システムソフトウェア特論'17 # 11 実行時環境とコード生成

久野　靖 *

2017.10.13

1　実行時環境と記憶域の配置

1.1　実行時環境とは

実行時環境 (runtime environment) とは，簡単にいえばコンパイラが生成した目的コードが実行されるときの約束ごとです。具体的には，次のようなものが実行時環境の規約に含まれます。

- 記憶領域の配置や割り当て方
- 変数の種類ごとのアクセス方法
- 関数の呼び出し方/呼び出され方や引数の渡し方/渡され方
- 生成コードと実行時ライブラリの分担

これらの約束ごとを設ける主な目的は、ソースコードに現れてくる名前 (変数名、手続き名、ラベル等) とそれらが表す実体 (記憶領域や命令列の特定の場所) の対応ないし**束縛** (binding) の管理を効率よく実現することだと言えます。

例えばソースプログラム中の変数 `x` は、実行時にはいずれかのメモリ番地に無ければなりません (または CPU レジスタのどれかに置かれることもあるかも知れませんが)。`x` がグローバル変数であれば、特定の番地に**割り当てる** (allocate) ことができます。

ローカル変数であれば、手続きが実行開始される時にローカル変数群を配置する領域割り当て、実行終了時にそれを**解放** (deallocate) しますが、その領域内での「何番目の位置 (オフセット)」ということはコンパイル時に決めておき、その領域の先頭を指すレジスタからどれだけ先、というアクセス方法を使うことで、効率よく扱います。パラメタについても同様のことをします。

このような、コンパイル時に変数の位置 (領域内のオフセットも含む) を決めてしまうことを、**静的束縛** (static binding) ないし**早期束縛** (early binding) と呼びます。

一方、名前とその実体の結び付きがコンパイル時には決まっていなくて、実行時に行う場合は**動的束縛** (dynamic binding) ないし**遅延束縛** (late binding) といいます。動的束縛が必要な例として、オブジェクト指向言語 (Java がそうです) のメソッド呼び出しが挙げられます。ある名前でメソッドを呼び出しても、実際にどのメソッドが呼ばれるかは実行時にしか決まらないからです。

動的束縛については回を改めて扱うこととし、以下では静的束縛を前提として説明していきます。まず記憶域の種別と配置について述べ、スタックとそれに関連する引数や返値の受け渡し、環境の切換えについて説明します。

1.2　記憶領域の種別と割当て

プログラムが実行時に参照する記憶領域は、基本的には次のように分類できます。

- コード領域 — プログラムの命令を保持する。

*電気通信大学 情報理工学研究科

- 定数領域 — 定数 (初期設定され、書き換えられないデータ) を保持する。
- 初期設定データ領域 — 初期設定され、書き換えられるデータの領域。
- 非初期設定データ領域 — 初期設定されないデータの領域。
- ヒープ領域 — 実行時に動的に割り当てられるデータの領域。
- スタック領域 — 局所変数など手続き呼出しに付随して割り当てられるデータ、戻り番地、その他管理情報の領域。

これらの区分は OS やハードウェアによってサポートされる場合もありますし、処理系の中だけの規約として実現される場合もあります。多くのシステムではハードウェアの機能を活用してコード領域や定数領域を書換え不可能なように保護し、誤りによってプログラムが書き換わってしまうことを防ぎます。

図 1 に、典型的な記憶域配置の例を示します (CPU の機能や OS の作り方によっては、もっとアドレス空間がバラバラになっている場合もあります)。ここでは**スタック** (stack) はアドレスが若い方向に伸びるように描きましたが、その配置や伸びる方向についても、通常ハードウェアやオペレーティングシステムによってはこれと違っています。

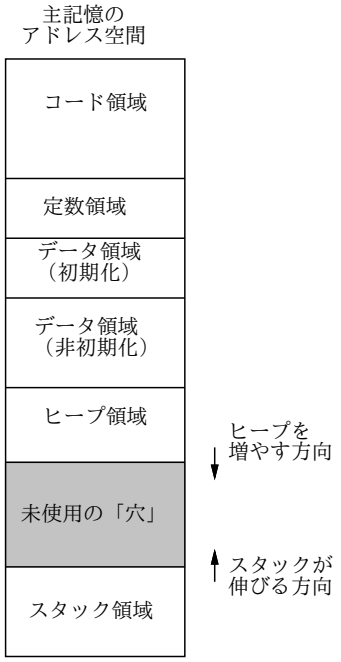

図 1: 典型的な実行時記憶域配置の例

実行時に動的に大きさが変化するのはスタックと**ヒープ** (heap) なので、この図のようにそれらを「向かい合せに」配置することがよくあります。現在では**メモリマッピング** (memory mapping) の機能が CPU に備わっていることが多いので、これらをアドレス空間上では離してとっておき、必要のつど実際のメモリページを割り当てることが通例です。

スタックとヒープ以外の領域については、大きさが実行時に変化することはないので、単に設計した配置に従って位置を割り当てるだけで済みます。さらに、リンカを用いる場合 (これが普通) は、目的コード中に「コード」「初期化データ」「非初期化データ」などの種別を含めるようになっていて、リンカが同種の領域をまとめながら詰め合せます。

また、記憶配置にはデータの種類に応じた**境界揃え** (boundary alignment) も考慮する必要があります。例えば多くのハードウェアでは 4 バイト長の整数や実数は 4 の倍数番地、8 バイト長の実数やアドレス値は 8 の倍数番地に配置する必要があります (そうしないとエラーになる CPU と、エラーにはならないけれど速度が低下する CPU があります)。

このため、変数に番地を割り当てる際には、そこに入るデータの種類に応じてあきを挿入して番地を境界に揃えます (または境界揃えを指定するアセンブラ用の指示を挿入します)。レコードのように複数のデータ型が混在する領域の場合には、その内部にも境界揃えのためのすきまが必要かもしれません。スタックやヒープ上の領域でも同様の配慮が必要です。

2 スタックとスタックフレーム

2.1 スタックの用途

スタック (stack) とは一般には、一番最後に割り当てられた領域が一番先に解放される (LIFO — last in, first out) ような割当てを実現するデータ構造です。

プログラミング言語における手続き呼出しでは、手続きからの戻りでは、一番最後に呼び出された手続きの呼び出し地点に戻るので、call 命令 (手続き呼び出し命令) は戻り番地 (call 命令の次の命令の番地) をスタックに積んでから手続きにジャンプし、ret 命令 (戻り命令) はスタックから戻り番地を取り降ろしてそこにジャンプするようにするようになっています。[1]

そして、一番最近に呼び出された手続きが一番最初に戻るので、手続き実行に付随する記憶領域 (局所変数や一時変数) もスタックに割り当てるのが自然です。これを含めて、手続き呼出しに付随する情報としては次のものがあげられます。

