プログラミング言語論2012 ＃1 —
言語の基本要素と概念

久野　靖*

2012.4.12

0　はじめに

本科目「プログラミング言語論」の目的は、「情報技術の基盤概念であるプログラミング言語全般に対する体系的な知識を身につけて頂く」ことです。また、その具体的な到達目標としては、次のものを掲げます。

- 主要な言語の体系、各種の言語が持つ主な特徴や基本概念について理解する。
- 本講義で取り上げる主要な言語について、簡単なプログラムを書いて動かす経験を持つ。

書いて動かすのは大変に思えるかも知れませんが、あくまで「体験」が目的でそんなに高度なことまでやるつもりはないので、ご安心ください。

また、今日「お仕事」で使われる主要な言語はオブジェクト指向言語なので、その近辺の話題も出てきますが、オブジェクト指向言語そのものは別の (表裏の) 科目でいろいろ取り上げているので、本科目では「言語体系的に」特徴的な (普通と違う) 部分を中心に扱う予定です。

まだ資料が全部できていないので:-)、各回の内容は確定していませんが、おおむね次の順番で計画しています。

- ＃1 プログラミング言語の基本概念、計算モデル、手続きの実装
- ＃2 構文と意味、Lisp 言語とその実装、環境とスコープ
- ＃3 オブジェクト指向とジェネリクス
- ＃4 プロトタイプ型、言語の表現
- ＃5 関数型、高階性

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 つ選んで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　プログラミング言語の基本概念と計算モデル

1.1　プログラミング言語の定義と要件

まず最初の質問として、「プログラミング言語とは何ですか?」から始めましょう。次のような順番で考えたらよいかと思います。

*経営システム科学専攻

(1) コンピュータは「(一群の) 指令に従って動作する」装置である (与える指令に応じて異なる動作を行う→汎用性を持つ)。その指令=プログラム。

(2) プログラムは人間が記述するので、その記述の「規則」がプログラミング言語。

では次に、プログラミング言語に求められる要件 (必要な/望まれる性質) は何でしょうか。

(a) コンピュータに読み込ませて動作させられること。

(b) 人間にとって読みやすく書きやすいこと。

(z) コンピュータに行わせる任意の動作を記述する能力を持つこと。

まず (a) については、もともとプログラムが計算機に読み込ませて動作させるためのものだから、「必要な」条件ということになる。一方で (b) については、プログラムを書くのは人間であり、また書いている最中には当然書きかけのプログラムを読むことになり、また完成した後でも保守・改造のために自分や他人の書いたプログラムを読むことから、読みやすく書きやすいことが「望ましい」ことになります。

あくまで「望ましい」なので、この要件を満たさないような言語も存在はします。典型的には、機械語 (CPU が直接する裸の命令列そのものを、16 進表記などで記述したもの) が、読みづらく書きづらい「言語」に相当するでしょう。また、機械語よりは読めるとしても、アセンブリ言語 (CPU の各命令を、命令や使用するレジスタ・メモリ番地などを適宜名前で書き表したもの) も、かなり繁雑です。これらはいずれも、CPU という裸の実行メカニズムに密着したものであり、実行効率やメモリ効率などを優先して設計されているため、人間にとっての読みやすさや分かりやすさは無視 (ないし低い優先順位) となっているからです。[1]なお、機械語やアセンブリ言語のように CPU の命令体系に依存するプログラミング言語を「低水準言語」と呼びます。

機械語やアセンブリ言語が繁雑で人間に苦労を強いるものだったことから、計算機のごく初期 (1950 年代) から、人間にとってより扱いやすい表記法でプログラムを記述し、それをアセンブリ言語や機械語に翻訳することで最終的に CPU で動かす、という方法が探究されました。この「扱いやすい言語」は特定の CPU の命令体系に依存しないようなものであり、低水準言語と対比して「高水準言語」と呼ばれます。初期の高水準言語の代表的なものとして、FORTRAN、COBOL、Algol-60 などがあります。現在の手続き型言語 (代表は C 言語) も、これらとそんなには違っていません。

1.2 チューリング機械

ところで、あるプログラミング言語が (a) コンピュータで動作させられ、(b) 人間にとって読みやすく書きやすいとしても、もしその言語でコンピュータが行える動作すべてが記述できないならば、その言語の機能は十分ではない、ということになりそうです。そこで (z) の要件を入れるわけですが、これはどのようにして判断すればよいでしょう? それには、コンピュータが行う計算の「モデル」を用意し、ある言語による記述がその「モデル」による任意の動作 (計算) を記述できればよい、というふうに考えます。

ここでそのようなモデルとして、チューリング機械を取り上げます。最も基本的な 1 テープ決定性チューリング機械の構成を図 1 に示します。チューリング機械は、次の 2 つの構成要素から成ります。

