coolda

だってクールだ コンパイラー PLM

plm

Image courtesy of NRAO/AUI

ステージ１　コンピュータ言語は自分で作る。

自分で使うコンピュータ言語は、自分が使いやすいように、自分で作る、
そうでなければ、使いやすいように、どんどん改造していく。
そこで、拡張可能なスターターとしての、小さなコンパイラー作りから始めます。
ここでご紹介するコンパイラーは、

1. コンパイラを初めて作ると思いたって、コードを見ても、
   何だか分からない、ということが無いように、心がけています。
2. プログラムのサイズが多少大きめになっても、親として使うC++の、
   便利なライブラリーは、躊躇することなく、どんどん使います。
   親がC++、といっても、マップ、ベクタなど、ライブラリー
   を使うのが主ですので、C++は初めて、でも問題ありません。
3. 全体サイズの目安は、最大でも400～500kb程度に止めます。
   しかし、64ビットでコンパイルすると、ほぼ1.4倍ほど増加になります。
4. 最初は、plmcore として用意した小振りのファイルから、
   見て行きます。ベース部分を小さくしたところから始めますと、
   コードを見るのは楽になりますが、プログラムは、実のところ、何もできません。
   作っただけで、利用はしない、のではなく、
   実際、自分で使って楽しめる程度までの、実用性を持たせたいものです。
   そこで、plmmedという中程度のサイズにまで、拡張していきます。
5. plmmedで予定しているサービスは、
   100、3.45、"hello" などを収納する配列、イフ文、ワイル、ドゥワイル文、
   フォー文、スイッチ、ケース文、プロシージャ、関数、中間言語を直接書き込むASM文、
   シェルの呼び出し、ドットスタイルの利用などです。
   ドットスタイルでは、
   a=100; a.s="hello"; write(a); write(a.s); すると、
   $ 100
   $ hello　が出力されます。
6. 30+40を、30 40 + に変換する方法、中間言語の表記スタイルなどは、
   ニクラスワースさんが、PL0で、発表されたものを使用しています。

それでは、作戦開始です。

ステージ２　**plmcore**をギットハブから入手する。

はじめに、出発点として用意した、ソースコードを入手します。
githubというサイトに用意してあります。
https://github.com/coolda/plmcore/downloads
Download as zip または、Download as tar.gz どちらかをクリックします。
これでうまくいかない場合、登録をおすすめします。
プログラムの開発などに特化したサイトで、簡単サービスがウリです。
すでに、２百万を超えるレポが存在するとのこと。
https://github.com/　にて、
ど真ん中の、Plans, Pricing and Signup をプッシュ。
一番上の$0/mo Free for open source の　右にある　Create a free account をクリック。
入力は、
1. Username : このサイトで使うユーザー名を入力します。
2. Email Address : あなたのEメールアドレス。ご自分で作ったプログラムも、自分の場所に置けますので、パソコンのメールアドレスが便利かと思います。
3. Password : パスワードは、７文字以上、その中には、最低、小文字アルファ１つ、数字１つを含めます。
4. Confirm Password : 3を繰り返す。

そして、Create an account を押します。
以上で終了です。
これで、次回から、ログインすることができます。
ログインしましたら、
https://github.com/coolda/plmcore/downloads
にアクセスして、ページの５行目辺りにある、
Download as zip または、Download as tar.gz どちらかをクリックします。
ターファイルのサイズは、16KB、ジップで、20KB程度です。

ステージ３　**plmcore** をテストする。

ダウンロードしてきたファイルを、ホームなどに置き、
ダブルクリックして、展開します。
出てきたフォルダー名は、長いので、plmcoreのみにリネームします。
ターミナルから、
$ cd plmcore
$ make
これで、実行可能なファイル、plm が出来ています。
フォルダー、progの中に入っている、テスト用のプログラムを試します。
$ ./plm prog/add
出力された内容は、

1. 入力されたコード、尻尾にダラーマークが追加されています。
   plmcoreを、irbのように、コマンドラインから利用する場合、
   入力をcinに変更しますが、入力終了の合図として$を使います。
   入力終了の合図として$を使った場合、C++の仕様では、
   この$は取り込まれません。しかし、
   ループ終了の合図として、内部でも、$を使用しているため、
   ダラーマークを自動的に追加するようにしています。
   例えば、パスカルでは、プログラム開始の合図として、programという
   単語を配置し、終了は、END. エンドのあとにピリオドをつけます。
   このように何がくるというのがはっきりしていると、
   コンパイラーの作成は楽です。
   問題は、利用者が、programと書くのが面倒だと思うことです。
2. ディバッグなどのために、コードに書きこんだ、cout << "fix it!:" << endl;
   pl_error();などがあれば、ここに出ます。
3. 変換された中間言語です。
4. インタープが実行され、結果が表示されます。

