An Overview of

KPL

A Kernel Programming Language

Harry H. Porter III
Department of Computer Science
Portland State University

October 14, 2003

Introduction

This document introduces the KPL programming language, which was designed for use in the BLITZ Operating System Project. KPL has many of the features in C++ and Java. This document is written for someone who is already familiar with C++ or Java.

The primary design criterion of the language was simplicity. The intent was to create a language that can be understood and acquired quickly, e.g., during a University-level course. A secondary design criterion was to create a language that facilitates readability and reliability of programs. As a consequence, the syntax emphasizes readability, at the expense of terseness and ease of typing.

Table Of Contents

The Hello-World Program

The "Hello-World" program prints a message and stops. The code for this program is broken into two files.

The first file is named "Hello.h" and is called the header file. Header files have an extension of ".h".

```
-- This is the header file for the "Hello-World" program...
header Hello
  uses System
  functions
    main ()
endHeader
```

The second file is named "Hello.c" and is called the code file. Code files have an extension of ".c".

```
-- This is the code file for the "Hello-World" program...
code Hello
  function main ()
      print ("Hello, world...\n")
    endFunction
endCode
```

The keywords of the language (symbols like **if**, **else**, **while**, **header**, **endHeader**, etc.) are shown in boldface.

Packages

Each program is broken into packages and every package has a name. In this example, there is one package and it is named "Hello". For each package, there must be one header file and one code file. The package and its files will have the same name, except for the extensions of ".h" and ".c".

Within the header file, there will be exactly one instance of the "header" syntactic construct, which has this general form:

```
header ID ...other things... endHeader
```

Likewise, the code file will contain a syntactic construct that has the following form:

```
code ID ...other things... endCode
```

A package will use other packages. In this example, the "Hello" package uses the package named "System". The relationship between packages is made explicit in the **uses** clause.

The **uses** clause has the following general form:

```
uses ID, ID, ..., ID
```

The **uses** clause appears only in the header file and appears directly after the **package** keyword and package name.

```
header Hello
  uses System
  ...
endHeader
```

The code in the Hello package calls a function named "print". This function is defined in the System package. If the programmer had failed to include "**uses** System" in the header file, the compiler would produce an error when compiling the package, to the effect that "The name print is undefined".

Compiling

Next let's compile and run this program. In the examples that follow, assume the BLITZ system is installed on a Unix/Linux system and that the shell prompt is "%".

The unit of compilation is the package. In other words, each package must be compiled separately and each compilation will process exactly one package. Here (in pseudo-code) are the steps we must take:

```
For each package...
    Compile the package to produce a ".s" file
    Assemble the package to produce a ".o" file
Link all the object files together to produce an "a.out" file
Invoke the BLITZ virtual machine to execute the "a.out" file
```

First, let's compile the "Hello" package. The KPL compiler is named "kpl". (User-typed input is shown **like this**.)

```
% kpl Hello
```

This will either print some compile-time error messages or will produce a file called

```
Hello.s
```

containing BLITZ assembly code. During the compilation, the compiler will notice that the Hello package uses the System package. The compiler will read and process the header file for System. The compiler must have access to the file named "System.h". However, the code file for System (i.e., the file named "System.c") does not need to be accessed when compiling Hello. In fact, the code file may not

yet have been written. Furthermore, if the System package happens to use other packages (actually, it does not), then the header files for those packages would also be read and processed by the compiler.

Next, we need to assemble the ".s" file. The BLITZ assembler is named "asm" and can be run with this command:

```
% asm Hello.s
```

The assembler will produce an object file, with the name "Hello.o". Normally, the System package will have been compiled and assembled already, so the files "System.s" and "System.o" will already exist. The System package is supplied as part of the KPL language and the programmer should not modify it. Here are the commands to create these files:

```
% kpl System -unsafe
% asm System.s
```

Safe and Unsafe

For the most part, the KPL language is strongly and safely typed. Bugs created by the programmer may cause erroneous behavior or generate error messages, but they should not cause a "crash" or core dump. However, KPL is a systems programming language and, like C and C++, the programmer can use the language in ways that will cause a crash. For example, the programmer can set a pointer to an arbitrary value and use it to store arbitrary data into any location in memory.

In KPL, several constructs in the language are considered "unsafe". Their use could lead to a crash if the programmer makes a mistake. If the KPL program never uses any unsafe constructs, all failures of the program will be tightly controlled. Either the program will produce erroneous results, or the runtime system will catch the bug and print a nice, clean error message. So, if unsafe constructs are avoided, the programmer should not be able to crash the system, no matter how bad the bug. On the other hand, if the programmer uses unsafe constructs, then it is possible for a bug to result in a program crash.

The compiler can be used in two modes. In "unsafe" mode, the full language is allowed, including unsafe constructs. In the "safe" mode, the compiler will not allow any unsafe constructs. If the programmer uses an unsafe operation, the compiler will print an error message saying that an unsafe operation appeared in the package.

By default, the compiler is in "safe" mode. The command line flag "-unsafe" must be used when compiling a package that uses any unsafe constructs.

Linking and Running

Now that we have compiled all the necessary packages, we need to link them together into one executable file, which is called the "a.out" file.

The KPL system includes a collection of runtime support routines written in assembly language. All of this code is included in a single hand-code assembly language file called "Runtime.s". This file contains routines involved in program start-up and error handling, as well as some basic character I/O routines. The programmer should never modify the "Runtime.s" file. Normally, the runtime routines will be assembled only once, producing a file called "Runtime.o", with a command like this:

```
% asm Runtime.s
```

The next step is to combine all of the ".o" object files into an executable file. This step is called linking and is done with a program called "lddd". Here is the command line:

```
% lddd System.o Hello.o Runtime.o -o Hello
```

The "-o" option indicates that the new file is to be named "Hello"; without it, the file would be named "a.out".

Finally, we can run the program with the BLITZ virtual machine emulator. This tool is called "blitz" and here is the command line to invoke it on our executable, followed by the output. The "-g" option means to load the executable file into memory and begin executing it.

```
% blitz -g Hello
Beginning execution...
================  KPL PROGRAM STARTING  ================
Hello, world...

================  KPL PROGRAM TERMINATION  ================
```

Without "-g", the emulator will enter a command mode, where the user can do things like single-step the program, examine memory and registers, etc. The BLITZ virtual machine emulator is discussed in the document titled "The BLITZ Emulator".

After execution completes, the virtual machine emulator enters the command mode. Normally, the user would simply "quit" at this point.

```
> q
Number of Disk Reads     = 0
Number of Disk Writes    = 0
Instructions Executed    = 169917
Time Spent Sleeping      = 0
    Total Elapsed Time   = 169917
%
```

The BLITZ Machine

The BLITZ machine architecture was designed to closely follow contemporary RISC architectures. Although the architecture and instruction set are somewhat simpler than what is found in most processors, the architecture is intended to be complete and realistic enough for the implementation of a complete operating system.

The BLITZ virtual machine emulator completely and accurately models all aspects of a real BLITZ machine. In fact, the large number of instructions indicated in the example above is due primarily to delays associated with the serial character device I/O. Like a real computer, I/O devices like disks and character devices take time to transmit data and these delays are simulated.

The BLITZ machine has 32 general purpose registers, of 32 bits each, and 32 floating-point registers, of 64 bits each. At any time, the CPU is executing in one of two modes. It is either in system mode (sometimes called kernel mode) or user mode (sometimes called program mode). Certain privileged instructions can only be executed while in system mode. The CPU has page-table hardware, so that virtual memory can be implemented. The basic machine has two I/O devices: a disk and a serial character interface, such as that used to drive a terminal or modem. There are a number of hardware traps, interrupts, and exceptions that can occur during instruction execution. Examples include (1) page fault, (2) I/O completion, (3) privileged instruction violation, (4) alignment error, and (5) the system trap instruction. There is also a timer interrupt which makes it possible to implement time-sliced multitasking.

The BLITZ machine architecture is described in the document “The BLITZ Architecture”, which also gives the complete instruction set and describes the assembly language.

The Header and Code Files

A program is made of several packages and each package is described by a header file and a code file.

The header file is the specification for the package. It provides the external interface to that package, giving all information other packages will need about what is in the package. In the Hello-World example, the file “Hello.h” specifies the package will contain a function called “main” and tells what

parameters this function takes and returns. (The main function takes no parameters and returns no results.)

The code file contains the implementation details for the package. All executable code appears in the code file. In the Hello-World example, the "Hello.c" file contains the actual code for the main function.

When a package is compiled, its header and code file will be parsed and processed. Also, the header files for any packages that are used will be parsed and processed. This also includes packages that are used indirectly, as for example when package A uses package B, which uses package C in turn. However, only one code file—the code file for the package being compiled—is parsed and processed during a compilation. In fact, the code files for the "used" packages may not even have been created yet.

For example, assume that package Hello uses package System, as in the above example. When compiling Hello, the file "System.c" need not even exist. It can be created later and, as long as it implements the specification given in "System.h", it can be compiled and linked with Hello with no risk of error.

What if a header file is used in one compilation and then altered before being used within the compilation of another package? For example, what if we compile package Hello, then change "System.h" and compile the System package? To prevent the errors that such a sequence of events might cause, the runtime system uses a hash-based check at start-up time to ensure (with high probability) that the object files are all consistent.

In our example, we asked what happens when System.h is changed after Hello has been compiled. The resulting object files (System.o and Hello.o) can still be linked. It is possible that the linking will fail, but it may complete without error. For example, the link step would fail if the "print" function were eliminated altogether from the System package, but the link step would complete if the change only involved altering the number or types of parameters to the print function. However, when the user tries to execute the resulting executable file (the "a.out" file), the runtime system will detect the inconsistency during program start-up and initialization. It will print an error message, and terminate execution.

