# An Introduction to

# KPL:

# A Kernel Programming Language

*Harry H. Porter III*
*Portland State University*

**HHPorter3@gmail.com**

*17 June 2021*

This document describes the KPL programming language, which is an integral part of the Blitz-64 Computer System. KPL has many of the features in Java, C, and C++. This document is written for experienced programmers who are familiar with those languages.

Like C and C++, KPL is a compiled language and is intended for writing systems software, such as OS kernel code. KPL is meant to be usable in isolation, with zero dependencies on outside software.

# Table of Contents


**Introduction** **7**
The Blitz-64 Project 7
Design Philosophy 8
Familiar Features 9
Advanced Features 9
Novel and Unusual Features 10

**About This Document** **13**
Prerequisite Background 13
Related Documents 13
Document Revision History / Permission to Copy 15
Versioning System 15
Related Software Tools 16
Notation and Terminology 16

**The Hello-World Program** **18**

**Packages** **19**

**Compiling** **20**
Safe and Unsafe 21
Linking 22

**The Header and Code Files** **24**
The Header File 25
The Code File 27

**Syntax and Grammar** **29**
Missing Semicolons 30
Comments 31
Lexical Tokens 31

**Statements** **33**
The If Statement 33
The While Statement 34
The Do-Until Statement 35

The For Statement 36
The Switch Statement 37
The Return Statement 38
Other Statements 38

**Basic Data Types 39**
Integer Constants 41
Character Constants 41
Floating Point Constants 42
Conversions 43

**Variables 44**
Variable Declarations 44
Local Variables 45
Global Variables 45

**Complex Data Types 47**
Type Definitions 48

**Arrays 50**
Creating Arrays 51
Array Representation 53
Array Sizes 54
Initializing Arrays 55
Dynamic Arrays 57
Multidimensional Arrays 58
Array Equality 59

**Strings 60**
String Equality 61
Unicode and UTF-8 61

**The Struct and Union Types 63**
Struct 63
Union 64
Data Representation and Alignment 64

**Pointers 66**

Creating New Objects 66
The “null” Pointer 68

**Functions 71**
Pointers to Functions 72
External Functions 75

**Objects and Classes 76**
Visibility Control 79
Fields 80
Methods 81
Creating Objects 84
Object Representation and Layout 87
Dispatch Tables and Runtime Class Representation 89

**Interfaces 96**

**The Assignment Statement 98**
Operators += and -= 99
Operators ++ and -- 101

**Type Checking and Subtypes 102**
Type Conversions 102
Object-Oriented Type Checking 106
Dynamic Type Checking 107
Pointer Casting 108
Subtyping Among Array and Struct Types 110

**The SwitchOnClass Statement 112**
Implementation 113

**Operators and Expression Syntax 115**
64 Bit Signed Arithmetic 117
Integer Division 117
Arithmetic Shifting 118
Syntax Exception Regarding ‘*’ 119

**Constants and Enumerations 121**

Constants 121
Enumerations 122

Errors and Try-Throw-Catch 125
The Try and Throw Statements 125
Declaring Errors 127
Uncaught Errors and Debugging 128

Naming and Scope Rules 131
The Unique Name Rule 131
The Renaming Clause 132

Parameterized Classes 135
Parameterized Interfaces 140

Conclusion 142

Appendix 1: Predefined Functions 143
upcastToHalfword, upcastToWord, upcastToInt 145
upcastToDouble 145
asByte, asHalfword, asWord 145
forceToByte, forceToHalfword, forceToWord 146
forceToDouble, forceToInt 147
copyBitsToDouble, copyBitsToInt 147
asInteger 148
asPtrTo 148
ptrToBool 149
isKindOf 149
isInstanceOf 150
sizeOf 151
initializeArray 151
setArraySize 153
arrayMaxSize 153
arraySize 153
initializeObject 154
CPUControl and CPUControlUserMode 155
CAS: Compare and Swap 156

The “fence” Memory Barrier 157

**Appendix 2: Printing with “printf” 159**

Introduction 159

Examples 159

Format Codes 160

Differences With C / C++ 162

**Appendix 3: Alternate Method Syntax 166**

Infix and Prefix Methods 166

Keyword Methods 167

**Appendix 4: Style Recommendations 169**

**Appendix 5: Memory Management 181**

Stack Usage 181

The Max_Stack_Usage Clause 183

The Memory Heap 185

**Appendix 6: KPL Syntax 187**

The Notation Used in this Grammar 187

A Context Free Grammar of KPL 189

**Appendix 7: Lexical Details 193**

Source File Encoding 193

Comments 193

White Space 194

Identifiers 194

Integers 195

Floating Point Constants 196

Character Constants 197

String Constants 199

Regular Expressions for Token 203

**Appendix 8: Recent Changes 206**

**About the Author 207**

# Introduction

## The Blitz-64 Project

The KPL language is part of the Blitz-64 project, which includes

- The Blitz-64 Processor Computer Architecture
- Machine Emulator
- Assembler
- Linker
- Library Tools
- KPL Language Design
- KPL Compiler
- Documentation

The Instruction Set Architecture (ISA) describes the Blitz-64 processor.

The ISA of a modern computer is complex, but we can summarize the Blitz-64 machine. Blitz-64 is a modern processor core featuring 16 general purpose registers, 16 Control and Status Registers (CSRs), and a bank of Translation Lookaside Buffer registers (TLBs). All registers are 64 bits each. At any time, the core is executing in one of two modes: either in kernel mode or in user mode. A few security-critical instructions are privileged and may only be executed in kernel mode. Access to the CSRs is privileged and they are used by the kernel to control interrupt processing, syscalls, and various exception handling. The TLB registers are used to implement virtual memory address spaces. The architecture is a RISC design, aimed at everything from a large, secure, multitasking OS kernel to embedded micro-devices.

A hardware implementation of Blitz-64 is underway. As of October 2019, work on a Verilog implementation of the ISA is proceeding smoothly. All non-privileged instructions and most privileged instructions are completed and functional. The microarchitecture uses a novel out-of-order scheduler and is achieving one clock cycle per instruction.

The Blitz-64 ISA is also implemented by a machine emulator, which executes the full ISA and also includes a built-in debugger. KPL code can be compiled, assembled, and

linked using these existing tools. Then, the KPL code can be executed on the emulator. All of the complex examples in this document have been executed this way.

The existing toolset includes an assembler, a linker, and tools to create object libraries. All tools are written in C and the KPL compiler is written in C++. The plan is to port all tools and the compiler to KPL, but this work has not begun.

The following documents provide more information:

*“KPL Syntax”*
*“Blitz-64: Instruction Set Architecture Reference Manual”*
*“Blitz-64: Assembler, Linker, and Object File Format”*
*“Personal Statement: The Goals of the Blitz-64 Project”*

## Design Philosophy

The primary design criterion of the KPL language was simplicity. The intent was to create a language that can be understood and acquired quickly by any skilled programmer. Over time, KPL has grown, yet this goal remains number one. Simplicity helps to achieve our other goals.

A secondary design criterion is 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.

A third design criterion was safety and error reporting. KPL places an emphasis on catching and reporting errors.

The Blitz-64 project is specifically targeted at applications that require high-reliability and malware resistance. KPL was designed with this objective in mind. A key goal is to design a programming language that puts reliability first. Of course, execution efficiency is important and tradeoffs between reliability and efficiency cannot be avoided. Traditionally, low-level languages like C and C++ made design choices in favor of efficiency, leaving high-reliability to interpreted languages. Within the class of systems programming languages, KPL is more aimed at reliability and fault-tolerance than familiar languages.

## Familiar Features

Many of the KPL **data types** will be familiar. For example, the types **int**, **double**, **bool**, **struct**, **union** are well-known. KPL makes it easy and natural to use **pointers** and **arrays** and the expressions like ***p** and **a[i]** will be familiar to C programmers.

Many of KPL's **control structures** will be familiar. For example, KPL includes **if**, **for**, **while**, **return**, **switch**, **break**, and **continue** statements.

KPL includes **classes** and **interfaces**. C++ and Java programmers will be at home. KPL is closer to C++ than Java, since KPL allows explicit manipulation of pointers, although the KPL syntax is simple and closer to Java.

KPL is designed to be **a compiled language**, not an interpreted language. KPL is compiled into assembly code, assembled into machine code, and executed directly on the hardware. KPL is not a scripting language and is not intended to run on top of other software.

KPL supports **separate compilation**, and is intended to accommodate extremely large and complex programs, made of smaller, individually compiled and tested pieces.

KPL is a systems programming language (like C and C++) and not an interpreted language. It is **meant for low-level programming**, and intended to run on bare hardware.

The KPL **memory heap model** requires the programmer to explicitly free memory (like C and C++); automatic garbage collection (as in many interpreted languages) is not possible.[1]

## Advanced Features

KPL includes the ability to create **parameterized classes**. In C++, these are called "**template classes**". In KPL, the code is not copied; **code sharing** is supported.

KPL includes interfaces, and **parameterized interfaces** are supported. Interfaces are also related in a hierarchy which allows **multiple inheritance**.

---

[1] More accurately, "Automatic GC is not reasonable". After all, clever researchers have figured out how to do automatic garbage collection for C and C++, proving that anything is possible.

KPL includes the ability to manipulate **pointers to code** and to create **nameless functions** which can be stored, passed as arguments, etc.

KPL includes features to manipulate bits. In KPL, the programmer has complete and direct control over whether data is represented in **8, 16, 32, or 64 bits**.