(a) 手続きに局所的な作業領域

(b) 戻り番地の情報

(c) 引数の情報/戻り値の情報

(d) 呼出し元の領域を示す情報

(e) 外側のスコープを示す情報

(f) 呼びに伴って壊れると困るレジスタ内容の写し

多くの手続き型言語の実現では戻り番地用とその他の領域用のスタックを兼ねて 1 本ですませますが、Prolog などでは呼出しと領域割当て/解放が同期しないため、呼出しスタックとデータ領域スタックを分ける必要がある。また Lisp のようにごみ集めを必要とする場合は、一般のデータを割り当てるスタック (データスタック) と手続きの呼び/戻り情報を積むスタック (制御スタック) を分けることもあります。以下では 1 本のスタックを用いた実現について説明していきます。

2.2 スタックフレームとフレームポインタ

1 本のスタックを用いる実現では、前節 (a)〜(f) の情報を各手続き呼出しごとに 1 組にまとめてスタック上に割り当てます。これを**スタックフレーム** (stack frame) ないし**活性レコード** (activation record) と呼びます。典型的なスタックフレームの形を図 2 に示しました。フレームの大きさは、局所変数に大きさが実行時に変化するもの (可変長配列など) を含まない限り、手続きごとに翻訳時に決まります。

引数の受け渡しと環境の切換えについては次節以降で説明するので、以下ではそれら以外の部分について説明します。

まず手続きが呼び出された時点でのスタックの状態について考えてみます。多くの命令セットアーキテクチャでは、スタックの先頭を指すレジスタである**スタックポインタ** (stack pointer、SP) が決められています。[2]

[1] 極めて古い CPU ではそういう技法が開発されていなくて色々変わった動作のものがありましたが、今はほとんどの CPU がそのような設計になっています。

[2] ハードウェアとしては決まっていなくて、オペレーティングシステムの規約としてスタックポインタに使うレジスタを規定する場合もあります。

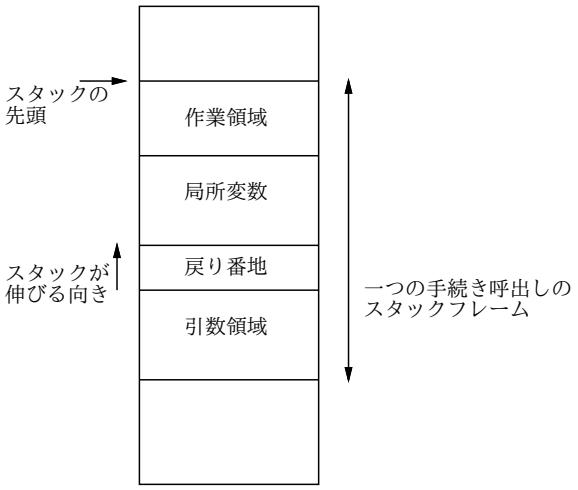

図 2: 典型的なスタックフレームの例

上で述べたように、手続き呼出し命令 (ないし呼出し規約) によって、戻り番地がスタックの上に積まれます。この様子を図 3(a) に示しました。

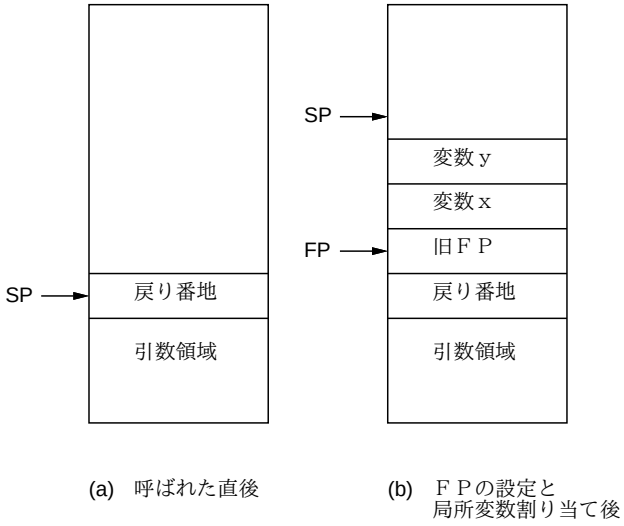

図 3: フレームポインタ

次にこの上に局所変数の領域を確保しますが、それにはスタックポインタを必要なだけずらせばよいです。もっとも、スタックは無限にあるわけではないので、領域を確保する際にスタック領域に十分なあきがあるかどうか検査する必要があります。ただし、メモリマッピングを用いてスタック領域を自動管理してくれる OS の場合には、あらかじめ決められた大きさまでスタックは自動拡張され、それを越えるとシステムがその旨通知してくれるので検査は不要です。

スタック上に確保した領域のアクセスは、この領域の番地が手続き呼出しごとに変わるため、固定番地による指定では行えません。ほとんどの CPU ではこのために「あるレジスタの指す場所から n バイト先」という番地指定ができます。

そこで、局所変数領域の先頭を指すレジスタを用いて、そのレジスタを起点にスタック上の領域にアクセスします。このレジスタを、スタックフレームの起点を指すことから**フレームポインタ** (frame pointer、FP) ないし**ベースポインタ** (base pointer) と呼びます。この様子を図 3 (b) に示します。例えばある手続きで大きさ 4 バイトの局所変数 `x` を起点の 4 バイト上、`y` を 8 バイト上に割り付けたとすれば、`y` の値を `x` に代入するには次のようなコードを出せばよいのです (命令やレジスタ名は x86-64

のもの。x86-64 については後述します)。

```
movel  -4(%rbp),%eax
movel  %eax,-8(%rbp)
```

しかし、フレームポインタは別の手続きを呼ぶとそこでも同様に利用するので、戻って来たときには別の値に書き変わっています。そこで、どの手続きでも呼ばれたらまずフレームポインタをスタックに格納し、その場所をフレームの起点だと考えてフレームポインタにはその番地を入れるようにします。手続きから戻るときには起点に入っている旧フレームポインタ値をフレームポインタに入れ直すことで、もとの値が復旧できます。

このようにすると、フレームポインタは常に 1 つ前のスタックフレームのフレームポインタを保存した番地を指しているので、フレームポインタから始まる連鎖をたどることでスタック上のフレームを新しいものから順にたどることができます。これを利用して、誤りや実行中断点 (ブレークポイント) への到達によって停止したプログラムの状況を調べ、その時点で各手続きの局所変数などを調べるデバッガをつくることができます。

