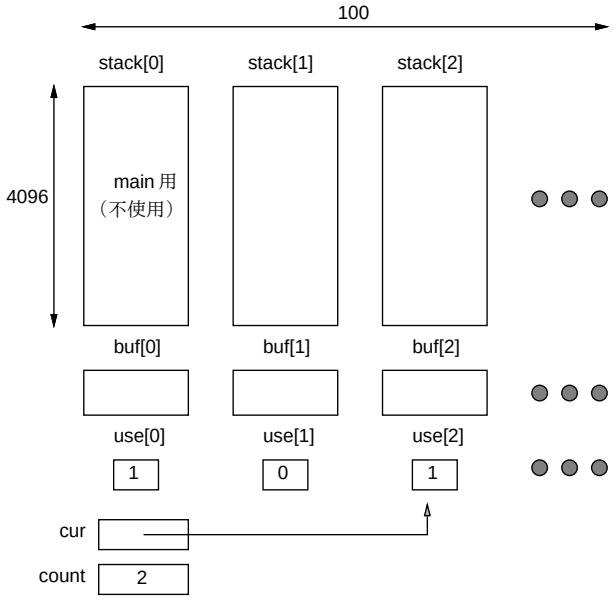


図 9: スレッドライブラリもどきのデータ構造

そこで、`main` 以外のスレッド用のスタックは配列を使って別に取るようにします。[9] また、スレッドが切り替わる際にはレジスタ類を全て保存し、あとで再開する時に戻す必要がありますが、それには `setjmp`/`longjmp` を使いますから、そのための `jmp_buf` もスレッド数ぶん要ります。そして、これらの領域が現在使用中かどうか (つまりスレッドが活きているかどうか) をフラグで表現します。これらのデータ構造を図 9 に、またその宣言部分を以下に示します。変数 `cur` は現在動いているスレッ

[8] 別のやり方として、タイマーを用いて一定時間ごとに別のスレッドに切り替える、という方式もありますが、こちらではいつ切り替わりが起きるかが制御できないので、並列プログラミングと同様な難しさが生じます。

[9] `main` のスタックは取らなくていいのですが、変数の宣言を分けると面倒なので他のスレッドと同じに場所は確保してあります。

ドが何番なのかを保持し、変数 count は活きているスレッドが (main を含めて) いくつあるのかを保持します。

```
#include <stdio.h>
#include <setjmp.h>
#define MAXTHREAD 100
#define STACKSIZE 4096
int stack[MAXTHREAD][STACKSIZE];
jmp_buf buf[MAXTHREAD];
int use[MAXTHREAD];
int cur = 0, count = 0;
```

スレッドを開始するときには、呼び元のスレッドとは関係なく新しいスタックで実行を開始します。そのための下請けルーチン_call を示します。スレッドとして実行開始させる関数のアドレス、スタックの底のアドレス、スレッド番号を受け取ります。冒頭で cur に番号を入れ、その番号の use 配列を 1 にして、count を増やします。その後は%eax に関数、%esp と%ebp にスタックの底のアドレスを入れ、間接 call 命令で関数を呼びます。そして戻って来たらこのスレッドは終わるので、use 配列を 0 に戻し、count を減らし、別のスレッドに切り替えます。以後、use が 0 のスレッドが選ばれることはないので、この呼び出しから戻って来ることはありません (仮に戻って来られても、戻り番地も何も全部壊して捨ててしまっているのでどうにもなりません)。

```
_call(int (*func)(), int *stk, int id) {
  cur = id;
  use[cur] = 1; ++count;
  asm("movq %rsi,%rsp");
  asm("callq *%rdi");
  use[cur] = 0; --count;
  gt_yield(); // NO RETURN
}
```

スレッドを生成する手続き gt_create は関数ポインタを 1 つだけ受け取り、use が 0 のスレッド番号を探して、その番号のスタックを指定して_call を呼ぶだけです。ループから出なくていいのかと言われそうですが、前述のように_call からは決して戻って来ませんからこれでいいのです。[10]

```
gt_create(int (*func)()) {
  int i;
  for(i = 0; i < MAXTHREAD; ++i) {
    if(!use[i]) _call(func, &stack[i][STACKSIZE-16], i); // NO RETURN
  }
}
```

では最後に、スレッドを切り替えるルーチン gt_yield を見てみます。観察用に何番から何番に切り替えているかを表示するコードが挿入してありますが、コメントアウトになっています。さて、切り替えるためにはまず setjmp で現在の自分の状態を buf に保存し、次に (保存した後なら)cur を順番に進めて use が 1 になっている (実行させられる) スレッド番号を探します。見つかったら (必ず見つかるはず)、その番号の buf で longjump すると当該スレッドのこの関数の setjmp の所へ返値 1 で戻って来ます。返値が 1 のときは何もせずにそのまま戻るので、このスレッドが最後に gt_yield を呼んだ箇所の次に戻ってスレッドの実行が続くわけです。

[10] スタックの底のアドレスを渡すときに、STACKSIZE ギリギリでなく 16 語 (64 バイト) あけていますが、これはなんとなくギリギリだと恐いから:-) なのと、このあいているところに追加の情報 (たとえばスレッドに渡す引数…演習参照) を入れるなどのことも考えたからです。

```
gt_yield() {
  /*printf("switch from %d ", cur);*/
  if(setjmp(buf[cur]) == 0) {
    do { cur = (cur+1) % MAXTHREAD; } while(!use[cur]);
    /*printf("to %d \n", cur);*/
    longjmp(buf[cur], 1);
  }
}
```

以上で API の説明は終わりで、この先は例題です。func1 は 0 から 9 までの数を表示しますが、1 個表示するごとに他のスレッドに切り替わります。func2 は同様ですが、1 個表示するごとに他のスレッドに 2 回切り替わるので、進行の速度は半分になります。

```
func1() {
  int i;
  for(i = 0; i < 10; ++i) {
    printf("func1 %d\n", i); gt_yield();
  }
}
func2() {
  int i;
  for(i = 0; i < 10; ++i) {
    printf("func2 %d\n", i); gt_yield(); gt_yield();
  }
}
main() {
  printf("main enter.\n");
  use[0] = 1; ++count;
  if(setjmp(buf[0]) == 0) gt_create(func1);
  if(setjmp(buf[0]) == 0) gt_create(func2);
  while(count > 1) gt_yield();
  printf("main exit.\n");
  return 0;
}
```

main はまず自分の番号 0 番の use を 1 にし、count を 1 増やしてから、2 つのスレッドを実行開始させます (実行開始させるとそのスレッドのコードに入ってしまうので、続きをやるために setjmp しておく必要があります)。スレッドを全部生成し終わったら、あとはスレッド数が 1 より多い (自分以外のスレッドがある) 間は、自分は何もせず他のスレッドの実行を続けさせます。自分だけになったらおしまいです。

演習 1-11 このコードをコピーして動かせ。動いたら、次のような修正を行ってみよ。

a. フィボナッチ数列が 20 個、数列 1、2、4、…が 20 個、交互に表示されるようにする。
b. 上と同様だが、20 個スレッドを作って、フィボナッチ数列の各値が 20 回ずつ表示されるようにする。
c. スレッドライブラリの機能を少し拡張して、スレッドが開始されるときに整数値のパラメタを 1 つ渡せるようにしてみなさい。それを活用して、2 の倍数、3 の倍数、5 の倍数を 20 個ずつ互い違いに打ち出す例題を作ってみなさい。
d. その他、このスレッドライブラリを使って何か面白い例題を作ってみなさい。