The Header File

The following things can go into a header file:

```
Constant Definitions
Global Variable Declarations
Type Definitions
Error Declarations
Enumerations
Function Prototypes
Class Specifications
Interfaces
```

Here is an example header file containing examples of all of these sorts of components. These constructs are described in detail in subsequent sections.

```
header MyPack
  uses System
  const
    pi = 3.1415
    MAX = 1000
  var
    x, y: int = -1
    perList: ptr to PERSON_LIST
  type
    PERSON_LIST = record
                    val: Person
                    next: ptr to PERSON_LIST
                  endRecord
  errors
    MY_ERROR (id: int)
    OTHER_ERROR (a,b,c: char)
  enum
    NO_ERR = 0, WARNING, NORM_ERR, FATAL_ERR
  functions
    foo (a1: int, a2: char) returns double
    bar (a1, a2: char)
    printErrMess (errCode: int)
  class Person
    superclass Object
    fields
      name: ptr to array of char
      id_num: int
      birthdate: int
    methods
      printID ()
      getAge () returns int
  endClass
  interface Ordered
    messages
      less (other: ptr to Ordered) returns bool
      greater (other: ptr to Ordered) returns bool
  endInterface
endHeader
```

The Code File

The following things can go into a code file:

```
Constant Definitions
Global Variable Declarations
Type Definitions
Error Declarations
Enumerations
Function Definitions
Class Specifications
Class Implementations
Interfaces
```

Any construct that may appear in a header file may also appear in a code file. In addition, the code file will contain function definitions and class implementations. All these things will be discussed later, but here is an example code file. (Some material is replaced with "..." to shorten this example.)

```
code MyPack
  var
    privateVar: ptr to PERSON_LIST
    privErr: int = NO_ERR
  function foo (a1: int, a2: char) returns double
      ...Variable declarations...
      ...Statements...
    endFunction
  function bar (a1, a2: char)
      ...Variable declarations...
      ...Statements...
    endFunction
  function printErrMess (errCode: int)
      ...Variable declarations...
      ...Statements...
    endFunction
 behavior Person
    method printID ()
       ...Variable declarations...
       ...Statements...
     endMethod
   method getAge () returns int
       ...Variable declarations...
       ...Statements...
     endMethod
  endBehavior
endCode
```

Within the header and code constructs, the various components may appear in any order; they need not be in the order shown here.

Syntax and Grammar

The full syntax of KPL is given in the document titled "Context-Free Grammar of KPL". In the present document, a few of the grammar rules are given informally, using ellipsis to suggest missing information. For details, you'll want to have the grammar document handy.

In many places, the grammar makes use of "end" keywords. In such cases, there are two matching keywords: the first serves to identify a syntactic construct and the second serves to terminate the construct.

For example, a class definition has the form:

```
class ...material describing the class... endClass
```

Here are some other examples.

```
header ... endHeader
record ... endRecord
interface ... endInterface
if ... endIf
while ... endWhile
```

Note that some keywords contain uppercase characters, which serve to make those keywords more readable. KPL is case sensitive, so this makes the language a little more difficult to type. However, it follows the general KPL philosophy that readability is more important than writability. The goal is a language whose programs are easier to read, comprehend, and debug.

Missing Semicolons

Many programming languages (like C++ and Java) use the semicolon as a statement terminator. However, in KPL, there is no statement terminator. The grammar has been designed carefully to avoid any ambiguities that might arise.

Normally, every statement would be placed on a different line, although this is not required. For example, the following two statements:

```
a = b + c
d = e * f
```

could be placed on the same line:

```
a = b + c    d = e * f
```

Although placing two statements on one line is not recommended, the compiler parses it the same as if they were on separate lines. The lack of statement terminators in the language is intended to make the resulting programs more readable by reducing typographic clutter.

Comments

KPL uses two styles of commenting. In the first style, everything after two hyphens through end-of-line is a comment.

```
x = y - 2   -- Adjust y a little
```

This is similar to the comment convention in C++ and Java, which use //, but the hyphen is used since it stands out more.

The second comment convention is /* through */ which is also used in C++ and Java.

```
x = y - 2   /* This comment can
                span multiple lines */
```

The second style of comments can be nested, unlike in C++ and Java. This makes it easy to disable a block of code which itself contains comments or disabled code.

```
/* Disable this code...
  x = a-2
  y = c*7  /* multiply by seven */
  z = b+5
*/
```

Lexical Tokens

The KPL grammar uses several types of tokens. Here are some examples of the different types of tokens. Lexically, KPL is quite similar to Java and C++.

	Examples
KEYWORD	**if, while, endWhile**
INTEGER	42, 0x1234abcd
DOUBLE	3.1415, 6.022e23
CHAR	'a', '\n'
STRING	"hello", "\t\n"
ID	x, my_var_name, yPos
OPERATOR	<=, >, +, -
MISC PUNCTUATION	(,), :, ., ,, ;, =

The If Statement

The **if** statement in KPL differs from Java or C++ in that it uses the "**if** ... **endIf**" syntax instead of braces for grouping. Here is an example.

```
KPL                     Java and C++
================        ====================
if x > y                if (x > y) {
  max = x                 max = x;
  min = y                 min = y;
else                    } else {
  max = y                 max = y;
  min = x                 min = x;
endIf                   }
```

In Java and C++, the conditional expression must be enclosed in parentheses, but in KPL the parentheses are not required. Of course, they may be included since all expressions may be enclosed in parentheses.

If there is only one statement in the "then" or "else" part, the braces can be omitted in Java or C++. However, in KPL, the **endIf** keyword is always used.

```
KPL                     Java and C++
================        ====================
if x > y                if (x > y)
  max = x                 max = x;
endIf
```

In Java and C++, braces are used to group multiple statements so they can be used in contexts requiring a single statement. Other languages use "begin...end". KPL is different; it has no such syntactic construct for grouping statements. Instead, any context where a statement may be used (such as the body of an **if** or **while**) may contain a sequence of zero or more statements. The proper grouping is always determined by the placement of keywords like **endIf**, **endWhile**, and so on.

KPL also has a single **elseIf** keyword, which can be used to make nested **if** statements more readable:

```
Nested if example
=================
if x == 1
  z = a
else
  if x == 2
    z = b
  else
    if x == 3
      z = c
    else
      if x == 4
        z = d
      else
        z = e
      endIf
    endIf
  endIf
endIf
```

```
Equivalent, using elseIf
========================
if x == 1
  z = a
elseIf x == 2
  z = b
elseIf x == 3
  z = c
elseIf x == 4
  z = d
else
  z = e
endIf
```

The While, Do-Until, For, and Switch Statements

The **while** statement in KPL looks similar to Java and C++. One difference is that KPL uses the **while** and **endWhile** keywords instead of braces to group the statements of the body. Also, the conditional expression does not have to have parentheses.

```
KPL
===============
while n > 0
  y = y*2
  ...
endWhile
```

```
Java and C++
===================
while (n > 0) {
  y = y*2;
  ...
}
```

KPL has a **do**-**until** statement, which is similar to the **do**-**while** statement in Java and C++. (The only difference is that the termination condition in KPL is reversed from Java/C++, and KPL uses the keyword **until** instead of **while**.)

```
KPL                          Java and C++
================             ====================
do                           do {
  n = n-1                      n = n-1;
  ...                          ...
until n <= 0                 } while (n > o);
```

In KPL, the **for** statement looks similar to Java and C++, except the **endFor** keyword is used instead of braces.

```
KPL                          Java and C++
================             ====================
for (n=1; n<MAX; n=n*2)      for (n=1; n<MAX; n=n*2) {
  ...                          ...
endFor                       }
```

There is a second form of the **for** statement which is given in the next example. We also give an equivalent in Java and C++.

```
KPL                          Java and C++
================             ====================
for i = 1 to 100 by 3        for (i=1; i<=100; i=i+3) {
  ...                          ...
endFor                       }
```

The general form is:

for LValue = Expr1 **to** Expr2 **by** Expr3 *...statements...* **endFor**

The "**by** *Expr3*" clause is optional; an increment of 1 is the default if it is missing. The loop always counts upward. In other words, the termination test is:

```
if LValue > Expr3 then terminate the loop
```

The *LValue* and the 3 expressions should be of type **int**. There is also a form where *LValue*, *Expr1*, and *Expr2* have type pointer.

This second form of the **for** loop is not really necessary since the programmer can always achieve the same effect by using a traditional, C-like version of the **for** statement. The primary reason for including the second form is that it makes some loops a little easier for beginning programmers to read and get right.

In KPL, the **break** and **continue** statements work the same as in Java and C++. They may be used in **while**, **for**, and **do**-**until** statements. For example:

```
KPL                          Java and C++
```

```
================          ====================
while n > 0               while (n > 0) {
  ...                       ...
  if ...                    if (...) {
    break                     break;
  endIf                     }
  ...                       ...
endWhile                  }
```

The **switch** statement looks similar to the **switch** statement in Java and C++.

```
KPL                       Java and C++
================          ====================
switch i                  switch (i) {
  case 2:                   case 2:
  case 4:                   case 4:
    ...statements...          ...statements...
    break                     break;
  case 1:                   case 1:
  case 3:                   case 3:
  case 5:                   case 5:
    ...statements...          ...statements...
    break                     break;
  default:                  default:
    ...statements...          ...statements...
endSwitch                 }
```

Just as in Java and C++, the **break** statement is used to jump to the end of the **switch** statement; execution will fall through to the next group of statements if there is no **break**.

Basic Data Types

KPL has the following basic types of data.

```
Type                      Example Values
================          ======================
int                       123, -57, 0xabcd4321
double                    3.1415, -5.2e10
char                      'a', '\n'
bool                      true, false
```

Values of type **int** are always represented as 32-bit signed values, stored in two's complement. Values of type **double** are always stored using the IEEE 64-bit floating-point standard.

Even though KPL was designed for one particular CPU architecture, the language makes it clear exactly how **int** and **double** values will be represented so that each program will execute predictably and identically, regardless of which machine it runs on.

KPL uses the ASCII system, and the usual back-slash escapes may be used in character and string constants.

There are two values of type **bool**, represented by the keywords **true** and **false**.

C++ traces its roots back to C, in which **int**s were used for Boolean values. KPL is a little more particular about conditional expressions than C and C++. In KPL, there is no implicit coercion from **int**s to **bool**s; the programmer must make the test explicit.

```
KPL                         C++
================            ========================
if i != 0 ...               if (i) ...
```

Variable Declarations

Variables are declared using a syntax that is shown in the next example:

```
KPL                         Java and C++
================            ========================
var x: int                  int x;
```

Several variables can be declared at once, however the **var** keyword must appear only once, as shown next. Also, variables may be given initial values, if desired.

```
KPL                         Java and C++
================            ========================
var
  x,y,z: char               char x,y,z;
  a,b: double = 1.5         double a = 1.5, b = 1.5;
  i,j: int = f(a)           int i = f(a), j = i;
```

Any variable that is not explicitly initialized will be set to binary zeros. Thus, it is more difficult in KPL than in C++ to pick up random data values from uninitialized memory. Here are the zero values for the basic types:

Type	Default Initial Value
int	0
double	+0.0
char	'\0'
bool	**false**
ptr to ...	**null**

Records and objects will have their fields initialized to their zero values. Arrays are initialized by default to have size zero, which will trigger an error if an attempt is made to access an element.