コード上の各場所でスタックポインタとフレームポインタがどれくらい離れているかは、局所変数として大きさが実行時に変化する領域を割り当てない限りは、翻訳時に分かります。したがって、スタックポインタとフレームポインタという 2 つのレジスタをこのために割り当て管理するのは無駄であり、どちらか 1 つだけですませることも可能ではあります (可変長配列などを割り当てる手続き内では特別に両方使うようにすればよい)。実際、そのようなコンパイラも存在しますが、ただし、その場合にはデバッガなどのために「どの番地を走っているときはスタックポインタとフレームポインタはどれだけ離れているか」の情報を別に用意して参照させる必要があります。ここでは当面簡単さを優先させて、2 つのレジスタを使用することを前提としておきます。

2.3 引数と返値の受け渡し

引数 (arguments) と**返値** (return value) は呼ぶ手続ないし**呼び側** (caller) と呼ばれる手続きないし**呼ばれ側** (callee) の間で受け渡されるため、両方の手続きからアクセスされます。言い換えれば、スタックフレームはある手続きに固有の環境を表しますが、例外として引数だけは隣接フレームと共有されます。返値も手続き間で受け渡されますが、戻りによって呼ばれ側のスタックフレームは消滅するので、共有は起きません。

以下では引数の各種受け渡し機構および返値の渡し方について説明します。なお、今日のシステムでは高速化のため、引数を多数あるレジスタに載せて渡しますが、ここではまずスタック (メモリ) 経由の方法を基本として説明しています。また、引数について呼び側から見る場合と呼ばれ側から見る場合で区別するため、前者を実引数、後者を仮引数と記します。

a. 値呼び

値呼び (call by value) は最も単純かつ基本的な受け渡し機構であり、呼び側では任意の式の値を実引数として渡します。呼ばれ側では仮引数は渡された値を初期値として持つような局所変数として扱います (ただし Pascal などでは値渡し引数への代入を許しません)。

値呼びを実現するには、各実引数の値を計算し、それを順にスタック上に積むだけですみます。例えば `sub(100, -10)` という呼出しの場合、図 4(a) のように、`100` と`-10` をスタックに積み、そのまま `sub` を呼び出します。`sub` の方では仮引数を `a`、`b` という名前でアクセスするとすれば、それらは (旧フレームポインタと戻り番地が 84 バイトの領域であるとすれば) フレームポインタ起点で 16 バイトおよび 20 バイト手前 (大きい番地) にあることになります。そこで、例えば `a + b` という式であれば次のように命令を出力します。

```
movl   16(%rbp),%eax
addl   20(%rbp),%eax
```

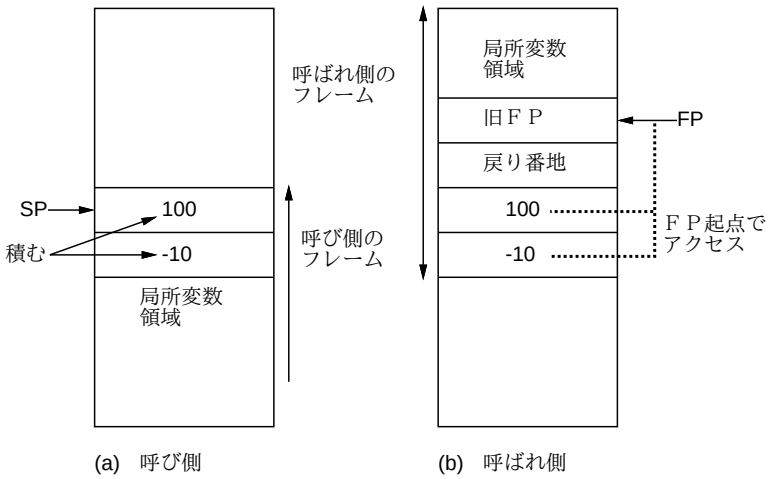

図 4: 値呼びの実現

つまり、他の局所引数と同等の命令でアクセスすることができるわけです。なお、この例ではスタック上に実引数を右から順に積んでいますが、こうしておくと引数の数が可変で、実際に渡した数は第 1 引数を調べるとわかるような関数 (C の `printf` など) が素直に実現できます (第 1 引数の位置は常に `16(%rbp)` で、残る引数はそれに隣接するから)。可変引数を使わなかったり、実引数の数を別の方法で受け渡すなら左から積むのでもかまわないことになります。

b. 参照呼び

値呼びは単純で分かりやすいですが、呼ばれ側から呼び側に情報を渡すうえでは不便です。これに対し、仮引数に対する更新がただちに実引数にも及ぶような呼出し機構が**参照呼び** (call by reference) です。

参照呼びでは図 5(a) のように、スタックには実引数の入った場所の番地を積んで渡し、呼ばれ側ではこの番地情報を読み出して実引数にアクセスします。

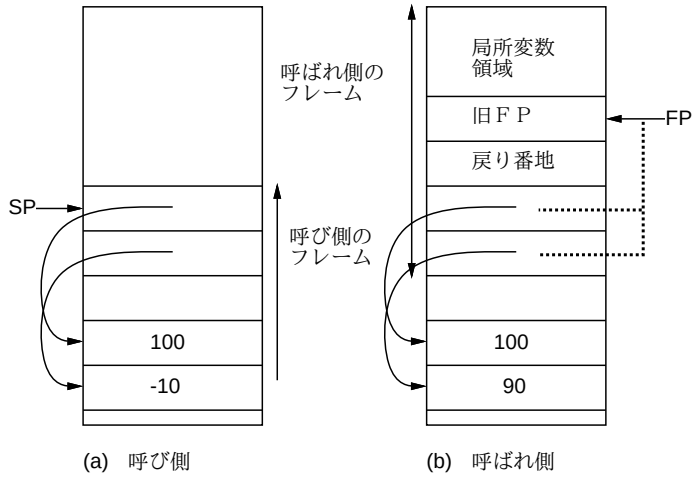

図 5: 参照呼びの実現

例えば上の例で呼ばれ側において `a = a + b` を実行するコードは次のようになります (アドレスが 8 バイトとすると、2 つのパラメタのオフセットは今度は 16 と 24 になります。また q がついた命令は 64 ビットを扱います)。

```
movq  24(%rbp),%rax
```

```
movl  (%rax),%edx
movq  16(%rbp),%rax
addl  (%rax),%edx
movl  %edx,(%rax)
```

参照呼びにおいて実引数が定数の場合、その定数 (例えば-10) が呼ばれ側で更新されてしまうのでは困ります。この問題に対しては、Pascal のように参照呼びの実引数には変数しか書けないという言語仕様で対処することもありますが、そうでなければ定数や式を渡すときには値を適当な作業領域に格納してその番地を渡す必要があります。

c. 名前呼び

参照呼びのような引数渡し機構を採用する動機の 1 つは前述のように「呼ばれた側から読んだ側への情報伝達」ですが、もう 1 つの考え方として「手続きを、その呼出し箇所に手続き本体のコードを (適切な名前置換えの後) 埋め込むものとして理解する」という立場があり得ます。仮にそのように考えることを許すとして、例えばつぎのような手続きがあったとします。

```
procedure sums(int k, a, b; real x, result)
begin
    result := 0.0;
    for k := a to b do result := result + x;
end;
```

