■電子ブロックの復活

私が小学生だったころ、電子ブロック(電子ボードだったかも？)を買ってもらって、説明書がボロボロになるくらいに遊びまくっていました。そして最近、学研では「大人の科学」という科学キットを売り出し始め、第7弾として「電子ブロック」が商品化されることになったようです。

そのWebサイト(記事末のResource【1】を参照)で見ると、私が持っていたものよりずっとスマートな商品になっています。私のは、白い基板に透明プラスチックのブロックでアンプやスピーカはなく、クリスタルイヤホンとクリスタルマイクが主な入出力装置だったと記憶しています。当時は回路の知識もなく説明書の通りに配線を接続するだけでしたが、本当にあのキットの面白さを理解するには中学生か高校生程度の数学と電子工学の知識が必要でしたね。

中学生くらいになると「ラジオの製作」という雑誌を読みながら、自分で揃えた部品を使って回路を作っていたのでキットの出番はなくなりましたが、自分の子供にも1度は遊ばせたいと思う一品です。読者の皆さんの中にも電子ブロックで遊んだ世代がいるのではないかと思いますが、お子さんがいる方はぜひ子供達にあの感動を味あわせてあげてください。

衣食足りて礼節を知ると言われますが、日本の1人あたりの生産性は先進国中最低だとか、IT不況だとか、主要電機メーカーがリストラ(傍からみていると構造改革って意味じゃなくただの人員削減みたいですね)とか明るい話題が少ないからか、悲惨なニュースが増えているような気がしています。この記事が掲載されるころには少しは景況も上向いているといいですね。

私のところでは、企業からの唯一のサポートであったAlphaマシンの貸し出し機も返却することになりました(これはAlphaの連載を止めたから自業自得ですけど)。OSのカーネルの試験などでは、自分のマシンでないと思うようにデバッグできないのですが、OSDLあたりを使って細々とプロジェクトを続けていこうかと思うこのごろです。

■コードの生成

コンパイラの作成のために基本的な道具立てを揃えてきましたが、いよいよ中心部分であるコード生成の話をします。コード生成というのは、コンパイラがプロセッサの命令を生成することをいいます。今までの準備の中でもコンパイラのソースコードとプロセッサの命令の関連については議論してきましたが、実際にプロセッサの命令列を作成するにはレジスタの割り当てやプログラムの流れの制御などいくつかの技術課題があります。

yacc/lexによるコンパイラの設計ではLR法によるコード生成が簡単でよく使われます。この方法では文法の解析中に解析木を作成し、解析が終わった解析木に対してコードの生成を行います。文法解析で問題となるのが演算子の優先順位や括弧などによる優先順位の変更です。出てきた順に計算するならばコード生成は簡単なのですが、演算の順序が解析木に出現する順序と異なるときには順序の変更に対応したコードを出す必要があります。

このとき、ターゲットのプロセッサがスタックマシンなら簡単で、スタックに途中結果を保留しておいて後から使えばよいだけです。以下の説明は2001年12月号で示したコンパイラのyaccソースを参照しながら読んでください。

例えば、次のような式を考えます。

```
10+20*4
```

これを解析すると最初の項は定数ですから、INTEGERとして切り出されます。演算子は「+」と「*」となっているので、この式は

```
INTEGER + INTEGER * INTEGER
```

のようにトークンが切り出されることになります。このトークンの列に対して文法の解析を行いますが、「+」も「*」も二項演算子として定義されているので、

```
(INTEGER + INTEGER) * INTEGER
```

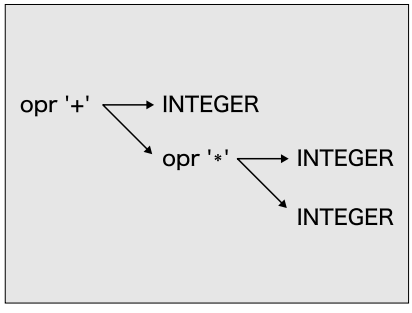
になるのか、

```
INTERGER + (INTEGER * INTEGER)
```

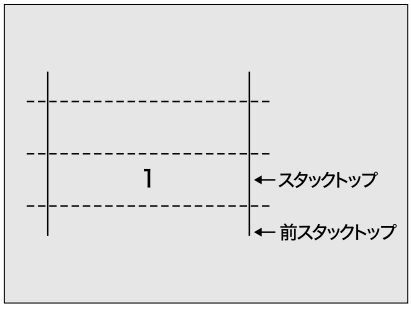
なのかを判定しなくてはなりません。yaccのファイルでは後から宣言された演算子ほど高い優先順位を持っているので、我々の例では「*」が優先され後者が解析結果となります。そこで、解析木は図1となります。

