Duskul実行時のスタックについて

Version 1.0.3 (2019.06.02) 荻原剛志

1. 概要

Duskul 処理系では、スタックを利用して式の評価、サブルーチンの呼び出しなどを実現しています。この文書では、スタックを使った式の評価の概要と、サブルーチン呼び出しの際の挙動について説明します。

まず、スタックは実行開始直前に固定長のメモリ領域が確保されます。領域の大きさは long 型で4096個分で、マクロ STACK_SIZE で定義されています（変更可能）。この領域を参照する変数名は stack です。大きさが変化しないため、stackは配列と見なして扱えます。

スタックの最も上の要素（スタックトップ）を指すためのスタックポインタ（以下、SPと呼ぶ：stack pointer）は変数 sp で表します（図1）。スタックはアドレスの大きい方から小さい方へ伸びるように構成し、spの初期値は STACK_SIZE です。

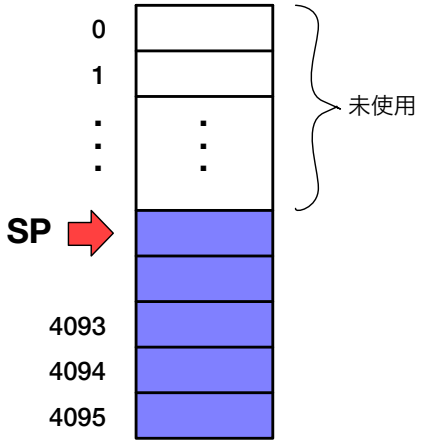


図1. スタックの概要

push操作は次のように記述できます。

```
stack[--sp] = 式;
```

同様にpop操作は次のように記述できます。

```
変数 = stack[sp++];
```

2. 式の評価

2.1 木構造としての式

コンピュータでは、メモリ装置から取り出されたデータはレジスタに置かれ、四則演算や論理演算もレジスタ上の値に対して適用されます。ただし、レジスタは数が少ないため、複雑な計算を行う場合、計算の途中結果をメモリに一時的に記憶させる必要が出てきます。このために式を木構造として捉え、式の評価（計算）を再帰的に行うと同時に、途中の計算結果をスタックに記憶するようにします。

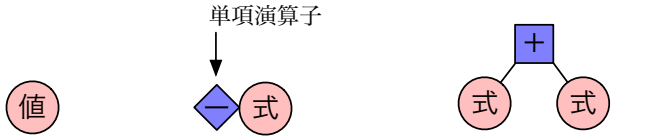


図2. 式の構造

基本的にコンピュータでの演算は、四則演算のように２つの値に対して適用するものと、符号の反転や論理否定のように１つの値に対して適用するものの組み合わせでできています。これを整理すると次のように言うことができます。

(a) 式は値（定数または変数）である

(b) 式は別の式に単項演算子（ − や not）を適用したものである

(c) 式は別の２つの式に二項演算を適用したものである

これを図で表すと図２のようになります。式は再帰的な構造を持ち、木構造で表現できることが分かります。

2.2 式の再帰的な評価

例として、x*(s-t)+y*z という式の構造を図3に示します。Duskulでは、式は構文解析によってこのような木構造を持つデータとして表現されます。

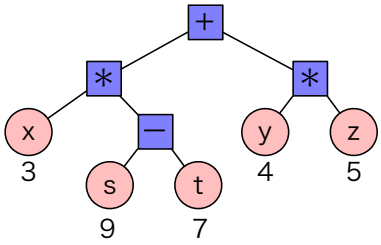


図3. x*(s-t)+y*z の木構造

この木構造に従って式の値を評価できます。まず、構文木を評価して値が得られたら、その値はスタックにプッシュすることにしておきます。ある部分木の値を評価するには、その左右の部分木（または葉）をそれぞれ評価します。すると、左右に対応する２つの値がスタックに積まれますので、それらをポップして二項演算子を（さらに単項演算子があればそれも）適用します。得られた値はスタックにプッシュします。この手順を再帰的に繰り返します。

例えば、図3の例で、各変数が図に示すような値を持っているとします。この場合に、どんな順序で評価が進み、スタック上にどのように値がプッシュ、ポップされるかを示したのが図4になります。