ステージ４　**plm**のメモリーシュミレーション。

コンピュータープログラムでは、データをメモリに収納し、
それから、そのデータを操作するという順番になります。
メモリとして使うのは、
1. CPUに組み込まれた、レジスタ。
2. 購入して搭載されているメモリデバイス。
3. 簡単なデータ構造として用意されているスタック。
スタックには、データを操作する方法として、プッシュとポップが用意されています。
プッシュで、データをテールから押し込み、
ポップでは最後にプッシュしたものを、テールから取り出します。
オリジナルのピーエルゼロでは、何もかも１つの配列で済ませてしまう、
という方法になっており、経済的ですが、ちょっと複雑です。
C/C++の配列は、スピード重視のため、
１種類のデータ型のみを入れるようになっています。
配列をスタックのシュミレーションとして採用した場合、
イント型しかサポートしなければ、イントのスタックを作ればよいのですが、
フロートも、ストリングも、となりますと、少し考える必要があります。
計算の操作は、定石通り、スタックでの操作にしたいので、
plmでは、C++のライブラリーにあるスタックに、
エラー出力サービスを加えたスタックを使います。
アセンブリー言語では、何らかの操作を行い、結果が得られた場合、
レジスターEAXに収納しておきますよ、という手法がとられます。
これに似せて、plmでは、グローバルに、変数reg1とreg2を作り、
レジスタのつもりとします。
パスカル、Cのような高級言語では、
なんでも名前をつけることができるサービスが、充実しています。
名前をつけた場合、その名前と値、メモリの番地などの関連をメモしておくリストが
必要で、これをネームテーブル、名前表と呼び、
名前が出てきたら、すぐさま、これに登録することから始めます。
plmでは、ネームテーブルにデータを収納するメモリ機能を加えた、構造体の配列を用意して、
これがデータセンターとなり、操作はフロート型のスタックで行うようにしています。

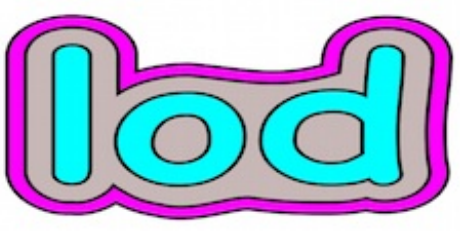
ステージ５　中間言語は、ワース先生お手製**+a**。

1975年ごろ、ニクラスワースさんが、学校での練習用に、
簡単なコンパイラの例を示されました。
その名前は、PL0で、3文字で表記されたオプコードが８ケと、
'<'に対してはlssのように、比較記号に対応する名前が3文字で表記されています。
これは、この分野の第一人者が設定しただけのことはあり、
とても分かりやすいので、ベースとして採用しています。
拡張する場合、3文字ではうまく出来ないので、すぐにこの原則は崩れてしまいます。
従って、覚えるのが大変になるので、あまり増やさないという方向でいきたいと思います。
実際の80x86対応のasmでは、ものすごい数のインストラクションが規定されています。
これは型に応じて別の名前を振り付けているためで、
スピードを稼ぐための方法としては有効で、かつ、
アセンブリを直接、おもいっきり使う人は殆どいないので、OKとしても、
コンパイラーを作る場合は、これと向き合うことになりますので、
相当面倒なことになります。
最低でもマニュアルを何度も見る必要があります。
plmcoreには、現時点で、オプコードOPが14、演算用のOP_OPRが20 存在します。