ここまでは、12月号で示したyaccソースか

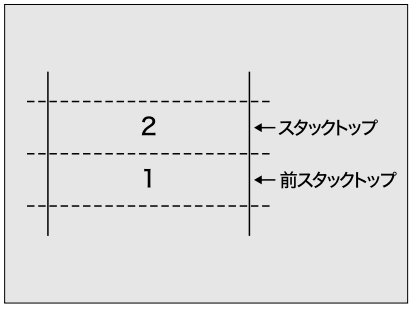
【図1】



【図2】



【図3】



ら自動的に生成できるものです。次にこの解析木からプロセッサの命令を生成することになります。話を簡単にするために、命令生成が簡単なプロセッサの例として、まずはスタックマシンを考えます(コンパイラの本などではスタックマシンをターゲットプロセッサとしているケースが多いような気がします。その方が簡単だからというのが大きな理由でしょうね)。

実は、我々のプロセッサはレジスタマシンなので、スタックマシンをターゲットとして考えて作ったコードはそのままでは動かせません。しかし、スタックマシンの動作は比較的簡単にレジスタマシンでもエミュレーションやコード変換ができるのでご安心ください。

スタックマシンとは、演算に必要なオペランドをスタックと呼ばれる「後入れ先出し(LIFO)」のバッファに積み上げ、演算はバッファの先頭のデータに対して行うアーキテクチャです。

例えば、スタックマシンで「1+2」を計算したい場合には

```
push 1
push 2
add
```

のように命令を並べます。これは1をスタック上に押し込んで(push)、次に2を押し込んで、最後にスタック上の2つのデータの加算を指示する(add)というような流れで計算します。計算結果は一般にスタックの一番上、これは次に取り出した(pop)とき1番先に取り出す要素に格納されます。

このタイプでは、データの場所を明示的に指定しなくても良いため、プログラムの扱いが簡単になります。その代わり、スタックにデータを入れたり出したりするたびに、ハードウェアがデータの操作以外にスタックのアドレス操作を行うことになって、ハードウェアの負担が若干増えます。

スタックを直接論理回路で実現することは普通行なわれず、メモリとアドレスを示すレジスタ(スタックポインタ)の組み合わせで実現します。多くのプロセッサではスタックはメモリの大きなアドレスがスタックの底になって、スタックにデータを押し込むごとにスタックポインタの値が自動的に小さくなっていきます。

先ほどの命令列を実行するときのスタックの動きを図示してみましょう。

```
push 1
```

の命令を実行すると図2のようになり、次に

```
push 2
```

の命令を実行すると図3のようになります。プロセッサによっては、スタックトップのメモリアドレスをスタックポインタに記憶するものと、スタックトップの次のアドレスをスタックポインタに記憶するものがありますが、動作上は、どちらもプッシュ動作によってスタックポインタが小さな値に更新されて、データがメモリに書き込まれます。

次に

```
add
```

の命令を実行すると、この命令は2つのデータを使用するので、スタックから2つのデータを取り出します。その後加算を実行して演算結果のデータを1つスタックに書き込みます(図4)。

スタックからデータを取り出すたびにスタックトップは大きなアドレスに変更されます。2つ取り出したら1つを書き込むので、結果としてスタックにあるデータは1つ減る(スタックポインタのアドレスは増加する)ことになります。

解析木は演算子とそのオペランドの組み合わせになっていますが、スタックマシンではスタック上のデータに対して指定された演算を行うため、スタックにデータを積み上げてから演算が指定されます。

前出の例では解析木で「+」演算子を見つけた後、2つの演算対象データ(オペランド)をスタックに積みにいく命令を生成します。ここで、最初のデータは定数なので、スタックに積み上げるpush命令を生成すればいいのですが、他方のデータは定数でなく、解析木のノードを指しています。このノードには「*」演算子があって演算結果を「+」のもう1つのオペランドとして扱うことになります。「*」側の演算を行うためにはやはり2つのオペランドがスタックに積まれている必要があって、解析木をたどって2つのINTEGERをスタックに積みます。その後、「*」を実行して積の結果をスタックに積んだ後、「+」の演算を実行します。

このように解析木を頭から見ていって再帰的にコード生成を繰り返せば、最終的に動作する命令コードが得られます。従って、先ほどの解析木をスタックマシンの命令操作に変換すると、

```
push 10
push 20
push 4
'*'
'+'
```

のようになります。スタックマシンの便利なところはプロセッサがデータのある場所を直接管理する必要がないところです。この命令列を見てもデータがどこにあるのかを指定する部分はありません。

