Primitive Operations in Hi

ăă

1. Literals
2. Parameters
3. Monadic
4. Dyadic
5. Conditional
6. Array indexing

Literal atoms

We have the grammar rule

```
atom          = {literal} literal
              | {paramname} id
              | {comp}comp;
```

And we need a visitor method for literal atoms:

```
//atoms
public void
 outALiteralAtom(ALiteralAtom node)
{
    types.put(node,
       types.get(node.getLiteral()));
}
```

this just records the type of the atom node to be the same as the type of the literal

The grammar defines literals to be strings or numbers

```
literal     = {string} string
            | {scalar} number;
```

this means we need visitor methods for string literals and number literals, thise will be called inAStringLiteral, outAStringLiteral etc

Here is an example visitor to handle numeric literals

```
public void outAScalarLiteral(AScalarLiteral n
    {
        types.put(node, new Integer(0));
        writer.println("push "+node);
    }
```

This first associates the node with the rank 0 in the type table since it is a scalar. It then outputs assembler to push the scalar onto the main stack of the processor.

Word size

In past years the Hi compiler has been targeted at the Pentium.

For simplicity all values, whether integer, pointer, or character are stored in a single machine word.

In the past this was a 32 bit word. Due to the upgrade of machines to Opteron class processors, we now use 64 bit words. This is rather wasteful, but we retain it to allow a simple compiler to be built.

String literals issues:

1. Converting from source form to actual string

2. Internal representation

Source form

'abc' , 'ab\ndef' , 'a \'quoted\' string'

If one performs a toString() call on the nodes returned by the Sable lexer you get the source of the lexeme followed by a space. Thus for the above we get back as Java strings:

"'abc' ", "'ab\\ndef' ",

"'a \\'quoted\\' string' "

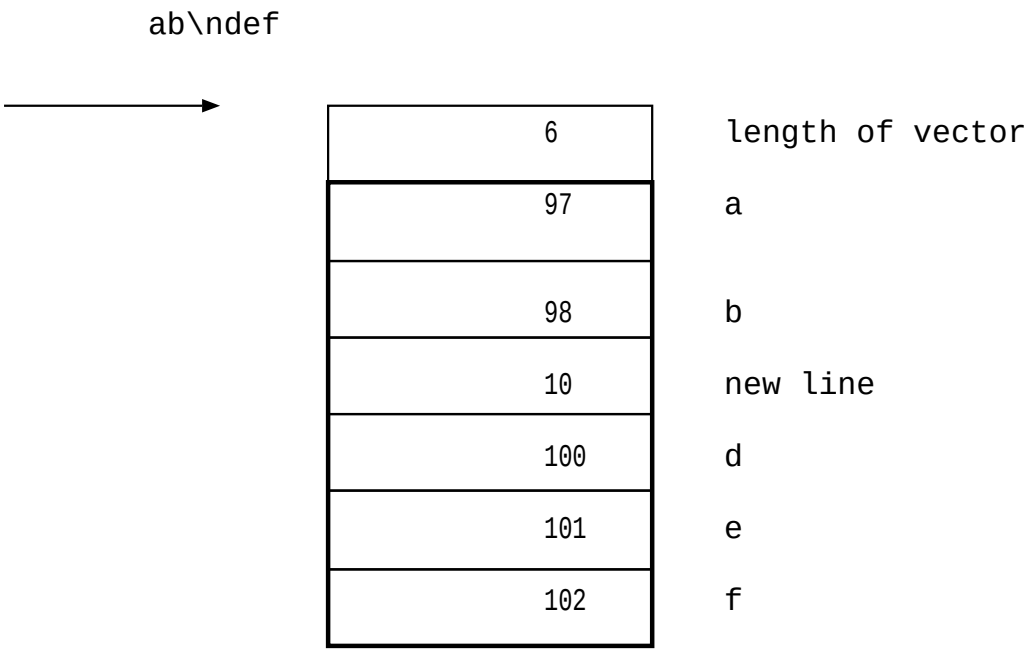
We must

1. strip of trailing spaces and single quotes

2. deal with escape chars '\\' in the source string.