- テープ — 片方向に無限に延びたテープがある。テープの上はます目に分かれていて、有限個の記号のどれか 1 つを記録している。この記号の集合を Γ で表す (作業用アルファベット)。その中には空白記号 B が含まれており、テープの使われていない部分には最初すべて B が書かれている。また、Γ の部分集合 Σ(入力アルファベット) があり、テープの冒頭部分にはこの Σ の要素を並べた入力データが書かれている。テープには読み書きヘッドが接していて、有限状態機械によりヘッドの位置を左右に動かすことができる。

[1]DEC VAX-11 のように、人間にとっても分かりやすいことをめざして設計された命令語を持つ CPU もありましたが、結果的には実行速度が高くできず、あまり成功しませんでした。

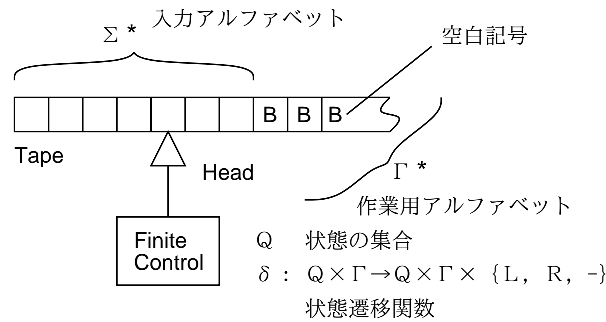

図 1: 1 テープ決定性チューリング機械

- 有限状態機械 — 有限個の状態の集合 Q を持つ装置で、遷移関数 $\delta\colon\ Q\times\Gamma\to Q\times\Gamma\times\{L,R,-\}$ により動作する。その意味は、現在の状態 q とヘッド位置に書かれている記号 s に応じて、新しい状態 q' に遷移するとともにヘッド位置を記号 s' に書き換え、読み書き位置を左 (L) または右 (R) に 1 ます動かすかまたは動かさない $(-)$、ということである。Q には状態 q_0 が含まれていて、実行開始時には有限状態機械はこの状態にある。また Q の部分集合 F があり、計算が停止したとき状態が F の要素であるなら、チューリング機械は入力を「受理した」と呼ぶ。[2]

では簡単な例として、入力が「$N(>0)$ 個の 0 と同数個の 1 の並び」であるかどうかを判定する (そのようなものであった場合に受理状態で停止する) チューリング機械を構成してみましょう。[3]

- $Q=\{q_0,q_1,q_2,q_3,q_4\}$, $F=\{q_4\}$
- $\Sigma=\{0,1\}$
- $\Gamma=\{0,1,X,Y,B\}$
- δ は次の表による

	0	1	X	Y	B
q_0	(q_1,X,R)	$-$	$-$	(q_3,Y,R)	$-$
q_1	$(q_1,0,R)$	(q_2,Y,L)	$-$	(q_1,Y,R)	$-$
q_2	$(q_2,0,L)$	$-$	(q_0,X,R)	(q_2,Y,L)	$-$
q_3	$-$	$-$	$-$	(q_3,Y,L)	$(q_4,B,-)$
q_4	$-$	$-$	$-$	$-$	$-$

これが動作する様子を図 2 に示しました。まず、状態 q_0 で 0 を見ると X に書き換え、状態 q_1 で 1 が見つかるまで右に移動していきます。1 があったらそれを Y に書き換えて q_2 で今度は X が見つかるまで左に移動していき、X があったら 1 つ右に移動して q_0 に戻ります。これにより、1 は X、0 は Y にペアで置き換えられていきますが、もし 1 の方が少ないと q_1 で B に遭遇して行き止まりで停止します。それ以外の場合、0 が無くなるので q_0 で Y が読めることになり、q_3 で Y がある限り右に移動します。そこでもし 1 に遭遇したら 1 の方が多かったので行き止まりで停止しますが、ぶじに B に遭遇したら q_4 に進み、受理状態となります。

このようにチューリング機械では、テープを書き換えつつ左右に移動することを有限の状態のみで制御することで、さまざまな「計算」が行えるわけです。

演習 1 次のようなチューリング機械を構成してみなさい。

[2]ある状態について、δ が「すべての入力についてまた同じ状態に進み、テープは動かさない」だと、この状態から進むことはないので、このような状態に到達したら停止したものとして扱えます。