ステージ６　中間言語を見る。

progフォルダーの中にある、addというテキスト、
a=30+40; write(a);
を読み込むと、以下のコードが出力されました。
0: lit　30
1: lit　40
2: opr　add
3: sto　0
4: lod　0
5: wrt　0
6: opr　ret
litは、リテラルを省略したもので、即値の意味です。
ゼロ行目は、スタックに具体的な値、数字の30を積んで下さいという命令です。
1行目は同じく、数字の４0をスタックに積む命令です。
最初に'='の右側部分の操作を行なっていることに注目して下さい。
2:は、演算addで足し算命令です。足し算ではデータが２つ存在します。
そこで、先ほどのデータ２本を下ろして、計算機に渡します。
アセンブリー言語では、CPUの中にある部品、ALUが演算操作を行いますが、
plmには、C++という強力な親がいますので、親に渡します。
C++から、帰ってきた値、ここでは70を、スタックに積みます。
現時点で、'a'は、a=70; のような形に変化していると考えます。
'='はアサインメントオペレータで、'a'という名前の場所に、
右側のデータを収納することを求めています。
収納はストアで、その略語がstoです。
名前'a'を見た時、実際、名前テーブルに登録する操作は、
ステートメント関数の中で、すでに行われています。
その時、'a'は最初なので、ゼロ番地に収納すると決められました。
ですから、ここで、isin_tab("a"); と尋ねれば、ゼロと帰ってきます。
ゼロは、何もなし、ではなく、ゼロ番地にデータはいますよ、
と云っています。
よく、'アレッ'っと思ってしまいますが、これでいいのだ、です。
本当にいない場合は、-1が返ってきます。

７０は再度スタックに積まれました。



4: lod 0 で、７０は再度スタックに積まれました。
この最後に積まれたものを下ろして、
cout すれば、結果が出力されます。
スタックに積まれたものを下ろす操作をポップといいます。
5: wrt 0 では、
stack.pop(fnum1);
cout << "wrt int: " << fnum1 << endl;
のような操作をして、70が出力されます。
6: opr ret では、reg2に、コールする時に入れておいた値を、
コードの配列のインデックス、cpにコピーすれば、戻ります。
ここでは関数を使っていませんので、コールがありません。
なのでゼロが入り、終了します。

ステージ７　中間言語へ変換する仕組み。

plmでは、演算の操作をスタックで行います。
わたしたちも、計算するとき、

```
    30
  + 40
```

のように書くことを小学校でやっていました。
プラス記号を一段下げます。

```
    30
    40
    +
```

これを１行に並べると、
30 40 +
スタックを使った演算操作を行う場合、30+40は
30 40 + の並びに変換しておきます。
PL０では、この操作を、立体的な方法で、とても上手に行います。
その様子を観察しておきます。関数を３つ作成します。
名前は、expression(); term(); factor(); となっています。

```
void expression() {
    term();
   while ( isAdd(look) ) {
      dolook();
      term();
      gen(OP_OPR, 0, OPR_ADD);
   }
}
```

エクスプレッシヨンは、呼ばれると、何もしないで、そのままタームを呼び出します。
これは、一体何をしているのでしょうか。これは、タライ回しの練習ではなく、
呼び出し元に制御は戻るという、関数の特性を利用するためです。
ここに戻ってきた時、もしデータが'+'なら、働こうとしています。
タームは、これと全く同様に、そのままファクターを呼びます。
それで、制御はファクターに降りてきます。

ファクターでの操作。

ファクターには、数字を処理するコードが書いてありますので、
最初の30を見て、lit 30 という命令を生成しますが、
'+' のことは、何も書いてないので、操作できません。
それで呼び出し元に戻ります。
タームは、'*'と'/'を見た時だけ働きますので、
ここでは、そのままエクスプレッションに戻します。
エクスプレッションでは、'+'を見ますが、ここでは意外にも、
単純にその'+'記号を捨て、再びタームを呼びます。
'+'記号の後にあるべきデータを生成させるためです。
'+'記号を捨てたので、現在は数字が見えています。
ファクターでは、この数字部分を調べ、lit 40を生成して、
制御を戻します。
エクスプレッションに制御が戻ったところで、すかさず、
gen(OP_OPR, 0, OPR_ADD);　で、opr add を生成します。
これで、30+40の部分は、
lit 30
lit 40
opr add　に変換されました。
最初にこのコードを見た時、なんだか分からないわけですが、
よく見ると、その手法にすっかり感心してしまいました。
まるでアーティストの作品のように思えてしまいます。
plmでは、勿論のこと、この仕組みを採用しております。

ステージ８　プロシージャ。

プロシージャとファンクションは大体似たようなものを指す言葉で、
日本語では、手続き、関数と云います。
これは、中間言語のある部分をブロック化して名前をつけ、
見やすく、そして管理しやすく改良したもので、
そこには、さらに色々なサービス、もしくは、ルールが盛り込まれています。
そのサービスとは、