これを次のように呼び出すとします。

```
sums(i, 1, 10, a[i,i], r);
```

これは、以下のように書くのと等しいわけです。

```
r := 0.0;
for i := 1 to 10 do r := r + a[i,i];
```

したがって大きさ 10 の行列 a の対角要素の和が求まるはずです。しかし、実際には参照呼びの場合、仮引数 x に相当する番地を呼出し時に計算して以後それを用いるため、上記のようなことはできません。これを可能にするには**名前呼び** (call by name) とよばれる引数渡し機構が必要です。名前呼びは Algol-60 言語で最初に採用されました。

名前呼びでは、呼ばれた側で仮引数が参照されるたびに、その仮引数のありかや値を計算し直す必要があります。そのため、参照呼びのように実引数の番地を渡す代りに、実引数の番地や値を計算する手続きを渡します。この手続きを伝統的に**サンク** (thunk) と呼びます。実現方法にもよりますが、サンクは各引数ごとにその値が参照されたとき (右辺値) 用とその場所に代入するとき (左辺値) 用の対で用意することが自然です。

例えば上の例だと、k、res に対応するサンクは呼ばれると常に i、r の値 (右辺値用) と場所 (左辺値用) を返します。また a、b に対応するサンクは右辺値用のみで、常に 1 や 10 を返します。一方、x に対応するサンクは呼ばれるごとにそのときの i の値に応じて適切な a[i,i] の値や場所を計算して返します。したがって、このサンクは変数 i を参照できなければなりません。そしてもし呼ばれた手続きが別の i という変数をもっていたとしても、間違ってそれを参照してはいけません。これを実現するには、後で述べる環境の切換えを正しく行う必要があります。

名前呼びの実現は複雑であり、効率上も不利であるので、最近の言語での採用例はほとんどありません。ただし、マクロ機構 (構文上は手続き呼出しに見えるが、その部分を字面上で定義本体により置き換えたうえで翻訳する機構) を用いる場合には、起きることは名前呼びと等価になります。ただ

しマクロ機構ではその場に本体を展開してしまうので、呼出し機構もサンクも不要です。逆に、参照呼びや値呼びの言語で実行効率向上のためにその場展開を行う場合には、これらの呼出し機構と名前呼びとで結果が異なる場合に留意する必要があるわけです。

d. 複写復元呼び

呼ばれ側から引数を通じて呼び側に情報を返したいが、番地の間接参照が遅いなどの理由で参照呼びにしたくないときに代りに使われる方法として**複写復元呼び** (copy-restore linkage) があります。この方式では図 6 に示すように、値呼びと同様に値を渡してしまい、呼ばれ側でその場所を参照/更新し、戻り時にはその場所から対応する実引数の場所に値をコピーし戻すことで呼び側に情報を返します。

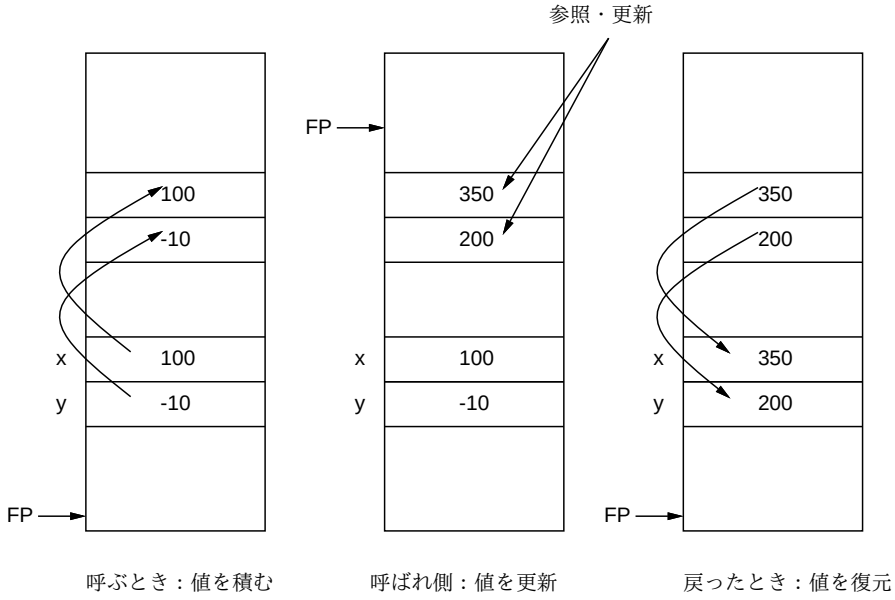

図 6: 複写復元呼び

複写復元呼びと参照呼びは常に同じ結果をもたらすとは限りません。例えば、次のような手続きを考えます。

```
procedure sub(integer i, j) begin i := i + 1; j := j + 1 end;
```

そして、呼ばれ方が以下のようだったとします。

```
x := 10; sub(x, x);
```

参照呼びでは sub の中で x が 2 回増やされるので、戻ってきたときの x の値は 12 になります。しかし複写復元呼びでは x の値 10 が複写されて渡され、それらが別個に増やされて 11 になり、戻りのときに x の場所に重ね書きで復元されるので x は 11 になります。言語によってはこの種の (引数渡し機構によって動作が変化する) コードを禁止し、引数渡しを参照呼びと複写復元呼びのどちらで実現してもよいものもある。

e. レジスタによる受け渡し

ここまででは引数は全てスタック上に積まれて受け渡されるものとして扱ってきました。しかし、毎回全引数をスタックに積み (すなわち主記憶上のどこかに書き込み)、呼ばれ側でまたレジスタに読み出すのは無駄です。そこで、引数の一部または全部をレジスタに入れたまま渡す工夫も多く行われまする。その際は、どんな場合どの引数をどのレジスタで渡すかについ規約が必要になります。

ここまでは引数についての話でしたが、返値についてはとくに、特定のレジスタに入れて戻る方法が簡単で公立もよいのでよく使われます。ただし、レジスタに入らない大きさのデータの場合が問題になりますが、対策としては、そのような例外的な場合のみ呼び側で領域を確保したり、どこか適当な場所 (たぶんスタックの上の方) に返値を置き、その番地をレジスタに入れて返すなどがあります (その場合には呼び側がただちに返値を適切な場所にコピーする必要があります)。

2.4 レジスタの退避回復

レジスタはデータのアクセスや演算のために多様に使われますが、ある手続きから別の手続きを呼んだときには、その呼ばれ側でも同じようにレジスタを使用するでしょうから、呼ぶ前と戻ってきた後では各レジスタの値は違っているかも知れません。