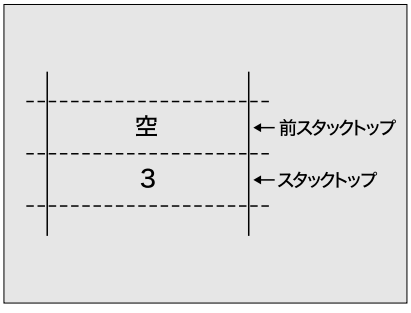
我々のターゲットプロセッサはレジスタマシンなので、この方法は直接は使えません。しかし、あまり複雑に考える必要はありません。スタックマシンが命令中にデータの場所を指定していないというのは、暗黙の指定を使っているからなのです。これは、常にスタックポインタという、レジスタで指定されるメモリアドレスからの相対的なデータの位置がプログラムで把握できており、その暗黙のデータ位置を使って命令列を構成しているということです。従って、スタックマシンの命令であっても命令ごとに扱うデータの場所は決っているといえます。どのデータをどこに置くのかは、スタックマシンがスタック上のデータの並びを暗黙のうちに仮定していることと通じるところがありますが、レジスタマシンではデータを明示的にレジスタに置く必要があります。先ほどの命令列を使ってスタック上のデータの場所を考えてみましょう(表1)。

ここではスタックの深さを「+n」の形に表わしてみました。この命令列をレジスタマシンに移植するには、スタック上のデータをレジスタに割り付けていきます。例えば、4つのレジスタ$0、$1、$2、$3を有するレジスタマシンを想定します。$0は避けてレジスタの割り付けを行うと

```
+0  -> $1
+1  -> $2
+2  -> $3
```

のように割り付けることができます。すると、この命令列は定数ロード命令(LDI)、乗算命令(MULT)、加算命令(ADD)を有する仮想レジスタマシンの命令を用いて、

【図4】



【表1】

命令	読み出し	書き込み
push 10	-	+0
push 20	-	+1
push 4	-	+2
'*'	+2 +1	+1
'+'	+1 +0	+0

```
LDI   $1, 10
LDI   $2, 20
LDI   $3, 4
MULT  $2,$3,$2
ADD   $1, $1, $2
```

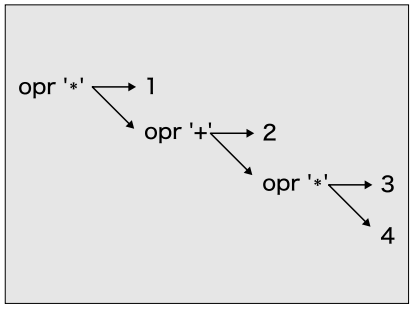
のように変換することができます。仮想といっているのは、我々はこのままの命令を持つレジスタマシンを作っているわけではなく、この例の説明のために導入したアーキテクチャであるという意味です。これでレジスタマシンの命令を使ってスタックマシンの命令コードを変換することができることが分かったと思います。

というところで、ちょっと待ってください。では、もっと複雑な式を変換するときにはどうなるのでしょうか？

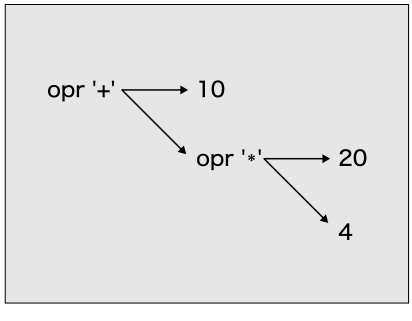
```
1*(2+3*4)
```

という演算を考えてみましょう。これを解析木に変換すると図5のようになります。スタックマシンを使って命令列を生成すると、表2のようになります。それではレジスタマシンの命令に変換しましょう……と、$0を使わないことにしてレジスタを割り付けようとすると、1つ足りません。このように、演算式を展開してレジスタを割り付けようとしたときにレジスタの本数が足りなくなることを、レジスタ溢れ(レジスタスピル)と言います。レジスタスピルが発生したときには、仕方がないので使っていないレジスタを1つメモリに退避します。レジスタのデータをメモリに書き込む命令はストア命令(ST)です。逆に、メモリのデータをレジスタに読み出す命令のことをロード命令(LD)と呼びます。メモリへの退避を含めて、レジスタマシンへの命令変換をしてみましょう。ここでは退避先のメモリアドレスをsaveとします。

【図5】



【図6】




```
LDI $1, 1
LDI $2, 2
LDI $3, 3
ST  $1, save
LDI $1, 4
MULT $3, $1, $3
ADD  $2, $2, $3
LD  $1, save
MULT $1, $1, $2
```

もっとデータを使うプログラムでは、メモリの退避領域をさらに追加すればいいのですが、このやり方ではどれだけ退避領域が必要なのかがプログラムを全部解析してみないと分かりません。それでも構わないのですが、スタックマシンの考え方を一部導入して、メモリの管理を簡単にすることを考えましょう。まず、前出の命令列においてSTとLD命令で退避していた部分をスタックへのpush及びpopにします。スタックへのpush、pop命令は、メモリのアドレスを明示的に示す必要はないためメモリの管理が楽になります。

```
LDI $1, 1
LDI $2, 2
LDI $3, 3
PUSH  $1
LDI $1, 4
MULT $3, $1, $3
ADD  $2, $2, $3
POP $1
MULT $1, $1, $2
```