[3]この例は「ホップクロフト他著, 野崎他訳, オートマトン　言語理論 計算論 II(第 2 版), サイエンス社, 2003」によります。

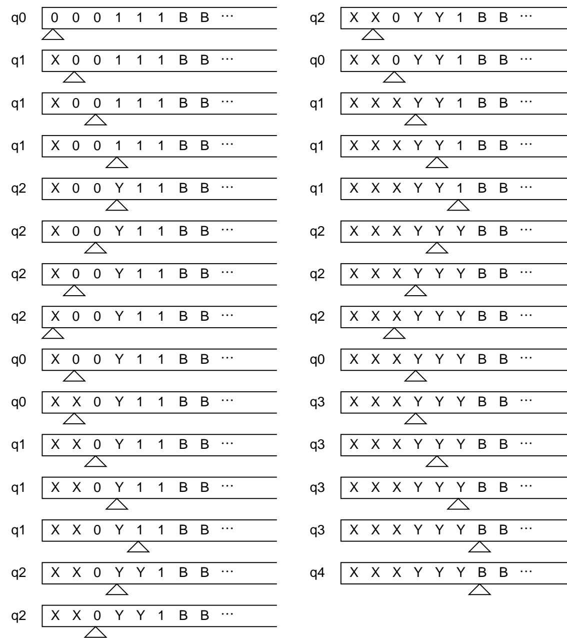

図 2: 0^n1^n を受理するチューリング機械の遷移関数

a. 入力が 0 が $N(>0)$ 個、1 が同数個、2 が同数個並んだ列であることを判定する。
b. 0 が N 個、1、0 が M 個並んでいた時に、テープのどこかに 0 が $M \times N$ 個並んだ列を作って止まる。
c. その他、自分が思い付いた適切な問題を実行する。

1.3 チューリング機械のさまざまな拡張

このようにチューリング機械はごく原始的なモデルですが、このモデルで行える計算と今日のコンピュータで行える計算は基本的に同等です。A と B が同等であるということを示すには、A が行える計算はすべて B でも行えることと、B が行える計算はすべて A でも行えることの両方を示せば十分。これはいいですね? そして次に、今日のコンピュータでチューリング機械のシミュレータを作って動かすことは何ら問題ないから、A → B も言える。これもいいですね? [4]

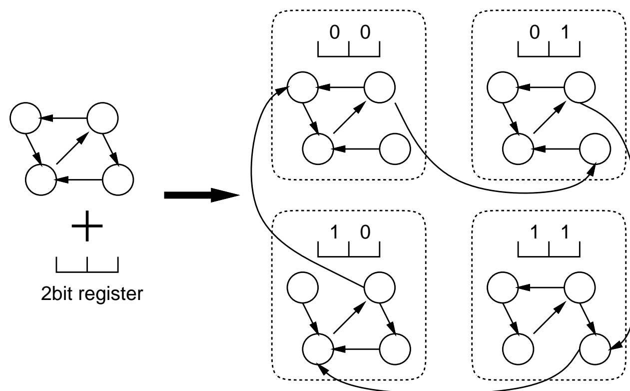

図 3: レジスタの追加

[4] ここでポイントは、テープに書かれる記号の種類が有限であること、状態数が有限であることです。無限はコンピュータで扱えませんから。テープが無限というのはちょっと困るのですが、メモリに入り切るだけのます目で止めておくなら、そこまでは同等、というふうに考えることができます。

逆方向ですが、上で述べたチューリング機械を「拡張」していく形で見て行きます。まず、上で説明した有限状態には記憶機能は無いのですが、ここに有限のビット数の記憶装置 (レジスタのようなもの) があるものと考えても構いません (図 3)。というのは、たとえば 16 ビットのレジスタならそこには $2^{16} = 65536$ 通りの値が入りますが、それに対応して元の状態 Q を 65536 個コピーします。その $65536 \times |Q|$ 個の状態のどれにいるかに応じて、元の Q の状態プラスレジスタの値が表現できるわけです。有限なものを 65536 倍しても有限には違いない、というのがポイントですね。

では次に、これまでテープにはます目が 1 列個ずつ並んでいたのですが、これを K 個並んでいるように拡張しましょう (K トラックチューリング機械、図 4)。これも簡単で、元のます目を K 個ずつグループとして扱い、拡張されたチューリング機械において次の状態に遷移するときに、元のチューリング機械ではこの K 個のます目を順次書き換えて行く状態列を経てやればいいのです。このようにしても、状態の数も記号の数も有限であることは変化していません。

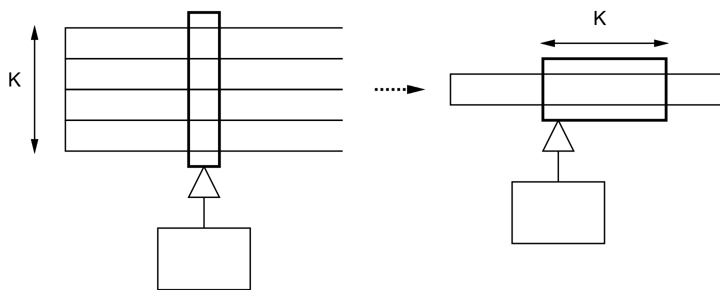

図 4: K トラックチューリング機械

では次に、テープが K 本あってそれぞれ独立に動かせるように拡張します (K テープチューリング機械、図 5)。それには、$2K$ トラックチューリング機械を用いて、2 トラックをペアで扱います。その片方には、K 本のテープそれぞれに対応する記号を書きます。そして他方には、現在ヘッドがある位置には「ここ」、それ以外は「ヘッドは右」「ヘッドは左にある」という記号を書きます。こうすれば、拡張されたチューリング機械で各ヘッドを読み書きする時には、元の ($2K$ トラックの) チューリング機械でそのヘッドの位置までテープを動かしてから読み書きすればいいわけです。もちろん状態数は増えますが、これでも有限であることに変わりはありません。

