Compiler Course Level IV: Practical exercise

Paul Cockshott

Contents

CHAPTER 1

Introduction

In this course you will have to produce a compiler written in java for the simple functional Language Hi. This document describes the language and the steps that you must undertake to perform the practical work. More details on the practical work can be found in section 5.1.

CHAPTER 2

Hi

This describes the tiny programming language Hi, which is designed to be a minimal example language for teaching compilers.

It illustrates in minimal form:

(1) An infinite type system.
(2) Functions and parameter passing.
(3) Computationally complete control structures.
(4) Operator overloading.
(5) Separate compilation.
(6) The possibility of parallel evaluation of code.

In this chapter I give an informal description of the syntax and semantics of Hi. In the exercises you will be given part of an implementation of Hi that uses integer representation for its numbers, you will have to produce an implementation that uses floating point representation.

2.1. Types

2.1.1. Scalars. Hi has a single base type: the scalar. These may be treated as either ASCII characters or as numbers. The representation of numbers - the number of bits used or whether reals or integers are used is up to the implementor. I will present an implementation using 32 bit integers, you may be asked to use a different implementation.

In the source code of the pogram words can be represented as decimal numbers. Thus `7, 42, 99` are valid literal representations of scalars. Minus numbers are written thus:

```
_50
```

using the underbar character. This is done to make parsing easier by ensuring that - is always treated as a binary operator.

The type of a scalar is named `scalar`.

2.1.2. Vectors. Vectors of arbitrary dimension are supported. A one dimensional vector constructor can be written thus: `[ 67, 90, 12 ]`

or thus`'here we go'`

In the latter case any sequence of ASCII characters can be embeded in the string. The Pascal sort of escape mechanism is used, thus "" is the representation of the single quote character.

An n dimensional vector constructor for $n > 1$ is written as a comma separated list of $n - 1$ dimensional vector values in [].

The type of an n dimensional vector is named `vec`n, thus: `vec1, vec3` are valid vector type names. The name `vec0` is an alternative name for the type name

`scalar`. The number which follows the word `vec` in a type is called the *rank* of the type.

2.1.2.1. *Null vectors.* A null vector is written either as : `[ ]` or ''. A null vector is of type `vec1`.

2.2. Functions

A function maps a list of arguments to a result. Function calls have the form: `foo(p,q,r)` Where `foo` is a function name and `p,q,r` are values.

An example function declaration would be

```
dbl(scalar a->scalar) a+a
```

In this `dbl` is the name of the function, `(scalar -> scalar)` is the type of the function and `a+a` is the expression that evaluates to give the function result. The identifier `a` names the parameter to the function.. Parameters must be typed.

Parameter names are sequences of alphabetic characters. Function names are either sequences of alphabetic characters or, sequences of one or more characters drawn from the operator alphabet

```
+ - = \ / . ! # & @ * ~ | % _ ? < > :
```

A sequence drawn from the operator alphabet is an *operator name*.

A function declared with an operator name must have two parameters.

External functions (written in C for instance) can be denoted using the word `external` in place of the expression that is evaluated to give a result.

2.3. Expressions

An expression is a formula that is evaluated to yield a result. Expressions are either:

(1) primary values, which are:
  (a) literal values (section 2.1.1)
  (b) parameter names (section 2.2)
  (c) subscripted vectors
  (d) vector constructors (section 2.1.2)
  (e) conditional values
  (f) function calls
  (g) bracketed expressions
(2) operator expressions

2.3.1. Primary values.

2.3.1.1. *Subscripted vectors.* If x is a `vec`n valued expression and y is `vec`m valued expression then $(x\#y)$ is a `vec`l valued expression where $l = m + n - 1$.

Examples.

```
[ 1,2,3,5,7](4) = 5
[ 1,2,3,5,7]([3,2,4]) = [3,2,5]
'hello'(3)= 'l'
'hello'([2,3,5,1]) = 'eloh'
['me','too'](2)= 'too'
[[9],[8.7]](2) = [8,7]
['me','too'](2)([2,1])= 'ot'
['me','too']([[1,2],[2,1]])= [['me','too'],['too','me']]
```

```
'zot'([[1,2],[2,1]])= ['zo','oz']
```

If a vector has length n then attempting to subscript by numbers outside the closed interval $[1..n]$ will yield undefined results.