KPL includes a **try-throw-catch** mechanism, which will be familiar to Java programmers. All errors (including arithmetic exceptions and overflow) can be caught and handled within the program, if the programmer chooses.

KPL is designed to accommodate huge programs that are broken into pieces which are **compiled separately** and then **linked**, along with **library functions**, to create an executable.

Since KPL is a systems programming language intended for software that accesses the hardware directly, linkage to **external functions** (often coded in assembly) is supported and arguments and returned values of invocations are fully type-checked on the KPL side. Upcalls (in which assembly code invokes KPL functions) are also supported.

## Novel and Unusual Features

Most features of KPL are **safe operations** and, regardless of how bad a bug is, the program should not result in a system crash. Any possible runtime error will be caught and a descriptive error message will be displayed. However, several language features are specifically identified as being **unsafe operations**, and buggy use of them can cause unpredictable / implementation-dependent behavior. The programmer must say explicitly if any of the unsafe features are used, thus dividing programs into two broad reliability classes.

Although KPL is a rather low-level language appropriate for bit fiddling and pointer manipulation, the approach to arrays is high-level. All **arrays are bounds-checked**. The well-known buffer-overflow danger doesn't plague KPL. All array operations are "safe", in that any bugs will be caught immediately and descriptive error messages will be displayed.

Various approaches to breaking large programs into independent modules have been proposed. This language introduces KPL **packages**, which form a unit of encapsulation larger than classes and functions. Packages are an integral part of the

language and support full compiler checking for huge programs which involve separate compilation of independent modules. One goal is to eliminate consistency errors across multiple packages. Another goal is to make all dependencies explicit. The ultimate goal is to prevent accidentally breaking previously debugged code in one package by modifying code in another package.

KPL introduces the **switch-on-class** statement. This is something I have not encountered before and believe to be both novel and useful.

KPL was designed in conjunction with the **Blitz-64 Instruction Set Architecture** (ISA), which is quite unusual. Modern language designers always assume a fixed and unchangeable legacy software and hardware base. At most, they can encourage the additions to the ISA, but never simplifications. Modern architecture work takes it as a given that the full C language must be supported efficiently and, in many cases, that already compiled code must continue to run flawlessly. At most, hardware designers can add additional functionality to handle special cases.

With the Blitz-64 project, we had the flexibility to change the ISA to better support the language; **the ISA and the language grew up together** and evolved jointly in the exploration of new points in the ISA and language design spaces. Their joint evolution made previously unreachable regions of the design space accessible.

KPL **syntax** is much simpler than C++ or Java syntax. Complex programs do not require languages with complex syntax. Complex syntax only hinders programming and this really matters for complex programs. Technically, KPL is “LL(k)” which means that it is much easier for humans to deal with than “LR(k)” grammars like C, C++, and Java. I want to emphasize how important I feel this is.

The KPL language has support for **compile-time verification that stack usage** limitation requirements are met. While this feature would not be needed by most programs, some high-reliability codes in imbedded systems with limited memory, must guard against all errors, including stack overflow.

Programmers of the languages Smalltalk and Squeak will be familiar with **keyword syntax** for invoking methods:

```
x.foo (y, z)    -- Traditional syntax
x at: y put: z  -- Keyword syntax
```

The KPL language supports both styles and allows the programmer to use whichever style he or she is most comfortable with.

Furthermore, both styles may be intermixed in the same program. My thinking was that in creating a language that allows both styles, it would be possible to evaluate and understand the pros and cons of each. My conclusions are (1) neither style is clearly superior and both have their advantages, and (2) mixing the two styles in a single programming is a disaster. It is difficult to remember which method uses which style.

```
x.myMethod (y)    -- Was the method defined like this...
x myMethod: y     --    or this?
```

KPL syntax has been simplified and reduced in several ways, including the elimination of the ubiquitous **semicolon** at the end of every statement, the elimination of **unnecessary parentheses** in many statement types, the replacement of the **braces { }** for statement grouping by descriptive **"end-" keywords**, and the use of a different notation **"--" for comments**. These features will be unfamiliar and initially annoying. However, these are insignificant nuances for advanced programmers and the resulting simplification of program code will be quickly appreciated.

KPL includes the type **double**, but leaves out **single precision floating point**. Certainly a language occasionally needs floating point capability, but with numerical computation increasingly being offloaded to specialized processors, the additional complexity of two similar data types has been avoided.

# About This Document

## Prerequisite Background

This document introduces the KPL programming language. We assume:

- No prior knowledge of KPL or the Blitz-64 project

We assume you are familiar with:

- Programming and programming language concepts

In addition, familiarity with the following is desirable:

- Bitwise logical operations (AND, OR, XOR, NOT)
- Boolean logic (AND, OR, IF-THEN-ELSE)
- Hexadecimal notation
- Signed Numbers (i.e., two's complement binary number representation)
- Sign-extension

## Related Documents

The following documents contain additional information:

**"KPL Syntax"**
The grammar of KPL, expressed precisely. Intended as a handy reference.
**"Blitz-64: Software Reference Manual"**
Documents various KPL functions and discusses algorithms and design decisions.
**"Blitz-64: Summary of the Machine Architecture"**
Overview of the Blitz Architecture.
**"Blitz-64: Instruction Set Architecture Reference Manual"**
Describes the Blitz-64 core, including registers and machine instructions.
**"Blitz-64: Assembler, Linker, and Object File Format"**
Describes the Blitz-64 Assembly Language and related topics.
**"Personal Statement: The Goals of the Blitz-64 Project"**
Motivations and objectives for the project.

## Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name are used. The document history is:

| **Date** | **Author** | |
|---|---|---|
| 13 October 2019 | Harry H. Porter III | *<document created>* |
| 10 November 2019 | Harry H. Porter III | *<initial version completed>* |
| 17 June 2021 | Harry H. Porter III | *<current version>* |

Details about updates to this document are in the appendix titled "Recent Changes".

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

Any material lifted should be referenced.

## Versioning System

In the Blitz-64 project, version numbers are not used for programs and documents. Instead, dates are used.

By comparing dates, you can determine whether this document matches the tools you are using or, if not, which is more recent.

## Related Software Tools

The following programs are relevant:

| Tool | Description | Coding Status |
| --- | --- | --- |
| `kpl` | KPL Compiler | Completed |
| `asm` | Blitz-64 Assembler | Completed |
| `link` | Blitz-64 Linker | Completed |
| `createlib` | Blitz-64 Library Tool | Completed |
| `blitz` | Blitz-64 Hardware Emulator | Completed |

## Notation and Terminology

Syntax and Grammar  KPL has a formally specified syntax. However, we don't want to bog down with it, so in this document we give the grammar informally.

For repetition, we use "..." as in:

```
var ID, ID, ..., ID : Type
```

More formally, we use { }* to mean "zero or more repetitions", as in:

```
var ID { , ID }* : Type
```

We also use { }+ to mean "one or more repetitions", as in:

```
methods { MethodPrototype }+
```

In other situations, braces { } are meant literally, not as grammar meta-symbols:

```
new array of int {11, 22, 33}
```

For optional constructs, we use brackets [ ] as in:

```
var ID : Type [ = Expression ]
```

We also use brackets [ ] literally when we describe array accessing. In array examples, the brackets do not mean optional. We assume it will be obvious when the brackets are meant literally and when they indicate optional material.

```
a[i] = 123
```

We also leave out material, as in:

```
if Expression
  ... Statements ...
elseIf Expression
  ... Statements ...
else
  ... Statements ...
endIf
```

Terminology We use the terms **byte**, **halfword**, **word**, and **doubleword**, to refer to various sizes of binary data.

| | number of bytes | number of bits | example value (in hex) |
|---|---|---|---|
| byte | 1 | 8 | A4 |
| halfword | 2 | 16 | C4F9 |
| word | 4 | 32 | AB12CD34 |
| doubleword | 8 | 64 | 0123456789ABCDEF |

We use the terms **KiByte**, **MiByte**, and **GiByte** instead of KByte, MByte, and GByte to mean:

| Unit | Value | | |
|---|---|---|---|
| KiByte | $2^{10}$ | 1,024 | $\sim 10^3$ |
| MiByte | $2^{20}$ | 1,048,576 | $\sim 10^6$ |
| GiByte | $2^{30}$ | 1,073,741,824 | $\sim 10^9$ |

# The Hello-World Program

The “Hello-World” program prints a message and stops. The code for every program is broken into two files.

For this example, 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 ()
    printf ("Hello, world\n")
  endFunction
endCode
```

We’ll show the keywords of the language (symbols like **if**, **else**, **while**, **header**, **endHeader**, etc.) 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 may 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 name.

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

The code in the Hello package calls a function named “printf”. 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 printf is undefined”.

# Compiling

Let's compile and run the "Hello-World" program.

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 to produce the "executable" file
• Execute the program
```

Versions of the development tools to prepare a program for execution can be run on a Unix/Linux/Mac computer. If using versions of the tools written in KPL itself, these commands would be run directly on a Blitz-64 computer.

Regardless of where the program is compiled and executed, we'll use "%" in this document to stand for the shell prompt. User-typed input will be shown **`like this`**.

First, let's compile the "Hello" package. The KPL compiler is named "kpl".

```
% kpl Hello
```

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

```
Hello.s
```

containing Blitz-64 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, 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-64 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.

Nevertheless, 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" (core dump, segment fault, etc.). 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

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 "link". Here is the command line:

```
% link 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-64 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
%
```

Without "-g", the emulator will enter a command mode, where the user can do things like single-step the program, examine memory and registers, look at variables in the runtime stack, etc.