もちろん、これではせっかくのレジスタマシンにスタック操作用の特別な命令が必要になるので、この命令をレジスタマシンの命令に置き換えます。レジスタSPがスタック領域の先頭を表わすレジスタだと仮定して、スタックに格納する1要素にアドレスが1ずつ付加されているとします。すると、PUSH、POPの命令を表3のようにレジスタマシンの命令に書き換えることができます。

【表2】

命令	読み出し	書き込み
push 1	-	+0
push 2	-	+1
push 3	-	+2
push 4	-	+3
mult	+2 +3	+2
add	+1 +2	+1
mult	+0 +1	+0

【表3】

PUSH X --->	SUBI SP,4 ST X, (SP)
POP X --->	LD X, (SP) ADDI SP,4

この表3のSUBIはレジスタからオペランドで示した数値を減算する命令で、ADDIはレジスタにオペランドで示した数値を加算する命令であるとします。また、「(SP)」はレジスタSPの示すアドレスにあるメモリ上の領域を意味すると考えます。つまりスタックにデータを格納する場合には、あらかじめスタックポインタ(SP)を4減らした上でスタックポインタの示すアドレスにデータを書き込み、スタックからデータを取り出す場合には、スタックポインタの示すアドレスからデータを読み出した後、スタックポインタに4を加えます。多くのプロセッサでは整数データは4バイトとなるので、この加減の数値は4となることが多いのですが、プロセッサによって違いがあります(この連載で利用するプロセッサでは1となります)。

レジスタの割り当ての方法についてもう少し考えてみましょう。はじめの演算式である

```
10+20*4
```

の解析木は図6のようになりました。コンパイラが、演算式に対応する解析木を抽出してからレジスタ割り当てを行う場合に、処理している解析木の場所によって、オペレータに対するレジスタの割り当てを変更する必要が生じます。解析木からコードを生成していくときには

1　第1オペランドを生成
2　第2オペランドを生成
3　オペレータを生成

の順序でコードの生成をします。ここで、第1及び第2オペランドの生成は、さらにオペレータの呼び出しに再帰的に展開されることがあることに注意してください。スタックの図にオペレータの第1、第2オペランドの別を書くと次の表4のようになります。

これから分かることは、同じ二項演算子の処理を行う場合であっても、オペランドとなるレジスタが切り替わるということです。そこで、命令コードの生成は、命令の置かれる場所により、生成するコードの利用するレジスタを切り替える必要が生じてきます。

さて、ここで我々のプロセッサの仕様に戻ると、まず、レジスタは全部で4つです。そのう

【表4】

命令	読み出し	書き込み
push 10	-	+0 --> 第1オペランド
push 20	-	+1 --> 第1オペランド
push 4	-	+2 --> 第2オペランド
'*'	+2 +1	+1 --> ../第2オペランド
'+'	+1 +0	+0

【表5】

A	LDI $1, 10	--> 第1オペランド
B	PUSH $1	--> 第1オペランド退避
	LDI $1, 20	--> 第1オペランド
	LDI $2, 4	--> 第2オペランド
	MULT $2, $1, $2	--> ../第2オペランド
	POP $1	--> 第1オペランド回復
A	ADD $1, $1, $2	

ち$0は命令によって特別な使い方をするので、コンパイラが自由に使うことは難しく、スタックポインタに1つレジスタを確保すると、残りは2つだけになってしまいます。

解析木の演算は二項演算が主なので、レジスタは最低2つあれば済みますから、コード生成はこの2つのレジスタをスタックとして利用することにします。

プロセッサの演算に2つのレジスタが必要ですが、この2つをスタックのトップのエントリとみなして、必要ならスタック領域を確保してレジスタを退避させます。スタック操作をPUSH、POPの擬似命令を使って書くと表5のような擬似コードが生成できます。

表5でAとBの領域の命令列を見てください。Aは「+」のオペレータのための命令列で、演算結果出力レジスタは$1になります。一方、Bは「*」のオペレータの命令列ですが、演算結果はAの命令列の第2オペランドとならなくてはいけないため、$2に演算結果を出力しています。このプロセッサでは、コンパイラが自由に利用可能なレジスタは$1と$2の2つだけになるので、演算結果をどのレジスタに出すのかはコンパイラの状態変数一つで簡単に切り分けられます(レジスタが多いときには、さらに多くの状態を記憶する状態変数を持てば良いのですが、どうやるのかは各自の楽しみにとっておきましょう)。

結果を$1に出力するのか、$2に出力するのかのレジスタ番号を記憶する変数をregとすると、他方のレジスタは

```
3 - reg
```

で計算できます(3-2=1、3-1=2というだけのことです)。解析木を再帰的にたどってコード生成する関数exの引数には、生成されたコードの結果がどちらのレジスタに出力するべきかを引数で渡す必要があります。