2.3.1.2. *Function calls.* A function call has the syntactic form $f(x,y,,,,,z)$ where f is a function name, and $x,y,$..etc are values. The types of the parameter names declared in the function definition must exactly match the types of the corresponding values supplied as actual parameters except as described in the section on operator overloading 2.3.3.

2.3.1.3. *Bracketed expressions.* Any expression p can be enclosed in brackets thus (p). Bracketing imposes an association order on operator expressions.

2.3.1.4. *Conditional Evaluation.* An expression can be conditionally evaluated using the `if` .. `then` ... `else ...  fi` thus where `x=4`

```
if x>2 then a else b fi
```

will evaluate to `a`. The conditional expression evaluates to the `then` branch if the expression between `if` and `then` is non zero. Otherwise it evaluates to the `else` branch.

2.3.2. Operator expressions. In Hi all operators are of the same priority. Expressions are evaluated right to left. The pre-defined operators and their signatures are:

Operator	Signature	
+	(scalar , scalar → scalar)	signed addition
*	(scalar , scalar → scalar)	integer multiplication
/	(scalar , scalar → scalar)	integer division
-	(scalar , scalar → scalar)	signed subtraction
<	(scalar , scalar → scalar)	less than
>	(scalar , scalar → scalar)	greater than
&&	(scalar , scalar → scalar)	and
||	(scalar , scalar → scalar)	or
==	(scalar , scalar → scalar)	equal to
#	(vec$n$,scalar →vec$n-1$)	column subscription
##	(vec2,scalar →vec1)	matrix row subscription
|	(vec$n$,vec$n$→vec$n$)	concatenate
~	(scalar →scalar)	not, maps 0→1, 1→ 0
_	(scalar →scalar)	negate, multiplies by -1

Examples of operator expressions are:

```
2+2                 = 4
a-b*c               evaluates as a-(b*c)
1-2*3               = _5
a*b-c               evaluates as a*(b-c)
2*3-1               = 4
[1,2,3]|[7,11,13]   = [1,2,3,7,11,13]

[[11,12],[21,22]]#2 = [12,22]
[[11,12],[21,22]]##2= [21,22]
[1,2]#1             = 1
'hi'|'lo'           = 'hilo'
```

2.3.3. Overloading of operators. All primitive and user defined operators are overloaded to work on vectors of higher dimension than those for which they are defined. Examples using predefined operators are:

```
2+[9,11,7]              =[11,13,9]
[2,3]*[3,4]             =[6,12]
[1,2,3]+[4,5]           =[5,7]
[1,2]*[2,3,2]           =[2,6]
[[1,2],[3,4]]-[1,0]     =[[0.2],[2.4]]
```

Similarly functions are overloaded so that given

```
foo(scalar a, vec1 b, vec2 c-> vec1)a+b*c(a)
```

the expression

```
foo(1,[[3],[4,5]],[[10,20],[30,40]])
```

will yield a result of type `vec2: [[31],[41,101]]`.

2.4. Primitive functions

The following functions have to be provided by a runtime system and can not be written in Hi itself.

length. The `length(vec1 → scalar)` function returns the number of elements in a vector.

getChar. The function `getChar(→ scalar)` returns the next character from standard input.

putChar. The function `putChar(scalar→ scalar)` outputs a scalar to standard output as a character. It returns its input parameter.

iota. The function `iota(scalar →vec1)` generates a vector containing an ascending sequence of integers. Thus `iota(4)`$\rightarrow [1,2,3,4]$

The following functions should be provided in a Hi implementation either in a library written in Hi itself or as externals.

getNum. The function `getInt(→ scalar)` parses the input to read the next decimal number.

putNum. The function `putInt(scalar→ scalar)` outputs a scalar to standard output as a decimal number. It returns its input parameter.

2.5. HiMain function

A Hi program is a sequence of function declarations. One of these must be called `HiMain`. This should be declared to have a type that matches its actual return value. This function will be executed as the main program.

CHAPTER 3

Semantic discussion

The salient features of this language are:

- It is applicative, there are no assignments in the language.
- It is partially functional, in that it supports functions but not higher order functions.
- It is an array language which provides operations on whole arrays.

Many of the semanic problems with implementing the language stem from the array operations, but when considered in conjunction with its applicative character, these array features open up considerable scope for parallelism in code generation.

3.1. Overloading again

It is worth looking a bit further at the semantics of the array overloading of functions and operators.