If there are errors during execution, the virtual machine emulator will enter a command mode, allowing the programmer to begin debugging.

# 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
```

Below 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 = struct
                    val: Person
                    next: ptr to PERSON_LIST
                  endStruct
  errors
    MY_ERROR (id: int)
    OTHER_ERROR (a,b,c: byte)
  enum
    NO_ERR = 0, WARNING, NORM_ERR, FATAL_ERR
  functions
    foo (a1: int, a2: byte) returns double
    bar (a1, a2: byte)
    printErrMess (errCode: int)
  class Person
    superclass Object
    fields
      name: ptr to array of byte
      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: byte) returns double
    ...Variable declarations...
    ...Statements...
  endFunction
  function bar (a1, a2: byte)
    ...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 grammar is given in an appendix. The grammar is also repeated in the document:

*"KPL Syntax"*

In the present document, the grammar is suggested informally. For example, ellipses are used for repetition:

```
var ID, ID, ..., ID: Type
```

We also leave out information when not relevant.

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.

```
if        ... endIf
for       ... endFor
while     ... endWhile
switch    ... endSwitch
header    ... endHeader
code      ... endCode
interface ... endInterface
behavior  ... endBehavior
function  ... endFunction
method    ... endMethod
struct    ... endStruct
union     ... endUnion
try       ... endTry
```

As a matter of style, please try to place the "end…" keyword directly under the keyword it matches and line them up by indenting the same amount:

```
function (...) returns ...
  ...Statements...
  while Expr
    if Expr
      ...Statements...
    endIf
    ...Statements...
  endWhile
  ...Statements...
endFunction
```

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 KPL uses the hyphen since it stands out more visually.

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

```
x = y - 2   /* A 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        endFunction      int
INTEGER                42        0x1234abcd       10_000
DOUBLE                 3.1415    6.022e23         1.000_001
CHAR                   'a'       '\n'             '∉'
STRING                 "hello"   "\t\n"
ID                     x         my_var_name      yPos
OPERATOR               <=   >    +    -
MISC PUNCTUATION       (    )    :    .    ,    ;    =
```

Identifiers may contain letters, digits, and the underscore. They must begin with a letter. Case is significant for all identifiers and keywords.

Integer and double constants may contain underscores, which can be used as separators to make lengthy values more readable. Underscores are ignored.

```
10000000
10_000_000
```

— *Identical value and easier to read*[2]

Here is the recommended way to write 64 bit values in hex:

```
0x1234_5678_9abc_def0
```

[2] If underscores are used, they must be used correctly, by which we mean every three digits.

# Statements

## 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, like this:

```
if (x > y)
  max = x
  ...
```

The preferred KPL style is to avoid parentheses in order to reduce syntactic clutter, make the code visually cleaner, and not obfuscate the underlying algorithm.

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 an **elseIf** keyword, which can be used to make nested **if** statements more readable:

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

## The While Statement

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

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

```
KPL                         C / C++ / Java
================            ===================
while n > 0                 while (n > 0) {
  ...                         ...
  if ...                      if (...) {
    break                       break;
  endIf                       }
  ...                         ...
endWhile                    }
```

The **break** and **continue** statements may be used, just like in C, C++, and Java. For example:

```
while true
  ...
  if ...
    break
  endIf
  ...
endWhile
```

## The Do-Until Statement

KPL has a **do-until** statement, which is similar to the **do-while** statement in Java and C++. (The difference is that the termination condition in KPL is reversed from Java/C++, and KPL uses the keyword **until** instead of **while**.[3])

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

[3] It would have been preferable to to design KPL to use **do-while**, but because of the syntactic simplicity of KPL, the keyword **while** would cause problems. Every design involves compromise.

# The For Statement

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 shown in the next example. We also give an equivalent in Java and C++.

```
KPL                             Equivalent in 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; if it's missing, the default is an increment of 1. 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.

The **break** and **continue** statements can be used in both forms of **for** statement.

```
KPL                              Equivalent in Java and C++
=====================            ==========================
for i = 1 to 100 by 3            for (i=1; i<=100; i=i+3) {
  ...                              ...
  if ...                           if (...) {
    continue                         continue;
  elseIf ...                       } else if (...) {
    break                            break;
  endIf                            }
  ...                              ...
endFor                           }
```

## The Switch Statement

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**.

## The Return Statement

The **return** statement can be used to return from within a function or a method.

```
return
```

If the function or method is expected to return a value, then it must be provided:

```
return Expression
```

## Other Statements

Assignment statements look the same as in C, C++, and Java.

```
x = y * (z + 4)
```

Function invocation looks the same as in C, C++, and Java.

```
foo (3, x, "hello")
```

Method invocation is always done with the “dot” syntax:

```
personPtr.ComputeTax (0.22, rateTable)
```

The “->” operator from C and C++ is not used in KPL

```
p->meth (x, y);    // This is not KPL
```

# Basic Data Types

Here are the basic data types:

```
Type                      Example Values
================          =================================
int                       123    -57    0x1234abcd1234abcd
double                    3.1415        -5.2e10
bool                      true          false
byte                      123           0x7f
halfword                  32767         0x12ab
word                      -2147483648   0x1234abcd
```

The first three of these (**int**, **double**, **bool**) are the most important and widely used. The remaining three are not as commonly used.

Values of type **int** are always represented as 64-bit signed values, stored in two's complement and can be written in either decimal or hex[4]. The legal range for **int** values is:

```
Decimal
    -9,223,372,036,854,775,808 … 9,223,372,036,854,775,807
Hex
    0x8000,0000,0000,0000 … 0x7FFF,FFFF,FFFF,FFFF
```

Values of type **double** are always stored using the IEEE 64-bit floating-point standard.

Values of type **bool** are stored using a single byte.

In addition, KPL has 3 additional "representation" types. These types are not widely used, but are available for situations where the programmer needs full control over data representation:

```
byte
halfword
word
```

[4] We use comma separators in this document for clarity, but separators are not allowed in KPL code.

By default, all integer values are of type **int** and all arithmetic computation is performed with 64 bits. To use the types **byte**, **halfword**, and **word**, the data must be explicitly moved and the programmer must choose either range checking or truncation when data is downsized.

We also mention the **ptr** data type here, although it is not considered a "basic" type.

```
KPL                      C / C++
================         ========================
ptr to Type              * Type
ptr to int               * int
ptr to MyClass           * MyClass
```

KPL uses the familiar operators * and & to dereference pointers and to obtain the address of data:

```
var p: ptr to int
p = &x
*p = 123
```

Here are the data sizes:

```
Type                     Representation
================         ====================
bool                      8 bits  =  1 byte
byte                      8 bits  =  1 byte
halfword                 16 bits  =  2 bytes
word                     32 bits  =  4 bytes
int                      64 bits  =  8 bytes
double                   64 bits  =  8 bytes
ptr to ...               64 bits  =  8 bytes
```

Even though KPL was designed for one particular CPU architecture, the language makes it clear exactly how all values will be represented in memory. The goal of portability is that a program must execute predictably and identically, regardless of which machine it runs on.

There are two values of type **bool**, represented by the keywords **true** and **false**. As in C / C++, **false** is represented with the byte 0x00. Any other value is interpreted as **true**, with 0x01 being the usual value.

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) ...
```

## Integer Constants

An integer value can be specified in either decimal or hex, whichever the programmer finds most convenient or clear.

```
var i: int
...
i = 38
i = 0x26    -- Equivalent
```

Sign-extension of hex constants does not occur:

```
i = 0xff                    -- This...
i = 0x00000000000000ff      --   ... is equivalent to this
i = 0xffffffffffffffff      --   ... and not to this
```

All integer constants are represented as 64 bit signed values:

```
i = -957
i = -0x3bd                  -- Equivalent
i = 0xfffffffffffffc43      -- Also equivalent
```

## Character Constants

An individual character is represented using an integer value. For example, ‘a’ is stored as the value 97 (hex 0x61). An individual character is stored in a variable of type **int**; the “char” data type is unnecessary and is not included in KPL.

KPL uses Unicode and individual characters are represented using their Unicode "codepoint" values. In KPL, characters are stored in **int** variables.

```
var c: int
c = 'a'       -- Same as c = 97
c = '€'       -- Same as c = 8364
c = '😀'       -- Same as c = 128512
```

Escapes may be used; the following are all equivalent and result in the same value being stored:

```
c = '\n'
c = 10
c = 0x0a
c = '\x0a'
```

The usual backslash escapes may be used in character and string constants.

```
message = "Hello, world\n"
```

Since the codepoints of all ASCII characters are within the range 0 … 127 (0x00 … 0x7f) , every ASCII code fits in a variable of type **byte**.

```
var c2: byte
c2 = 'a'       -- Same as c2 = 97
```

In legacy code, programmers often store ASCII characters in an 8-bit data type called "char". Although KPL programmers can certainly use variables of type **byte**, using an **int** is the way it should be done in KPL.

## Floating Point Constants

Floating point constants are coded in the usual ways.

