An introduction to KP Language - Ch2

Tae Yano
tae.yano@hunter.cuny.edu
ty2142@columbia.edu

Oct/21/2004

CSMS W4115
Programming Language and Translator
Fall 2004

Document Type: language Reference Manual -- first draft

2.1 Introduction (and example to illustrate the language)

The KP language compiler takes a KP source code as a input and produces a Java executable file as output. When the executable runs on JVM, it creates unique "knit" objects and does actions (e.g., printing, resizing, saving, etc) on them as instructed in the source. This is about all, by design, that the KP language is good for.

The "Knit" object is, in its most basic sense, an abstract data structure representing knitted/knittable items. This data structure is a collection of elements called "Stitch". Stitch, like Knit, is a data type unique to KP language. It is not unlike the Boolean type but has three states instead of two;: it is either "k" (for knit stitch), "p" (for pearl stitch), or "s" (for skip or not care). All the properties of a given Knit object is defined by the values, or absence, of its stitches and how they are connected to each other.

An easier way to picture knit data structure is to think it as a string of characters from a character set with only three values (k and p and s), with capacity to extend vertically as well as horizontally. Probably the readiest abstract data type is a multi-dimensional, doubly-linked, list such as this one:

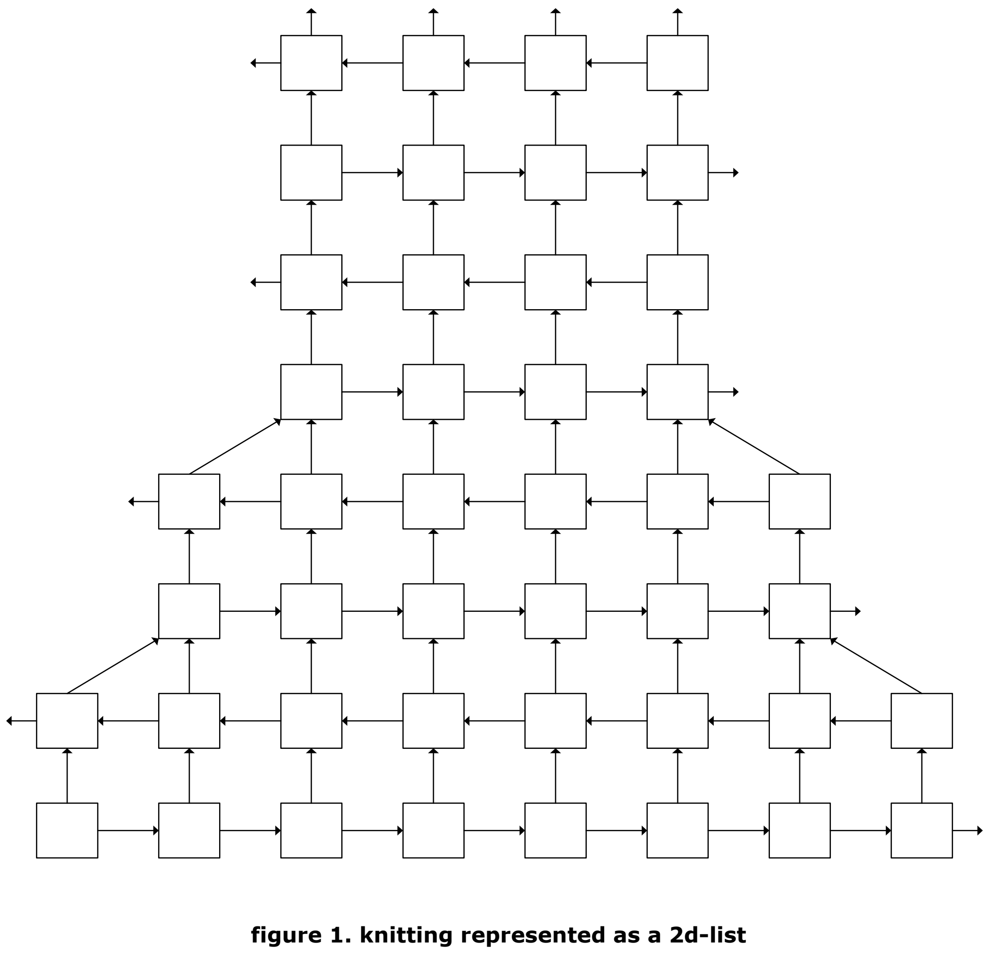

figure 1. knitting represented as a 2d-list

A code to create this instance of the data structure would look something like the code snippet below. Here I used so-do c++ code. It shows pointer operations which, with java, would be hidden by API. This is not the only implementation there is.

```
struct knit_element{
       char sts,
       knit_element *up,
       knit_element *side
}
knit_element
       stitch0[8], stitch1[8],
       stitch2[6], stitch3[6],
       stitch4[4], stitch5[4], stitch6[4], stitch7[4];

knit_element *this_knit = stitch0;

stitch0[0].sts = 'k';            //this is the very first stitch