Given an *n*ary function g whose result is of rank p we can describe the ranks of its formal parameters with an n element vector of integers. Given a function call we can similarly describe the actual parameters' ranks. Let us call the first vector $\mathbf{f}$ and the second $\mathbf{a}$. We define the overloading degree δ as

$$\delta = max(\mathbf{a}_i - \mathbf{f}_i)\forall_{i=1..n}$$

The result of applying g in this context is then of rank $p + \delta$. In the case where the function has more than one overloaded parameter, then the lengths of the overloaded result vector $\mathbf{v}$ can be defined as

$$length^j(\mathbf{v}) = min(length^{j-p}(\mathbf{o})\forall_{\mathbf{o}\in\omega_j}\forall_{j=p+1..p+\delta}$$

where ω_j is the set of parameters overloaded by degree j.

3.1.1. examples.

(1) Let a function f be of rank $[0,1,2] \rightarrow 1$. If it is supplied with actual parameters of rank $[1,1,3]$ then clearly $\delta = 1$, and the rank of the result will be 2. Suppose the lengths of the first dimension of the actual parameters are $[3,2,5]$ then $length(\mathbf{v})$ the length of the result will be $[x,y,z]$ for some integers $,x,y,z$, and the length of the first dimension of the result, $length^2(\mathbf{v})$, in the equation above, will be $length([x,y,z]) = 3 = min([3,5])$, the minimum of the first dimension of the overloaded parameters.

(2) If the function f above is applied to actual parameters a,b,c of ranks $2,2,2$ then the result will be of rank 3. If $length(a) = [3,10]$ and $length(b) = [4,5,6]$ then clearly $length^1(\mathbf{v})$ will be a 2 dimensional array $[[h,k,l],[r,s,t,u,v]]$ for some non-negative integers h,k,l,r,s,t,u,v. Furthermore

$$length^2(\mathbf{v}) = [3,5] = min([length^1(a), length^1(b)])$$

$$= min([[3, 10], [4, 5, 6]])$$

Finally the length of the first dimension of the result $length^3(\mathbf{v})$ will be 2 since

$$2 = length([3, 5]) = length(length^2(\mathbf{v}))$$

.

3.2. Compile time versus run time storage allocation

The language Hi can be statically type checked. That is to say the compiler can always determine if an expression returns a scalar or a vector, and if it is a vector, what the rank of that vector will be. On the other hand the compiler can not be sure at compile time what the length of a vector produced by an expression will be.

For this reason it is necessary for vectors to be allocated space on the heap rather than on the stack. Furthermore, since there is no explicit vector de-allocate operation, it is desireable to have a garbage collector available.

CHAPTER 4

Syntax specification and Sable

4.1. Hi Syntax

I give below a syntax specification for Hi written in the Sable grammer definition language.

```
Package Hi;
/*\end{verbatim}
\subsection{Helpers}
Helpers are regular expressions macros used in the definition of terminal symbols of t
\begin{verbatim}*/
Helpers
  letter = [['A'..'Z']+['a'..'z']];
  digit = ['0'..'9'];
  alphanum = [letter+['0'..'9']];
  cr = 13;
  lf = 10;
  tab = 9;
  digit_sequence = digit+;
  letseq = letter+;
  opsym =  '+'|'-'|'='|'\'|'.'|'!'|'#'|'&'|'@'|'*'|'~'|'|'|'%'|'_'|'?'|'<'|'>'|'/';
  opseq = opsym+;
  eol = cr lf |cr |lf;        // This takes care of different platforms
  not_cr_lf = [[32..127] - [cr + lf]];
  quote = '';
  all =[0..127];
  schar = [all-''];
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];
Tokens
arrow= '->';
bra= '[';
comma= ',';
def='def' ;
dot='.';
else= 'else';
external= 'external';
fi= 'fi';
if= 'if';
inline= 'inline';
ket= ']';
```

```
lparen = '(';
number= digit_sequence;
rparen = ')';
scalar= 'scalar';
semi= ';';
then= 'then';
vec= 'vec';
/* composite tokens */
id = letseq;
op=opseq;
comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
string = quote schar+ quote;
blank = (' '|cr|lf|tab)+;
Ignored Tokens
        blank,comment;

Productions
program     = fndecl + ;
actuallist  = expr comma;
actualparams= {emptyparams}| {paramlist} actuallist* expr;
atom        = {literal} literal| {paramname} id| {comp}comp;
body        = {expression}expr semi| {external} external  semi| {inline} inline strin
comp        = {vector} vectorcon | {if} conval | {bracketed} lparen expr rparen;
cond        = expr;
conval      = if cond then true else false fi;
expcomma    = expr comma;
expr        = {opexpr} opexpr | {secondary}secondary ;
false       = expr;
fndecl      = id  lparen params arrow type rparen body |{opdec} op lparen opparam arr
literal     = {string} string| {scalar} number;
map         = id  lparen actualparams rparen;
opexpr      = primary op expr ;
opparam     = [l]: param comma [r]:param|{monop} param;
param       = type id;
paramlist   = param comma;
params      = {emptyparams} | {paramlist}paramlist* param;
primary     = {map} map|atom;
secondary   = {primary}primary| {monad} op atom;
true        = expr;
type        = {scalar} scalar| {vector} vec number ;
vectorcon   = bra actualparams ket ;
```