レジスタを式の途中結果や引数/返値の受け渡しのみに使用するのであればあまり問題はありません (式の評価の途中で関数を呼んだりするときは注意が必要)。しかし、効率よいコードのためにはよく参照される使われる値をできるだけ長くレジスタに置いて主記憶アクセスを避ける必要があります。その場合、それらのレジスタの値が手続き呼出しによって変わってしまうのは不都合です。これに対処するやり方としては、次の 2 つの方法があります。

- **呼び側での保存** (caller-save) — 手続きを呼ぶ側で、内容を壊されては困るレジスタの内容を保存してから呼び、返って来たらその内容を復元する。
- **呼ばれ側での保存** (callee-save) — 呼ばれた手続きの側で、呼ばれた直後に自分が内容を壊すレジスタを保存し、戻る直前に復元する。

どちらにも固有の利点と欠点があります。前者は、実際に壊されると困るレジスタのみを保存できますが、手続き呼出しが多数あると保存/復元用コードが多量に生成されます。後者は、実際には使っていないレジスタを保存してしまうかもしれませんが、保存/復元のコードは手続きの入口と出口に 1 箇所ずつで済みます。いずれにせよ、1 つの言語処理系ではどの方法を採用し、具体的にどのレジスタを退避回復するかを統一する必要があります。上記の特徴を考慮して、数個のレジスタは呼び側での保存、他の数個は呼ばれ側での保存とすることもあります。

3 コード生成

3.1 x86-64 CPU のデータサイズとレジスタ

以下では実際の CPU のコード生成を (アセンブラ出力により) 行ってみます。その場合、命令語の細かい仕様はアセンブラに任せればよいのですが、CPU がどのようなデータサイズを扱い、どのようなレジスタを持ち、どのような命令を持っているかは知っておく必要があります。

ここでは広く普及している x86-64 命令セットアーキテクチャについて簡単に説明します。厳密にはこのアーキテクチャでも Intel と AMD で違うところがあるのですが、主に OS 関係の部分なので、ここで説明する範囲では違いはありません。

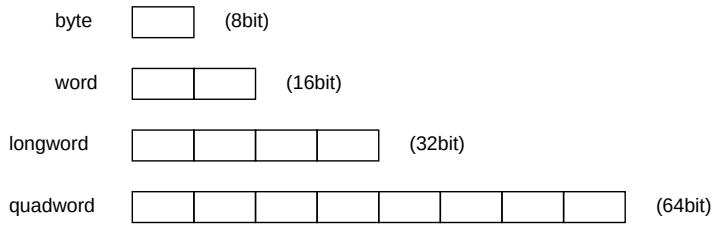

図 7: x86-64 CPU が扱うデータサイズ

図 7 に x86-64 CPU が扱うデータのサイズを示します。このアーキテクチャは 8 ビットの 8086 から順次進化 (無理矢理拡張?) してきたので、名前にもその名残りがあります。バイトは普通ですが、

その倍の 16 ビットが「ワード」で、普段整数に使っている 32 ビットは「ロングワード」になります。そして 64 ビットアーキテクチャなので、アドレスのビット数が 64 ビットですが、これは「クワド (4 倍) ワード」と呼ばれます。

そして、これらのデータを扱うためのレジスタが図 8 のように構成されています (浮動小数点用のレジスタは略)。これも歴史的経緯のためたいへんやっかいです。もともと 8086 は特定用途向けの名前のついた少数のレジスタを持っていたのですが、レジスタが多い方が性能的に有利なのでどんどんレジスタを増やし、なおかつ上記のデータ幅も増やしたことによっています。

ともあれ、一番多く使うアキュムレータは昔は a レジスタ (8 ビット) と ax レジスタ (16) ビットでしたが、ax の下半分の 8 ビットが a レジスタと一緒でした。そして 32 ビット、64 ビットと増やしたときに上にビットを追加して eax と rax ができました。今回主に使うのは 32 ビットと 64 ビットですから、この 2 つを使います。

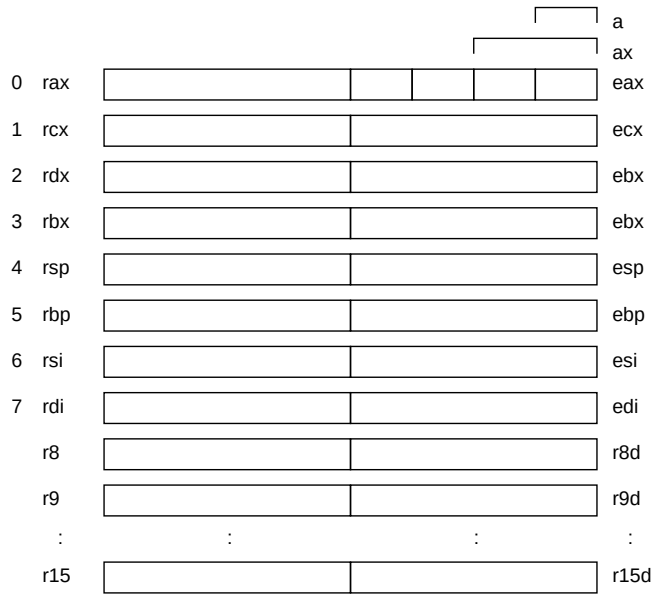

図 8: x86-64 CPU が持つレジスタ群

図にあるように、残りのレジスタも個々の役割と名前を持っていましたが、今では特定目的に使うのは rsp と rbp くらいで、あとは「同等の汎用目的レジスタが沢山ある」ものと思ってよいです。x86-64 になったときにレジスタを 8 個増やしたので、これらはもう固有の名前をつけずに r8-r15 となっています。すべて、32 ビットの名前 (右側) と 64 ビットの名前 (左側) を持つのに注意。機械語の上ではレジスタは番号で表すので、その番号も付記してあります。

特殊目的のレジスタが 2 つと書きましたが、そのうち rsp についてはハードウェア命令と関係があります。命令「pushq レジスタ」は、rsp を 8 減らしてから、指定したレジスタの内容 (8 バイト) を rsp の指している番地以下 8 バイトに書き込みます。「popq レジスタ」はその逆です。これにより、スタックにレジスタ内容を退避/回復できます。なお、スタックに積むときに rsp を「減らす」ということは、x86-68 ではスタックは上向き (低い番地の向き) に延びるということになります。

また、call(手続き呼び出し) 命令も同様に、現在の命令ポインタを戻り番地としてプッシュしてから指定宛先にジャンプし、ret(手続きからの戻り) 命令は戻り番地をポップして命令ポインタに戻すことで呼んだ箇所に戻るようになっています。

rbp については、先に説明したフレームポインタとして使用します。これについては次節で説明しましょう。

3.2 呼び出し規約

プログラムを実装するときに手続き呼び出しは不可欠ですが、ハードウェア命令として提供されている call/ret では引数や返値には感知していません。そこで、呼ぶ側と呼ばれる側でどのようにして引数や値を受け渡すかを決めてそれに従いコードを生成することになります。この約束ごとのことを**呼び出し規約** (calling convention) と呼びます。