■コード解説

ここまでの準備で基本的な枠組みはできてきたので、実際のコードの解説に入ります。まずは、ソースコードの先頭部分ですが、リスト1のようになります。標準入出力とコンパイラ/インタプリタ共通の解析木に関する定義を利用し、さらにlexで生成したトークンの定義を用いるため、3つのファイルをインクルードしています。

また、スタティック変数を2つ定義しています。lblはラベルの通し番号を記憶する変数です。命令中で用いるラベルのアドレス解消はアセンブラにまかせるため、コンパイラではユニークなラベルを生成しておけばいいので、順番に番号を付けた「L014」のような文字列をラベルとして利用します。この文字列の中に出力する数値を変数lblに記憶しておきます。

frametopという変数はここではあまり役に立っていません。実は関数に局所変数を定義するときに、スタック領域の先頭と局所変数領域の先頭がどれだけずれているかを記憶する変数として用意していますが、そもそも局所変数として関数引数しか作っていないため出番があまりないのでした。プロセッサによっては、フレームポインタとスタックポインタを物理的に分けて持つものもありますが、コンパイラで命令生成する場合には、コンパイラの内部状態として、生成中の命令においてフレームポインタとスタックポインタはどれだけずれているかさえ分かればすむことが多いのです。

リスト2は、プログラム処理の先頭で呼ばれる初期化関数です。

この関数ではスタックポインタである$3の値をセットしています。コメントアウトしていますが、必要ならここからMAIN関数に分岐すると、よりC言語らしくなります。言語構文の方でMAIN関数を定義していないため利用しないようにしていますが、必要ならコメントから外しておいてください。MAIN関数に初期化部分から分岐するようにすれば、2002年1月号で書いた、インタプリタとコンパイラで関数の書く場所が異なるという問題は解決できます。

■関数ex

さあ、いよいよ解析木からコードを生成する本体の関数であるexの説明になります(リスト3)。

関数exには3つの引数があります。1つ目は解析木へのポインタp、2つ目は結果を返すレジスタ番号regです。3つ目は、結果を返さない、レジスタの値を呼び出し側で保存するか否かのフラグであるpresです。最後のpresは、解析木を再帰的に呼び出す場合に、呼び出された側が全部のレジスタをコード生成のために利用する必要がないことから、コードを圧縮するために用意しています。我々のコンパイラでは、レジスタの扱いはいわゆるcallee saveレジスタとしています。そこで呼ばれた側は、レジスタの内容を書き換える場合には、スタックフレーム上に退避する必要があるのです。ところが、必ず退避する必要があるのかと考えると必ずしもそうではないのです。

例えば、単項演算や定数の生成を考えれば分かりますが、こういった操作や演算ではレジスタは結果を返す1つだけしか必要としないため、callee saveのレジスタの退避回復の操作はコードとしては冗長になります。そこで、本当に必要な演算のときだけレジスタ退避を行うことにします。退避が必要なケースというのは、上の階層のコードで結果を返す以外のレジスタを使用している場合だけになりますから、再帰的に解析木のコード生成関数を呼び出すときに、自分がレジスタを使用中であればpresを1にして呼び出し、片方が空いていればpresを0にして呼び出します。これによって、コード生成関数ではpresが0の場合には両レジスタを自由に使ったコードを生成すればよくなり、レジスタの退避が必要なくなります。

次にローカル変数の定義です(リスト4)。lbl1とlbl2はラベルに対応する数値のワーク用変数です。regxは返り値を返さない方のレジスタ番号になります。valueとtopはワーク変数で、iとjはループ変数としています。

【リスト1】

```
#include <stdio.h>
#include "toyp.h"
#include "y.tab.h"

static int lbl = 0;
static int frametop = 0;
```

【リスト2】

```
void sinit(int val) {
printf("    lda $3, %d($0)\n", val);
/*
printf("    bal $0, MAIN\n");
*/
return;
}
```

【リスト3】

```
int ex(nodeType *p, int reg, int pres) {
```

【リスト4】

```
int lbl1, lbl2, regx, value, i, j, top;
```

```
regx = 3 - reg;
```

これは、reg が 1 なら 2 となり、2 なら 1 となる計算を行なっています。

```
if (!p) return 0;
```

この書き方が気になる人は、p==NULL にしてください。解析木が NULL の場合にコードを生成せずに関数から抜けます。

```
switch(p->type) {
```

解析木の構造体は union で複数の型を表わせますが、型によらず先頭には必ず type というメンバを定義してあります。そこで、このように関数の処理切り分けに type というメンバを使うことができます。まずは、定数の処理から見ていきましょう（**リスト 5**）。