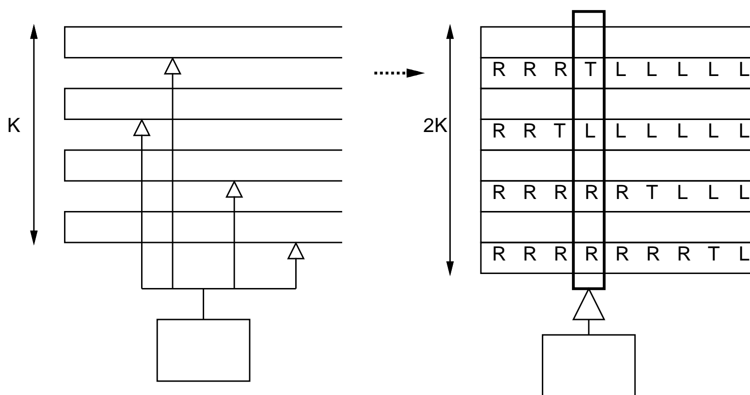

図 5: K テープチューリング機械

1.4 チューリング完全性

では最後に、万能チューリング機械、つまり任意のチューリング機械の動作を真似ることのできる (それと同じ動作ができる) チューリング機械を構成してみましょう。その前に準備として、任意の

チューリング機械を作業アルファベットが $\{0, 1, B\}$ だけから成るチューリング機械に変換できることを示します。それは簡単で、変換前のチューリング機械のアルファベット (有限個です) を表すのに N ビットあれば済むような N を決めて、各記号を N ビットの 0 と 1 の並びで符合化し、N 個のます目を組にしてそこに書けばよいわけです (B だけは最初にテープに書かれているという点で特別扱いなので B に対応させます)。

では次に、0、1 だけを使うように変換したチューリング機械の情報をテープ上に表現します。それには、新しい文字 c を追加し、表現するチューリング機械が持つ N 個の状態の情報を表すブロックを cc で区切って並べ、両側を ccc で囲みます。各状態の記述は c で区切った 3 つの副ブロックから成り、それぞれその状態でテープの記号が B、0、1 の時の情報を表し、次の 2 つの場合があります。

- 0 — この場合に対応する動作が未定義。
- $wd111\cdots1$ — w はテープに書く記号、d は $L/R/-$ のいずれかでテープを動かす向き (ないし動かさないこと)、最後の 1 の列は「次に何番の状態に行くか」を 1 の個数で表します。

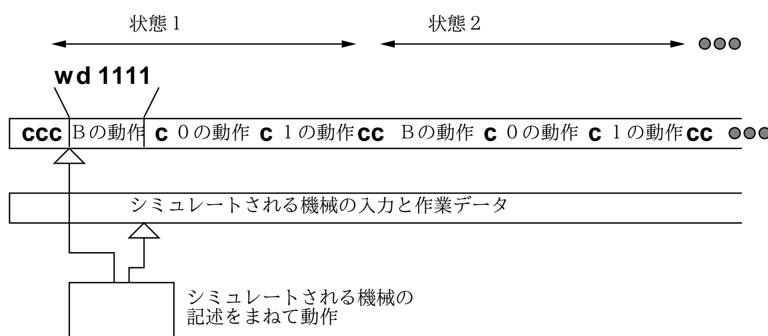

図 6: 万能チューリング機械の構成の例

万能チューリング機械は 2 テープを持ち、1 本目にシミュレートするチューリング機械の情報を上で説明した表現で与え、2 本目にシミュレートするチューリング機械への入力を与えます。[5]万能チューリング機械は実行開始すると現在の状態と入力テープの文字に基づいて指定された動作を実行し、次の状態に移ることを繰り返します。

このようにして構成した万能チューリング機械では、任意のチューリング機械の表現を与えることで、そのチューリング機械と同じ動作を行う (シミュレートする) ことができます。つまり「プログラムを与えるとそのプログラムに応じた動作をする」わけです。計算理論の人たちの次の興味は、ではどれだけ簡潔な (状態数の少ない) 万能チューリング機械が構成できるか、ということになりました。Marvin Minsky は 1962 年に「4 記号、7 状態」の万能チューリング機械の存在を示しました。今日ではさらに状態数や記号数の少ない万能チューリング機械も示されています。