ライブラリの手続きとも整合が取れている必要があるため、呼び出し規約は 1 つの OS の中で統一されていることが普通です (OS のライブラリを呼ばない閉じた言語処理系ではこれと別でもよい)。

ここでは x86-64 が動く Unix 系 OS の規約を説明します。上で説明したように、もともとは手続きの引数はスタックで渡すのが通常でしたが、レジスタが多数ある CPU ではメモリ (スタックは結局メモリの一部です) に格納するよりレジスタで渡す方が高速なため、先頭のいくつかの引数はレジスタで渡すようになりました。

具体的には先頭から 6 個の引数は順に`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` に載せられて渡されます。それらより多い引数はスタックに置かれます (実際には引数がそんなに多いことはあまりないですが)。返値については`%rax`(128 ビット値のときは加えて`%rdx` も使う) に置かれて返されます。[3]

レジスタの退避回復については、`%rsp`, `%rbp`, `%rbx`, `%r12`-`%r15` は呼ばれ側保存なので (最初の 2 つはスタックとベースポンタですが) 使う場合には退避回復が必要です。あとのレジスタは値を変更して構いません。

では練習として、「2 つの整数を受け取りその和を返す」手続きを作り、C 言語から呼び出してみましょう。まず C 言語側を示します。

```
#include <stdio.h>
int sub(int a, int b);
int main(void) {
  printf("%d\n", sub(3, 9)); return 0;
}
```

簡単ですね。これと一緒に使うアセンブリ言語で書いた `sub` を示します。

```
        .text
        .globl  sub
        .type   sub, @function
sub:    pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, %eax
        addl    %esi, %eax
        leave
        ret
```

最初の 3 行はアセンブリ言語への指示で「実行コード (テキスト) セグメントであること」「`sub` は外部参照される記号であり」「種別は関数であること」を示しています。これらの情報は翻訳後のファイルに含まれてリンカに渡されます。

その後の `sub:`からがコードです。まず`%rbp` をスタックに積み、その場所は`%rsp` が指しているのでそれを`%rbp` にコピーします。これで動的チェインができました。次に`%rsp` を動かしてローカル変数の場所を確保するのが通常ですが、ここでは全部レジスタでやっていてスタック上の場所は使わないので何もしません。

[3] いずれも 64 ビットレジスタの名前を挙げましたが、32 ビットの値では 32 ビット側だけを使います。以下同様。

次の 2 命令が sub の本体で、1 番目のパラメタは%edi(整数なので 32 ビット) に入っているので、それを%eax にコピーし、次の命令で%esi に入っている 2 番目のパラメタを加算します。これでぶじ、返値を入れるべきレジスタ%eax に和が入りました。

あとは戻りですが、%rsp は変更しなかったので、すぐに leave 命令 (スタック位置から%rbp をポップ) と ret 命令 (スタック位置から戻り番地をポップしてジャンプ) を実行します。

では動かしてみましょう。

```
% gcc sam_b1.c sam_b1sub.s
% ./a.out
12
```

もう少し込み入った、配列アクセスとジャンプ・条件ジャンプのある例を示しましょう。今度は sub は配列内の値の最大値を求めます。

```
#include <stdio.h>
int sub(int n, int a[]);
int b[] = { 5, 1, 6, 8, 2, 7, 4, 3 };
int main(void) {
  printf("%d\n", sub(8, b)); return 0;
}
```

ジャンプ命令は jmp、条件ジャンプ命令は演算命令か比較命令 (cmpl 等) の直後でのみ利用でき、jl(より小)、jle(以下)、jg(より大)、jge(以上)、je(等しい)、jne(等しくない) があります。

「0(%rsi)」は前にやったようにレジスタの指している場所から 0 ずれた位置 (ということはレジスタの指している場所) をアクセスするので、ここでは 2 番目のパラメタで渡された配列の先頭要素を取り出します。そして、「0(%rsi,%rdi,4)」というのは、上記位置からさらに%rdi に添字が入っているものとしてその添字に対応する位置をアクセスします (4 は 1 つの要素が 4 バイトであることを表す)。問題はこの場合 64 ビットの%rdi を使う必要があることで、そのため%edi を演算したあと (dec は 1 減らす演算)、cltq(convert long to quad) 命令で 32 ビットを 64 ビットに拡張しています。

```
        .text
        .globl  sub
        .type   sub, @function
sub:    pushq   %rbp
        movq    %rsp, %rbp
        movl    0(%rsi), %eax
.L1:    dec     %edi
        cltq
        jl      .L2
        movl    0(%rsi,%rdi,4), %edx
        cmpl    %edx, %eax
        jge      .L1
        movl    %edx, %eax
        jmp     .L1
.L2:    leave
        ret
```

プログラムの構造としては、まず%eax に配列の先頭要素を入れ、それから配列の各要素を順に取り出して%eax とくらべ、大きいようならその値を%eax にコピーします。順に取り出すには、%edi に要素数が入っているので、それを 1 ずつ減らし、負なら終わるというループを使っています。

実行のようすを示します。あまり面白くないですが、要素数が 100 万でもちゃんと動きます。

```
% gcc sam_b2.c sam_b2sub.s
% ./a.out
8
```

演習 1 上の 2 つの例題を打ち込んで動かせ。動いたら次のことをやってみなさい。

a. 1 番目の例題で、演算を足し算以外のものにして動かしてみよ。
b. 2 番目の例題で、演算を「配列要素の合計」にして動かしてみよ。
c. 2 番目の例題を改造して「配列を全部クリアする」「配列の要素を全部 1 つずつ前に動かす (先頭にあったものは末尾に移す)」などを作ってみよ。
d. C 言語で同じ処理を書いたものと実行時間を比較してみよ。C 言語側で最適化 (-O4) をオプションを指定した場合としない場合で違うので両方比べること。
e. gcc では「-S」オプションでコンパイルして出力されたアセンブリ言語コードを残すようになっている。これを用いて、C コンパイラの出力コートと手で書いたコードの比較をおこなってみよ。C 言語側で最適化 (-O4) をオプションを指定した場合としない場合で違うので両方比べること。