Internal representation

String literals are a denotation for Hi vectors of rank 1. The elements of the string are numbers. This requires us to know about the internal format of vectors. This is what we want to achieve:



The following code sequence would put a pointer on the stack to a vector of the appropriate format for the string 'ab\ndef'

```
push $llll1str
jmp llll1 # jump past the chars
llll1str:.quad 6
         .quad 97
         .quad 98
         .quad 10
         .quad 100
         .quad 101
         .quad 102
llll1: # continue from here
```

```
public void outAStringLiteral(AStringLiteral node)
    {
        String past=newlab();
        String s=node.toString();
        // count backslashes
        int backs=0;
        for(int i=1;i<s.length()-2;i++)
         if(s.charAt(i)=='\\'){backs++;i++;}
        writer.println(
         "push $"+past+"str\n"+
         "jmp "+past+"\n"+
         past+"str:.quad "+(s.length()-3-backs));
        for(int i=1;i<s.length()-2;i++){
         if(s.charAt(i)!='\\')
           writer.println(".quad "+(int)s.charAt(i));
         else{
           i++;
```

```
        if(s.charAt(i)=='n')
          {writer.println(".quad "+(int)'\n');}
        else
          {writer.println(".quad "+ (int)s.charAt(i)
      }
    }
    writer.println(past+":");
    types.put(node, new Integer(1));
}
```

Parameters

Identifiers occur in two contexts in Hi, as parameters to functions and as function names. We deal next with the the translation of names.

We associate with each name currently in scope a description. If it is a parameter this description is of class param

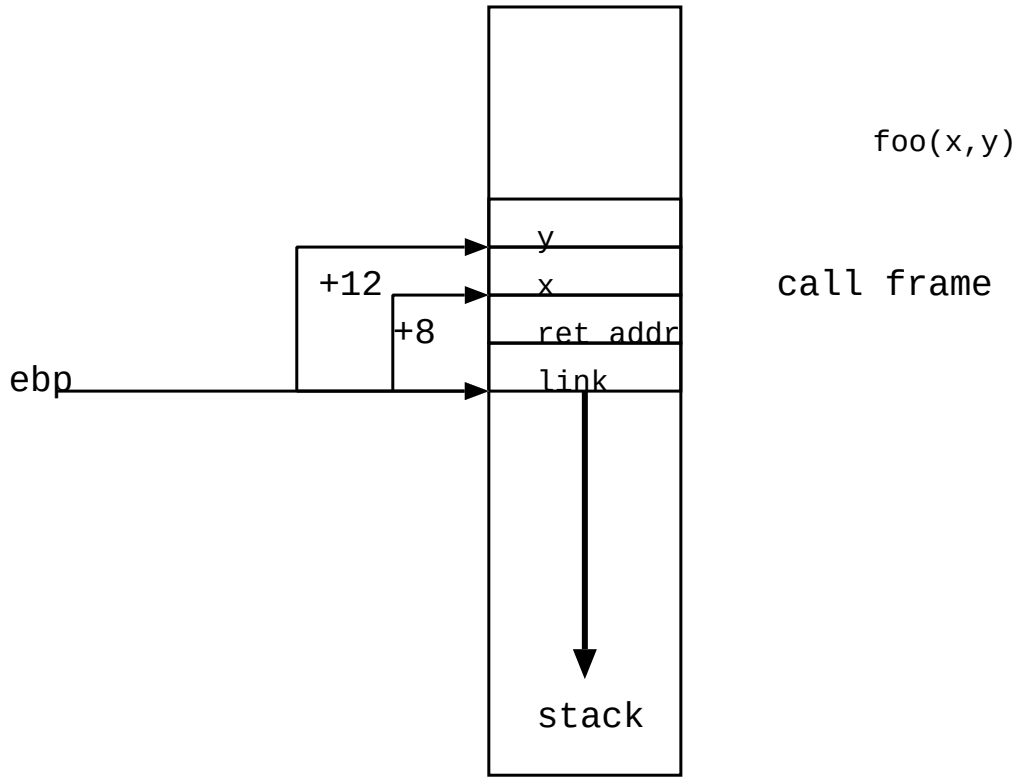
```
class param{
        int rank,offset;
        param(int r,int o){rank=r; offset=o;}
}
```

If the name denotes an nary function then we associate with it a Vector of length n+1, for which the elements 0..n-1 indicate the parameter ranks and element n indicates the rank of the result.

In what follows I initially present the parameter mechanism that are used on 32 bit Pentium family processors.

Subsequently I will look at how parameters are passed for 64 bit Opteron family (x86-64) processors, since calling works differently on these.

Interpreting the offset field of a param



Thus for x we would generate

```
push dword[ebp+8]
```

etc

In a Pentium you just push parameters on the stack and then make a call.

Opteron style calling

In an Opteron class processor parameters are mainly passed in registers.

If the parameter is INTEGER or POINTER, the next available register of the sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9 is used

If the parameter is floating point the next available SSE register is used, the registers are

taken in the order from %xmm0 to %xmm7

If insufficient registers are available, then remaining parameters are passed on stack

Saving parameters

Since functions may be recursive we can not leave parameters in registers.

We thus have to save them on the stack after function entry.

We give them an address below the frame pointer.