さて、話を元に戻すと、チューリング機械によってコンピュータ向けの任意のアルゴリズムを実行させることは (具体的に書くのは大変ですが) 問題なくできそうです。そして、万能チューリング機械は任意のチューリング機械と同等の能力を持っています。ですから、プログラミング言語で万能チューリング機械の動作を書ければ、そのプログラミング言語でコンピュータが実行する任意の計算が記述できるわけです。このような性質を持つプログラミング言語を「チューリング完全」と言います。

厳密な証明という形で示すのは難しいところがありますが、一般的には通常のプログラミング言語はチューリング完全とされています。一方、チューリング完全ではない言語の例としては、HTML とか CSS などが挙げられます (再帰かループが書ければそれでチューリング機械の動作が書けるのでチューリング完全というのが一般的な考え方です)。

[5]実際にはシミュレートする機械の記述は読み取るだけなので、それをテープの先頭に置き、その後ろを入力と作業に使うことで、テープ 1 本で済ませることができます (そのように説明する方が普通)。

1.5 一般的な CPU のアーキテクチャ

ここまでは理論的なモデルとしてのチューリング機械について見て来ましたが、これを私たちが普段使っている CPU の構成と対比してみましょう (図 7)。

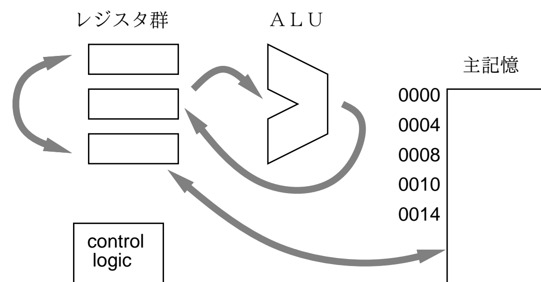

図 7: 一般的な CPU の構成

今日の CPU では、内部に複数のレジスタを備えていて、ここに現在計算している値を蓄えたり、現在実行している命令の場所を保持しています。数値演算や論理演算を行う回路 (ALU) はここことデータをやりとりします。一方、大量のデータやプログラムは主記憶に入っていて、番地を指定することで読み書きされます。CPU の動作そのものは主記憶に書かれた命令語を制御ロジックが解釈し、命令語に指定されている動作を実行する、ということを繰り返すだけです。

この CPU の動作は、万能チューリング機械とよく似ていると言えます。レジスタについてはチューリング機械でもモデル化できることを既に述べました。最後にデータやプログラム (シミュレートされる機械の記述) は、チューリング機械ではテープ、CPU では主記憶に入っています。チューリング機械でテープが使われているのは、番地のような込み入ったものをモデルに入れないためですが、テープを動かして読むのは明らかに遅いので、CPU でこれが番地を割り当てた主記憶に変更されているのは自然だと言えるでしょう。

CPU が備えている個々の命令は、データの転送、演算、制御 (無条件ジャンプ、条件ジャンプ) が基本です。その具体的なレパートリは (そしてレジスタの構成や名称なども)CPU の種類によって違うわけですが、これについては出て来たところで最小限扱うことにして、先に進みます。

2 プログラミング言語の基本機能

2.1 プログラミング言語の基本要素

ではここから実際のプログラミング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 2 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。

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

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

この、最後のものが分かりにくいですね。実際に上の C プログラムのアセンブリ言語コードを見ながら考えてみましょう。FreeBSD に備わっている GCC(Gnu C Compiler) では、次のようにするとファイル `test1.s` にアセンブリ言語コードができるので、それを眺めます。

```
gcc -S test1.c
```

実際にその抜粋を見てみます (ここでは CPU は IA32 アーキテクチャ)。

```
     (冒頭略略)
main:
     (途中略)
.L2:
 movl -4(%ebp), %eax
 cmpl -8(%ebp), %eax
 je .L3
 movl -4(%ebp), %eax
 cmpl -8(%ebp), %eax
 jle .L4
 movl -8(%ebp), %edx
 leal -4(%ebp), %eax
 subl %edx, (%eax)
 jmp .L2
.L4:
 movl -4(%ebp), %edx
 leal -8(%ebp), %eax
 subl %edx, (%eax)
 jmp .L2
.L3:
 subl $8, %esp
 pushl -4(%ebp)
 pushl $.LC3
 call printf
 addl $16, %esp
```

```
leave
ret
.size main, .-main
.ident "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"
```

このアセンブリソースを読む上で頭に入れておくべきことは次の通り。

- レジスタ%ebp(ベースポインタ)は「ローカル変数を置く場所の起点」を指すように調整されているので、ローカル変数(整数の場合はサイズ4バイト)は「-4(%ebp)」「-8(%ebp)」「-12(%ebp)」等々となる。
- レジスタ%eax、%edxは作業用の32ビットレジスタ。(%eax)のようにかっこに入っているときは「間接アクセス」(レジスタにメモリ番地が入っていて、その番地をアクセスすること)を表す。
- $1、$2等は定数1、2等を表す。
- 出て来る命令はmov(値の移動)、cmp(値の比較)、sub(引き算)、lea(メモリ番地の取得)、jl等(条件ジャンプ)、jmp(無条件ジャンプ)。