Global Variables

Every variable is either a local variable or a global variable. Local variables are declared at the beginning of functions and methods. Local variables only exist while the function or method is being executed.

Any variable that is declared outside of a function or method is called a "global variable". Each global variable is placed in a fixed, unchanging memory location. Consequently, each global variable exists throughout the execution of the program.

Global variables may be declared either in a header file or in a code file. Where it is declared determines the visibility of the global variable.

Global variables declared in a header file can be accessed from anywhere in that package and anywhere in any package that uses the package containing the declaration. However, variables declared in a code file are accessible only from the code portion of the package containing the declaration. Thus, there is a facility for information hiding. A global variable is either shared with other packages (by placing its declaration in the header file) or the variable is private and local to a single package (by declaring it in the code file).

All variables—local and global—will be initialized. If an initializing expression is provided, it is used; if not, the variable will be initialized to its zero value.

Constructed Data Types

KPL has the following complex data types:

Type	Examples
Arrays	`array [10] of double`
Pointers	`ptr to array [10] of double`

Records	`record`
	`  val: double`
	`  next: ptr to MY_REC`
	`endRecord`

Functions	`function (a,b: int) returns bool`

Classes	`class Person`
	`  ...`
	`endClass`

Interfaces	`interface Taxable`
	`  ...`
	`endInterface`

These will be discussed in subsequent sections of this document.

In addition, there are three somewhat unusual types, which are not used as frequently as other types:

```
anyType
typeOfNull
void
```

The type **anyType** subsumes all other types. It can only be used in certain contexts. For example, we cannot have a variable with type **anyType**, since the compiler cannot know how many bytes will be needed to store the value. But we might have the following type:

```
ptr to anyType
```

The type **typeOfNull** would not normally be used by the programmer. This type has only one value, the null pointer, which is represented with the keyword **null**. The **null** value is a pointer whose value is zero.

The type **void** is used in only in conjunction with pointers, as in

```
ptr to void
```

Normally, all pointer types are type-checked. The use of **void** effectively turns off type checking for pointers. A value of type **ptr to void** can be assigned to/from any other pointer type. The use of type **ptr to void** is “unsafe” in the sense discussed earlier in this document.

Type Definitions

KPL has a "type definition" construct. Here is an example:

```
type MY_REC = record
                val: double
                next: ptr to MY_REC
              endRecord
```

Such a definition then allows "MY_REC" to be used instead of having to re-type the full type everywhere it is needed.

The general form is

```
type ID = ...type...
     ID = ...type...
     ...
     ID = ...type...
```

Here is an example showing that several type definitions can follow the **type** keyword.

```
type
  MY_PTR = ptr to PERSON_LIST              -- Used here
  MY_ARRAY = array [100] of PERSON_LIST    --   ...and here
  PERSON_LIST = record ... endRecord       -- But, defined here
```

Notice that the type PERSON_LIST is used before it is defined. This is okay and this occurs in other places as well. For example, a class may be used at one point in a source code file and defined at a later point in that file.

Arrays

Array types have the following general form:

```
array [ ...SizeExpr... ] of ...type...
```

Here is an example variable declaration using an array type:

```
var a: array [100] of double
```

The *SizeExpr* gives the number of elements in the array and must be statically computable so that the compiler can determine how many bytes to allocate for this variable.

The numbering of the array elements begins at zero, so this array has elements

```
a[0], a[1], ... a[99]
```

Array elements can be accessed (i.e., read and updated) using the normal bracket notation. For example:

```
a[i] = x
y = a[foo(j)+k]
```

In KPL, arrays always carry their sizes along with them. Every attempt to access an array element will be checked at runtime to ensure the index expression is within the bounds of the array. If an attempt is made to access an array element that is "out of bounds", a runtime error will halt execution at that moment. Therefore, the notorious "buffer overrun" errors from C/C++ cannot occur in KPL when arrays are used.

There are two ways to create an array: the **new** expression and the **alloc** expression. Both have a similar syntax differing only in the keyword used, although each has a very different meaning.

```
new   ...ArrayType... { ...Initialization... }
alloc ...ArrayType... { ...Initialization... }
```

(Here the braces are used directly, and do not indicate multiple occurrences.)

The **new** expression creates a new array value and returns it. For example, the following will initialize the variable "a" by setting all its elements to the value –1.23.

```
a = new array of double { 100 of -1.23 }
```

A **new** expression is an R-Value, not an L-Value. In this way it is similar to an **int** constant. For example, you cannot ask for its address, although you could ask for the address of variable "a".

Note that the *SizeExpr* in the *ArrayType* after the **new** keyword is normally left out, since it is redundant.

```
a = new array of double { 100 of -1.23 }
a = new array [100] of double { 100 of -1.23 }    -- Equivalent
```

The **alloc** expression allocates memory on the heap, initializes the array, and returns a pointer to it. Here is an example of the **alloc** expression:

```
var p: ptr to array [5] of int
...
p = alloc array of int { 0, 11, 22, 33, 44 }
```

Elements of an array allocated on the heap may be accessed using the bracket notation. Whenever brackets are applied to a pointer to an array—instead of to an array directly—a pointer dereferencing operation will be automatically inserted by the compiler.

```
p[i] = ...          -- p is  a pointer
... = p[j]
```

Every element of a newly created array (whether created in a **new** expression or an **alloc** expression) must be given an initial value. The values are listed in order between the braces. A single value may be copied many times using the syntax

CountExpression **of** *ValueExpression*

For example "100 **of** –1.234" will initialize 100 elements to the same floating point value. Both the *CountExpression* and the *ValueExpression* may be complex expressions, evaluated at runtime. Here are more examples of array creation and initialization:

```
arr1 = alloc array [13] of double { 1.1, 2.2, 10 of 3.3, 4.4 }
arr2 = alloc array [n+m] of double { n of 0.0, m of 9.999 }
arr3 = alloc array [f(k)] of double { f(k) of g(x)*0.5 }
```

Often the programmer will work with "dynamic arrays", whose size is not known at compile time. In such cases, pointers to arrays must be used and the array must be placed on the heap.

In a dynamic array type, the *SizeExpr*, along with the brackets, is left out:

array of *...type...*

Here is an example:

```
var dynArr: ptr to array of double
...
dynArr = alloc array of double { n+m of 0.0 }
...
dynArr [i] = dynArr[j]
```

The array size will be inferred from the number of initial values in the initialization part. When the array is created at runtime, the expression "n+m" will be evaluated to determine the amount of memory to be allocated.

To determine the size of a dynamic array at runtime, the programmer can use the built-in postfix operator **arraySize**. This expression returns the number of elements in the array.

```
i = dynArr arraySize    -- In this example, returns n+m
```

In the previous examples, the type of the array elements has been a simple type like **int** or **double**, but the element type can be any type, even another array. Here is an example of a 2 dimensional array, which is nothing more than an array of arrays:

```
var arr_2D: array [100] of array [500] of double
```

There is a "syntactic shorthand" for specifying arrays of several dimensions. For example, the previous example could also be written as follows, with no change in meaning. The compiler will simply expand the following code into the code shown directly above.

```
var arr_2D: array [100,500] of double
```

For array types with more than one dimension, the programmer must specify all sizes, except possibly the first. In other words, only the first dimension may be dynamic. The remaining dimensions must be statically known so that compiler can create the proper address calculations.

The grammar allows the asterisk to be used for array types of higher dimension, when the first dimension is dynamic. The asterisk may only appear in the first dimension. For example:

```
var arr_3D: ptr to array [*, 5, 25] of double
```

Here is some code to initialize the array:

```
arr_3D = alloc array [*,5,25] of double
             { 100 of new array [5,25] of double
                      { 5 of new array of double
                               { 25 of -9.999 } } }
```

To access elements in a multi-dimensional array, several index expressions must be provided, separated by commas. For example:

```
d = arr_3D [a,b+3,c]
```

Array accessing, as illustrated above, is also nothing more than a syntactic shorthand for a more complex expression. For example, the following expressions are completely synonymous:

```
arr_3D   [a,   b+3,   c]
arr_3D   [a]  [b+3]  [c]
((arr_3D [a]) [b+3]) [c]
```

In the case of arrays with dimension greater than 1, the **arraySize** expression returns the size of the first dimension only.

```
i = arr_3D arraySize
```

Since each element in a 3 dimensional array is itself a 2 dimensional array, we could always write something like:

```
j = arr_3D[0] arraySize      -- Sets j to 5
k = arr_3D[0,0] arraySize    -- Sets k to 25
```

Each singly-dimensioned array is stored along with a 4-byte integer giving the number of elements. This count precedes the first element. For example, the array

```
var a: array [100] of double
```

would require 4 + 100*8 bytes of storage (i.e., 804 bytes), since each double value requires 8 bytes.

When asking for the address of arrays and array elements, note that the address of the array is always 4 bytes less than the address of the first element, since the array size is stored directly before the first element.

```
&a[0] == &a + 4
```

Strings

Strings are represented as pointers to arrays of characters. For example, the following is type-correct:

```
var str: ptr to array of char
...
str = "hello"
```

We can access the elements of the string, just as we access any array:

```
var ch: char
...
ch = str[1]    -- sets ch to 'e'
str[3] = 'k'   -- now str points to "helko"
```

Note that this differs substantially from how strings are dealt with in C/C++. In KPL, there is not necessarily a terminating ASCII “null” character, although it is certainly possibly to add one:

```
str = "hello\0"
```

The “System” package includes the following routines to print data:

```
print (s: ptr to array of char)
printInt (i: int)
printHex (i: int)          -- prints, e.g., 0x0012ABCD
printChar (c: char)        -- prints non-printables as, e.g., \x05
printBool (b: bool)        -- prints "TRUE" or "FALSE"
printDouble (d: double)
nl ()                      -- Short for printChar ('\n')
```

Here is an example:

```
print ("The value of 'i' is ")
printInt (i)
nl ()
```

Each new string constant appearing in a program will cause a new array to be created. For example, the following code will create two arrays, even though the string constants contain the same characters:

```
str1 = "hello"
str2 = "hello"
```

All string arrays will be placed in the ".data" segment and may therefore be updated.