```
var d: double
...
d = 123.456
d = 1.4999999e-57
d = 42            -- Integers are converted as necessary
```

All floating point constants are represented as double precision numbers (that is, with 64 bits) according to the IEEE standard. KPL does not support single precision floating point numbers. This was not an oversight.

## Conversions

Conversions are discussed in depth elsewhere but we can say this:

Integer constants generally have type **int**. However, small integer constants can be used in contexts requiring **byte**, **halfword**, or **word** values if they fall within the range of the type.

All integer arithmetic and bitwise logical operations will be performed using 64 signed values. When applied to **byte**, **halfword**, or **word** data, all arithmetic and logical operators will produce a 64 bit result of type **int**.

Integer constants will be converted to floating-point values whenever required, but only if the value can be represented exactly as a **double** value.

# Variables

Every variable is either a local variable or a global variable. Of course data may also be placed in memory which has been dynamically allocated on the heap, but this is discussed in a later section.

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.

## Variable Declarations

Variables—both local and global—are declared using the keyword **var**:

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

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

```
KPL                         Java and C++
================            ========================
var
  x, y, z: byte             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;
  isDone: bool = false      bool isDone = false;
```

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
bool                     false
ptr to ...               null
byte                     0 (= 0x00)
halfword                 0 (= 0x0000)
word                     0 (= 0x00000000)
```

Variables of type **struct** and **union** will be initialized to zero values, which means all fields will be set to their default initialization of zero values.

All objects and arrays contain hidden information. Every object contains a hidden field which indicates the object's class and is used in message dispatching. Every array contains an indication of the current and maximum sizes of the array.

Objects and arrays are initially in an uninitialized state. Every object and array must be initialized before use. In the uninitialized state, these hidden fields are zeros and a zero indicates that the object or array is uninitialized.

Any attempt to access or use an uninitialized object or array will result in an error message saying as much.

## Local Variables

Local variables are declared at the beginning of functions and methods. Local variables only exist while the function or method is being executed.

When a function or method is executed (that is, when the function is "called" or the method is "invoked"), a stack frame is allocated on the stack. Local variables are always placed in such stack frames.

## Global Variables

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.

Sometimes these variables are called “static data”, but KPL uses the term “global variables.”

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).

# Complex Data Types

A type is said to be complex if it somehow involves other types. For example, an array type always involves a “base type” (e.g., **double**) which is the type of the elements.

KPL has the following complex data types:

```
Type                     Example
================         ================================
Arrays                   array [10] of double
Pointers                 ptr to array [10] of double
Structs                  struct
                           val: double
                           next: ptr to MY_STRUCT
                         endStruct
Unions                   union
                           valAsDouble: double
                           valAsInt: int
                         endUnion
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** is not normally 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_STRUCT = struct
                   val: double
                   next: ptr to MY_STRUCT
                 endStruct
```

Such a definition then allows "MY_STRUCT" 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 = struct ... endStruct           -- 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 be thrown.

Typically, any error will result in a message and execution will be immediately terminated. Therefore, the notorious "buffer overrun" errors from C / C++ cannot occur in KPL when arrays are used.

## Creating Arrays

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. For example:

```
new array of double { 1.111, 2.222, 3.333, 4.444 }
new array of double { 100 of 1.111, 500 of 2.222 }
```

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[5]. 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
```

[5] The terms "R-Value" and "L-Value" come from compiler technology. An R-Value is any expression that can occur on the righthand side of an assignment. When evaluated at runtime, it yields a data value that can be copied or used in some way. An L-Value is any expression that can appear on the lefthand side of an assignment. When evaluated, it gives a location or memory address into which a data value can be stored.

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.[6]

```
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 }
```

[6] Generally speaking in KPL, a pointer dereference will be inserted whenever necessary. So “p[i]” is equivalent to “(*p)[i]”. The programmer can use either, but the first is the preferred style because it is shorter and simpler.

# Array Representation

Each array is stored along with a 8-byte header giving the number of elements. This header field precedes the first element. For example, this array

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

would require 8 + 100*8 bytes of storage (= 808 bytes), since each **int** value requires 8 bytes.

The purpose of the array header field is to allow all array accesses to be checked. If an attempt is made to fetch or modify an element that is not within the array, a runtime error will be thrown. These checks are inserted by the compiler.[7]

The presence of the array header also makes it possible to safely copy arrays. For example:

```
var
  x: array [10] of double = ...
  y: array [5] of double = ...
...
x = y
```

When asking for the address of arrays and array elements, note that the address of the array is always 8 bytes less than the address of the first element, since the array size is stored directly before the first element.

```
&a[0] == &a + 8
```

[7] The Blitz-64 Instruction Set Architecture (ISA) includes special instructions (INDEX0, INDEX1, INDEX2, ... INDEX32) which perform array index scaling and bounds checking in a single instruction. Thus, the execution overhead for bounds checking an array access is minimal.

The array header actually stores two values, a MAXIMUM size and a CURRENT size. The header consists of 8 bytes. The first word is the MAXIMUM size and the second word is the CURRENT size.[8]

One benefit of this arrangement is that extra space can be preallocated for an array. The array can then change size and grow at runtime, as long as it remains within its preallocated maximum size.

In practice, the KPL approach to array seems convenient. If, however, the overhead of the array header is considered unacceptable, the programmer can always work with pointers directly.

## Array Sizes

A header is present on every array and it contains two integer values: the CURRENT size and the MAXIMUM size of the array. Both are always element counts, not byte counts.

Every read and write to the array will be checked to ensure

```
0 <= index < currentSize
```

If this test fails, an error will be thrown at runtime.

An array header has the format:

| **Offset** | |
|---|---|
| 0 | MAX: word |
| 4 | CURR: word |
| 8 | First element: … |

Both MAX and CURR count elements, not bytes.
KPL imposes a ceiling limit of 2,147,483,647
elements for any single array. For an array of

[8] An array header has this format:

| **Offset** | |
|---|---|
| 0 | MAX: word |
| 4 | CURR: word |
| 8 | First element: … |

Both MAX and CURR count elements, not bytes. KPL imposes a ceiling limit of 2,147,483,647 elements for any single array. For an array of **int**s, this limits the array size to 16 GiBytes.

The CURRENT size of an array can be determined with **arraySize**:

```
i = arraySize (arr)     -- The argument can be an array variable
i = arraySize (arr_ptr) --    ...or a pointer to an array
```

The MAXIMUM size of an array can be determined with:

```
i = arrayMaxSize (arr)
i = arrayMaxSize (arr_ptr)
```

The CURRENT size of an array can be changed with **setArraySize**. The new size must be within 0 … MaxSize or an error will be thrown.

```
setArraySize (arr, 30)
setArraySize (arr_ptr, 30)
```

The CURRENT size of an array can be set to the MAXIMUM size with:

```
setArraySize (arr, arrayMaxSize (arr))
setArraySize (arr_ptr, setArraySize (arr_ptr, 30))
```

Since it is common to deal with Strings and arrays of bytes, there is an additional function: **maximizeString**. This will set the CURRENT size to the MAXIMUM size.

```
maximizeString (str)
```

The argument (of type **ptr to array of byte**) should have been initialized previously or an error will be thrown.

## Initializing Arrays

As previously mentioned, every array carries a hidden 8 byte header which tells the size of the array. Consequently, every array must be initialized before use.

When initial values are given, the initialization of the header is implicit. In the next example, we demonstrate initialization in two ways: with **new** and on the heap with **alloc**.

```
var
  a: array [100] of double = new array of double { 100 of -1.23 }
  p: ptr to array of double
...
```

```
p = alloc array of double { 300 of 5.67 }
```

With both **new** and **alloc**, the CURRENT is initialized to the MAXIMUM value.

In some cases, it may be desirable to initialize arrays explicitly and the built-in function **initializeArray** can be use for that, as in:

```
var
  arr: array [100] of double
...
initializeArray (arr)
```

This function will only set the header; it will not alter the elements of the array. The programmer can rely on the fact that all storage will be set to zero before use. Since **initializeArray** will set both the CURRENT and MAXIMUM sizes to the same value, this will have the same effect as:

```
arr = new array of double { 100 of 0.0 }
```

The **initializeArray** function can also initialize an array using a pointer, as in:

```
var
  arr_ptr: ptr to array [100] of double
...
arr_ptr = ...         -- for example: arr_ptr = &arr
initializeArray (arr_ptr)
```

In the above example, “arr_ptr” had better point to a block of memory large enough to contain all the array elements, plus the 8 byte header which every array has. Since the CURRENT size will be set to the MAXIMUM size, the new array will pick up whatever values were previously in that memory region.

## Dynamic Arrays

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 function **arraySize**. This expression returns the number of elements in the array.

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

## Multidimensional Arrays

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 = arraySize (arr_3D)
```

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

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

## Array Equality

Arrays can be compared using the == and != operators.

In order for two arrays to be equal, they must have the same CURRENT size and each of their elements must be pairwise equal.

The CURRENT size is used to determine how many elements are to be compared. Their MAXIMUM sizes are ignored. If the MAXIMUM size is larger than the CURRENT size, the additional elements are ignored.

Two uninitialized arrays are considered equal, since their CURRENT sizes are both zero.

All elements up to the CURRENT size are compared. Element comparison simply tests bit equality.[9]

[9] Note that if the array elements are themselves array elements, this equality check will compare all bytes of each element, up to the MAXIMUM size. So the equality test used on the elements is not "array equality" as defined in the section. Instead, the elements are compared in their entirety, byte-by-byte.