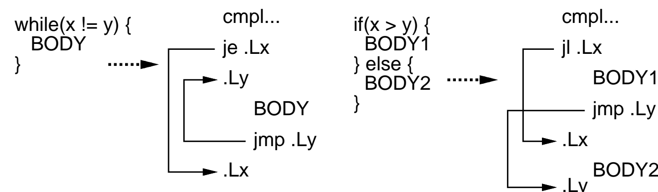

図8: Cの制御構造とアセンブリ言語コードの対応

こうして見ると、変数の参照・代入、演算・比較演算はそれぞれCの記述と直接対応して生成されていますが、制御構造については図8のように条件判定に対応する条件ジャンプ命令とそれに対応するラベル、およびループや枝の合流を実現する無条件ジャンプ命令の組み合わせに翻訳されていることが分かります。

なぜこういうややこしい(直接的でない)対応になっているのでしょう? それは先に述べたように、CPUの命令は基本的にCPUが効率よく実行される基本的なものが用意されているのであって、それを人間が直接使うのが分かりやすいとは言えないからです。実は昔の高水準言語(たとえばFORTRAN)では、条件ジャンプや無条件ジャンプの命令文しかなく、プログラマはそれらを組み合わせて制御構造を実現していました。

しかしそのような手間をプログラマに掛けさせるよりも、プログラミング言語が「人間にとって扱いやすい」書き方や概念(入れ子にできる制御構造)を提供し、言語処理系がそれをCPUの言葉に翻訳してあげる、という方がずっと効率的です(処理系は1度作るだけ、それを使ってプログラミングをする人は多数いて何個もプログラムを作るから)。というわけで、このような制御構造は「CPUの機能を直接反映」というよりも「プログラミング言語が作り出してくれる人間のための概念」という意味合いが強くなっているわけです。

演習3 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。

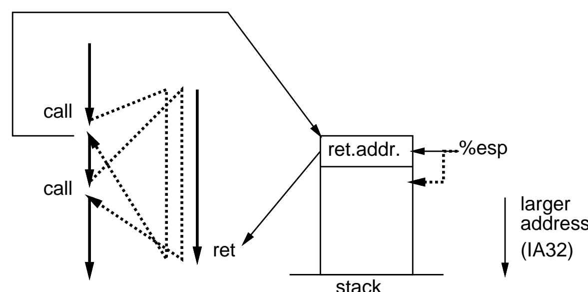

図 9: call 命令と ret 命令

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 9 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。

しかし、戻り番地はどこに「保存」するのでしょうか。それはスタックポインタ (IA32 では`%esp`) というレジスタがあって、メモリ上の適切な場所を指しています。call 命令は、まずこのスタックポインタを 4 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。[6]

実は古い CPU ではスタック領域を使用せず、手続きごとに決まった場所を戻り番地用に用意していました。しかしその方法だと、呼ばれた手続きの中から自分自身を呼び出すと (つまり再帰呼び出し)、先に保存した戻り番地を上書きして壊してしまうので、再帰が使えません。それは不便なので、今日の CPU ではスタックを使うように命令が設計されています。

さてでは、引数 (パラメタ) はどうやって渡すのでしょうか (図 10)。C では、パラメタはパラメタリストの後のものから[7] 順にスタックに積み (1)、続いて call します (2)。すると呼ばれた手続きの先