Records

Here is an example using record types:

```
type MY_REC = record
                val: double
                next: ptr to MY_REC
              endRecord
var r: MY_REC
```

The **new** expression can be used to create a record value. Each field of the record must be initialized within the braces.

```
r = new MY_REC { val=1.5, next=null }
```

To access a field in the record, the infix-dot operator is used:

```
x = r.val
r.val = 2.56
```

The **alloc** expression can be used to allocate a record and place it in the heap:

```
var recPtr: ptr to MY_REC
...
recPtr = alloc MY_REC { val=1.5, next=null }
```

KPL guarantees to the programmer exactly how all data values (including records, arrays, and objects) are represented in memory. In the case of records and objects, the fields are placed in memory sequentially in order, with extra padding bytes inserted where necessary to ensure proper alignment. Every data value will be word aligned, except **char** and **bool** values, which are each stored in a byte.

Pointers

KPL is similar to C++ in that all pointers are explicit and the programmer can choose whether to work with data or with pointers to data. In Java, the pointers are all implicit and the programmer has less control over representation.

Consider these two variables:

```
var r: MY_REC
    p: ptr to MY_REC
```

The variable "r" will require 12 bytes (the size of a MY_REC record) while "p" will require only 4 bytes since all pointer values are 4 bytes.

To get the address of a variable, use the & operator, which is also used in C and C++:

```
p = &r
```

To refer to the data that the pointer points to, the prefix operator * is used, just as in C and C++:

```
r = *p
```

Records, arrays, and objects may be created using either the **new** expression or the **alloc** expression.

The syntax of the **alloc** construct and the **new** construct is the same. The **new** construct creates a new value which must be used (e.g., copied into a variable) while the **alloc** construct allocates memory in the heap, initializes it, and returns a pointer to the allocated memory.

For example, the following **alloc** expression will create a new record on the heap:

```
p = alloc MY_REC { val=1.5, next=null }
```

To initialize variable "r", the **new** expression would be used:

```
r = new MY_REC { val=1.5, next=null }
```

The syntax to chase pointers is the same as in C or C++, so we can access the fields of the record pointed to by "p" with statements like these:

```
x = (*p).val
(*p).val = 2.5
```

C++ uses a shorthand of "->" to make dereferencing clearer. KPL uses the infix-dot operator. The semantics of the dot operator is "If the left operand is a pointer, dereference it first".

```
KPL                     Equivalent in C++
===============         ==================
x = p.val               x = p->val;
p.val = 2.5             p->val = 2.5;
```

The above discussion of pointers used records and pointers to records, but pointers to objects work the same way, as illustrated below.

In the next example, assume there is a class called “Person”. This example shows that KPL code can look a lot like Java code. (Assume that “name” is a field in class Person and that “computeAge” is a method from the class.)

```
KPL                     Java
===============         ==================
var p: ptr to Person    Person p;
p = new Person {...}    p = new Person (...);
p.name = ...            p.name = ...;
p.computeAge (...)      p.computeAge (...);
```

The programmer can copy entire records, objects, or arrays with code that looks like C++:

```
*p = r
r = *p
```

The null pointer is symbolized with the keyword **null**. The **null** value is represented with the value 0x00000000.

```
if p != null ...
```

Pointer expressions will automatically be coerced to **bool** values if necessary, so the above test could also be coded in a way that looks like C++.

```
if p ...
```

When coerced to **bool** values, non-**null** pointers are treated as **true** and **null** pointers are treated as **false**.

Whenever a pointer is dereferenced, a runtime check will make sure the pointer is non-null. For example, this code

```
p = null
...
p.name = ...      -- Error here
```

will result in the runtime error

```
Attempt to use a null pointer!
```

When a runtime error like this occurs, the BLITZ virtual machine will halt emulation and go into command mode. The user can then type the "st" command to see the activation stack. The top line shows where in the source code the program was executing when the error occurred.

```
Enter a command at the prompt.  Type 'quit' to exit or 'help' for
info about commands.
> st
   Function/Method        Frame Addr   Execution at...
   ====================   ==========   ====================
   foo4                    00FFFD80    MyPack.c, line 75
   foo3                    00FFFD90    MyPack.c, line 71
   foo2                    00FFFDA0    MyPack.c, line 67
   foo1                    00FFFDB0    MyPack.c, line 63
   main                    00FFFEF8    MyPack.c, line 89
```

The programmer may also work with pointers to other sorts of data, as in:

```
var p1: ptr to int
    p2: ptr to char
```

Pointers can be converted to integers and vice versa, using the **asInteger** and **asPtrTo** constructs. The **asInteger** operator uses a postfix syntax and **asPtrTo** uses infix, where the second operand is a type.

```
i = p1 asInteger
p2 = i asPtrTo char
```

Pointers can also be incremented and decremented directly, as in these examples:

```
p1 = p1 + 4
p2 = p2 - 1
```

The increment or decrement is always in terms of bytes. This is a subtle difference with C++. In C++, the expression "p+1" may increment the pointer by an amount that is different than 1 byte, and which is in fact implementation dependent.

KPL provides the **sizeOf** operator to determine the size of a type of value. This is a prefix operator, applied to a type. Since each class is a type, the **sizeOf** operator can be applied to a class, as in...

```
p1 = p1 + sizeOf Person
```

The use of some of the pointer operations is unsafe. In particular, **asPtrTo**, pointer increment, and pointer decrement are all unsafe, while the normal operations of copying, dereferencing, and comparing for equality are safe. It is safe to take the address of a global variable but unsafe to ask for the address of a parameter or local variable.

Functions

The general form of a function is

```
function ID ( ...Parameters... ) [ returns ...Type... ]
    ...Variable declarations...
    ...Statements...
  endFunction
```

Here is an example function:

```
function foo (a, b: int) returns bool
    var c: int
    c = a + b
    return c > 10
  endFunction
```

The **returns** clause is optional. Here is another example in which there are no parameters or return value:

```
function foo2 ()
    print ("hello")
  endFunction
```

Functions may be invoked (i.e., called) using syntax like Java or C++, except the semicolon is not used.

```
myBoolVar = foo (x, y)
foo2 ()
```

All functions must appear in the code portion of a package, since they constitute implementation.

Within the header file, the programmer may optionally include a function declaration (i.e., a function prototype). Here is an example:

```
header MyPack
  ...
  functions
    foo (a, b: int) returns bool
    foo2 ()
    foo3 (a: int, b: char, c,d: double)
  ...
endHeader
```

For every function that is declared in the header file, there must be a matching function definition in the code file. However, every function in the code file need not have a matching declaration in the header file. Whether or not a function has a declaration in the header file determines its visibility.

For example, if a function is declared in the header file of package MyPack, then that function can be called from code in MyPack and from code in any other package that uses MyPack. If a function does not have a declaration in the header file, then that function is private. It may only be called from the code file of MyPack.

KPL also provides a way to invoke functions written in assembly language. Consider the function named "Switch" which is coded in assembly language. This function must be assembled separately and included in the link phase. Within the header file of MyPack, the function "Switch" must be declared using the **external** keyword.

```
header MyPack
  ...
  functions
    external Switch (x, y: ptr to ThreadControlBlock)
    foo (a, b: int) returns bool
    ...
```

Note that the function name "Switch" is capitalized to distinguish it from the keyword **switch**.

KPL does not allow overloading of function names. All functions must have distinct names.

Classes

Classes in KPL are similar to the classes in C++ and Java. Each object is an instance of a class. The class describes which fields the instances will have and provides methods for operating on the instances of the class.

KPL also has interfaces, which are similar to the interfaces of Java. Interfaces are discussed later.

A class is defined in two parts or pieces, called the *specification* and the *implementation*.

The first part specifies which fields will be in the class and which methods are in the class, but does not supply the actual code for the methods. The keyword **class** is used for the specification part.

The second part, which gives the implementation of the class, includes only the code bodies for the methods. The keyword **behavior** is used for the implementation part.

As an example, consider a class called "Person". Here is the specification part for Person:

```
class Person
  superclass Object
  fields
    name: ptr to array of char
    id_num: int
    birthdate: int
  methods
    printID ()
    getAge () returns int
endClass
```

Here is the implementation part of Person:

```
behavior Person
  method printID ()
      ...Variable declarations...
      ...Statements...
    endMethod
  method getAge () returns int
      ...Variable declarations...
      ...Statements...
    endMethod
endBehavior
```

The specification part of a class may be placed in either the header file or the code file. Where the specification is placed determines the visibility of the class. If placed in the header file, then the class may be used by any packages that use this package. If placed in the code file, then the class may only be used within that package.

The behavior part of a class must always be placed in the code file.

The general syntax for the specification part of a class is given next. The notation [...] means “optional”. The notation {...}+ means “one or more occurrences”. (The actual grammar rule is simplified a little here.)

```
class ID [ ...Type Parameters... ]
  [ implements ID, ID, ... ]
  superclass ID
  [ fields { FieldDeclaration }+ ]
  [ methods { MethodPrototype }+ ]
endClass
```

Type parameters are discussed in a later section.

Here is the syntax for the implementation part.

```
behavior ID
  { Method }
endBehavior
```

Each class has exactly one superclass, which is given following the **superclass** keyword. The root superclass is called "Object". It is included in the "System" package and does not have a superclass.

A class may implement zero or more interfaces. These would be given following the **implements** keyword. For example:

```
class MyClass
  superclass Object
  implements InterA, InterB, InterC
  fields
  ...
```

The class specification lists the methods that are implemented in the class. However, it only includes a prototype for each method. A method prototype includes the method name, parameters, and return type, but does not include the code for the method.

There must be a one-to-one correspondence between the methods listed after the **methods** keyword in the specification and the methods provided in the implementation part. In other words, if the specification says there is a method called "printID" in the class, then an implementation of method "printID" (with matching parameters and return type) must appear in the **behavior** construct for the class.

A class will inherit any and all methods from its superclass, and its super-superclass, and so on up to "Object". When a method is included in a class, it must be given a new name or else it will override a method inherited from the superclass.

KPL does not allow overloading of method names. Two methods may only have the same name if they are in different classes.

In Java and C++, methods and fields have visibility control (private, public, etc.). KPL does not have this layer of complexity. Every field and every method can be used wherever the class itself can be used. In Java and C++, the visibility mechanism is used to restrict and constrain what the programmer can do with objects; in KPL it is up to the programmer to use objects correctly. For example, if the programmer feels that some field should be accessed only from code within the class, then it is the programmer's responsibility to discipline him or herself and avoid accessing the field from outside the class.

If the programmer really needs a mechanism to disallow some parts of the program from accessing certain methods or fields, there are several ways of doing it. First, the class specification can be placed in the code file, making the entire class private to a package. Second, the programmer can work with interfaces, which can be used to control which methods may be invoked on an object. Finally, the

programmer can create a new wrapper class to allow only certain kinds of access to an underlying object.