1. プロシージャの先頭にはジャンプ命令が取り付けてあるので、
   プロシージャ本体は、そのままでは実行されない。
2. 利用する時は、gotoとか、jmp命令で、先頭のジャンプの次の行にジャンプしますが、
   例えば、プロシージャに、p1という名前をつけた場合、
   goto p1; と書かないで、ただp1; とか、p1();とすれば反応するようにしてある。
3. プロシージャは'{' '}' とか、begin … end でブロック化されており、
   このブロック内は、他とは独立した領域なので、ここで生成した変数の名前は、
   他に影響しない。つまり他ですでに使用している名前を、
   この中では、別の用途に使えます。
   これは、ここ専用の名前テーブルが用意されることを意味します。
   ここで生成した変数のことをローカル変数という。
4. ローカル変数は、このブロックを抜けるときには、チャラになる。
5. プロシージャ内で使う変数の名前を外に知らせる位置が決めてあり、
   プロシージャを呼び出すとき、値を入れて渡すと、それを使って作業をしてくれる。
   このデータをパラメタと呼んでいます。

例:

```
procedure p1(a b) { write(a); write(b); }
p1(100 200);
```

(a b)は、データの受け取り窓口のような働きをしています。

名前の先頭に**'&'**をつけておくと。

6. プロシージャが使う変数の名前の先頭に'&'をつけておくと、
   渡した変数自体が変更されます。
   例:
   procedure p1(&a &b) { a=a+100; b=b+200; }
   x=10; y=20;　p1(x y);
   write(x); write(y);
   110 220 と出力。
7. プロシージャは実行されたあとは、呼び出しの次の行に必ず戻る。
8. プロシージャでの作業の結果は、消えてしまう前にどこかにコピーするなりして、
   セーブされる。この操作をリターン文と呼んでいる。

plmcoreのプロシージャは、1と２および７のみに対応しています。
何もついていないので、コードも短くとても気持ちがいいものです。
問題は、パラメタの渡しを実装すると、コードのサイズが急膨張することです。
それに対処するために、一工夫が必要です。
plmcoreのプロシージャは、グロ–バル扱いです。
ディフォルトがグローバルな言語は、少数ですが、存在します。
PL0、Scheme、Luaなどの教育、小型、軽量級のものです。
メモリの節約はともかく、名前の重複が可能なのは便利なので、
ブロックごとの名前表は、結局作る必要があります。

ステージ９　ワイル文の実装を見る。

ワイル文からエラー表示などをはずしたものを見ます。
入力されるプログラムは、例えば、
// a=0; while(a<3) a=a+1; write(a);
が入ってきた時の実装は、

```
void whilestmnt() {
    if (isLParen(look))
        dolook();
    wx1 = cx;
    condition();
    if (isRParen(look))
        dolook();
    wx2 = cx;
    gen(OP_JPC, 0, 0);
    statement();
    gen(OP_JMP, 0, wx1);
    code[wx2].operand2 = cx;
}
```

状況は、こうです。
statemennt()にて、whileというキーワードを読み込んでから、
この関数に来ますので、ここで見えているlookは、'(' となります。
この括弧の中に入っているのは、条件式と呼ばれているもので、
条件式を調べる操作は、コンディション関数に書いてあります。
なので、ここは単純にcondition();を呼び出し、必要なコードを生成してもらいます。
条件式は、ここでは、a<3 ですが、ステートメントを実行したら、そのつど、
ここに舞い戻り、条件とのつき合わせをする必要があります。
現在のコードインデックスの値は、cxです。この値をメモしておきます。
そうすれば、条件式のコードの先頭の配列にアクセスできます。
cxの値は、中間コードの配列のカウンターですから、コードを生成すれば、
増加していきます。ですから、ここでメモしておかないと、条件の確認をするために、
条件式の先頭にジャンプする、その飛び先の行が分からなくなってしまいます。
ここでは、wx1にしっかりメモしましたので、だいじょうぶ、というわけです。

制御がコンディションから帰ってきました。

制御がコンディションから帰ってきました。
条件式は、例えば、a<3　が成立しているか調べ、
成立していれば、1を、成立していないとゼロを返してきます。
aの値が３になれば、不成立なのでゼロを返してきます。
これで、もうこの場所に留まる必要はないので、
一気にジャンプして、抜け出るわけですが、
この時点では、飛び先とすべき、このプログラムの尻尾のインデックスは、
これからコードの生成が続くわけですから、ここでは分かりません。
つまり、どこどこに飛べ、という指定が、未だ出来ないのです。
そこで、またメモを取り出し、現在のcxをメモします。
この値が、jpcを生成した、コードの配列のインデックス、code[cx]; です。
jpcは、条件ジャンプで、条件不成立の時の飛び先を、オペランド２に書き込みます。
今は、この値は暫定値ゼロとしておきます。
ステートメントで必要なコードを生成し、戻ってきましたら、
gen(OP_JMP, 0, wx1);
を生成して、条件式を見に行くループを作ります。
これで、コードの生成はおしまいです。
ということは、コードの尻尾はこれで決定しました。
それで、現在のcxが底ということになりますので、
先ほど、暫定でゼロにしておいた値を、
確定したjpcの飛び先の値として書き込みます。
code[wx2].operand2 = cx;
配列を現在作成の真っ最中というイメージが湧きにくいようで、
このバックパッチというのが、分かりにくいという評判です。
少し時間をとって、じっくりとイメージをつかんで下さい。