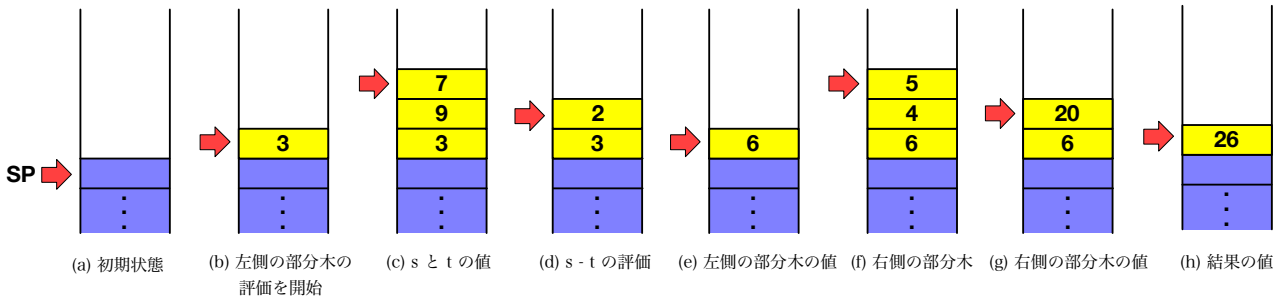


図4. 木構造の再帰的な評価とスタック

3. サブルーチンの呼び出し

3.1 サブルーチン呼び出しの手順

サブルーチン（手続き、または関数）を新たに実行するためには、サブルーチンに与える実引数を用意し、サブルーチンの実行に必要な局所変数（ローカル変数）の領域を確保しなければなりません。また、現在実行中の情報を保存しておき、サブルーチンの実行が終了したらその実行に復帰する必要があります。これらの一連の手順は、C言語などの、スタックを利用する一般の言語に共通しています。概要は次のようになります。

(1) 実引数をスタック上に用意する

(2) サブルーチン側に分岐（マシン命令のcallなど）

(3) SPなどを操作して、スタック上に制御用の情報と局所変数の領域を確保する

(4) サブルーチンのコードを実行する

(5) SPなどを操作して、スタックをサブルーチンを呼びだす前の状況に戻す

(6) サブルーチンの呼び出し元に復帰する（マシン命令のrtnなど）

(7) 関数からの戻り値があれば利用できるようにする

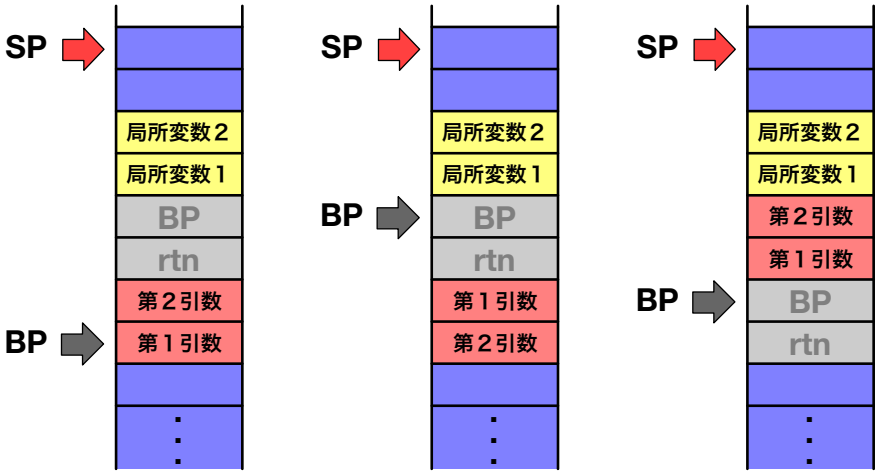
ただし、詳細は各処理系によって異なります。以下ではDuskul処理系での実装を中心に述べますが、C言語の代表的な実装についてもコメントします。

3.2 スタックフレーム

サブルーチンの実行のための情報は、スタック上に一定の決まりに従って配置されます。この構造のことをスタックフレーム(stack frame)と呼びます。サブルーチンを呼び出す手順の中で、新しいスタックフレームが構築されます。

スタックフレームの実装の詳細は、言語やコンパイラによって異なります。図5(a)はDuskulで採用している方式を示しています。図5(b)は、教科書などで解説されることが多い一般的なC言語のスタックフレームです。参考までに、図5(c)はXcode標準のコンパイラであるclangが生成するスタックフレームです。

SPは式の評価などによって増減しますので、サブルーチンの引数、局所変数にアクセスするために別の基準点が用意されます。これがベースポインタ(base pointer)、またはフレームポインタと呼ばれるもので、サブルーチンの実行中には変化しません。図5(a)の方法では、ベースポインタ（以下、BPと記述）にマイナスの変位を加えることで実引数と局所変数にアクセスできます。一方、図5(b)では、実引数はプラスの変位、局所変数にはマイナスの変位を加えてアクセスします。この方式では、実引数はスタックに逆の順序で積みます。



(a) Duskulの実装　(b) 一般的なC言語の実装　(c) LLVMの実装

図5. スタックフレーム

スタック内の"BP"は、そのサブルーチンが呼び出される前のBPの値で、サブルーチンから復帰した時にこの値をベースポインタに戻します。rtnは、マシン語のサブルーチン呼び出し命令（callなど）によってスタック上にpushされる復帰アドレスです。Duskulでは使いません。図5(a)、(b)では、実引数と局所変数の間にBPとrtnが挟まっています。これは、実引数をスタックに積んだ後でcall命令が実行されるためです。