Fields

Classes may contain fields. For example:

```
class MyClass
  ...
  fields
    myField: int
  ...
endClass
```

From outside the class, the fields may be accessed using the dot notation:

```
var m: MyClass
...
i = m.myField
...
m.myField = j
```

The field would also be accessed the same way if a pointer to the object is used, instead of the object itself.

```
var p: ptr to MyClass
...
i = p.myField
...
p.myField = j
```

From within the class, the fields of the class may be accessed directly. For example:

```
behavior MyClass
  ...
  method foo (...) returns ...
      ...
      i = myField
      ...
      myField = j
      ...
    endMethod
  ...
endBehavior
```

A class will inherit any and all fields from its superclass, and its super-superclass, and so on up to "Object". Fields may be added to a class, but overriding or overloading of fields is not allowed. When a new field is added to a class, its name must be distinct from the names of all inherited fields.

Java and C++ allow "static" fields, but KPL does not have static fields. A static field is nothing more than a global variable whose visibility is limited. KPL provides only one concept: the global variable. If a programmer wants to have a static field called "x" in some class "MyClass", he or she can simply create a global variable and give it a name that suggests its use.

```
var MyClass_x: ...type...
```

Then the programmer can simply write "MyClass_x" instead of "MyClass::x".

Methods

A method is declared in the class's specification part and implemented in the class's behavior part.

```
class MyClass
  ...
  methods
    ...
    foo (a,b: int) returns int
    ...
endClass

behavior MyClass
  ...
  method foo (a,b: int) returns int
      var x: int
      x = a + b
      return x
    endMethod
  ...
endBehavior
```

The general form of a method is

```
method ID ( ...Parameters... ) [ returns ...Type... ]
    ...Variable declarations...
    ...Statements...
  endMethod
```

If there are no parameters and the method does not return a value, the method looks like this:

```
method bar ()
    ...
  endMethod
```

If the method returns a value, then the method should contain at least one **return** statement with a value. If the method does not return a value, then the method may contain a **return** statement without a value or may fall out the bottom. This routine does both:

```
method printWithNULL (p: ptr to array of char)
    if p == null
      print ("NULL!")
      return
    endIf
    print (p)
  endMethod
```

Within a method, the keyword **self** may be used to refer to the receiver object. The type of **self** is

```
ptr to CLASS
```

where "CLASS" is the class containing the method.

In the next example, "recur" is a recursive method.

```
method recur (...)
    ...
      self.recur (...)
    ...
  endMethod
```

The keyword **self** may also be used in other ways. For example:

```
m.foo (j, self, k)
...
p = self
```

There is also a keyword **super**, which has exactly the same type as **self**. However, **super** can only be used in one way; it is used within a method to invoke the inherited and overridden version of the method.

For example, assume class "Person" has a method called "meth".

```
class Person
  ...
  methods
    meth (...)
  ...
endClass
```

Now assume that a subclass called "Student" overrides "meth".

```
class Student
  superclass Person
  ...
  methods
    meth (...)
  ...
endClass
```

Within the implementation of "meth" in Student, the overridden method can be invoked by using **super** instead of **self**:

```
behavior Student
  ...
  method meth (...)
      ...
      super.meth (...)
      ...
    endMethod
  ...
endBehavior
```

Creating Objects

Instances of classes can be created in one of two ways. The object can be allocated in the runtime heap or the object can be placed directly into a variable.

To allocate and initialize an object on the heap, the **alloc** expression is used. For example:

```
var perPtr: ptr to Person
...
perPtr = alloc Person { name = "Smith",
                        id_num = nextNum+1,
                        birthdate = 2003 }
```

Here is an example of creating an object with the **new** keyword and storing the result directly into a variable.

```
var per: Person
...
per = new Person { name = "Smith",
                   id_num = nextNum+1,
                   birthdate = 2003 }
```

The **alloc** expression returns a pointer while the **new** expression returns a object.

These examples assume that the Person class has three fields, called “name”, “id_num”, and “birthdate”. After the object is created, the fields are given their initial values, which are listed between the braces.

Whenever an object is created, the programmer has two choices: either all the fields can be initialized explicitly or they can all be set to their zero values by default.

In the above examples, the fields were initialized. The fields and their initializing expressions are listed between the braces. The fields need not be listed in order, but all fields must be listed, including any inherited fields.

If the programmer doesn't want to initialize the fields, then the braces and everything between them should be omitted. The object will be created and each of the fields will be initialized to its zero value. For example:

```
perPtr = alloc Person
```

It is often the case that creating and initializing an object should be accompanied by some additional computation. Java and C++ have "constructor" methods for this, but KPL does not have any special syntax for constructors.

Instead, the KPL convention is to create a method—typically called "init"—to perform all necessary initialization and computation associated with object creation. Thus, the same effect as constructors is achieved with features already in the language.

Here is an example using an "init" function. To create an object, the **alloc** or **new** expression is used with no explicit field initialization. Then the init method is immediately invoked. The init method can be designed to take however many arguments make sense in the application. In our example, we will pass one argument and initialize the remaining fields with computed or default values.

```
perPtr = alloc Person.init("Smith")
```

Here is a possible definition of the init method:

```
class Person
  ...
  methods
    init (n: ptr to array of char) returns ptr to Person
    ...
endClass

behavior Person
  ...
  method init (n: ptr to array of char) returns ptr to Person
      name = n
      last = last + 1
      id_num = last
      -- birthdate defaults to zero
      return self
  endMethod
  ...
endBehavior
```

In the above example, the "init" method returned a pointer to the object; this allows us to invoke it in the same statement that creates the object:

```
perPtr = alloc Person.init(...)
```

However, if the object is not allocated on the heap but is created with a **new** expression, we would have to invoke "init" in a separate statement. Since KPL requires the programmer not to ignore a returned value, we must create a dummy variable to absorb the value:

```
var ignore: ptr to Person

per = new Person
ignore = per.init(...)
```

Another approach is to design "init" not to return anything. Here are examples showing how we would invoke the "init" method in such a design:

```
perPtr = alloc Person
perPtr.init(...)
per = new Person
per.init(...)
```

Interfaces

KPL has interfaces, which are similar to the interfaces of Java. Here is an example:

```
interface MyInter
  messages
    foo1 (a,b: int) returns int
    foo2 (d: double)
endInterface
```

Any object that implements the interface "MyInter" must provide at least these two methods, although the class may have other methods as well. Furthermore, these two methods must have types on their parameters and return value that match the specification in the interface.

Just as in Java, an interface can extend zero or more other interfaces. Thus, there is multiple inheritance in the interface hierarchy.

The general syntax of an interface is:

```
interface ID [ ...Type Parameters... ]
  [ extends ID, ID, ... ]
  [ messages { MethodPrototype }+ ]
endInterface
```

Type parameters are discussed in a later section.

Note that the keyword here is **messages**, not **methods**. Methods are chunks of behavior and therefore occur in classes; messages describe the protocol for interacting with objects and are therefore specified in interfaces. Messages are implemented by methods.

A class may implement zero or more interfaces, and this is given in the class specification. For example:

```
class ExampleClass
  implements MyInter, AnotherInterface
  superclass ...
  fields
    ...
  methods
    ...
endClass
```

The **implements** clause is optional and may list one or more interfaces. Of course a class will necessarily implement all the interfaces its superclass implements.

Here is a variable declaration which uses an interface instead of a class.

```
var p: ptr to MyInter
```

The constraint on "p" is that it must point to an object which has (at least) two methods called "foo1" and "foo2", with appropriate parameter typings. Thus, the programmer may invoke those methods on p:

```
i = p.foo1 (...)
```

One class that implements this interface is "ExampleClass", but there may be other completely unrelated classes that also implement this interface. The compiler guarantees that, at runtime, p will point to an instance of ExampleClass or some other class that implements this interface.

Note that the programmer cannot create a variable of type "MyInter" since the compiler has no way to know how much space to allocate for the variable. The programmer must use a pointer instead.

Infix and Prefix Methods

The traditional method syntax involves parentheses with comma-separated arguments.

```
w = x.foo(y,z)
```

KPL also allows methods to use a "binary operator syntax". Here is the invocation of a method named "**" on receiver "x". There is one argument, indicated by "y".

```
w = x ** y
```

While this looks different than the invocation of "foo", it is essentially the same. The method "**" is invoked on the object "x" and a result is returned.

There is also a "unary operator syntax". In the next example, the prefix method "~" is invoked on the object named "x". Here, a method with no arguments is invoked on object "x".

```
w = ~x
```

In the binary operator syntax, there is always one argument, while in the unary operator syntax, there is no argument. In both cases, a result is always returned.

For each operator, the class must contain a corresponding method. For the above examples, let's assume that "x" has type "MyClass"; then the definition of "MyClass" will need to contain methods for "foo", "**", and "!" as in:

```
class MyClass
  ...
  methods
    foo (p1, p2: MyClass) returns MyClass
    infix ** (p1: MyClass) returns MyClass
    prefix ~ () returns MyClass
    ...
endClass
```

The types of the arguments and returned values in this example all happen to be the same "MyClass", but in other programs, they could be any type.

The name of a binary or unary method may be any sequence of the following characters:

```
+  -  *  /  \  !  @  #  $  %  ^  &  ~  `  |  ?  <  >  =
```

with the exception that the following tokens may not be used as operators:

```
/*   */   --   =
```

Keyword Methods

In addition to the normal method syntax and the infix and prefix operator syntax, KPL has another method syntax which is unlike anything in Java or C++. It is called "keyword syntax" and it was introduced in the Smalltalk language.

With keyword methods, the name of the method contains colons. Consider the method named

```
at:put:
```

For each colon, there is a single argument. Therefore, we can tell that "at:put:" takes two arguments. Here is an example where this method is invoked on receiver "x" with arguments "y" and "z".

```
x at: y put: z
```

Keyword methods may or may not return a result. They may be used in expressions and mixed with the other methods forms, as shown in the next example:

```
myTable at: (myTable lookup: (x ** y)) put: ~z
```

For each keyword method, the class must contain a corresponding method. Here is a class with methods for "lookup:" and "at:put:".

```
class MyClass
  ...
  methods
    lookup: (p: MyClass) returns MyClass
    at: (p1: MyClass) put: (p2: MyClass)
    ...
endClass
```

One advantage of keyword syntax over the traditional syntax is that it can be employed to identify arguments. As an example, contrast the following two method invocations. Both methods are intended to do the same thing; the only difference is that in one case the programmer has chosen to use the traditional syntax while, in the other, the keyword syntax has been used and the method renamed accordingly.

```
a.compile (b, c, d, e)
a compile: b withEnvironment: c outputTo: d optimizations: e
```

In the first, no clue is given about the identity and meaning of the arguments, but in the second, the reader can make some guesses about the meanings of the arguments. This sort of intuitive help can make some programs vastly easier to read and understand.