# Strings

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

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

The “System” package includes the following type definition:

```
type String = ptr to array of byte
```

Thus, it is more common to define variables like this:

```
var str: String
```

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

```
var ch: int
...
ch = str [1]     -- sets ch to 'e' = 101 = 0x65
str [3] = 'k'    -- now str points to "helko"
```

We can print the string using the built-in **printf** statement:

```
printf ("str = %s\n", str)
```

The **printf** function is discussed elsewhere, but much of the functionality comes from C. Here are examples which C programmers will understand:

```
printf ("My int = %d\n", i)   -- Print in decimal
printf ("%20d", i)            -- Pad to field width
printf ("%-20.5s", str)       -- Truncate, left-justify, pad
printf ("%x", i)              -- Print in hex
printf ("%#016X", i)          -- Misc formatting
printf ("%f", dVal)           -- Print a double
printf ("%-23.8f", dVal)      -- Precision, left-justify, pad
printf ("%d \t %s\n\n %d", i, str, i) -- Print several things
printf ("\t\"\n\\")           -- Escape sequences
```

The **printf** function is understood by the KPL compiler and it will check the above formatting strings. The compiler will also verify that all value expressions are type-correct.

In KPL, arrays always carry their sizes with them. There is no need for a terminating ASCII "null" character and it is not normally used.

This is a major difference from how strings are dealt with in C / C++, where a forgotten terminating "null" is a familiar mistake. Of course it is certainly possibly to add a terminating "null" in KPL, if you happen to encounter any reason to:

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

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 writable segments and may therefore be updated in place.

# String Equality

Strings are represented as arrays, so comparisons using == and != follow the rules for pointer and array comparison:

```
var str1, str2: String
if str1 == str2 …            Compare pointers
if *str1 == *str2 …          Compare all characters
```

Other comparisons (such as "<" for lexicographic ordering) are complex for UTF-8 strings, so these must be done by calling specialized functions.

# Unicode and UTF-8

String constants, character constants, and comments may use the full Unicode character set[10].

```
-- Note that ∞ ≠ 😀
review_4 = "Café of the Naïve (€€€)"
ch = 'π'
```

However, all keywords and identifiers are restricted to the ASCII subset.

```
var π: double   -- Compile-time error
```

In KPL, a "String" is an array of bytes. Strings are stored using the UTF-8 encoding system. In UTF-8 encoding, each character requires between 1 and 4 bytes.

In UTF-8, every ASCII character occupies a single byte. Therefore, if a string contains only ASCII characters, there is a one-to-one correspondence between characters in the string and elements in the array. The length of the array is exactly the number of characters in the string.

If the string contains characters beyond the ASCII subset, then there will be more elements in the array than characters in the string. These two assignments are equivalent and do exactly the same thing:

```
str = "∞ ≠ 😀"
str = "\xe2\x88\x9e\x20\xe2\x89\xa0\x20\xf0\x9f\x98\x80"
```

This will print "12":

```
printf ("%d", arraySize (str))
```

This will print "∞ ≠ 😀":

```
printf ("%s", str)
```

[10] Unicode and UTF-8 encoding are introduced and described in the document "*Blitz-64: Assembler, Linker, and Object File Format*".

# The Struct and Union Types

## Struct

A **struct** is a collection of several data items, bound together into a record. Each data item in the struct is labeled with a field name.

In this example, a new **struct** type is defined and given the name “MY_STRUCT”. Then a variable “r” is defined.

```
type MY_STRUCT = struct
                   val: double
                   next: ptr to MY_STRUCT
                 endStruct
var r: MY_STRUCT
```

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

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

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

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

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

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

## Union

A **union** is similar to a **struct** except that all fields are placed on top of one another. The size of a **union** value is determined by the length of the longest field.

In the following example, the length of the data will be 4 bytes, since values of type **word** require 4 bytes, while the others require fewer bytes.

```
type MY_UNION = union
                  b: byte
                  w: word
                  h: halfword
                endUnion
var myUnion: MY_UNION
...
myUnion.w = 0x1234abcd        -- change all 4 bytes
... = myUnion.b               -- retrieve the first byte, 0x12
myUnion.h = 0x7fff            -- change first 2 bytes
... = myUnion.w               -- retrieves 0x7fffabcd
```

## Data Representation and Alignment

KPL guarantees to the programmer exactly how all data values (including **struct**s, **union**s, **array**s, and objects) are represented in memory. In the case of **struct**s and objects, the fields are placed in memory sequentially in order, with extra padding bytes inserted where necessary to ensure proper alignment. Thus, every field value will be properly aligned.

The alignment requirements for the Blitz-64 core are discussed in the ISA document. The alignment of data in KPL is mandatory and is not "implementation dependent". This is so programmers can depend on how their data is stored, regardless of platform.

Here are the KPL alignment rules.

- There is no alignment requirement for byte-sized data (**byte** and **bool**).
- **halfword** values must be located on even addresses, i.e., halfword aligned.
- **word** values must be on addresses divisible by 4, i.e., word aligned.
- **int**, **double**, and **ptr** values must be doubleword aligned, that is, on addresses divisible by 8.
- Objects and arrays will always be a multiple of 8 bytes in size and will be doubleword aligned.
- The size of a **struct** or **union** may be as small as 1 byte. The alignment of a **struct** or **union** is determined by its size. If the size is 1, there is no alignment requirement. If the size is 2, then it must be halfword aligned. If the size is 3 or 4, then word alignment will be used; otherwise the **struct** or **union** will be doubleword aligned.[11]

Normally the compiler will take care of alignment and the programmer can ignore all alignment issues. However, if the programmer writes code like this, there will be an alignment error.

```
var
  p: ptr to int
p = asPtrTo (100007, int)
i = *p
```

This code will throw the error "ERROR_UnalignedLoadStore" since the address 100,007 (i.e., 0x0,0001,86A7) is not divisible by 8.

[11] Note that this is slightly different from saying that the alignment of a **struct** or **union** is determined by the alignment requirement of the largest field it contains. A **struct** of size 3 bytes, containing three fields, each of type **byte**, will be loaded and stored using LOAD.W and STORE.W and therefore must be word aligned.

# 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_STRUCT
    p: ptr to MY_STRUCT
```

The variable "r" will require 16 bytes (the size of a MY_STRUCT record, as previously defined) while "p" will require only 8 bytes since all pointer values are 8 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
```

## Creating New Objects

Objects, arrays, structs, and unions 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 **struct** on the heap:

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

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

```
r = new MY_STRUCT { 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 sruct pointed to by "p" with statements like these:

```
x = (*p).val
(*p).val = 12.345
```

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, then dereference it first".

```
KPL                         Equivalent in C++
====================        ===================
x = p.val                   x = p->val;
p.val = 12.345              p->val = 12.345;
```

The above discussion of pointers used structs and pointers to structs, 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 structs, objects, or arrays with code like this:

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

# The “null” Pointer

The null pointer is symbolized with the keyword **null**. The **null** value is represented as the integer 0, i.e., 0x0000000000000000.

```
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**, as in C.

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

```
A "NULL ADDRESS" exception has occurred!
```

When a runtime error like this occurs, the Blitz-64 virtual machine will halt emulation and go into command mode. The user can then type the “stack” command to see the activation stack. Generally speaking, the fifth 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.
> stack
   Function/Method                Execution at...  File
   ============================   ===============  =================
   EmulatorDebuggingRequested                      KPL_lib/runtime.s
   invokeDebugger                 CALL   line 762  KPL_lib/System.c
   RuntimeErrorNullAddress        CALL   line 649  KPL_lib/System.c
   _runtimeErrorNullAddress                        KPL_lib/runtime.s
   AnotherFun                     ASSIGN line 116  examplePgm.c
   MyClass.myMethod                                examplePgm.c
   MyFunction                     SEND   line 110  examplePgm.c
   main                           CALL   line 58   examplePgm.c
   _kplEntry                                       examplePgm.c
   _entry                                          KPL_lib/runtime.s
```

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

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

Pointers can be converted to integers and vice versa, using the **asInteger** and **asPtrTo** functions.

```
i = asInteger (p1)
p2 = asPtrTo (i, byte)
```

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** function to determine the size of a type of value.  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 pointers for equality are safe. However, the **sizeOf** and **asInteger** operation 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.[12]

The functions **asInteger**, **asPtrTo**, and **sizeOf** are built-in to the KPL language, since they take types instead of values as arguments.

[12] Note that an assignment such as "*p = 123" is a safe operation. But what if p is allowed to point to a variable local to some function and that function has returned before this assignment? The memory region where the variable was previously stored may have been re-used for a completely different stack frame, so the assignment could cause a system crash by randomly overwriting memory. It is also unsafe to ask for the address of fields and array elements, since those might be within local variables.

# 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 ()
  printf ("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: bool, p: ptr to byte, x, y, z: 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.

# Pointers to Functions

Functions 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, byte)
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 8 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; therefore, it 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.[13]

The compiler will recognize that this means the function is to be called and executed.

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

## External Functions

KPL also provides a way to invoke functions written in assembly language. Consider the function named "Fork" 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 "Fork" must be declared using the **external** keyword.

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

KPL does not allow overloading of function names. All functions must have distinct names. Furthermore, the names of external functions are used as is, without any modification, so they must not coincide with other identifiers used in assembly files, such as the names of machine instructions, registers, etc.

[13] The two forms are syntactically very similar and the absence of statement terminators in KPL necessitates an awkward syntactic detail. Since the next thing after any assignment statement may legally begin with a left parenthesis, KPL has an additional syntax rule: "The opening parenthesis "(" must be on the same line as the function name."

The following function invocation is problematic:

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

It must be re-written somehow to move the "(" onto the same line as the function name:

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

# Objects and 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 an example class we will name "Person". Here is the specification part for Person:

```
class Person
  superclass Object
  fields
    name: ptr to array of byte
    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
```

For every **class**, there must be a corresponding **behavior**, and vice-versa. You may not have one without the other.

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 the class specification is 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 "zero or more repetitions". 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
  implements InterA, InterB, InterC
  superclass Object
  fields
  ...
```

The class specification lists the methods that are implemented in the class. However, the specification includes only 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”.