Suppose we have f(a,b), then on entry we have the following code

```
enter 0,0
push %rdi # save a offset will be %rbp -8
push %rsi # save b offset will be %rbp -16
```

Opteron style returning of results

Integer and pointer results return in %rax

Floating point results in %xmm0

Monadic operations

These are of the form

```
{monad} op atom;
```

for instance we might have !5 for the factorial function if we had implemented that.

```
!(scalar x->scalar)
 if x<=1 then 1 else x* !(x-1) fi;
HiMain(->scalar)putNum(!5)*0;
```

This will print

```
120
```

Sequence of operations:

1. Look up the operator and report an error if it was not declared

2. Check that it takes only 1 argument

3. Check that the type of the actual argument matches the type of the declared argument

4. Look up the implementation of the operator

5. Print the implementation

Operator implementations:

1. Create a hash table indexed on the operator name

2. If the operator is declared inline, store the assembler string for the operator as the implementation.

3. If the operator is declared as a high level operator then allocate a unique label - for instance the hex expansion of the characters of the operator, so ! might be represented as OP21

4. Place call to this in the implementation "call OP21\n"

5. Preceed the code generated for the body of the operator with this label.

Example

```
So putNum(!5) compiles to
push $5
pop %rdi
call OP21
push %rax
pop %rdi
call putNum
```

If we are smart we can optimize this to:

```
mov $4 , %rdi
call OP21
mov %rax, %rdi
call putNum
```

and the operator declaration

```
!(scalar x->scalar)
 if x<=1 then 1 else x* !(x-1) fi;
```

compiles to

```
OP21:enter $0,$0
# followed by the body of the ! code
# followed by the return code
pop %rax
leave
ret 0
```

whereas the expression x-1

would compile simply to

```
push $1
pushq -8(%rbp)     # push x
pop %rax
subq 0(%rsp),%rax # subtract top of stack (1)
movq %rax,0(%rsp) # store result on stack
```

All this is assuming that we have a prior definition of binary - as:

```
-(scalar x,scalar y->scalar)inline 'pop %rax
subq 0(%rsp),%rax
movq  %rax,0(%rsp)';
```

```
push $1
pushq -8(%rbp)    # push x
pop %rax          # x now in rax
subq 0(%rsp),%rax # subtract top of stack (1)
movq %rax,0(%rsp) # store result on stack
```

can be simply optimised to

```
push $1
mov  -8(%rbp)  , %rax  # load x into rax
subq 0(%rsp),%rax     #-
movq  %rax,0(%rsp)
```

A more sophisticated optimisation might be

```
mov  -8(%rbp)  , %rax  # load x
dec %rax
push %rax
```

Note that the above optimisation involves spotting that there is a special decrement instruction available.

Conditionals

Let us look in more detail at what we generate for the conditional expression in the definition of the factorial function:

```
if x<=1 then 1 else x* !(x-1) fi;
```

Remember we are generating code implements the Sable Reverse depth first visitor class. This means that

1. each subtree is visited without being aware of what it is located in, the visiting of the x*!(x-1) generates the code for this without being aware it is in an if expression

2. the visiting is done from right to left (we need this for parameter passing), this implies the else-code is generated followed by the then-code followed by the if-code

if x<=1 then 1 else x* !(x-1) fi;

So we start with a jump to the if which comes at the end:

```
jmp llll1if
```

followed by the else code

```
 jmp llll1if
llll1else:
push $1
mov  -8(%rbp)  , %rax
subq 0(%rsp),%rax     #-
movq  %rax,0(%rsp)
pop %rdi
        call OP21
push %rax
mov  -8(%rbp)  , %rax
imulq 0(%rsp),%rax
movq %rax,0(%rsp)    # *
jmp llll1fi
```

Next we handle the then code

```
llll1then:
push $1
jmp llll1fi
```

Finally we handle the conditon

```
llll1if:
push $1
mov  -8(%rbp)  , %rdi
pop %rsi
        call OP3c3d   # do comparison
and %rax,%rax
jnz llll1then
jmp llll1else
llll1fi:
```

To alter the order of this towards a more natural order we would have to overide the visitor method in ReversedDepthFirstAdaptor for if then elses from:

Reverse order

```
public void caseAConval(AConval node)
    {
        inAConval(node);
        if(node.getFalse() != null)
        {node.getFalse().apply(this);}
        if(node.getTrue() != null)
        {node.getTrue().apply(this);}
        if(node.getCond() != null)
        { node.getCond().apply(this); }
        outAConval(node);
    }
```

Normal order

```
public void caseAConval(AConval node)
    {
        inAConval(node);
        if(node.getCond() != null)
        { node.getCond().apply(this);}
        if(node.getTrue() != null)
        { node.getTrue().apply(this);}
        if(node.getFalse() != null)
        { node.getFalse().apply(this); }
        outAConval(node);
    }
```

Maps

function application hsd the syntax

```
map          = id  lparen actualparams rparen;
```

Example:

```
HiMain(->scalar)putNum(indexfn(myarray(3),2));
indexfn(vec1 a,scalar i->scalar)a#(i);
myarray(scalar n->vec1)n*iota( n);
```

This will print out 6

```
myarray(3)
```

evaluates as

```
-> 3*iota(3)
-> 3*[1,2,3]
-> [3,6,9]
```

we then call indexfn with parameters set as

```
a<-[3,6,9], i<-2
a#(i)
```

We evaluate this to yield 6.