定数を表わす type 名は typeCon になっています。そこで、case 文で typeCon となる場合に、定数を reg で示されたレジスタに作成するコードを生成します。ここで、我々のプロセッサの命令には、定数を直接命令で指定する部分が 8 ビットの符号付きしかなかったことを思い出してください。そこで、127 までの定数であれば直接命令で生成できるのですが、それ以上の大きさの定数は複数の命令を組み合せることで作成します。127 というと 2 進の 7 桁になります。7 桁ずつ命令を生成する必要があるかどうかを、コンパイラが判断しながらシフトと加算によって 16 ビットの定数をレジスタに作成していきます。前に書いたように、我々のプロセッサにはシフト命令はないのですが、レジスタ加算命令でシフトを行ないます。

解析木で生成すべき定数値は p->con.value に記憶しています。この定数値を上の桁から命令生成が必要かどうか 7 ビットずつ調べていくのですが、上位桁が 0 であれば定数の生成やシフト動作をする必要がないため、生成する命令コードをなるべくコンパクトにするためにちょっと面倒なやりかたをしています。

生成コードは同一であっても、コンパイラのコードとしてはこのコードよりももう少しマシなコードも考えられると思いますので、もっと良いコードを考えてみてください。私としてはとりあえず、動いたので良しとしています。

次に大域変数の扱いです（**リスト 6**）。大域変数はメモリの固定番地に割り当てることにしたため、メモリ番地を変数名から作成した後ロード命令をコードとして生成します。ここで、メモリ番地を生成なんて書くと大袈裟ですが、メモリの a を 1 番地に、また z を 26 番地に順番に割り付けているだけです。a を 0、z を 25 として、記憶している解析木の定数ノードの定数番号である id.i の値に、1 を加えたアドレスから 1 ワードのデータをロードしているというわけです。すなわち、

```
p->id.i + 1
```

の計算をしているだけともいえます。

■演算、制御構文

さて、大物が残っていますね。演算や制御構文です。これらは解析木では typeOpr のノードを作ります。

```
case typeOpr:
 switch(p->opr.oper) {
```

このノードから演算や制御構文をさらに細かく分けてコード生成します。

まずは関数定義から。関数は引数がある場合とない場合に分かれます。引数がある場合、引数はレジスタに入れて呼び出しが行われます。しかし、引数を局所変数と同様に扱う必要があるため、関数の呼び出しが行われた後に引数の領域をスタックフレーム上に確保し、レジスタの実引数の値をスタックフレムーム上の領域に格納しておきます（**リスト 7**）。

我々の言語では、関数名としては foo という 1 つだけしか許可していないので、まず foo というラベルを出力します。次に、この関数が利用するスタックフレームの大きさをスタックポインタから引くのですが、関数からの戻り番地をスタックフレームに退避しておくことと、前出の仮引数の領域で 2 ワード分の領域が必要となるため、-2 をスタックポインタである $3 レジスタに加算しています。その後、スタックの先頭に戻り番地を、その次に実引数を書き込みます。これで関数のプロローグ部分の処理は終了し、関数の本体のコード生成を再帰的に呼び出します。この時関数の返り値は $1 レジスタと決めているのと、$2 レジスタの値はすでに退避済みなため、再帰呼出しの中で壊れても構わないということで、「reg=1, pres=0」で呼び出しを行います。関数本体の中で return 文を実行した場合には、fooexit というラベルの

【リスト 5】

```
case typeCon:
  value = p->con.value;
  top = 0;
  for(i=2; i > 0; i--) {
     if((p->con.value >> i*7) != 0) {
      value = (p->con.value >> i*7) & 127;
      if(top == 0 || value != 0)
    printf("    lda $%d,%d($%d)\n", reg, value, top);
    for(j=0; j<7; j++)
                  printf("  add $%d,$%d,$%d\n",reg,reg,reg);
        top = reg;
     }
    }
  value = p->con.value  & 127;
  if(value || top == 0)
  printf("  lda $%d,%d($%d)\n", reg, p->con.value & 127, top);
  break;
```

【リスト 6】

```
 case typeId:
  printf("  ld   $%d,%d($0)\n", reg, p->id.i + 1);
  break;
```

【リスト 7】

```
   case FDEFA:
     printf("foo:\n");
     printf("   lda $3, -2($3)\n");
     printf("   st  $2, 0($3)\n");
     printf("   st  $1, 1($3)\n");
     ex(p->opr.op[0], 1, 0);
     printf("fooexit:\n");
     printf("   ld  $2, 0($3)\n");
     printf("   lda $3, 2($3)\n");
     printf("   bal $0, 0($2)\n");
     break;
```

【リスト 8】

```
 case FDEF:
     printf("foo:\n");
     printf("   lda $3, -1($3)\n");
     printf("   st  $2, 0($3)\n");
     ex(p->opr.op[0], 1, 0);
     printf("fooexit:\n");
     printf("   ld  $2, 0($3)\n");
     printf("   lda $3, 1($3)\n");
     printf("   bal $0, 0($2)\n");
     break;
```