Consider two methods, one in a class “Person” and one in a superclass of Person. Either the methods have the same name or they have different names. If the method name is the same, then the subclass method will override the method from the superclass.

KPL does not allow “overloading” of method names. Every method in a given class must have a unique name. Two methods may only have the same name if they are in different classes.

## Visibility Control

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 a program from accessing certain methods or fields, there are three ways of doing it.

First, the class specification can be placed in the code file, making the entire class private to a given package. With this technique, a class—including its very existence —can be totally hidden from code in alien, possibly untrusted packages.

Second, the programmer can work with interfaces, which can be used to control which methods may be invoked on an object. With this technique, the client code uses pointers to interfaces, rather than pointers to classes. The client code can only only send messages present in the interface specification; it cannot know exactly which class is implementing the functionality and cannot access any fields unless access methods have been included for that purpose in the interface.

Finally, the programmer can create a new “wrapper class” to allow only certain kinds of access to an underlying object. Clients only have access to (pointers to) instances of the wrapper class; to access the protected objects, clients must send messages to the wrapper class which will control and monitor access to the protected objects. While this technique imposes execution overhead, it allows sophisticated control. For example, the wrapper class can require passwords and implement complex authorization protocols.

# 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—which is sufficient. 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 that it is related to MyClass.

```
var MyClass_x: ...Type...
```

In C++, the programmer accesses a static field by writing “MyClass::x”.

In KPL the programmer simply accesses “MyClass_x”.

## Methods

A method is declared in the class’s specification part. The **class** construct includes a list of method prototypes.

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

The method is implemented in the class's **behavior** :

```
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 int)
  if p == null
    printf ("NULL!")
    return
  endIf
  printf ("%c", *p)
endMethod
```

That example was illustrative; it might be clearer if coded as:

```
method printWithNULL (p: ptr to int)
  if p
    printf ("%c", *p)
  else
    printf ("NULL!")
  endIf
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 to invoke an inherited and overridden version of a 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**:[14]

```
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
...
```

[14] In this example, we are invoking an inherited implementation of “meth” from within the new, overriding implementation of “meth”. It is common to invoke the inherited version of a method from within the reimplementation of that same method, as was done here. But actually, we can use “**super**.meth (…)” from within any method in Student, not just “meth”. Regardless of where in Student it is used, the result is the same: the search for the implementation begins in the superclass, i.e., in Person. In all other method invocations (i.e., those not involving **super**), the search begins in the class of the object itself as determined at runtime, i.e., in Student or some subclass of Student.

```
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 an 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 all 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 byte) returns ptr to Person
    ...
endClass

behavior Person
  ...
  method init (n: ptr to array of byte) 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(...)
```

Programmers are free to choose whichever approach seems best for their applications.

## Object Representation and Layout

Each object is represented in memory with a sequence of one more doublewords. Each object contains a header doubleword, so the minimum object size is 8 bytes. The header word is followed by the bytes for the fields. All objects of the same class have the same size.

The address of the object is the address of the header doubleword. In other words, a pointer to the object points to the first byte of the header, not to the first field.

The header is a pointer to the “Dispatch Table” and is often called the Dispatch Table Pointer (DPT).[15]

The KPL compiler will lay out the fields of a class sequentially. In other words, the compiler will not alter the order of the fields. The compiler will also ensure that all fields are locate at properly aligned addresses. This may require the compiler to insert “padding bytes” between the fields. The compiler will also insert padding bytes at the end of the object if necessary, to round the size of the object up to a multiple of 8 bytes.

Consider this example[16]:

```
class MyClass_1
  superclass Object
  fields
```

[15] For other languages, the header is said to contain a “class pointer”, but we make a distinction between the dispatch table and the class descriptor. This will be discussed later.

[16] To simplify this example, we assume that class Object has no fields.

```
      f1: word
      f2: int
      f3: byte
  endClass
```

The compiler will lay out the fields for this class as follows:

| Offset | Size | |
|---|---|---|
| 0 | 8 | < Dispatch Table Pointer > |
| 8 | 4 | f1 |
| 12 | 4 | ... padding ... |
| 16 | 8 | f2 |
| 24 | 1 | f3 |
| 25 | 7 | ... padding ... |
| | Total size in bytes = 32 | |

If the a subclass is created, the fields of the subclass will be aded after the fields of the superclass. For the following example:

```
 class MySubClass
    superclass MyClass_1
    fields
      f4: word
  endClass
```

the compiler will create the following layout:

| Offset | Size | |
|---|---|---|
| 0 | 8 | < Dispatch Table Pointer > |
| 8 | 4 | f1 |
| 12 | 4 | ... padding ... |
| 16 | 8 | f2 |
| 24 | 1 | f3 |
| 25 | 3 | ... padding ... |
| 28 | 4 | f4 |
| | Total size in bytes = 32 | |

Note that padding at the end of the object was reduced and the compiler was able to find a properly aligned offset for the new field within the area that previously used as padding. In this example, it happened to be the case that the new class's objects

have the same size as the superclass, but generally speaking, the addition of fields will cause the objects to become larger.

In order to reduce the amount of padding added by the compiler, the programmer may want to re-order the fields. One easy technique is to order the fields by their alignment requirements. The programmer should place the fields in this order:

All fields requiring doubleword alignment
followed by
All fields requiring word alignment
followed by
All fields requiring halfword alignment
followed by
All fields requiring byte alignment

For example, by reordering the fields in "MyClass", we can eliminate some padding:

```
class MyClass_REVISED
  superclass Object
  fields
    f2: int
    f1: word
    f3: byte
endClass
```

The compiler still inserts padding at the end to increase the object's size up to a multiple of 8, but the object's size has been reduced:

| Offset | Size | |
|---|---|---|
| 0 | 8 | < Dispatch Table Pointer > |
| 8 | 8 | f2 |
| 16 | 4 | f1 |
| 20 | 1 | f3 |
| 21 | 3 | ... padding ... |
| | Total size in bytes = 24 | |

The fields in a **struct** are laid out following the same rules. However with **structs** and **unions**, there is no header.

# Dispatch Tables and Runtime Class Representation

Each object contains a header which points to a dispatch table. Each class will have exactly one dispatch table.

Consider these classes:

```
class MyClass
  superclass Object
 methods
    meth_1 ()
    meth_2 ()
    meth_3 ()
endClass

class MySubClass
  superclass MyClass
 methods
    meth_4 ()
    meth_2 ()
endClass
```

The compiler will create a dispatch table for each class, as follows:

**MyClass:**

| | Offset | |
|---|---|---|
| | 0 | Class Descriptor for MyClass |
| meth_1 | 8 | MyClass_1 . meth_1 |
| meth_2 | 16 | MyClass_2 . meth_2 |
| meth_3 | 24 | MyClass_3 . meth_3 |

**MySubClass:**

| | Offset | |
|---|---|---|
| | 0 | Class Descriptor for MySubClass |
| meth_1 | 8 | MyClass . meth_1 |
| meth_2 | 16 | MySubClass . meth_2 |
| meth_3 | 24 | MyClass . meth_3 |
| meth_4 | 32 | MySubClass . meth_4 |
| _super_meth_2 | 40 | MyClass . meth_2 |

The initial field (offset = 0) in the dispatch table always points to the "class descriptor" for the class. The class descriptor contains things like the name of the

class. The class descriptor also contains information about superclasses, which is needed for the **isKindOf** function.

The remaining fields in the dispatch table each contain a JUMP to a block of code. When a method is invoked, there will be a jump directly to some entry in the dispatch table which will then immediately jump to the first instruction of the method.

Each slot is 8 bytes. This is enough space for a long jump to any location in memory, although the JUMP instructions will often require only a single 4 byte instruction.

The methods for a class such as "MyClass" are laid out in the dispatch table one after the other, in the order they are listed in the **class** … **endClass** source code.

When subclassing occurs, new entries are added at the end of the dispatch. For "MySubClass", the initial 3 entries are for methods that were present in the superclass. Since one method "meth_2" was overriden, the entry for the subclass (offset 16) now contains a jump to the new method. The other methods ("meth_1" and "meth_3") were not overridden, so those entries point to the method code from MyClass.

Next, we see the entires for a new method "meth_4" which was not present in the superclass. We also see an entry for "_super_meth_2", pointing to the code that was overridden. This entry is needed for a "send-to-super" instruction, which will invoke the method form the superclass.

To take it one step further, let's introduce a sub-sub class:

```
class MySub_Sub_Class
  superclass MySubClass
  methods
    meth_4 ()  -- overrides method in MySubClass
    meth_2 ()  -- overrides methods in MyClass and MySubClass
    meth_5 ()  -- new
endClass
```

Here is the lay out for the dispatch table:

**MySub_SubClass:**

| Offset | |
|---|---|
| 0 | Class Descriptor for MySub_SubClass |

| | | | |
|---|---|---|---|
| meth_1 | 8 | MyClass . | meth_1 |
| meth_2 | 16 | MySub_Sub_Class . | meth_2 |
| meth_3 | 24 | MyClass . | meth_3 |
| meth_4 | 32 | MySub_Sub_Class . | meth_4 |
| _super_meth_2 | 40 | MySubClass . | meth_2 |
| meth_5 | 48 | MySub_Sub_Class . | meth_5 |
| _super_meth_4 | 56 | MySubClass . | meth_4 |
| _super__super_meth_2 | 64 | MyClass . | meth_2 |

Consider a send-to-super, such as:

```
super.meth2 ()
```

Assume it occurs in MySub_SubClass::meth2, so the intent is to invoke MySubClass::meth2().[17]

With a normal send-to-self, the object is first consulted and a pointer to the dispatch table is retrieved from the object's header. The code will use an offset (16 for "meth2") into that table. The actual dispatch table to be used will depend on the class of the object at runtime.

With a send-to-super, there is no need to consult the dispatch table at all; static binding can be used. In this example, the send-to-super occurs in class MySub_SubClass, so the compiler knows that this is a call to MySubClass::meth2. There is no need for dynamic dispatching. The compiler will generate a direct call to the right method.

Next, we consider "class descriptors".

We have already described the format of the dispatch table and now we describe the format of the class descriptor.

Dispatch tables and class descriptors come in pairs. For each class, there will be exactly one dispatch table and one class descriptor at runtime.

[17] Such a send-to-super can occur in any method in MySub_SubClass. That is, an invocation of "meth2" need not occur within the "meth2". The only constraint is that the method being invoked (meth2) must be overridden in MySub_SubClass. Otherwise, you should just use a normal send-to-self.

Each dispatch tables contains a pointer to its class descriptor and each class descriptor contains a pointer to its dispatch table.

For parameterized classes, there is only a single dispatch table / class descriptor pair. For example, for "List [T]" there is only one dispatch table and one class descriptor. This makes it impossible at runtime to distinguish between "List [int]" and "List [Person]", for example.[18]

The class descriptor contains this information:

The name of the class
The filename in which the class was defined
The line number in that file
The size of instances of the class
A pointer back to the dispatch table
The superclasses and super-interfaces of the class

In more detail, here is the layout of a class descriptor

| Offset | Size | |
|---|---|---|
| 0 | 8 | Magic Number |
| 8 | 8 | Ptr to String (name of class) |
| 16 | 8 | Ptr to String (name of source file) |
| 24 | 4 | Line number in source file |
| 28 | 4 | Size of instances, in bytes |
| 32 | 8 | Ptr to dispatch table |
| 40 | 8 | Ptr to super-classes (one ptr for each superclass) |
| | 8 | Ptr to interfaces (one ptr for each superclass) |
| | 8 | Zero (to end the list) |

The magic number field contains the value 0x434c415353646573, which is the ASCII code for the characters 'CLASSdes'. This field is used to make sure that class pointers are valid and catch errors.

The second field ("name of class") is a pointer to a String (that is **ptr to array of byte**). The String will contain the name of the class, such as "MySub_Sub_Class".

[18] However, this is not really necessary. W

The third field ("name of source file") is a pointer to a String containing the name of the source file in which this class was defined. For example, "tests_il/x36-inherit.c".

The fourth field is a word-sized integer giving the line number within that file on which the class definition begins.

The fifth field ("size of instances") is a word-sized integer giving the number of bytes in instances of this class.

The sixth field ("ptr to dispatch table") is the address back to the Dispatch Table for this class.

This is followed by zero or more pointers to the superclasses of this class. The class Object has no superclasses, so it will have zero pointers here. All other classes will have one pointer for each class in the hierachy, with the final pointer pointing to "Object" In particular, these are pointers to the dispatch tables for the other classes.

These will be followed by zero of more pointers for to interfaces. There will be one pointer for each interface that this class implements. These pointers will be the address of "Interface Descriptors", which are described below.

Finally, the class descriptor will end with a zero.

The vector of pointers in the class description is used to implement the **isKindOf** function. To determine whether a class C is a subclass of X or implements X (where X is either a class or an interface), it is sufficient to scan this vector looking for a match. Since all superclasses are included (not just the direct superclass), it is not necessary to visit other class descriptors. Likewise, since the vector contains all the interfaces that are implemented (not just those directly implemented), there is no need to visit Interface Descriptors in a recursive search.

asd
asd
asd
asd
asd
asd
asd

asd
asd

# 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 of a class that implements MyInter. Any such object must necessarily have two methods, called “foo1” and “foo2”, with appropriate parameter typings. Of course, the object’s class may have other methods, but these cannot be accessed through “p”, since they are not part of the MyInter interface.

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 such a variable. The programmer must use a pointer instead.

# 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[19]:

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

The asterisk is used to indicate pointer dereferencing:

```
*p = x
```

The dot is used to indicate field accessing in objects, structs, and unions:

```
myObj.field3 = x
```

The brackets are used to indicate array accessing:

```
a[k] = x
```

As usual, parentheses can be used for grouping:

```
(* p) [i] = x     -- Parens are required here
* (p [i]) = x     -- Parens are optional; same as *p[i]
```

[19] This is a slight simplification. For example, the thing following the * can be any expression, as long as it has type "**ptr to…**".

The precedence of all operators in KPL mimics C, C++, and Java and is documented elsewhere in this document.

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!
```

Furthermore, integers are not implicitly coerced to type **bool**. To achieve both an assignment and a comparison against 0, the following KPL code would be used, making clear that two operations are being performed:

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

## Operators += and -=

C, C++, and Java are likely to be familiar with += and -= and they can be used in KPL.

```
KPL                       Effect
================          ========================
i += k                    i = i + k
i -= k                    i = i - k
```

With += and -=, the first operand (x) is evaluated only once. If "x" is a simple expression, this doesn't make any difference, but if evaluating "x" causes side-effects, this can be important[20].

[20] For completeness, we should remark that in the case of multiple threads, there is a subtle difference. Consider "a[i] += 1"; if there is a possibility that "i" could be modified concurrently by another thread, this statement could function differently from "a[i] = a[i] + 1". However, concurrent programmers really ought to follow a discipline of locking shared data that precludes such race condition sensitivity.

Here is how the += statement is defined:

```
KPL                         Equivalent
================            ========================
xxx += Expr                 temp = & xxx
                            *temp = *temp + Expr
```

For example:

```
KPL                         Equivalent[21]
================            ========================
*foo(x) += k                temp = foo(x)
                            *temp = *temp + k
```

In C, C++, and Java, a number of operators can be combined with = to create a new assignment operator. In KPL, this can be done only with + and -. The assignment = cannot be combined with any arbitrary operator in KPL. However, this does not seem too common and the same result can be achieved easily enough in KPL.

```
KPL                         Java and C++
================            ========================
i = i * k                   i *= k
```

It would be unusual for the lefthand side expression to involve side-effects, especially from something besides addition and subtraction. However, if this situation does arise, the following KPL code may actually be clearer and less error-prone.

```
KPL                         Java and C++
================            ========================
temp = (complexExpr)        (complexExpr) %= k
temp = temp % k
```

In KPL, += and -= are statements, not expressions. Here are the three forms of assignment statement in KPL:

```
LValue  = Expression
LValue += Expression
LValue -= Expression
```

[21] Note that "& (* foo)" is the same as "foo".

## Operators ++ and --

The KPL language does not have the ++ and -- operators. However, the same result may be achieved with += and -=.

```
KPL                         Java and C++
================            ========================
i += 1                      i++;
i -= 1                      i--;
```

In C, C++, and Java, the ++ and -- operators can be used for both their value and for their side-effects. In KPL, these two steps must be made explicit[22].

```
KPL                         Java and C++
================            ========================
i += 1                      a[++i] = 0
a[i] = 0
```

[22] Originally, ++ and -- were included in C to take advantage of machine instructions that included incrementing/decrementing as side-effects. In modern RISC processors, this sort of instruction complexity is generally avoided in an effort to keep each instruction simple.

Given the best contemporary compiler technology, there is usually no efficiency to be gained from these operators. If the target machine includes instructions that have implicit incrementing/decrementing, the compiler will likely select these instructions regardless of how the programmer codes it.

# Type Checking and Subtypes

## Type Conversions

KPL provides several built-in, predefined functions to convert between the basic data types.

These functions 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 prototypes or definitions for these functions.

Here are the built-in conversion functions:

| Built-in Function | Argument Type | Result Type |
|---|---|---|
| **upcastToHalfword** | byte/halfword | halfword |
| **upcastToWord** | byte/halfword/word | word |
| **upcastToInt** | byte/halfword/word/int | int |
| **upcastToDouble** | byte/halfword/word | double |
| | | |
| **asByte** | int | byte |
| **asHalfword** | int | halfword |
| **asWord** | int | word |
| | | |
| **forceToByte** | int | byte |
| **forceToHalfword** | int | halfword |
| **forceToWord** | int | word |
| | | |
| **forceToDouble** | int | double |
| **forceToInt** | double | int |
| **copyBitsToDouble** | int | double |
| **copyBitsToInt** | double | int |
| | | |
| **asInteger** | ptr | int |
| **asPtrTo** | int/ptr , *Type* | ptr to *Type* |
| **ptrToBool** | ptr | bool |

The following functions are inserted automatically whenever necessary and would not normally be used by the programmer. Although the representation is changed, the underlying numerical value remains exactly the same.

```
upcastToHalfword    byte/halfword             halfword
upcastToWord        byte/halfword/word        word
upcastToInt         byte/halfword/word/int    int
upcastToDouble      byte/halfword/word        double
```

Integers are extended by sign-extension. Since sign-extension is automatic with the Blitz-64 machine instructions LOADB, LOADH, and LOADW, and since values in registers are kept in sign-extended form, there is typically zero overhead for the sign-extension of integers. (The Blitz-64 machine instructions SEXTB, SEXTH, and SEXTW are not normally needed.)