stitch0[0].side = & stitch0[1];
stitch0[0].up = & stitch1[7];

//"up" of the first stitch is the last stitch of the 2nd row...
//this is how knitting works.

stitch0[1].sts = 'p';
stitch0[1].side = & stitch0[2];
stitch0[1].up = & stitch1[6];

stitch0[2].sts = 'p';
stitch0[2].side = & stitch0[3];
stitch0[2].up = & stitch1[5];

//and so on and on...

stitch0[7].sts = 'p';
stitch0[7].side =NULL;           //turn the row
stitch0[7].up = &stitch1[0];

//and so on and on...

stitch7[2].sts = 'p';
stitch7[2].side = & stitch7[3];
stitch7[2].up = NULL;

stitch7[3].sts = 'p';            //this is the last stitch
stitch7[3].side =NULL;
stitch7[3].up = NULL;            //end of the binding (i.e., last row)
```

The KP language definition of this object would look like this:

```
//note--spacing is for ease of read. no semantic meaning in them.
//k, p, dec are all build-in (or atomic) knit objects.
//only build-in knit object can appear in the rows

knit this_knit = {
       row1:   k, p, k, p, k, p, k, p;
       row2:   k, p, k, p, k, p, k, p;
       row3: dec, p, k, p, k, p, k, dec;
       row4:    p, k, p, k, p, k;
       row5:  dec, k, p, k, p, dec;
       row6:       k, p, k, p;
       row7:       k, p, k, p;
       row8:       k, p, k, p;
}
```

The object, as it is now, does not do much that users can see. A data structure is constructed, and just sits somewhere in the memory.

From the knitter's stand point, one very useful thing would be to print out a graphical representation of this item. There is a well-defined, widely-used common set of notations among knitters (commonly referred as "symbolcraft" or "knitting chart" [*]) and KP has a built-in function to do just this:

```
printk( this_Knit );
```

or

```
printk( this_Knit, PIC );
```

will print out a knitting chart like this (those symbols are abbreviated version):

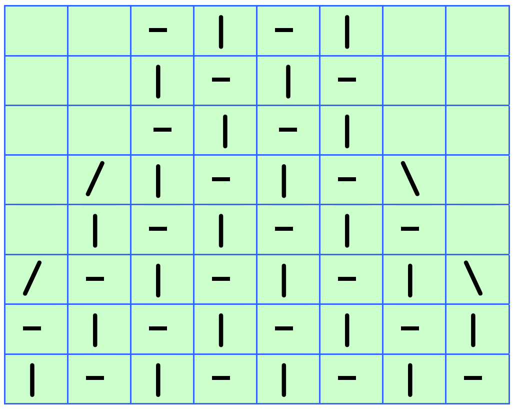

figure 2. knitting chart to knit figure 1

[*] ISO 8388:1998 "knitted fabrics vocabulary" delineates symbols as well as the use of them. More ready and user friendly ones are available from JSC (Japan Industrial Standards Committee) database (JISL0201).