Keyword syntax may seem rather strange at first, but the experience of Smalltalk shows that it works well in practice. It can be learned easily and is quickly accepted by novice programmers. KPL provides the keyword syntax, but if desired, the programmer can simply ignore it and continue to program in the Java / C++ style.

Pointers to Functions

Functions, which were discussed earlier, are defined with a syntax as suggested by this example:

```
function sqrt (a: double) returns double
    ...Variable declarations...
    ...Statements...
  endFunction
```

and invoked with syntax like this:

```
x = sqrt(y)
```

The function definition defines a name, such as "sqrt", and a function invocation uses the name. If the function returns a value, as does sqrt, then the invocation must appear in an expression and the value must be consumed. If the function does not return a value, the invocation occurs in a call statement.

In KPL, pointers to code may be stored in variables, by using function types. In the next example, a variable "f" is defined. This variable will contain a pointer to a function.

```
var f: ptr to function (double) returns double
```

Function types may only be used in conjunction with **ptr to**. The syntax of a function type is:

```
ptr to function (Type, Type, ... Type) [ returns Type ]
```

Here are some example function types:

```
ptr to function (int, int, int) returns Person
ptr to function (double, int, char)
ptr to function () returns ptr to Person
```

A function pointer may be assigned and compared like other pointers. In the following assignment statement, the variable f is set to point to the sqrt function.

```
f = sqrt
```

The compiler will check to make sure the type of the sqrt function is compatible with the type of variable f. In particular, the compiler will ensure the number of arguments is the same, the types of the arguments are pair-wise equal and that, if there is a return value, both sqrt and f have the same return type. In other words, two function types are incompatible if they differ in any way.

To invoke a function using a function pointer, the same syntax as a normal function invocation is used. For example, we can write:

```
x = f(y)
```

A variable such as "f" requires only 4 bytes and is represented as a pointer to the machine instructions for the function. And like other pointers, it may be null if it has not been set to point to any function. If an attempt is made to invoke a function using a **null** function pointer, the error will be caught immediately and the runtime system will print an error message.

Pointers to functions may be copied, stored, passed as arguments to methods and other functions, and used like any other value. For example, we might wish to store a number of different function pointers in an array. Here is the definition of an array called "a":

```
type MY_FUN = ptr to function (double) returns double
var a: array [10] of MY_FUN
```

We will need to initialize this array:

```
a = new array of MY_FUN { 10 of null }
```

Then we can store pointers to various functions in the array:

```
a[4] = sqrt
a[7] = cos
...
```

The syntax for invoking functions is

```
ID ( arg, arg, ... arg )
```

so to invoke one of the functions in "a" we cannot write:

```
x = a[i] (y)
```

Instead, the code would look like this:

```
f = a[i]
x = f(y)
```

When a name such as "sqrt" appears in the program, how does the compiler determine whether it signifies a pointer or a function invocation?

In the assignment statement

```
f = sqrt
```

the compiler will notice that there are no arguments; it therefore assumes "sqrt" means a pointer to code and it will copy a pointer. Contrast this to a function invocation, such as is shown next, in which arguments are present.

```
x = sqrt (2.25)
```

The two forms are syntactically very similar and the absence of statement terminators in KPL necessitates an additional syntactic detail. Since the next thing after any assignment statement may legally begin with a left parenthesis, KPL has an additional syntax rule that requires the arguments in a function invocation to be on the same line as the name of the function. This rule disambiguates what would otherwise be a parsing problem.

More precisely, the rule says that the opening parenthesis must be on the same line. To avoid parsing problems, the following sort of thing:

```
foo
  ( x,
    y,
    z )
```

must be re-written to:

```
foo (
    x,
    y,
    z )
```

The Assignment Statement

Here are some example assignment statements:

```
i = j * 4
*p = *q
p.name = "smith"
a[k] = foo (b,c)
```

The general form is

LValue = Expression

where *LValue* can have any of the following forms:

ID
** LValue*
LValue . ID
LValue [Expression]
(LValue)

The asterisk in the second form is used to indicate pointer dereferencing. The dot in the third form is used to indicate field accessing in objects and records. The brackets in the forth form are used to indicate array accessing. Parentheses can be used for grouping, as in:

```
(* p) [i] = x
* (p [i]) = x
```

Confusion between = and == by C++/Java programmers has been the source of many bugs.

```
if (i = max) ...          // A common C++/Java mistake
if (i == max) ...
```

In KPL, the assignment symbol (=) is not an operator, as it is in C++ and Java. In keeping with KPL's philosophy of emphasizing program correctness at the expense of conciseness and efficiency, it was decided that = would not be usable as an expression, which makes the following illegal:

```
if i = max ...          -- Syntax error!
```

Also, in KPL integers are not implicitly coerced to type **bool**. Thus, the above statement must be coded as:

```
i = max                 -- Use this instead
if i != 0 ...
```

Type Conversions

KPL provides several built-in, predefined functions. These use the standard function invocation syntax, but they are recognized by the compiler as special. The compiler will generate machine instructions to perform the operation and will insert this code directly inline. There are no corresponding function definitions for these operations.

Most of these built-in functions perform data type conversions. Here are the built-in conversion functions. (In the following, assume that the variables i, c, d, b, and p have types **int**, **char**, **double**, **bool**, and **ptr**, respectively.)

```
i = charToInt (c)     -- Convert ASCII into -128..127
c = intToChar (i)     -- Ignore high-order 24 bits
d = intToDouble (i)   -- Never a loss of accuracy
i = doubleToInt (d)   -- Truncates (e.g., -4.9 => -4)
b = ptrToBool (p)     -- Null=>false, Other=>true
```

Other built-in functions yield floating point values that cannot be easily obtained otherwise:

```
d = posInf ()        -- Returns positive infinity
d = negInf ()        -- Returns negative infinity
d = negZero ()       -- Returns -0.0, which differs from 0.0
```

The compiler will automatically insert the following type conversions whenever they are needed.

```
charToInt
intToDouble
ptrToBool
```

Here are some examples, showing the use of automatically inserted type conversions:

```
i = 'A'          -- Set i to 65
d = 123          -- Set d to 123.0
while p          -- Walk a linked list
  ...
  p = p.next
endWhile
```

Any other conversion must be programmed explicitly.

```
i = 123.0        -- Error: need to use 'doubleToInt'
c = 65           -- Error: need to use 'intToChar'
```

Static Type Checking

Next, we consider type checking when objects and classes are involved. For the following examples, assume that we have two classes called “Person” and “Student”. Assume Student is a subclass of Person.

Here are two variables:

```
var per: Person
    st:  Student
```

Each of these variables holds the entire object, not a pointer to the object. Even thought Student is a subclass of Person, the following assignments are not allowed:

```
per = st        -- Compile-time error!
st = per        -- Compile-time error!
```

The reason that these assignments is not allowed is that, in general, the variables will have different sizes. In order to perform the assignments, data would need to be discarded or added. If this is what is desired, the programmer must code it explicitly.

Next, consider using pointers to the objects:

```
var perPtr: ptr to Person
    stPtr:  ptr to Student
```

The following assignment is legal:

```
perPtr = stPtr        -- Okay
```

In KPL, any pointer that is declared to have type "**ptr to** C", where C is a class, will be guaranteed to point at runtime to an instance of C or one of C's subclasses. Likewise, any pointer that is declared to have type "**ptr to** I", where I is an interface, will be guaranteed to point at runtime to an instance of some class that implements interface I. This is the same type rule as in Java.

This guarantee is made as long as only safe constructs are used. The programmer can use constructs (such as **asPtrTo**, described below) to violate this invariant.

If perPtr points to an instance of class Person and we send a message using perPtr, we will invoke a method defined in class Person.

```
perPtr.meth(...args...)
```

However, perPtr might point to an instance of one of Person's subclasses, like Student (or even to an instance of a subclass of a subclass of Person, and so on). Assume perPtr points to an instance of class Student and we send the same message. If "meth" has been overridden and redefined in Student, then we will invoke the new method. Otherwise, if meth is inherited without being overridden, we will invoke the method defined in Person.

The following assignment is not allowed. While perPtr might point to a Student at runtime, the compiler cannot guarantee this.

```
stPtr = perPtr        -- Compile-time error!
```

Explicit casting, using the **asPtrTo** construct, is discussed later.

Subtyping Among Array and Record Types

One array may be copied to another. In the following example, all 10 elements will be copied.

```
var arr1, arr2: array [10] of Person
...
arr1 = arr2
```

To make the assignment, the arrays must have the same type. There is no subtype relationship between array types, even when pointers are used. For example:

```
var p1: ptr to array [10] of Person
    p2: ptr to array [10] of Student
...
p1 = p2       -- Compile-time error!
p2 = p1       -- Compile-time error!
```

Likewise, there is no subtype relationship between record types. Two record types are equal if and only if they have the same fields, with the same names in the same order, and the fields have pair-wise types that are equal.

```
var r1: record
          f1: ptr to Person
        endRecord
    r2: record
          f1: ptr to Person
        endRecord
    r3: record
          f1: ptr to Student
        endRecord
...
r1 = r2       -- Okay
r1 = r3       -- Compile-time error!
```

Dynamic Type Checking

Given a pointer variable (like “perPtr” above), KPL provides two built-in operations to determine what sort of object it points to at runtime: **isInstanceOf** and **isKindOf**.

The **isInstanceOf** operator uses binary infix syntax, where the first operand points to an object and where the second operand is a class. It returns a **bool** value. In the next example, **isInstanceOf** is used to determine whether perPtr points to an instance of class Person.

```
var perPtr: ptr to Person
...
if perPtr isInstanceOf Student
  print ("Got a Student!")
endIf
```

The **isInstanceOf** operator returns **true** if the first operand points to an instance of the named class. Note that if perPtr had pointed to an instance of some subclass of Student, the **isInstanceOf** expression would have returned **false**.

To perform the more inclusive test, KPL has another binary infix operator named **isKindOf**, whose first operand points to an object and whose second operand is a class or interface. In the next example, assume that "PartTimeStudent" is a subclass of Student.

```
if perPtr isKindOf Student
  print ("Got a Student or PartTimeStudent!")
endIf
```

Note that the **isInstanceOf** operator in KPL is different from the "instanceof" operator in Java. The **isKindOf** operator in KPL behaves the way Java's "instanceof" behaves.