[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。

[7]可変引数に楽に対応するため。可変引数が無ければどちら順でも構わないことになります。

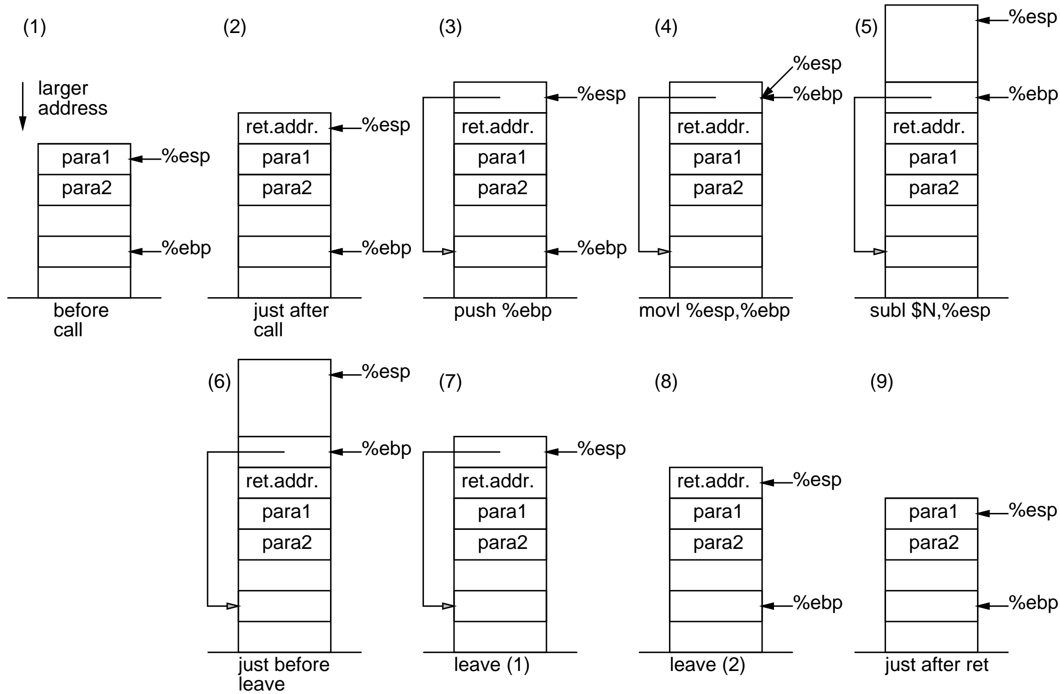

図 10: パラメタの受け渡しと bp の操作

頭に来ますが、ここでベースポインタをスタックに保存し (3)、その保存した番地をベースポインタに入れ (4)、スタックポインタはローカル変数などの分だけずらします (5)。こうすることで、新たなベースポインタがこの関数内でパラメタやローカル変数をアクセスする基準位置となり (パラメタは `8(%ebp)`、`12(%ebp)`、…でアクセスでき、ローカル変数は既に見たように`-4(%ebp)`、`-8(%ebp)`、…でアクセスできます)、そこからスタックポインタまでの間が自分のローカル変数領域になります。

さて、手続き内の動作が終わって戻る時は、leave という命令を使うとまずスタックポインタをベースポインタの値にし (7)、続いてスタックポインタの位置から旧ベースポインタを取り出してベースポインタを復元できます (8)。あとは ret により呼び出し側に戻ればよいわけです。

では実際にそうなっているか見てみましょう。C の例題プログラムは次のものです。

```
#include <stdio.h>

int main() {
  sub1(99);
  sub2(88);
}

int sub1(int x) {
  printf("sub1. %d\n", x);
/*asm("jmp tmp");*/
}

int sub2(int y) {
/*asm("tmp:");*/
  printf("sub2. %d\n", y);
}
```

コメントアウトした asm 命令は後で使います。これを先と同様にコンパイルした結果の抜粋を示します。

```
     (前略)
main:
     (途中略)
 pushl $99
 call sub1
 addl $16, %esp
 subl $12, %esp
 pushl $88
 call sub2
 addl $16, %esp
 leave
 ret
 .size main, .-main
 .section .rodata
.LC0:
 .string "sub1. %d\n"
 .text
 .p2align 2,,3
.globl sub1
 .type sub1, @function
sub1:
 pushl %ebp
 movl %esp, %ebp
 subl $8, %esp
 subl $8, %esp
 pushl 8(%ebp)
 pushl $.LC0
 call printf
 addl $16, %esp
 leave
 ret
 .size sub1, .-sub1
 .section .rodata
.LC1:
 .string "sub2. %d\n"
 .text
 .p2align 2,,3
.globl sub2
 .type sub2, @function
sub2:
 pushl %ebp
 movl %esp, %ebp
 subl $8, %esp
 subl $8, %esp
 pushl 8(%ebp)
```

```
pushl $.LC1
call printf
addl $16, %esp
leave
ret
.size sub2, .-sub2
.ident "GCC: (GNU) 3.4.6 [FreeBSD] 20060305"
```

sub1 でも sub2 でも冒頭の 3 命令 (プロローグ) でベースポインタの切り替えとスタック領域の確保を行い、最後の reave、ret で戻っています。また、他の手続きを呼ぶところでは、(ここでは使っていませんが) 戻り値を入れるかも知れないためにスタックを少し確保し、続いて引数を積み、call で呼び出し、戻って来たらさっき確保したぶんと引数を積んだぶんを合わせて元にスタックポインタを戻します。戻り値についてはここでは出て来ませんが、4 バイト (整数など) は`%eax` に入れて返し、大きい値は先に確保した領域に入れて返します。

これだけでは実行例が普通でつまらないので、関数 sub1 の出力が終わった直後に sub2 の中に jmp で飛んでみましょう (asm をコメントから出す)。実行したようすは次のようになります。

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

つまり、sub1 の途中から sub2 の中に飛んで sub2 の中の出力を行いますが、そこで y としてアクセスしているのは sub1 に渡された値になっています。ところで、sub1 から飛び出す前に x の値を変更したらどうなると思いますか。