位置に分岐するようにコード生成をするため、exの呼び出しの後にfooexitラベルを定義しています。後は戻り番地を回復し、スタックポインタを回復した後、戻り番地で示されるアドレスへ無条件分岐を行っています。

引数がない場合には、リスト8のようにスタックフレームは戻り番地の分だけ確保すればよくなります。関数から戻るreturn文の処理は簡単です（リスト9）。

return文に引数がある場合には関数値として引数を返さなくてはなりませんが、関数値は$1に返すと決っているため、ex関数を再帰的に呼出し、returnの引数を$1に作成します。その後、fooexitラベルへの分岐命令を生成します。

●制御構文のコード生成

制御構文の代表としてwhile文を用意してあります（リスト10）。

while文ではあらかじめ条件判定してループ本体を実行するかどうかを判断します。ループを再度実行するときには条件判定から行うため、ループの最後から先頭に分岐してきます。そこで、まずラベルを出力してから条件計算をします。条件が0でない場合にループから脱出するので、条件分岐命令BZを使って脱出ラベルのところに分岐させます。条件分岐で分岐しない場合には、ループ本体の実行が行われるため、この部分にループ本体のコードを生成しておきます。

本体の実行が終ったところで、ループの先頭に分岐しておきましょう。

●if文のコード生成

リスト11はif文のコード生成です。if文にはelseがある場合とない場合があります。elseがある場合には、else文と合わせて解析木のノードの子リンクが2となるため、子リンクの数を記憶するp->opr.nopsの値を判定して、どちらのパターンかを切り分けます。

●演算・操作のコード生成

変数への代入はリスト12のようになります。代入する右辺値をexの再帰呼び出しでレジスタに生成し、その値をメモリへストア命令を使って書き込みます。

単項演算子のマイナス記号は、リスト13のように反転して1を加えて負の値を生成します。関数の仮引数は、その関数のスタックフレームの先頭から1ワード加えたところに格納していますので、リスト14のようにフレームの先頭からアクセスするようにします。このframetopという変数はスタックを関数内で他に使用する構文があると値を増減して常にフレームトップへの参照ができるように調整されます。

関数呼び出しのコードはリスト15のようになります。まず、presが0でないときには、regxで示されるレジスタの内容は保存する必

【リスト9】

```
case RETURN:
    if (p->opr.nops > 0) {
       ex(p->opr.op[0], 1, 0);
    }
    printf("   bal $0, fooexit\n");
    break;
```

【リスト10】

```
case WHILE:
   printf("L%03d:\n", lbl1 = lbl++);
   ex(p->opr.op[0], reg, pres);
   printf("    bz  $%d,L%03d\n", reg, lbl2 = lbl++);
   ex(p->opr.op[1], reg, pres);
   printf("    bal $0, L%03d\n", lbl1);
   printf("L%03d:\n", lbl2);
   break;
```

【リスト11】

```
 case IF:
   ex(p->opr.op[0], reg, pres);
   if (p->opr.nops > 2) {
   /* if else */
   printf("    bz  $%d,L%03d\n", reg, lbl1 = lbl++);
   ex(p->opr.op[1], reg, pres);
   printf("    bal $0, L%03d\n", lbl2 = lbl++);
   printf("L%03d:\n", lbl1);
   ex(p->opr.op[2], reg, pres);
   printf("L%03d:\n", lbl2);
   } else {
   /* if */
   printf("    bz  $%d,L%03d\n", reg, lbl1 = lbl++);
   ex(p->opr.op[1],reg, pres);
   printf("L%03d:\n", lbl1);
   }
   break;
```

【リスト12】

```
case '=':
   ex(p->opr.op[1],reg, pres);
   printf("st   $%d,%d($0)\n", reg, p->opr.op[0]->id.i + 1);
   break;
```

【リスト13】

```
case UMINUS:
   ex(p->opr.op[0],reg, pres);
   printf("    not $%d,$%d\n",reg,reg);
   printf("    lda $%d,1($%d)\n",reg,reg);
   break;
```

【リスト14】

```
case ARGV:
   printf("    ld  $%d,%d($3)\n",reg,frametop+1);
   break;
```

【リスト15】

```
case FUNC:
   if(pres) {
     printf("  lda $3, -1($3)\n");
     printf("  st  $%d,0($3)\n", regx);
     frametop += 1;
     }
   ex(p->opr.op[0],1, 0);
   printf("    bal $2, foo\n");
   if(reg!=1) {
      printf(" lda $%d,0($1)\n",reg);
     }
   if(pres) {
      printf(" ld  $%d,0($3)\n", regx);
      printf(" lda $3, 1($3)\n");
      frametop -= 1;
     }
   break;
```