4 C から呼べる関数コードを生成する処理系

それではいよいよ、実際に x86-64 のコードを生成する処理系を作ってみます。といっても、型検査までは前回と同様におこなうので新しいところは生成部分だけです。ただ、言語の構文を少し変えていますので、その差分だけ見ていきます。まず SableCC の記述ファイルから。

```
Package samb3;    ←パッケージ名が違う
(途中略)
Productions
  prog = {main} int ident lpar dcls rpar stat  ←関数ぽい形
       ;
  dcls = {lst} dcls semi dcl      ←パラメタ部
       | {one} dcl
       ;
  dcl  = {idcl} int ident
       | {adcl} int ident lsbr rsbr
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
  stat = {idcl}    int ident semi    ←以下は同じだが read/print は無し
(以下略)
```

コンパイラドライバは Executor の変わりに Generator にしました。

```
package samb3;
import samb3.parser.*;
import samb3.lexer.*;
import samb3.node.*;
```

```
import java.io.*;
import java.util.*;

public class SamB3 {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JISAutoDetect"),
        1024)));
    Start tree = p.parse();
    Symtab st = new Symtab();
    HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
    TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(tck); st.show();
    if(Log.getError() > 0) { return; }
    Generator gen = new Generator(st, vtbl); tree.apply(gen);
  }
}
```

TypeChecker は同じですが、構文が変わったところとコード生成の準備に関して対応して少しだけ変更しています。まずメインの入口で記号表にダミー変数と関数名の変数を登録しています。後者は、この言語では関数が返す値は関数名と同名の変数に代入しておくという仕様にしたので、そのための場所に使います。前者は変数のオフセットが 0 のところは動的チェインの場所なので、そこをあけておくためです。あとは、関数のパラメタ部の宣言も後で使うのでそれぞれオフセットテーブルに登録するだけです。

```
package samb3;              ←パッケージが変更
import samb3.analysis.*;
import samb3.node.*;
import java.io.*;
import java.util.*;

class TypeChecker extends DepthFirstAdapter {
  (途中略)
  @Override
  public void inAMainProg(AMainProg node) {
    st.addDef("$dummy$", Symtab.ITYPE);
    st.addDef(node.getIdent().getText(), Symtab.ITYPE);
  }
  @Override
  public void outAIdclDcl(AIdclDcl node) {
    Symtab.Ent e = st.addDef(node.getIdent().getText(), Symtab.ITYPE);
    vtbl.put(node, e.pos);
  }
  @Override
  public void outAAdclDcl(AAdclDcl node) {
    Symtab.Ent e = st.addDef(node.getIdent().getText(), Symtab.ATYPE);
    vtbl.put(node, e.pos);
  }
```

(以下略)

では Generator を呼んでいきましょう。内部でファイル「asm.s」に書き込む PrintStream を保持しておきます。そしてメインの入口では先に見て来たような定型の部分を出力します。メインの出口では終わりの部分を出力します。なお、スタックの所要量については最後まで生成しないとわからないので、冒頭では.STSIZE という記号で値を参照しておき、最後でその実際の値を定義しています。

```
package samb3;
import samb3.analysis.*;
import samb3.node.*;
import java.io.*;
import java.util.*;

class Generator extends DepthFirstAdapter {
  HashMap<Node,Integer> pos;
  Symtab st;
  PrintStream pr;
  static String preg[] = {"%rdi", "%rsi", "%rdx", "%rcx", "%r8", "%r9"};
  int pcnt = 0, lcnt = 0;
  public Generator(Symtab s, HashMap<Node,Integer> p) throws Exception {
    st = s; pos = p; pr = new PrintStream(new File("asm.s"));
  }
  @Override
  public void inAMainProg(AMainProg node) {
    String name = node.getIdent().getText();
    pr.printf(" .text\n");
    pr.printf(" .globl %s\n", name);
    pr.printf(" .type %s, @function\n", name);
    pr.printf("%s: pushq %%rbp\n", name);
    pr.printf(" movq %%rsp, %%rbp\n");
    pr.printf(" subq $.STSIZE, %%rsp\n");
  }
  @Override
  public void outAMainProg(AMainProg node) {
    pr.printf(" movl -8(%%rbp), %%eax\n");
    pr.printf(" addq $.STSIZE, %%rsp\n");
    pr.printf(" leave\n");
    pr.printf(" ret\n");
    pr.printf(".STSIZE = %d\n", st.getGsize()*8);
  }
```

次は関数の入口部分の変数宣言ですが、配列も整数もスタック上のオフセットの位置 (すべて%rbp より上なのでマイナスの値です) にパラメタとして渡されて来た値をを格納します。どのレジスタが何番目かは配列 preg に入れてあります (スタック渡しには対応していません)。

```
  @Override
  public void outAIdclDcl(AIdclDcl node) {
    pr.printf(" movq %s, -%d(%%rbp)\n", preg[pcnt++], pos.get(node)*8);
```

```
}
@Override
public void outAAdclDcl(AAdclDcl node) {
  pr.printf(" movq %s, -%d(%%rbp)\n", preg[pcnt++], pos.get(node)*8);
}
```

さて、ここから様々な構文のコード生成です。この処理系では、すべての式の値はスタック上の変数に格納されるようにしています。このため、変数でない式があるごとに、その式の値を入れるテンポラリ (temporary、作業変数) を生成し、`setOut()` でそのオフセットを上に返すようにしています。なので、代入の場合は式の値を取り出して左辺の変数 (そのオフセットは表 `pos` に入っていましたね) に格納すればすみます。配列代入の場合は、添字式も同様に取り出しますが、取り出すレジスタは`%edx` にして 64 ビットに変換してアクセスに使用します。

```
@Override
public void outAAssignStat(AAssignStat node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" movl %%eax, -%d(%%rbp)\n", pos.get(node)*8);
}
@Override
public void outAAassignStat(AAassignStat node) {
  pr.printf(" movq -%d(%%rbp), %%rcx\n", pos.get(node)*8);
  pr.printf(" movl -%s(%%rbp), %%edx\n", getOut(node.getIdx()));
  pr.printf(" cltq\n");
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" movl %%eax, 0(%%rcx,%%rdx,4)\n");
}
@Override
```

いよいよ、制御構造の if と while です。いずれも、分岐先のラベルがいるのでそのラベル番号を変数 `lbl` に入れてカウンタを進めます。先頭でこれをやるのは、処理の途中で別の (条件や本体の) コードを生成するとそこでもラベルを生成するため番号がごちゃまぜにならないようにしたものです。if はまず条件式を持って来てその真偽 (0 でなければ真) に基づき本体を迂回するラベルにジャンプします。While もよく似ていますが、先頭へのジャンプと迂回のジャンプで 2 つラベルが必要なところが主な違いです。