Pointer Casting

KPL provides a built-in postfix operator called **asInteger**, which can be used to convert any pointer into an integer.

```
i = p asInteger
j = p.next asInteger + 4
```

To convert an integer into a pointer, the **asPtrTo** operator is used. This is an infix operator whose first operand is an integer expression and whose second operand is a type, not an expression.

Expression **asPtrTo** *Type*

For example:

```
(i+20) asPtrTo array of double
```

The **asPtrTo** operator will simply copy the 32-bit value, without any runtime type checking.

```
var p1: ptr to Person
...
p1 = i asPtrTo Person
```

Of course, the static type checking still occurs.

```
var p2: ptr to Person
...
p2 = i asPtrTo Person        -- Okay
...
p2 = i asPtrTo double        -- Compile-time type error
```

The **asPtrTo** operator may also be used to cast a pointer from one type to a pointer of another type.

A typical use of pointer casting is shown in the next example. Assume that "perPtr" may point to any kind of Person, including a Student object. Given a pointer, the programmer may wish to determine if the pointer points to a Student and, if so, do something with it.

```
var perPtr: ptr to Person
    stPtr: ptr to Student
...
if perPtr isKindOf Student
  stPtr = perPtr asPtrTo Student
  ...Do something using stPtr...
endIf
```

Since pointers can also be incremented and decremented directly, pointer casting is not necessary in some cases, such as:

```
p = p + 4    -- Increment a pointer
p = p - 4    -- Decrement a pointer
```

Pointers may also be subtracted from one another, resulting in an integer.

```
i = p - q    -- The difference between two pointers is an int
```

However, pointers may not be added:

```
i = p + q    -- Compile-time error
```

The **asInteger** operation is safe, but the **asPtrTo** operation is considered unsafe since an error in its use may lead to a system crash.

Operators and Expression Syntax

KPL has many of the same operators as C++ and Java. Furthermore, the syntax of expressions is very similar, so the operators from C++ and Java are parsed using the same precedence and associativity rules as in C++ and Java.

The various expression operators are listed here, along with a few remarks. The operators are grouped and listed from lowest precedence to highest precedence.

```
================ (Lowest Precedence) ================
  All keyword messages, e.g., x at:y put:z
=====================================================
  All infix operators not mentioned below
=====================================================
  ||     Short-circuit OR for bool operands
=====================================================
  &&     Short-circuit AND for bool operands
=====================================================
  |      Bitwise OR for int operands
=====================================================
  ^      Bitwise XOR for int operands
=====================================================
  &      Bitwise AND for int operands
=====================================================
  ==     Can compare basic types, pointers,
  !=        and objects, but not records or arrays
=====================================================
  <      Can compare int, double, and
  <=        pointer operands
  >
  >=
=====================================================
  <<     Shift int operand left
  >>     Shift int operand right arithmetic
  >>>    Shift int operand right logical
=====================================================
  +      Can also add ptr+int
  -      Can also subtract ptr-int and ptr-ptr
=====================================================
  *
  /      For int, always truncates down; -7/3 => -3
  %      Modulo operator for integers
=====================================================
  Prefix -        For int and double operands
  Prefix !        For int and bool operands
  Prefix *        Pointer dereferencing
  Prefix &        Address-of
  All other prefix methods
```

```
=======================================================
  .               Message Sending: x.foo(y,z)
  .               Field Accessing: x.name
  asPtrTo
  asInteger
  arraySize
  isInstanceOf
  isKindOf
  []              Array Accessing: a[i,j]
=======================================================
  ()              Parenthesized expressions: x*(y+z)
  constants       e.g., 123, "hello"
  keywords        i.e., true, false, null, self, super
  variables       e.g., x
  function call   e.g., foo(4)
  new             e.g., new Person{name="smith"}
  alloc           e.g., alloc Person{name="smith"}
  sizeOf          e.g., sizeOf Person (in bytes)
================= (Highest Precedence) ===============
```

Constants and Enums

The programmer may associate names with constant values, using the **const** construct. Here is an example:

```
const
  MAX = 1000
  HALF_MAX = MAX / 2
  PI = 3.14159265358979
  DEBUG = false
  NEWLINE = '\n'
  MESSAGE = "Hello, world!"
  EMPTY_LIST = null
```

The compiler must be able to evaluate all of the expressions in **const** definitions at compile-time.

Note that the value of "HALF_MAX" is an expression, but this is okay since it can be evaluated at compile-time. The expressions in **const** definitions may involve only immediate values and other **const** definitions, even though they may use a **const** definition that occurs later in the file. For example:

```
header
  ...
  const
    A = 100
  ...
  const
    B = C*3
  ...
  const
    C = A-9
  ...
endHeader
```

The expression after the "=" must have one of the following types:

```
int
double
bool
char
ptr to array of char    -- i.e., a string constant
null
```

The compiler will make an effort to evaluate all expressions in the program at compile-time. For example, the following statement

```
i = HALF_MAX + 3
```

will be simplified and compiled identically to

```
i = 503
```

KPL also includes an **enum** construct. Here is an example:

```
enum NO_ERR, WARNING, NORMAL_ERR, FATAL_ERR
```

The **enum** is shorthand for a sequence of **const** definitions, defining sequential integer constants. In this example, the above **enum** is equivalent to the following:

```
const
  NO_ERR = 1
  WARNING = 2
  NORMAL_ERR = 3
  FATAL_ERR = 4
```

The default starting value is 1, but you may specify a different starting value. The general form is:

```
enum ID [ = ...Expression... ] { , ID }
```

The *Expression* must be an integer expression which can be evaluated at compile-time. For example:

```
enum A=charToInt('A'), B, C, D, E
```

Both **const** and **enum** constructs may appear in either the header file or the code file. The placement of the **const** or **enum** determines the visibility of the names it defines. The recommendation is that the names defined **const** and **enum** definitions be fully capitalized, as in the above examples.

Errors and Try-Throw-Catch

KPL has a try-throw-catch mechanism like Java, but with somewhat simpler semantics.

The **try** statement includes a body of statements and a number of **catch** clauses. Here is an example:

```
try
    ...Body Statements...
  catch myError (i: int)
    ...Statements...
  catch error2 (a,b: char)
    ...Statements...
  catch FATAL_ERR ()
    ...Statements...
endTry
```

As in Java, the "body statements" are executed first. If no error is thrown, none of the **catch** clauses is executed. If an error is thrown during the execution of the body statements, then a matching **catch** is searched for.

If a matching **catch** clause is found in the **try** statement, the corresponding error handling statements are executed. The body statements are never re-entered or returned to; instead execution continues with the statement after the **endTry**. It is also possible that the error handling statements in the **catch** clause will contain a **return**, **break**, **continue**, or **throw** statement, which will cause execution to leave the **try** statement.

If no matching **catch** clause is found, then the **try** statement itself terminates and the error is propagated upward and outward.

An error can be thrown explicitly by executing a **throw** statement. Here is an example **throw** statement:

```
throw myError (5)
```

An error may also be thrown by the runtime system during the course of program execution.

Argument values may be passed to the error handling statements. In KPL, the throw-catch process is similar to a function invocation since argument values are copied to parameter variables. However, unlike Java, there is no "error object" and errors are not related in any hierarchy.

In KPL, each error must be declared. Here is an error declaration:

```
errors
  myError (i: int)
  error2 (a,b: char)
  FATAL_ERR ()
```

The error declaration tells the compiler how many and what types of arguments are expected when a given error is thrown or caught. The compiler then checks the **throw** statements and **catch** clauses, much like it checks function definitions and call statements.

An error declaration may occur in either a header or code file. If an error is declared in the code file, it may only be thrown and caught by code in that package. If declared in the header file, the error can be thrown and/or caught in any package that uses that package. Each error must be declared exactly once.

In Java, the **try** statement has a "finally" clause, with rather complex semantics. KPL does not have a "finally" clause.

In Java, each method must say which errors it might throw. These are listed in the method header after the "throws" keyword and the Java compiler ensures that all errors will be caught by some **try** statement. There is no corresponding construct in KPL. Thus, it is possible for a method or function to throw an error that is uncaught. When this occurs, the runtime system will throw a second general-purpose "uncaught-error" error. If this too is not caught, a fatal runtime error will occur and execution will halt.

Regarding Naming and Scope Rules

Many programming languages allow lexical scoping of variable declarations: variables in inner scopes will hide variables declared in outer scopes. For example:

```
begin              -- This is not KPL
  var x: int
  ...
  begin
    var x: int
    ...
  end
end
```

KPL adopts the exact opposite philosophy: in KPL, name hiding is not allowed. The names of parameters and locals must be different and must be different from global variables. Also, things of different sorts, like types, constants, errors, functions, classes, interfaces, and global variables must all have different names.

For example, a type definition and a global variable may not have the same name:

```
type t = record ... endRecord
var t: int                          -- Compile-time error
```

Here is another example, showing that local names may not collide with or hide global variable names.

```
var i: int
...
function foo (...)
    var i: int                      -- Compile-time error
    ...
  endFunction
```

While this might seem overly restrictive, the KPL scoping rules are intended to make programs clearer and more reliable. Having multiple entities with the same name may introduce confusion and increase opportunities for bugs. In KPL, the programmer has to do a little more work when writing programs, but the resulting programs are easier to read and understand.

After all, it is not hard to ensure that all variables have different names. A variable name can easily be made unique by adding a character or two to its name.

```
var i2: int
```

When some package uses several other packages, it is possible that two of the used packages both contain variables with the same name. In such case a “name collision” occurs. For example, assume that package X uses packages A and B. Assume that package A defines a variable called “myVar” and package B defines another variable, by coincidence also called “myVar”. Within package X, a problem arises: what does “myVar” refer to?

In KPL, name collisions are not allowed. Within each package, each different entity must have a different name.

Since it is not always practical to modify any package at will, the **uses** clause has additional syntax that allows entities from another package to be renamed. The **renaming** clause is used to avoid name collisions.

In this example, package X could be coded like this:

```
package X
  uses
    A renaming myVar to myVarA,
    B renaming myVar to myVarB
  ...
endPackage
```

Within package X, the variable from A will be referred to using the name “myVarA” and the variable from B will be called “myVarB”.