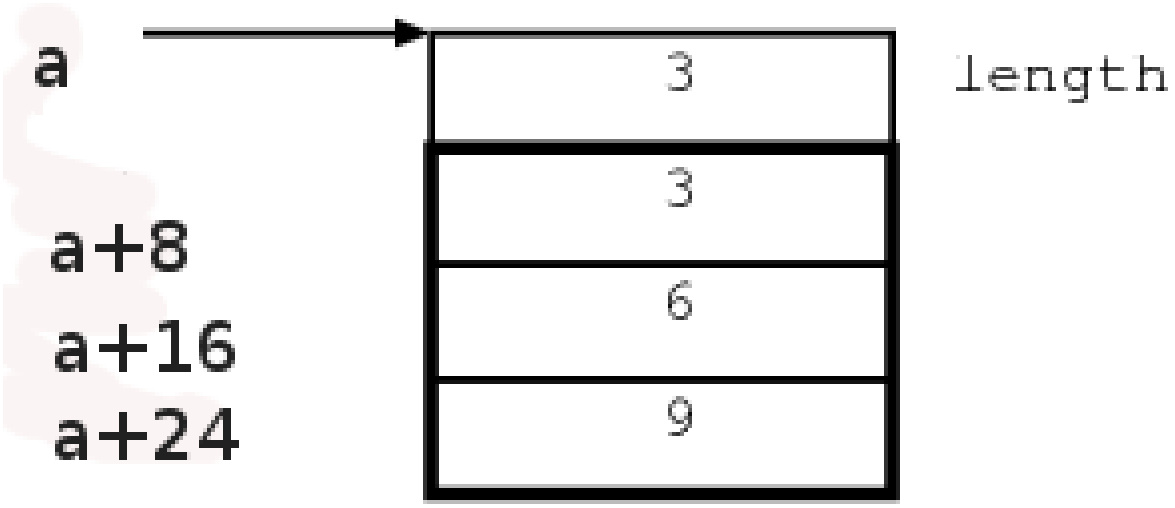
Here is the code for the # operation

```
 /* vector subscription operator */
#(vec1 v, scalar s->scalar)inline 'pop %rdi
pop %rsi
pushq 0(%rdi,%rsi,8)';
```

```
pushq 0(%rdi,%rsi,8)
```

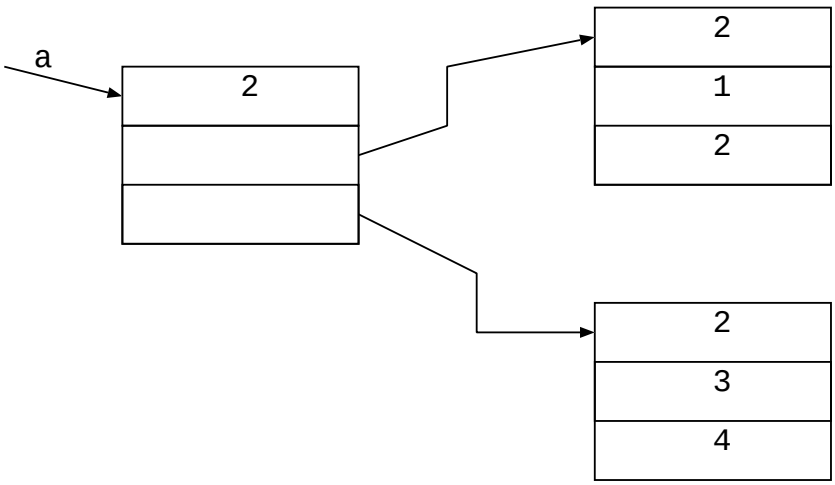
This is the crucial step, rax contains **a**, the address of the vector, rsi contains the index.

the vector looks like:



so if rsi contains 2 the instruction will load `mem[a+2*8]=mem[a+16]= 6` into rax as we require.

Structure of a 2d vector:



This representation of 2d arrays is refered to as an Iliffe vector after Iliffe an ICL designer who invented them

Issues for you to consider

1. How are we to represent vectors if our numeric representation moves from integers to reals?

2. How does this affect the code required to index vectors?

3. Should we use 4 or 8 byte reals?