いずれもジャンプ命令の後、本体を生成し、最後のラベルを生成します。このように子ノードの処理を呼び出すタイミングを制御する必要があるため、`case...` のメソッドをオーバライドしています。

```
public void caseAIfStat(AIfStat node) {
  int lbl = lcnt; lcnt += 1;
  node.getExpr().apply(this);
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
  pr.printf(" testl %%eax,%%eax\n");
  pr.printf(" jne .L%d\n", lbl);
  node.getStat().apply(this);
  pr.printf(".L%d:\n", lbl);
}
@Override
public void caseAWhileStat(AWhileStat node) {
```

```
    int lbl = lcnt; lcnt += 2;
    pr.printf(".L%d:\n", lbl);
    node.getExpr().apply(this);
    pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getExpr()));
    pr.printf(" testl %%eax,%%eax\n");
    pr.printf(" je .L%d\n", lbl+1);
    node.getStat().apply(this);
    pr.printf(" jmp .L%d\n", lbl);
    pr.printf(".L%d:\n", lbl+1);
  }
  @Override
```

条件式では、2 つの値を持って来て cmpl 命令で比較して正否により 0 または 1 を%eax に入れ、その値を最後にテンポラリに格納します。このため、格納する前にテンポラリを記号表に追加登録し、そのオフセットを使用しています。また、最後にそのオフセットを setOut() で登録して親ノードに渡します。

```
  public void outAGtExpr(AGtExpr node) {
    pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getRight()));
    pr.printf(" cmpl -%s(%%rbp), %%eax\n", getOut(node.getLeft()));
    pr.printf(" jg .L%d\n", lcnt);
    pr.printf(" mov $1, %%eax\n");
    pr.printf(" jmp .L%d\n", lcnt+1);
    pr.printf(".L%d: movl $0, %%eax\n", lcnt++);
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    pr.printf(".L%d: movl %%eax, -%d(%%rbp)\n", lcnt++, f.pos*8);
    setOut(node, new Integer(f.pos*8));
  }
  @Override
  public void outALtExpr(ALtExpr node) {
    pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getRight()));
    pr.printf(" cmpl -%s(%%rbp), %%eax\n", getOut(node.getLeft()));
    pr.printf(" jl .L%d\n", lcnt);
    pr.printf(" mov $1, %%eax\n");
    pr.printf(" jmp .L%d\n", lcnt+1);
    pr.printf(".L%d: movl $0, %%eax\n", lcnt++);
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    pr.printf(".L%d: movl %%eax, -%d(%%rbp)\n", lcnt++, f.pos*8);
    setOut(node, new Integer(f.pos*8));
  }
```

ここから先は演算命令なので、条件分岐がないぶん、先の条件演算子よりは簡単です。いずれも値を格納するためのテンポラリを割り当てます。構文上の必要から生じている単一規則では、テンポラリのオフセットを上にコピーする作業だけおこないます。

```
  @Override
  public void outAOneExpr(AOneExpr node) { setOut(node, getOut(node.getNexp())); }
  @Override
```

```
public void outAAddNexp(AAddNexp node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getNexp()));
  pr.printf(" addl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
@Override
public void outASubNexp(ASubNexp node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getNexp()));
  pr.printf(" subl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
@Override
public void outAOneNexp(AOneNexp node) { setOut(node, getOut(node.getTerm())); }
@Override
public void outAMulTerm(AMulTerm node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getTerm()));
  pr.printf(" imull -%s(%%rbp), %%eax\n", getOut(node.getFact()));
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
@Override
public void outADivTerm(ADivTerm node) {
  pr.printf(" movl -%s(%%rbp), %%eax\n", getOut(node.getFact()));
  pr.printf(" cltd\n");
  pr.printf(" idivl -%s(%%rbp)\n", getOut(node.getTerm()));
  Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
  setOut(node, new Integer(f.pos*8));
}
@Override
public void outAOneTerm(AOneTerm node) { setOut(node, getOut(node.getFact())); }
@Override
```

一番最後の因子のところですが、整数はその整数を割り当てたテンポラリに格納します。変数は新たな変数を使わなくても、もとの変数のオフセットをそのまま使えば済みます。配列アクセスは配列代入と類似ですが、こちらは取り出した値をテンポラリに格納する必要があります。

```
public void outAIconstFact(AIconstFact node) {
  Symtab.Ent e = st.addDef(".t" + pcnt++, Symtab.ITYPE);
  pr.printf(" movl $%s, %%eax\n", node.getIconst().getText());
  pr.printf(" movl %%eax, -%d(%%rbp)\n", e.pos*8);
  setOut(node, new Integer(e.pos*8));
```

```
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, new Integer(pos.get(node)*8));
  }
  @Override
  public void outAArefFact(AArefFact node) {
    pr.printf(" movq -%d(%%rbp), %%rdx\n", pos.get(node)*8);
    pr.printf(" movl -%s(%%rbp), %%ecx\n", getOut(node.getExpr()));
    pr.printf(" cltq\n");
    pr.printf(" movl 0(%%rdx,%%rcx,4), %%eax\n");
    Symtab.Ent f = st.addDef(".t" + pcnt++, Symtab.ITYPE);
    pr.printf(" movl %%eax, -%d(%%rbp)\n", f.pos*8);
    setOut(node, new Integer(f.pos*8));
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOut(node.getExpr())); }
}
```

では、小さな言語で書いた「バブルソート」のプログラムを示します。C とまったく一緒に見えますがちょっとだけ文法が違うのに注意 (どこでしょう?)。

```
int sub(int n; int a[]) {
  int remain; remain = 1;
  while(remain) {
    remain = 0; int i; i = 1;
    while(i < n) {
      if(a[i]>a[i-1]) { int z; z=a[i-1]; a[i-1]=a[i]; a[i]=z; remain=1; }
      i = i + 1; } } }
```

これを呼び出す C 言語側はたとえばこんな感じです。

```
#include <stdio.h>
int sub(int n, int a[]);
int main(void) {
  int a[] = { 7, 1, 6, 3, 2, 4, 8 };
  sub(7, a);
  for(int i = 0; i < 7; ++i) { printf("%d\n", a[i]); }
}
```

演習 2 「小さな言語」処理系を動かし、上の例題を翻訳して asm.s を生成し、gcc main.c asm.s で C 言語コードと結合して動かしてみなさい。動いたら次のことをやってみなさい。

a. もっと長い配列を渡すようにして (配列には逆順に並んだ値を入れておくとよい)、所要時間を計測してみなさい。C 言語で同じプログラムを (文法は少し修正必要) コンパイルして動かし、CPU 消費量を比較しなさい。-O で最適化してみるとさらによい。CPU 消費は「time ./a.out」のように time コマンドを使うことで計測できる。

b. 上記において生成された asm.s を見て「効率が悪い原因」を検討してみなさい。手で asm.s を修正して速度の向上を試み、実際にどれくらい向上したか計測してみなさい。

c. 「小さな言語」でもっと別のプログラムを作って動かしてみなさい。性能計測もできるとなおよい。

5 課題 11A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題# 11」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。
  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。
  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。