There are several other built-in functions to do essentials for knitting projects. One such is savek(), which creates an image file (in a local file system) of the image:

```
       savek( this_knit );
```
or
```
       savek( this_knit, PIC );
```

Optionally, printk and savek functions output the Knit object in row-by-row KP notation:

```
       printk( this_knit, INSTRUCTION );
       savek( this_knit, INSTRUCTION);
```

When the object is saved in this format, it can be read directory into a KP application with readk() :

```
       readk( this_knit );
```

This is a convenient format for knitters too. Because If this object is going to be knitted, row-by-row instructions, not a chart, is what they need. Although charts are useful for designing and visualizing finished products, since they do not convey the sequential flow of the stitches, it is very difficult to follow whey you are knitting.

Although the instructions appear in the file are written in the KP language, and it is not an exact replica of the common knitters' notation, the two are sufficiently close for all knitters to understand without much trouble.

2.2 KP language Lexicon

2.2.1 Character Set

The KP source code is a sequence of characters from a character set defined in Basic Latin block of ISO/IEC 10646. Any character from this set can appear in the source, though not all of them are meaningful in the language.

2.2.2 White Space and Comments

(Blank) spaces, vertical and horizontal tabs, new lines, and carriage returns are known, collectively, as white space. All white space will be ignored except insofar as they are used to separate adjacent "tokens" or when they appear in character or string constants.

Both C and C++ style comments are supported: "//" start a single-line comment. Multi-line comments appeared between "/*" and "*/". Texts appeared as comments will be treated as white space and ignored similarly. Comments can not be nested.

2.2.3 Tokens

The sequence of characters making up a KP source code are grouped into atomics elements, called *tokens*, according to the rules explained in this document. There are five classes of tokens: *operators*, *separators*, *identifiers*, *keywords*, and *constants*. Tokens are separated from adjacent ones with one or more of above mentioned white space.

2.2.4 Operators and Separators

The following character sequences when appears in a KP source code are considered as operator tokens:

```
        = + - * / ~ & | ! .
        ++ -- += -= *= /= %=
        >\ >/ </ <\ >> << && || <= >= == !=
```

And these are separator (punctuation) tokens:

```
        ( ) [ ] { } , ; :
```

2.2.5 Identifiers

An identifier is a sequence of upper and lower case letters, digits and under score ("_") character. An identifier must start with non-digit and not spell same as a keyword.

2.2.6 Keywords

The character sequences listed below are keywords n KP language:

```
knit row eye stitch up side
k p dec inc tg yo begin end
break else elseif loop if
char string int float func
```

2.2.7 Constants

The constant class tokens are further broke down into four kinds: integer constant, floating constant, character constant, and string constant.

A string constant is a sequence of characters enclosed between double-quotes (""). A character constant is a single character enclosed between single-quotes (''). An integer constant is one non-zero digit followed by a sequence of digits. The sequence is preceded, optionally, by "-" sign. A floating constant is an integer sequence followed by decimal point, and a sequence of digits.

2.3 KP language Syntax

2.3.1 Statement

A KP program is a collection of "statements". A statement is an executable block of code which, if appears independently in the body of the program, executed sequentially at the invocation in the order of appearance. There are five kinds of statement: *Knit definition*, *build-in function call*, *conditional statement*, *loop statement* and *assignment*. When statements appear inside of a block (within "{" and "}") they make up a syntactical collectives called *statement-list*. Those statements are explained farther below.

2.3.2 Data Type

There are not many data types in KP. There are "`char`" for character, "`string`" to name array of characters, "`int`" for integer, and "`float`" for double precision floating point numbers. (Though those are included in the language, more or less, for convenience.[†]) In addition to those conventional data types, there are "`knit`", "`eye`", and "`stitch`" data types unique to the KP language.

As mentioned, `knit` is an abstract data structure of knitting/knittable items. Internally it is a collective of objects of type `eye`. An eye consist of a `stitch` object, and pointers (or references) to two other eyes. Stitch is a trinaly data type whose state is either '`k`', '`p`', or '`s`'.