ステージ１０　ドゥワイル文を実装する。

コンピュータープログラムは、とにかく自分でやってみる、
というのが、一番です。
ここでは、ワイル文ととてもよく似ている、ドゥワイル文を、
忘れないうちに実装してみましょう。
ドゥワイル文は、例えば、コンパイラーを作る中で、
次にくるのは、変数の名前ですよ、という時に使うと、ドンピシャです。
この場合、イフ文と組み合わせ、

```
string s1;
if ( isAlpha(look) ) {
   do {
      s1 += look;
      dolook();
   } while ( isAlNum(look) );
}
```

変数の名前は、ファクターで数字か名前かを識別する時、
名前の先頭は、アルファに限る、としておけば、
先頭の１文字を調べるだけで、名前なのか数字なのかが判断できるので、
Cでは、名前の書き方に、このような制限をつけています。
まず、isAlpha()でアルファかどうかを確認し、１文字をストリングに収納します。
次からは、数字も使えますので、ワイルの条件式は、isAlNum()とします。
アルファか数字以外を見たら、中止してブロックから出てきます。
これで見事に名前を取り出せます。
それでは、実際に、do_whilestmnt()の実装を行って下さい。
void do_whilestmnt();　の骨組みは、以下のようになります。

1. dw1=cx;
2. statement();
3. paren_cond();
4. dw2=cx;
5. gen(OP_JPC, 0, 0);
6. gen(OP_JMP, 0, dw1);
7. code[dw2].operand2=cx;

別の小さなファイルに書くと考えをまとめやすい。

---

do_whilestmnt.cpp のようなファイルを作り、
そこで考えがまとまりましたら、
plm.cppにコピーするというのが、おすすめです。
ステートメント部分が完成しましたら、
1. int dw1, dw2; をグロ–バルエリアに作成します。
　　グローバルはなるべく使わないように、という方針がある場合は、
　　ローカルに作ります。
2. statement()の中の、else if (gstr == "for")　の 1 つ前辺りに、

```
else if (gstr == "do")
　　do_whilestmnt();
```

を加えます。
加えた後は、以下のようになります。

```
else if (gstr == "do")
　　do_whilestmnt();
else if ( gstr == "for" )
　　forstmnt();
```

ステージ１１　フォー文の実装。

C/C++のフォー文は、きれいにまとまっており、
使いやすく、覚え易いので、使う場合、問題はありません。
それで、plmでも、これをそのまま採用します。
しかし、forの後に括弧をつけるスタイルは、今や少数派かもしれません。
plmのフォー文は、
1. for (a=0; a<3; a=a+1) write(a);
もしくは、
2. for (a=0 a<3 a=a+1) write(a);
セミコロンなしでも使えるようにしています。
この操作は、セミコロンを捨てる時、if(isSemicolon(look))　dolook();
と、イフを加えておけば、セミコロンが無い時は、なにもしないので、
うまくいきます。
フォー文の括弧の中にある３つのデータは、エキスプレッションです、
というのですが、ハイハイと聞き流し、実際には、
statement(); condition(); statement();の関数を使って処理をしています。