4.2. Sable

Sable is a compiler compiler developed at the University of McGill in Montreal. It takes in syntax specifications and writes out a collection of Java source files. These Java files constitute a parser for the language described in the syntax specification. When given an input stream the parser will parse the input and, if

it is syntactically correct, build a syntax tree for the program so parsed. Details of how to use it are given in the lecture notes.

CHAPTER 5

Your exercise

5.1. Transformation of the program to prefix functional form

In Hi there are three different ways to write expressions that do calculations.

(1) You can use a function call as in f(x,y)
(2) You can use a prefix operator as in ~ a
(3) You can use an infix operator as in a+b

All of these can be considered syntactic sweetening on a single basic mechanism : function application. Thus in principle one could syntactically transform an expression of the form:

```
a +  (b*2)
```

to

```
add(a,times(b,2))
```

and the expression `~ a` could become the function call

```
not(a)
```

We would not want this in the source code but it may be desirable to perform the transformation when translating the code. If we do, then there will only be one place in the compiler that has to deal with generating code for calculations - where a function call occurs.

As a first phase of your project you are to do the following

(1) Build a parser using Sable for the Hi Grammar
(2) Write classes that will manipulate the syntax tree produced by Sable to rewrite all operator expressions as function application
(3) Demonstrate this working by printing out the transformed syntax tree.

5.1.1. Sample code. Here is sample code that will remove monadic expressions from the syntax tree. You can extend this class to allow the removal of dyadic operator expressions as well.

```
package Hi ;
import java.util.*;
import Hi.node.*;
import Hi.analysis.*;
public class MonadRemover extends DepthFirstAdapter  {
 public void outASecondaryExpr(ASecondaryExpr node)
    {
        PSecondary second = node.getSecondary();
        if (second instanceof AMonadSecondary){
                AMonadSecondary amonad = (AMonadSecondary)second;
                PAtom atm= amonad.getAtom();
```

```
                String id = amonad.getOp().toString() .trim();
                AMap map = new AMap();
                TLparen lp = new TLparen();
                map.setLparen(lp);map.setRparen(new TRparen());
                map.setId(new TId(id));
                AParamlistActualparams apl = new AParamlistActualparams();
                apl.setExpr( new ASecondaryExpr(new APrimarySecondary(new APrimary(at
                map.setActualparams(apl);
                node.setSecondary(new APrimarySecondary(new AMapPrimary(map)));
            }
        }
    }
```

5.2. Generate a full compiler

Having transformed the program to an internal form that contains only function applications, you can now go on to produce a full compiler with type checker and code generator.

(1) Assume that your first pass over the abstract syntax tree converts it to functional form.
(2) A second pass can find all the operator and function declarations and enter these into a symbol table along with their types.
(3) A third pass can walk over the modified syntax tree one more time and check as it goes that the type rules are not broken. It can also output assembler code equivalent to each function or expression as it walks over it.
(4) A fourth phase will use the gnu assembler to convert the assembler file into a .o file.
(5) A final phase will invoke the gcc compiler to link this .o file with the run-time library.
(6) It will also be necessary to write a shell script `hic`[1] that will :
    (a) append the postlude file to the source file and put the result in a temporary file
    (b) feed this temporary file into your compiler
    (c) invoke the assembler
    (d) invoke the gcc compiler
(7) It may also be handy to create a shell script `run` which will compile and run the hi program.

[1] For Hi Compile.