【リスト16】

```
default:
   ex(p->opr.op[0], reg, pres);
   if(pres) {
     printf("  lda $3, -1($3)\n");
     printf("  st  $%d,0($3)\n", regx);
     frametop += 1;
     }
   ex(p->opr.op[1], regx, 1);
```

【リスト17】

```
   switch(p->opr.oper) {
      case '+': printf("add $%d,$%d,$%d\n",reg,reg,regx);
                break;
      case '-': printf("not $%d,$%d\n",regx,regx);
                printf("lda $%d,1($%d)\n",regx,regx);
                printf("add $%d,$%d,$%d\n",reg,reg,regx);
                break;
```

【リスト18】

```
      case '*': printf("mul $%d $%d,$%d\n",reg,reg,regx);
                break;
      case '/': printf("div $%d $%d,$%d\n",reg,reg,regx);
                break;
```

【リスト20】

```
  if(pres) {
    printf("    ld  $%d,0($3)\n", regx);
    printf("    lda $3, 1($3)\n");
    frametop -= 1;
    }
```

【リスト21】

```
   }
  }
  return 0;
}
```

要があるので、スタックフレーム中に退避領域を確保し退避します。次に、関数では実引数は$1に設定する約束になっているので、引数の計算を$1をターゲットとして行います。その後、ブランチアンドリンク命令で関数に分岐します。関数から戻ってきたときには戻り値は$1に入ってくるので、この構文のターゲットが$1でないならばレジスタのコピーが必要になります。

また、presが0以外になっている場合には、退避してあったレジスタの値を回復しておきます。その他の二項演算もpresの処理が必要になりますから、一括して処理をまとめておきましょう（リスト16）。これで、二項演算に必要な値がregとregxに生成されたので、演算子に従って演算コードを生成します（リスト17）。

引き算は負の値の加算として実現します。引く数を反転して1を加えると-1を掛けたことになるため、その後加算を行っています（リスト18）。

乗除算については、実はプロセッサが命令を持っていないので、本来はコンパイラがそれなりのコードを生成す必要があります。ここでは、実行できない命令を生成していますので、このままでアセンブルするとエラーになります。あんまり全部作ってしまうと読者の楽しみがなくなるので、ここは読者への宿題としておきましょう。

次は論理演算です（リスト19）。特に論理演算関係に冗長に思えるところが多いかもしれませんが、ご容赦ください。

リスト20では、二項演算の前に退避されたregx側のレジスタを回復させる処理を行います。そして、リスト21まででコンパイラのコード生成が終了します。ちょっと長い説明でしたが、ここまで読んでいただきありがとうございました。

【リスト19】

```
      case '<': printf("  slt    $%d,   $%d,   $%d\n",reg,reg,regx);
                break;
      case '>': printf("  slt    $%d,   $%d,   $%d\n",reg,regx,reg);
                break;
      case GE:  printf("  slt    $%d,   $%d,   $%d\n",reg,regx,reg);
                printf("  not    $%d,   $%d\n",reg,reg);
                printf("  lda    $%d,   1($%d)\n",reg,reg);
                printf("  lda    $%d,   1($0)\n",regx);
                printf("  add    $%d,   $%d,   $%d\n",reg,reg,regx);
                break;
      case LE:  printf("  slt    $%d,   $%d,   $%d\n",reg,reg,regx);
                printf("  not    $%d,   $%d\n",reg,reg);
                printf("  lda    $%d,   1($%d)\n",reg,reg);
                printf("  lda    $%d,   1($0)\n",regx);
                printf("  add    $%d,   $%d,   $%d\n",reg,reg,regx);
                break;
      case NE:  printf("  not    $%d,   $%d\n",reg,reg);
                printf("  lda    $%d,   1($%d)\n",reg,reg);
                printf("  add    $%d,   $%d,   $%d\n",reg,reg,regx);
                printf("  bz     $%d,   L%03d\n", reg, lbl1 = lbl++);
                printf("  lda    $%d,   1($0)\n",reg);
                printf("L%03d:\n", lbl1);
                break;
      case EQ:  printf("  not    $%d,   $%d\n",reg,reg);
                printf("  lda    $%d,   1($%d)\n",reg,reg);
                printf("  add    $%d,   $%d,   $%d\n",reg,reg,regx);
                printf("  bz     $%d,   L%03d\n", reg,  lbl1 = lbl++);
                printf("  lda    $%d,   0($0)\n",reg);
                printf("  bal    $0,    L%03d\n", lbl2 = lbl++);
                printf("L%03d:\n", lbl1);
                printf("  lda    $%d,   1($0)\n",reg);
                printf("L%03d:\n", lbl2);
                break;
   }
```

Resource

[1] 大人の科学「学研電子ブロック EX-150」
http://kids.gakken.co.jp/kit/otona/7/