DuskulではBPは**localbase**という変数で表されます。また、引数と局所変数の変位（オフセット）は、構文解析の際に出現順に割り当てられます。

3.3 サブルーチン呼び出しの詳細

Duskulでサブルーチンを呼び出す手順を説明します。

まず、図6(1)はサブルーチンを呼び出す前のスタックで、BPはどこか適切な位置に設定されているとします。図6(2)から、実引数を必要な個数だけスタックに積んで行きます。実引数は一般の式ですので、実引数を得るためにスタックを使った計算や、別の関数呼び出しが行われることもあります。ここでは引数を3個積んでいます。通常のコンパイル言語では図6(3)の時点でマシン語のcall命令などでサブルーチンに分岐し、スタック上に復帰アドレスがpushされます。この部分はDuskulでは関係ありません。サブルーチン側の処理として、図6(4)でBPをスタック上にpushすると同時に、実引数の先頭を指すようにBPを更新します。次に、図6(5)のようにSPを操作して、サブルーチンが利用する個数分の局所変数の領域を確保します。これで、サブルーチンのコードを実行できます。

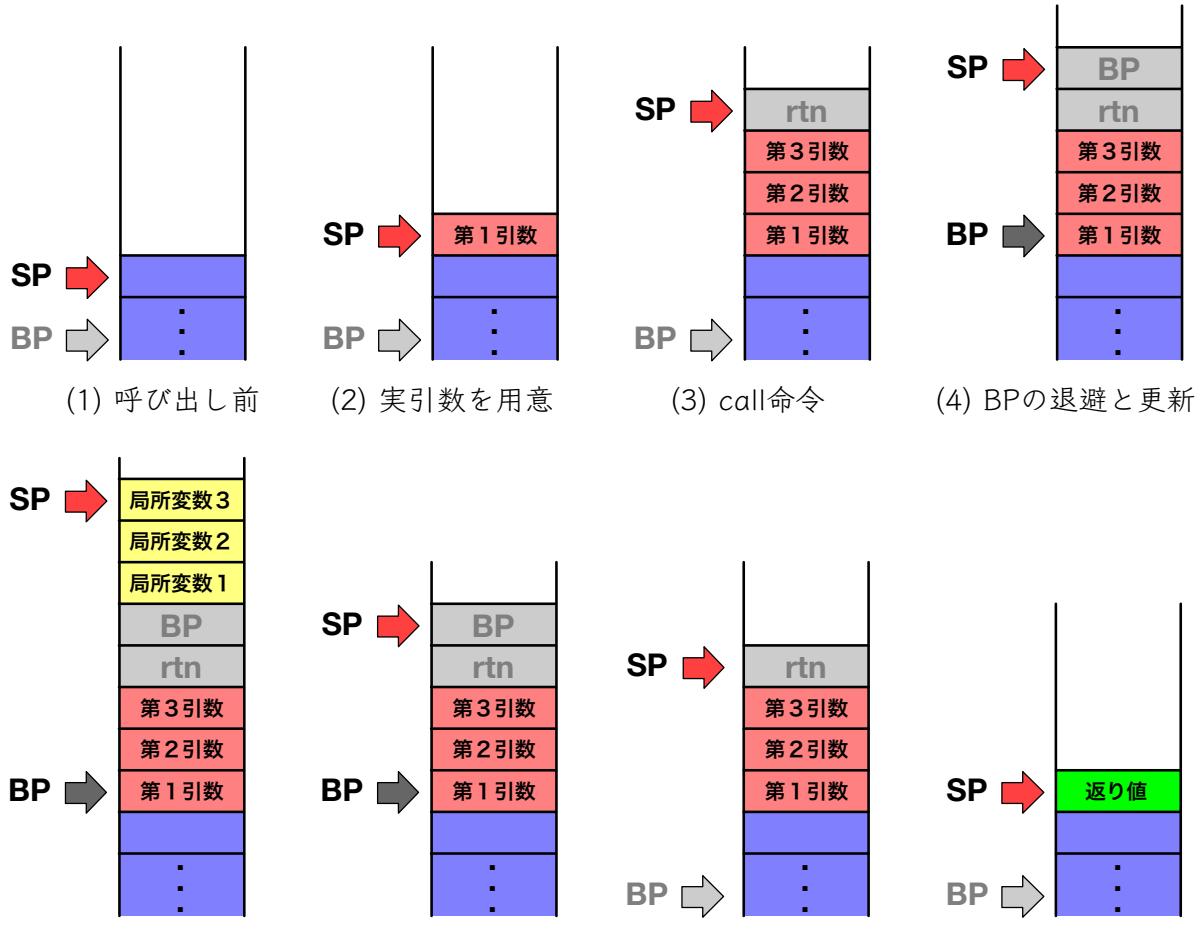


図6. Duskulのスタックの動作

サブルーチンの実行が終わったら図6(6)のようにSPを戻します。ここでBPをpopして復元すると図6(7)のようになります。通常のコンパイル言語ではこの時点でマシン語のrtn命令などを実行して呼び出し元のコードに復帰しますが、Duskulでは関係ありません（デバッグ用の情報が格納されています）。引数の個数分だけSPを戻せば呼び出しの手順は終了です。ただし、関数を呼び出した場合、図6(8)のようにスタックトップに返り値を積んでおく必要があります。