ステージ１２　イフ文はエルスイフが大変だ。

ifしか使わない場合のイフ文の実装は16行です。
if に、elseをつけた場合、37行になります。
ところが、if-elsif-else文になりますと、200行を越えます。
このバージョン１本だけにしてしまうと、
問題なのは、ifだけしか使わない場合でも、
最初に、パッチの手配の関係で、elsifが何個あるのかなどの、
調査をする必要があるので、コード量は増加します。
イフ文はいつもよく利用するものだけに、気にかかります。
そして、イフ文は遅い、というのは周知の事実でもあります。
そこで、当分の間は、ifのみの時専用のifz、if elseのとき専用に使う
ifeを残しておくことにしました。
しかし、ifzなんて忘れてしまうのは確実なので、
結局これは捨てることになると思います。
elsifのときにコード量が増えるのは、
1. バリエーションが増加するのでその追求が必要。
　　イフだけ、イフ＋エルス、イフ＋エルスイフ、イフ＋エルスイフ＋エルス、
　　イフ＋エルスイフ複数回、イフ＋エルスイフ複数回＋エルス。
2. この状況が確定したら、今度は、線の接続の準備をして、
　　そしてバックパッチを行う手間が必要。
ということで、イフ文はなるべく使わないようにする、というわけにも行きませんので、
もう少し上手に書く必要があると思っています。
plmcoreには、ifzとifeのみ入っています。

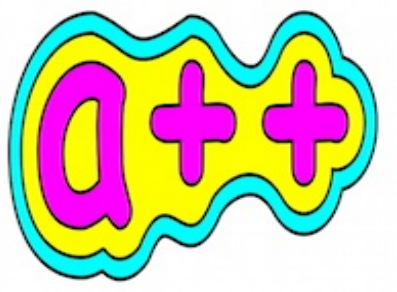
ステージ１３　**write**ステートメント。

write(a); の形を見ると、関数の呼び出しのようですが、実際は関数を使っていません。

```
else if ( gstr=="write" ) {
        paren_expre();
        gen(OP_WRT, wrtinfo, 0);
}
```

OP_WRTは、プリントすべきデータを、coutに渡します。
渡すデータは、実際の値を、直前にスタックにマウントしておきたいので、
先にエクスプレッションに回して、OP_LODを生成します。
例えば、ストリングの場合、スタックに載せるのは、アドレスになります。
どんなデータがスタックに載っているのかを、wrtinfoに書いておきます。
インタープでは、これを見て、データを用意、coutに渡します。

ステージ１４　**a++** と　**a--**。



C/C++には、
int a=1;　a++;　という方法が用意されています。これで、aは２になります。
'++' プラス演算子２つを固まりにしたもの、
これはポストフィックスエクスプレッション、後置式、つまり、式だ、と云われています。
式の尻尾にセミコロンとくれば、ステートメントではないか、と思ったりします。
で、結局これは何ですか、と云われると、さあ、何でしょう、となってしまいます。
一般に、スクリプト言語では、このような、ちょっとした曖昧さがありますと、
クリーンな言語、と云ってもらえなくなる、ということで、
実装しない、という風習があるようです。
しかし、あると便利で、いつも使っているものですから、用意します。
Cには、++a;というのもあります。
これは、現に実行している作業の、前処理として+1を行う、
というものですが、今回は、a++とa-- を利用できるようにします。
そして、当面は、イントのみのサポートとします。

```
int a=0, b=1, c=0;
a = 3 * b++;
cout << "a: " << a << endl;       //   3

c  = 3 * ++b;
cout << "c: " << c << endl;       //   9
```

ここでクイズです。
'a'の結果はいくつでしょうか。
右に3と書いてあるのが答えですが、ということは、
先にかけ算、次にインクリメント。
'c'の答えは9, ということは、++bの操作は、かけ算に優先してる。
C++の優先順位の表を、１度見ておくべきかと思います。

ステージ１５　ストリングデータの操作。

例えば、c++では、
const char* s1="aloha";
string s2("hello");
string s3="hola";
string s4=s1;
のようにして、ストリングデータをストアできます。
plmの中間言語では、これをどのように操作するのでしょうか。
現在、スタックはフロート型で、文字列を直接スタックには積めません。
しかし、データをどこかに置く必要があります。
それで、immediateのimm領域というのを確保して、一旦、ここに置き、
そこから、OP_STOにて、s1のアドレスにストアしたり、
writeの場合は、簡略化して、immが示すアドレスに、
ずばり取りに行く形式にしました。
imm領域は、現在、200から始まるようになっています。

```
/* imm_hello */
write("Hello, world!");
```

```
0: imm    200
1: wrt    0
2: opr    ret
Start pl_micro:
wrt immstr: Hello, world!
End pl_micro:
```

配列は、イント型の数字、フロート型の数字、ストリングが使えます。

```
/* myary */
myary[0]=100; myary[1]=3.45; myary[2]="hello";
for(a=0; a<3; a=a+1) write(myary[a]);
```

ステージ１６　ラップアップ。

コンピューター言語で、使い方が簡単なものは、
裏側が大忙しということがあります。
plmmedでは、かなり手応えのあるものも登場します。
お楽しみに。