PARROT - PROGRAMMING EXAMPLES

Parrot programing is similar to assembly language programing and you get a chance to work at lower level. Here is the list of programming examples to make you aware of the various aspects of Parrot Programming.

Classic Hello world!

Create a file called hello.pir that contains the following code:

```
.sub _main
   print "Hello world!\n"
   end
.end
```

Then run it by typing:

```
parrot hello.pir
```

As expected, this will display the text 'Hello world!' on the console, followed by a new line *duetothe*\n.

In this above example, '.sub _main' states that the instructions that follow make up a subroutine named '_main', until a '.end' is encountered. The second line contains the print instruction. In this case, we are calling the variant of the instruction that accepts a constant string. The assembler takes care of deciding which variant of the instruction to use for us. The third line contains the 'end' instruction, which causes the interpreter to terminate.

Using Registers

We can modify hello.pir to first store the string Hello world!\n in a register and then use that register with the print instruction.

```
.sub _main
   set S1, "Hello world!\n"
   print S1
   end
.end
```

Here we have stated exactly which register to use. However, by replacing S1 with $S1 we can delegate the choice of which register to use to Parrot. It is also possible to use an = notation instead of writing the set instruction.

```
.sub _main
   $S0 = "Hello world!\n"
   print $S0
   end
.end
```

To make PIR even more readable, named registers can be used. These are later mapped to real numbered registers.

```
.sub _main
   .local string hello
   hello = "Hello world!\n"
   print hello
   end
.end
```

The '.local' directive indicates that the named register is only needed inside the current compilation unit *thatis, between. suband. end*. Following '.local' is a type. This can be int *forIregisters*, float *forNregisters*, string *forSregisters*, pmc *forPregisters* or the name of a PMC type.

Summing squares

This example introduces some more instructions and PIR syntax. Lines starting with a # are comments.

```
.sub _main
   # State the number of squares to sum.
   .local int maxnum
   maxnum = 10

   # Some named registers we'll use.
   # Note how we can declare many
   # registers of the same type on one line.
   .local int i, total, temp
   total = 0

   # Loop to do the sum.
   i = 1

loop:
   temp = i * i
   total += temp
   inc i
   if i <= maxnum goto loop

   # Output result.
   print "The sum of the first "
   print maxnum
   print " squares is "
   print total
   print ".\n"
   end
.end
```

PIR provides a bit of syntactic sugar that makes it look more high level than assembly. For example:

```
temp = i * i
```

Is just another way of writing the more assembly-ish:

```
mul temp, i, i
```

And:

```
if i <= maxnum goto loop
```

Is the same as:

```
le i, maxnum, loop
```

And:

```
total += temp
```

Is the same as:

```
add total, temp
```

As a rule, whenever a Parrot instruction modifies the contents of a register, that will be the first register when writing the instruction in assembly form.

As is usual in assembly languages, loops and selections are implemented in terms of conditional branch statements and labels, as shown above. Assembly programming is one place where using goto is not a bad form!

Fibonacci Numbers

The Fibonacci series is defined like this: take two numbers, 1 and 1. Then repeatedly add together the last two numbers in the series to make the next one: 1, 1, 2, 3, 5, 8, 13, and so on. The Fibonacci number fib$_n$ is the n'th number in the series. Here's a simple Parrot assembler program that finds the first 20 Fibonacci numbers:

```
# Some simple code to print some Fibonacci numbers

        print   "The first 20 fibonacci numbers are:\n"
        set     I1, 0
        set     I2, 20
        set     I3, 1
        set     I4, 1

REDO:   eq      I1, I2, DONE, NEXT

NEXT:   set     I5, I4
        add     I4, I3, I4
        set     I3, I5
        print   I3
        print   "\n"
        inc     I1
        branch  REDO
DONE:   end
```

This is the equivalent code in Perl:

```
print "The first 20 fibonacci numbers are:\n";

my $i = 0;
my $target = 20;
my $a = 1;
my $b = 1;

until ($i == $target) {
   my $num = $b;
   $b += $a;
   $a = $num;
   print $a,"\n";
   $i++;
}
```

NOTE: As a fine point of interest, one of the shortest and certainly the most beautiful ways of printing out a Fibonacci series in Perl is perl -le '$b = 1; print a+=b while print b+=$a'.

Recursively computing factorial

In this example we define a factorial function and recursively call it to compute factorial.

```
.sub _fact
   # Get input parameter.
   .param int n

   # return (n > 1 ? n * _fact(n - 1) : 1)
   .local int result

   if n > 1 goto recurse
   result = 1
   goto return

recurse:
   $I0 = n - 1
   result = _fact($I0)
   result *= n

return:
   .return (result)
.end


.sub _main :main
   .local int f, i

   # We'll do factorial 0 to 10.
   i = 0

loop:
   f = _fact(i)

   print "Factorial of "
   print i
   print " is "
   print f
   print ".\n"

   inc i
   if i <= 10 goto loop

   # That's it.
   end
.end
```

Let's look at the _fact sub first. A point that was glossed over earlier is why the names of subroutines, all start with an underscore! This is done simply as a way of showing that the label is global rather than scoped to a particular subroutine. This is significant as the label is then visible to other subroutines.

The first line, .param int n, specifies that this subroutine takes one integer parameter and that we'd like to refer to the register it was passed in by the name n for the rest of the sub.

Much of what follows has been seen in previous examples, apart from the line reading:

```
result = _fact($I0)
```

This single line of PIR actually represents quite a few lines of PASM. First, the value in register $I0 is moved into the appropriate register for it to be received as an integer parameter by the _fact function. Other calling related registers are then set up, followed by _fact being invoked. Then, once _fact returns, the value returned by _fact is placed into the register given the name result.

Right before the .end of the _fact sub, a .return directive is used to ensure the value held in the register; named result is placed into the correct register for it to be seen as a return value by the code calling the sub.

The call to _fact in main works in just the same way as the recursive call to _fact within the sub _fact itself. The only remaining bit of new syntax is the :main, written after .sub _main. By default, PIR assumes that execution begins with the first sub in the file. This behavior can be changed by

marking the sub to start in with :main.

Compiling to PBC

To compile PIR to bytecode, use the -o flag and specify an output file with the extension .pbc.

```
parrot -o factorial.pbc factorial.pir
```

PIR vs. PASM

PIR can be turned into PASM by running:

```
parrot -o hello.pasm hello.pir
```

The PASM for the final example looks like this:

```
_main:
   set S30, "Hello world!\n"
   print S30
end
```

PASM does not handle register allocation or provide support for named registers. It also does not have the .sub and .end directives, instead replacing them with a label at the start of the instructions.