The general form of the **uses** clause is this:

```
uses OtherPackage { , OtherPackage }
```

Where *OtherPackage* has this form:

```
ID [ renaming ID to ID { , ID to ID } ]
```

A package may define and export the following kinds of things for use in other packages.

```
constants
types
global variables
functions
classes
interfaces
errors
```

Within any package, all of the above entities must have unique names, regardless of whether they are defined in the package or inherited from another package, and regardless of whether any reference to them actually occurs in the package. The **renaming** clause can be used to rename any of these, as necessary, to avoid name collisions.

Parameterized Classes

Classes may be parameterized. Here is an example with two parameterized classes:

```
class List [T:anyType]
  superclass Object
  fields
    first: ptr to ListItem [T]
    last: ptr to ListItem [T]
  methods
    Prepend (p: ptr to T)
    Append (p: ptr to T)
    Remove () returns ptr to T
    IsEmpty () returns bool
    ApplyToEach (f: ptr to function (ptr to T))
endClass

class ListItem [T:anyType]
  superclass Object
  fields
    elementPtr: ptr to T
    next: ptr to ListItem [T]
endClass
```

The idea is that instances of class "List" will contain a number of elements of type T, where the type of the elements can be any type. We can have lists of integers, lists of Person objects, and so on.

Each list will be represented with a single instance of class "List" which will contain pointers to the first and last elements in a linked list of instances of class "ListItem". Each time an element is added to the list, we will create a new ListItem object and link it into the list. We want to be able to place any kind of thing in the list, so this program stores and manipulates pointers to the elements, not the elements themselves. Each ListItem will point to a single element and also to the next ListItem in the list.

Whenever we use the List type, we must provide a "type argument" for the parameter T. For example, we can define a list of Persons, where Person is a class defined elsewhere:

```
var perList: List [Person]
```

We can also use other kinds of lists:

```
var intList: List [int]
    boolList: List [bool]
    otherList: List [AnotherClass]
    listOfLists: List [List [anyType]]
```

Every item on "perList" list will be a Person, or a subclass of Person. Every item on "intList" will be an **int**, and so on.

We can create new instances of a parameterized class using either **new** or **alloc**, just as with any non-parameterized class.

```
perList= new List[Person] {first = null, last = null}
```

We can manipulate the instances of parameterized classes in the same ways as non-parameterized classes. For example, we can send messages to the List object to add and remove elements from the list. Assume "perPtr"points to a Person object; we can add it to the list with this code:

```
var perPtr: ptr to Person = ...
...
perList.Append (perPtr)
```

Likewise, we can add elements to the "intList".

```
var i: int = 12345
...
intList.Append (&i)
```

Here is the code for the Append method:

```
behavior List
  ...
  method Append (p: ptr to T)
      var item: ptr to ListItem [T]
      item = alloc ListItem [T] { next = null, elementPtr = p }
      if self.IsEmpty ()
        first = item
        last = item
      else
        last.next = item
        last = item
      endIf
    endMethod
  ...
endBehavior
```

Within the implementation part of a parameterized class (i.e., within the **behavior** construct), we can use type parameters. In the above example, we see "T" being used as a type in the implementation of the List methods.

The List class also has a method called "ApplyToEach". This method is passed a function. It runs through the list and invokes that function once on each element in the list. Assume that we have a function that prints a Person object.

```
function printPerson (p: ptr to Person)
    print ("A Person with name = ")
    print (p.name)
    print ("\n")
  endFunction
```

In the next statement, this function is invoked for each Person in the list, thereby printing all their names.

```
perList.ApplyToEach (printPerson)
```

Here is the code for method "ApplyToEach":

```
method ApplyToEach (f: ptr to function (ptr to T))
    var p: ptr to ListItem [T]
    for (p = first; p; p = p.next)
      f (p.elementPtr)
    endFor
  endMethod
```

The compiler will type-check all expressions involving parameterized types. If the programmer makes a type error, it will be caught at compile time. For example, an attempt to add a Person to a list of integers will be caught:

```
intList.Append (perPtr)            -- Compile-time error!
```

Likewise, an attempt to apply "printPerson" to the elements of "intList" is in error:

```
intList.ApplyToEach (printPerson)  -- Compile-time error!
```

The general form of a class specification was given earlier as:

```
class ID [ ...Type Parameters... ]
  ...
endClass
```

The *TypeParameters* are optional. If present, they are enclosed in brackets. In more detail, here is how a class definition begins:

```
class ID [ '[' ID: type, ID: type, ... ID: type ']' ]
  ...
endClass
```

Within the brackets is a list of one or more type parameters. Each type parameter has an associated "constraint type" which follows the colon. Here are examples:

```
class List [T:anyType] ... endClass
class TaxableList [Txbl:Taxable] ... endClass
class Mapping [Key:Hashable, Value:anyType] ... endClass
```

Whenever a parameterized class is instantiated, the type argument must be a subtype of the constraint type. In this example, "Taxable" and "Hashable" must be interfaces or classes defined elsewhere. The keyword **anyType** signifies a predefined type that is the supertype of all other types. When used as a constraint type, it allows any type to be used when the parameterized class is instantiated.

Let us assume that "Taxable" is an interface. Whenever "TaxableList" is instantiated, the type argument must be a class or interface that implements the "Taxable" interface. Assume that "Company" is a class that implements the Taxable interface and that "Vehicle" is a class that does not implement the Taxable interface. Then a TaxableList of "Company"s is okay, but a TaxableList of "Vehicle"s would be in error.

```
var listA: TaxableList [Company]
    listB: TaxableList [Vehicle]   -- Compile-time error!
```

Now assume that the Taxable interface includes a "ComputeTaxes" message. All classes that implement the Taxable interface will have to provide a method for ComputeTaxes, although different classes may implement ComputeTaxes differently. Within the implementation of TaxableList, the message ComputeTaxes may be sent to any variable with type "Txbl", since whatever kind of object it is, it must implement the Taxable interface. Therefore, it must understand the message.

The KPL compiler implements parameterized classes using shared code. In other words, there will be only one copy of the code for methods like Append and ApplyToEach. This code will be shared and used by all lists, whether they are lists of Persons, list of integers, or whatever.

Parameterized interfaces can also be defined, in much the same way as parameterized classes. For example:

```
interface Collection [T:anyType]
  messages
    Append (p: ptr to T)
    Size () returns int
    IsEmpty () returns bool
endInterface
```

Given the above definition of Collection, we could alter the definition of List to make it implement Collection:

```
class List [T:anyType]
  implements Collection [T]
  superclass Object
  fields
    ...
  methods
    ...
endClass
```

The Collection interface requires a method called “Size”, which was not in our original definition of List. The compiler will produce an error, unless we add a Size method to class List.

Parameterized classes and interfaces are most useful in the implementation of general-purpose data structures such as sets, lists, look-up tables, and so on. Without parameterized classes, these sorts of classes, which must handle arbitrary types of data, must work around the compiler’s type-checking system. While this can be done in KPL (with features like **ptr to void** and **asPtrTo**), any program bugs may cause the program to crash catastrophically. However, if parameterized classes are used, the compiler can type-check much more of the program and thereby increase the reliability of the program.

The Debug Statement

KPL contains a statement which consists of the single keyword **debug**. Here is an example piece of code, containing a **debug** statement:

```
if perPtr
  i = 123456
  debug
  x = f1 (3)
endIf
```

The **debug** statement has two uses.

First, when executed, the **debug** statement will immediately halt program execution. The statement is compiled into the “debug” machine instruction, which will cause the BLITZ virtual machine to cease emulation and enter command mode. For example, if the above code is executed, the user will see the following:

```
****  A 'debug' instruction was encountered  *****
Done!  The next instruction to execute will be:
0015bc: 87d0001a    or   r0,0x001a,r13   ! decimal: 26, ascii: ".."
>
```

The user may then enter the “stack” command to see where execution was suspended and can then execute the “go” command to resume execution.

```
> st
   Function/Method        Frame Addr   Execution at...
   ====================   ==========   ====================
   main                    00FFFEF8    test0.c, line 25
Bottom of activation frame stack!
> g
Beginning execution...
```

The second use of the **debug** statement is when the programmer wishes to examine the target assembly code produced by the compiler. Since the target code can be lengthy and difficult to navigate, the **debug** statement can be used to flag a location of interest in the target file.

Below is a section of the target file produced by the KPL compiler for the above source code. This is a small portion of a large file, but it is easy to locate the code of interest. When examining this, note that all material following an exclamation (!) is commenting added by the compiler.

```
! IF STATEMENT...
        mov     23,r13          ! source line 23
!   if _Global_perPtr == 0 then goto _Label_21          (int)
        set     _Global_perPtr,r1
        load    [r1],r1
        mov     0,r2
        cmp     r1,r2
        be      _Label_21
! THEN...
! ASSIGNMENT STATEMENT...
        mov     24,r13          ! source line 24
!   i = 123456          (4 bytes)
        set     123456,r1
        store   r1,[r14+-52]
! --------------------  DEBUG  --------------------
        mov     25,r13          ! source line 25
        debug
! ASSIGNMENT STATEMENT...
        mov     26,r13          ! source line 26
!   Prepare Argument: offset=8  value=3  sizeInBytes=4
        mov     3,r1
        store   r1,[r15+0]
!   Call the function
        call    _function_11_f1
!   Retrieve Result: targetName=x  sizeInBytes=4
        load    [r15],r1
        store   r1,[r14+-56]
! END IF...
_Label_21:
```

The instruction following the “debug” instruction is a “mov” instruction which is a synthetic instruction. The instruction

```
        mov     26,r13
```

is expanded by the assembler into

```
        or      r0,26,r13
```

which is what was displayed as the “next instruction” when execution halted after the “debug”.

Conclusion

No programming language is perfect and no programming language can address all potential applications with equal facility. The KPL language has been designed for students to use to write a simple operating system within a college course of one or two terms. The features which have been included in the language have been selected with this in mind.

The KPL philosophy emphasizes reliability of the resulting programs at the expense of all other considerations. When reliability is emphasized, one effect is that overall coding time should be reduced.

One way this philosophy is manifested is in the greater degree of runtime checking. For example, all pointer and array operations are checked at runtime and errors are caught and reported immediately. Another manifestation is that the language was designed with the explicit aim of encouraging program readability, even if this seems to make the language more difficult to write.

Simplicity in programming languages is always a good thing, leading to better compilers, easier programming, greater efficiency, and increased program reliability. The overall goal of the BLITZ project is to create a simplified, but real, operating system for students to study. It is the hope that the KPL programming language is simple enough to be both useable and fun to code in, while being complete enough for the real work of implementing operating system kernel code.