演習 4 上の例を動かして確認せよ。その後次のことを行え。

a. 上の質問の回答を予想した後、実際に試してみよ。
b. 2 つの関数でパラメタ数が違うとどんなことが起きるか。また死なないためにはどのような手当てをすればよいか。考えた後実際に試して確認せよ。
c. `%ebp` の値をそのまま (整数のポインタとして) 返す関数を追加し、これを使って取得した値から引数やローカル変数がアクセスできることを確認してみよ。

2.3 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。

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 11 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、ベースポインタ) を A が動いているときの状況に戻さないといけ

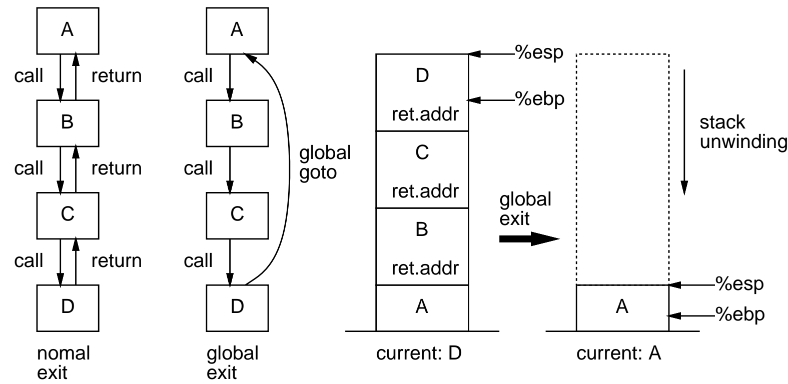

図 11: 大域脱出とスタックの巻き戻し

ません (図 11 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;

int main() {
  printf("main entry.\n");
  if(setjmp(buf) == 0) {
    sub1();
  }
  printf("main exit.\n");
}

int sub1() {
  printf("sub1 entry.\n");
  sub2();
  printf("sub1 exit.\n");
}

int sub2() {
  printf("sub2.\n");
  longjmp(buf, 1);
}
```

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 5 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 6 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。

3 並行性とスレッド

3.1 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。

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 12 右)。そし

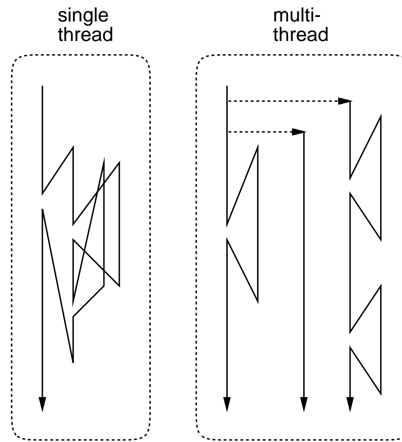

図 12: シングルスレッドとマルチスレッド

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 12 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]

3.2 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)。

[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。

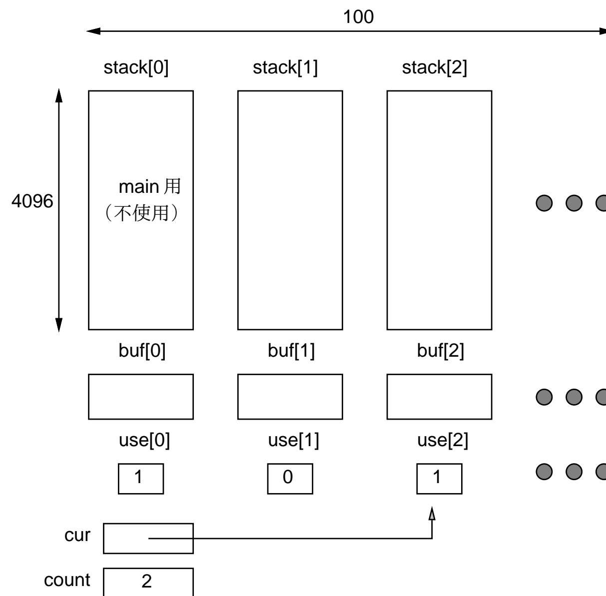

図 13: スレッドライブラリもどきのデータ構造

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 13 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)。

```
int _call(int (*func)(), int *stk, int id) {
```

[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。

```
    cur = id;
    use[cur] = 1; ++count;
    asm("movl 8(%ebp),%eax");
    asm("movl 12(%ebp),%esp");
    asm("movl %esp,%ebp");
    asm("call *%eax");
    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]

```
  int 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。

```
  int 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。

```
  int func1() {
    int i;
    for(i = 0; i < 10; ++i) {
      printf("func1 %d\n", i); gt_yield();
    }
  }
```

[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。

```
int func2() {
  int i;
  for(i = 0; i < 10; ++i) {
    printf("func2 %d\n", i); gt_yield(); gt_yield();
  }
}

int 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");
}
```

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 7 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。