The KP language has a set of pre-defined short-hands (`k, p, dec, inc, tg, yo`) to make up a rows of knit. Although quite a few knitting projects are possible with only this set, programmers can define their own eyes, connect them together and make up a knit item "from scratch".

2.3.3 Declaration and Scope

To declare is to give an entity a name - *identifier*- by which it can be referred throughout a given domain. In the KP language a block consists of one domain, or *Scope*. Identifiers are visible only within the block they are declared. Named block (such as knit definitions, below), variables or function declaration outside of any block are therefore globally visible.

A declaration appears *type specifier* followed by one or more identifiers, and optional initial assignment (*initialization*). Identifiers can not be used before it declared and must be unique throughout its scope.

2.3.4 Knit Definition

Knit definition is a named block which defines the properties of a knit object. There have to be at least one `eye` object in a given definition (i.e., empty definition is illegal). There are more than one way to define knit object within KP language. The list below is the specification for construction:

[†] Just as the original Bourne shell's not having numerical variable (every user decreased variables are character or character array)...

knit_definition :
'**knit**' id '=' '{' ('**begin**' ':' *proc_block*)? *unit_definition_list*
('**end**' ':' *proc_block*)? '}'
unit_definition_list:
(*row_definition)*+ | (*eye_definition*)+ | (*knit_reference*)+
row_definition:
'**row**' digit ':' ('**k**'|'**p**'|'**s**'|'**yo**'|'**tg**'|'**dec**'|'**inc**'|id) (',' '**k**'|'**p**'|'**s**'|'**yo**'|'**tg**'|'**dec**'|'**inc**'| id)*
';'
eye_defintion:
('**eye**')? (id | (id '.' ((**stitch** '=' ('**k**'|'**p**'|'**s**')) | (**up** '='id) | (**side** '=' id)) ';'
knit_reference:
('**row**' digit ':')? id (',' id)* ';'
proc_block:
statement | *statement_list*

For example, a simplest knit definition looks like this:

```
        knit garter = { row1: k; }
```
or:
```
        knit skitt = {
                row1: k, p, k;
                row2: k, p, k;       }
        knit r-skitt = {
                row1: p, k, p;
                row2: p, k, p;       }
        knit combine = {
                r-skitt, skitt;
                skitt, r-skitt;      }
```

If writing a knit "from scratch", this would look like:

```
        knit my_cable_unit = {
                begin:{
                        print( "define your own stitch..." );
                        print( "all un-assigned is initialized NULL ..." );
                }
                eye sts00.sts = k;
                eye sts01.sts = k; sts00.side = sts01;
                eye sts02.sts = k; sts01.side = sts02;
                eye sts03.sts = k; sts02.side = sts03;
                eye sts10.sts = k;
                eye sts11.sts = k; sts10.side = sts11;
                eye sts12.sts = k; sts11.side = sts12;
                eye sts13.sts = k; sts12.side = sts13;
                sts00.up = sts11; sts01.up = sts10;
                sts02.up = sts13; sts03.up = sts12;
        }
```

This makes up a knitting pattern below. This is a building block of a decorative pattern commonly called "Cable Knitting".

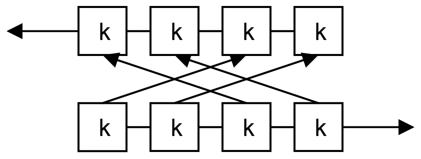

figure 3. atom of cable knit

2.3.5 Expression

Most of *Arithmetic* and *Comparison operations* does not take knit, eye, or stitch as *operands* but a few. ">>", or. *shift right operation* takes knit as an operands and shift the rows to right. "<<", or *shift left* works similarly but to the right.

There also are operators works only on knit objects. ">/", or *shift bottom up*, which augments the knit by one row by duplicating the bottom-most row, and "</" *shift top up,* duplicating the top-most row. ">\", or *shift bottom down* diminishes the knit by one row from the bottom-most, and "<\" *shift top down* from the top-most rows.

2.3.6 Conditional and Loop Statement

Conditional Statement construction takes the following form:

```
conditiona_statement:     if '(' expression ')' statement
                          elseif  '(' expression ')' statement
                          else statement
for_statement:   for '(' for_condition ')' statement
loop_statement:   loop '(' id ')' statement
```

loop_statemetn is often used with knit object to iterate through and examining the contents. for example:

```
//this su-do example go through the first row of the knit object
//and print out the stitch...

eye knit_itr;
stitch sts_holder;
knit_itr = this_knit;  //takes the first eye of the knit object
loop( knit_itr ){
        sts_holder = knit_itr.sts;
        print( "the contents of the eye is: %d \n", sts_holder );
        knit_itr = knit_itr.side;
}
```

2.3.7 Assignment

An *assignment*, when appears as *statemen*t, have to be terminated with ";", it otherwise treated as an *expression.* When it is operating on knit, eye, and stitch object, *Assignment operators* (=, +=, -=,*=, /=, %=) works differently from arithmetic data types such as int.

There is only one assignment, '=', works with knit item. When lvalue is a knit variable, it could take a knit identifier and this create a *reference* to the knit object. It could also take an eye variable, where the referencing object is a knit with the eye as a only component. When you need a duplicate of a knit object, you should use copyk() explained below.

2.3.8 Function Call

The language does not offer means for programmers to define there own functions therefore all the *function call* appears in a source code is call to a built-in function which is explained in the next section.

A function call appears with tailing ";" is treated *statement* (see previous section). It could appears where expression takes place without ";" and treated as one.

2.4 Build-in Functions

2.4.1 IO Functions

int printk(knit item, [format_id]);
Example: `int error = printk( this_knit, PIC );`
This function prints a knit object on the console (or, STDOUT the running program is associated to). printk function takes one, or two comma separated arguments. The first of the argument is the name (identifier) of the knit object to be displayed. The Optional "format_id" is either PIC or INSTRUCRION, and decides if the knit it printed as a knitting chart image or row-by-row instruction in KP. When this argument is omitted, it is assumed to be PIC.

int savek(knit item, [format_id], [filename]);
Example: `int error = savek( this_knit, PIC, "/root/myknit_dir/this_knit.k" );`
This function create a file in the local file system and write a knit object into the file. savek() function takes one, two or three comma separated arguments. The first of the argument is the name (identifier) of the knit object to be written. The Optional "format_id" is either PIC or INSTRUCRION, and decides if the knit it printed as a knitting chart image or row-by-row instruction in KP. When this argument is omitted, it is assumed to be PIC. The third argument "filename" is a name of a file it writes. When this is omitted, it uses an arbitrary chosen name and saves the file in the running program's current directory ("PWD"). The string must be within double quotation marks if a full path name is given. If the named file already exist, it return error.

knit readk([filename]);
example: `knit this_knit = readk( "/root/myknit_dir/this_knit.k" );`
This function read a knit object from a file in the local file system. readk function takes zero or one argument. "filename" is a name of a file it reads from. When this is omitted, it reads from STDIN. The "filename" string must be within double quotation marks if a full path name is given. If the named file does not exist, or the file is not of the KP format, it silently returns NULL object.

There is other more conventional print() function which takes a variable length argument list like C and C++.

2.4.2 Knit Functions

int widthk(knit knit_item, int skip_emp)
This function returns the widest across eye count (i.e., eye count of the longest row). If "skip_emp" is set to 1, it gives the count without skip stitches.

int heightk(knit knit_item)
This function returns the total number of rows.

int countk(knit knit_item, int skip_emp)
This function returns the total number of eyes in the given object. If "skip_emp" is set to 1, it gives the count without skip stitches.

int check(knit knit_item, int strict)
this function return 1 if the given knit coherent (can actually be knit). If "strict" is set to 1, it does not make any assumption when its checking so the knit have to be written precisely.

knit copyk(knit knit_item)
this function create a duplicate of the given knit object.

eye copye(eye eye_item)
this function create a duplicate of the given eye object.