In the following examples, we'll use these variables:

```
var
  d: double
  i: int
  w: word
  p: ptr to ...
```

Here are some example in which automatic conversions are inserted. The first demonstrates that integer constants can be used in contexts requiring 8, 16, or 32 bits, as long as they lie within the representable ranges.

```
w = -1      -- Sign extend 8 bit value 0xff to 0xffffffff
i = 'A'     -- Set i to 65, i.e., 0x0000000000000041
d = 123     -- Set d to 123.0
```

The general principle is that any conversion in which the value is guaranteed to be preserved will be inserted automatically and implicitly. However, whenever there is a possibility the value will not fit into the smaller type, the programmer must explicitly use a conversion function.

```
i = w       -- OK: numerical value will always be unchanged
w = i       -- Error: need to use 'asWord' or 'forceToWord'
```

The following functions will downsize an integer from 64 bits to 8, 16, or 32 bits. This is a checked operation: the original value must lie within the legal range of the

smaller representation. If it does not, a runtime error will be thrown. The Blitz-64 machine instructions CHECKB, CHECKH, and CHECKW are used.

```
asByte              int                        byte
asHalfword          int                        halfword
asWord              int                        word
```

For example, the following will throw an error if "i" is not within the range representable in a byte. Otherwise, the data will be copied with no change in value.

```
b = asByte (i)      -- Throw error if not within -128...+127
```

The following functions will downsize an integer from 64 bits to 8, 16, or 32 bits. If the original value lies outside the range of the smaller representation, the conversion will change the numerical value to fit within the smaller range. An error will never be thrown.

```
forceToByte         int                        byte
forceToHalfword     int                        halfword
forceToWord         int                        word
```

In the functions above, the upper bits are simply discarded. There is no execution overhead.

Any 8, 16, or 32 bit integer can be transformed into a double precision floating point representation without losing accuracy or changing its value. The **upcastToDouble** operation is done using the Blitz-64 machine instruction CVTFI. For example, the following are legal and will never change the numeric value:

```
d = 123                  -- Implicitly converts 123 to 123.0
d = upcastToDouble (w)   -- No loss of accuracy is possible
d = w                    -- Same (upcastToDouble is implied)
```

However loss of accuracy is possible whenever converting a 64 bit integer into a **double**; some extremal values cannot be represented with 100% accuracy as **double** values, since **double**s have limited precision:

```
i = d                    -- Compile-time error
i = forceToInt (d)       -- Use this instead
i = copyBitsToInt (d)    -- ... or this, if you want
```

To convert between 64 bit integers and floating point double representation, use these functions. These use the Blitz-64 machine instructions FCVTFI and FCVTIF.

```
forceToDouble       int                         double
forceToInt          double                      int
```

In the case of the **int→double** conversion, small integers can be converted and the exact value will be preserved. However, for integers outside the range

```
-9,007,199,254,740,992 ... +9,007,199,254,740,992
```

accuracy will be lost.

In the case of the **double→int** conversion, accuracy will be lost for any fractional value or any integral value beyond the range

```
-9,223,372,036,854,775,808 ... +9,223,372,036,854,775,807
```

You may also copy the bits between an **int** and a **double** with the following functions. The bits are copied directly, with no conversion. The value will be totally changed (except for +0.0, which happens to be represented as 0x0).

```
copyBitsToDouble    int                         double
copyBitsToInt       double                      int
```

Elsewhere we will discuss the following functions. Both copy the bits with no change. The **asPtrTo** function requires a second argument, which is a type. It is unsafe and requires the "-unsafe" command line option when compiling.

```
asInteger           ptr                         int
asPtrTo             int/ptr, Type               ptr to Type
```

The **ptrToBool** function is inserted automatically as needed. The **null** pointer is converted to **false** and all other addresses are converted to **true**.

```
ptrToBool           ptr                         bool
```

Here's a very typical example which makes use of the implicit conversion of pointers to **bool**:

```
while p          -- Walk a linked list
```

```
    ...
    p = p.next
  endWhile
```

## Object-Oriented 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 though 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 are 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 (...Arguments...)
```

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!
```

This assignment can be done by explicitly casting it, using the **asPtrTo** function. This is discussed later.

## Dynamic Type Checking

KPL provides two built-in functions to determine what sort of object are pointed to at runtime: **isInstanceOf** and **isKindOf**.

With the **isInstanceOf** function the first argument points to an object and the second argument names a class. It returns a **bool** value. In the next example, **isInstanceOf** is used to determine whether perPtr points to an instance of class Student.

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

The **isInstanceOf** function returns **true** if the first operand points to an instance of the named class. If perPtr points to an instance of some subclass of Student, the **isInstanceOf** function will return **false**.

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

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

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

## Pointer Casting

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

```
i = asInteger (p)
j = asInteger (p.next) + 8
```

To convert an integer into a pointer, the **asPtrTo** function is used. The first argument is an integer expression and the second argument is a type.

```
asPtrTo (Expression, Type)
```

For example:

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

The **asPtrTo** function will simply copy the 64-bit value, without any runtime type checking.

```
var p1: ptr to Person
...
p1 = asPtrTo (i, Person)
```

Of course, the static type checking still occurs.

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

The **asPtrTo** function 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 isKindOf (perPtr, Student)
  stPtr = asPtrTo (perPtr, Student)
  ...Do something using stPtr...
endIf
```

This sort of operation is so common that KPL provides the **switchOnClass** statement, which is discussed is a separate section of this document.

Pointers can also be incremented and decremented directly. Pointer casting is not necessary:

```
p = p + 8    -- Increment a pointer
p = p - 8    -- Decrement a pointer
```

In KPL, the manipulation of pointers is always in terms of bytes. This is different than in C / C++, in which an increment of 1 depends on the exact type of the pointer:

```
KPL:
  p = p + 1    Always increments by 1 byte

C / C++:
  p = p + 1    Increments by 1 byte for "char * p"
  p = p + 1    Increments by 8 bytes for "double * p"
  p = p + 1    Implementation dependent for "int * p"
```

While the sizes of basic types is known and fixed, the sizes of objects, structs, and unions will depend of the fields they contain. The built-in function **sizeOf** can be used to determine the size of values of a particular type.

```
perPtr = perPtr + sizeOf (Person)
```

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** and **sizeOf** functions are always safe, but the **asPtrTo** function is considered unsafe since an error in its use may lead to a system crash.

## Subtyping Among Array and Struct 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 **struct** or **union** types. Two structs 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. Likewise, for union types.

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

# The SwitchOnClass Statement

KPL introduces a new kind of statement called "switch-on-class", which I have not previously encountered, but which seems particularly useful.

To motivate this statement, imagine a program with a complex data structure involving a number of classes. Imagine a pointer where it is not known exactly what sort of an object it points to. Now imagine that we wish to write a function that, passed a such a pointer, will print out information about the object pointed to. This function must print something different for each class of object it might be passed.

Of course this could be done using the **isInstanceOf** and **asPtrTo** functions and a series of if-then-else tests. Since every class would need to be tested, this series of tests would be linear in the number of classes. In short, the code would be ugly and slow.

The syntax of the **switchOnClass** statement is similar to the **switch** statement, as it appears in C, C++, Java, and KPL itself.

In the following example, assume that a class "Student" is a subclass of "Person". Every Person has a "name" field and Students inherit this field. In addition, assume Students have a "grade" field. Also assume a class "Vehicle", which is unrelated to Person and Student, has a "model" field.

Here is an example of the **switchOnClass** statement:

```
var p: ptr to Object
...
switchOnClass p
  case Person:
    printf ("Person: name = %s\n", p.name)
  case Student:
    printf ("Student: name = %s\n", p.name)
    printf ("         grade = %d\n", p.grade)
  case Vehicle:
    printf ("Vehicle: model = %s\n", p.model)
  default:
    printf ("Unknown class\n")
endSwitchOnClass
```

There are several things to notice.

First, the keywords are **switchOnClass** / **endSwitchOnClass** instead of **switch** / **endSwitch**. However, the **case** and **default** keywords are the same.

Second, the **break** instruction is not used and is not allowed. There is an implicit **break** at the end of every **case** statement block.

Third, the expression following the **switchOnClass** keyword is restricted to be an ID, not a complex expression. This identifier must have type "**ptr to** *Class*" or "**ptr to** *Interface*".

Fourth, after each **case** keyword there must be a class name. In a traditional **switch** statement, a value will follow each **case** keyword; here a class name follows the **case** keyword.

Finally, within each case block, the identifier—in this example, "p"—can be used as if it had the more specific type.

For example, in the "**case** Student" statement block, variable "p" has type "**ptr to** Student". Because "p" has this type, the accesses to the "name" and "grade" fields, which only Students have, are allowable.

At runtime, the object pointed by "p" is checked and, depending on its actual class, a jump is made to the corresponding **case** block. If the object is a class that is not included among the options, the jump is made to the **default** block. If the pointer is null, the jump is made to the **default** block.

The test used is equivalent to **isInstanceOf**, not **isKindOf**. If, for example, there is a subclass of Vehicle (say "Bicycle") and "p" happens to point to an instance of Bicycle, then the jump will be made to the **default** block, since the class Bicycle is not among the cases listed. It doesn't matter that Vehicle appears and that every Bicycle is a kind of Vehicle, in the sense of being a subclass.

## Implementation

In the initial KPL compiler, the **switchOnClass** statement is implemented efficiently. The dispatch is performed in constant time, with only a few instructions. Every object's header points to a dispatch table and this is followed to get a hash value

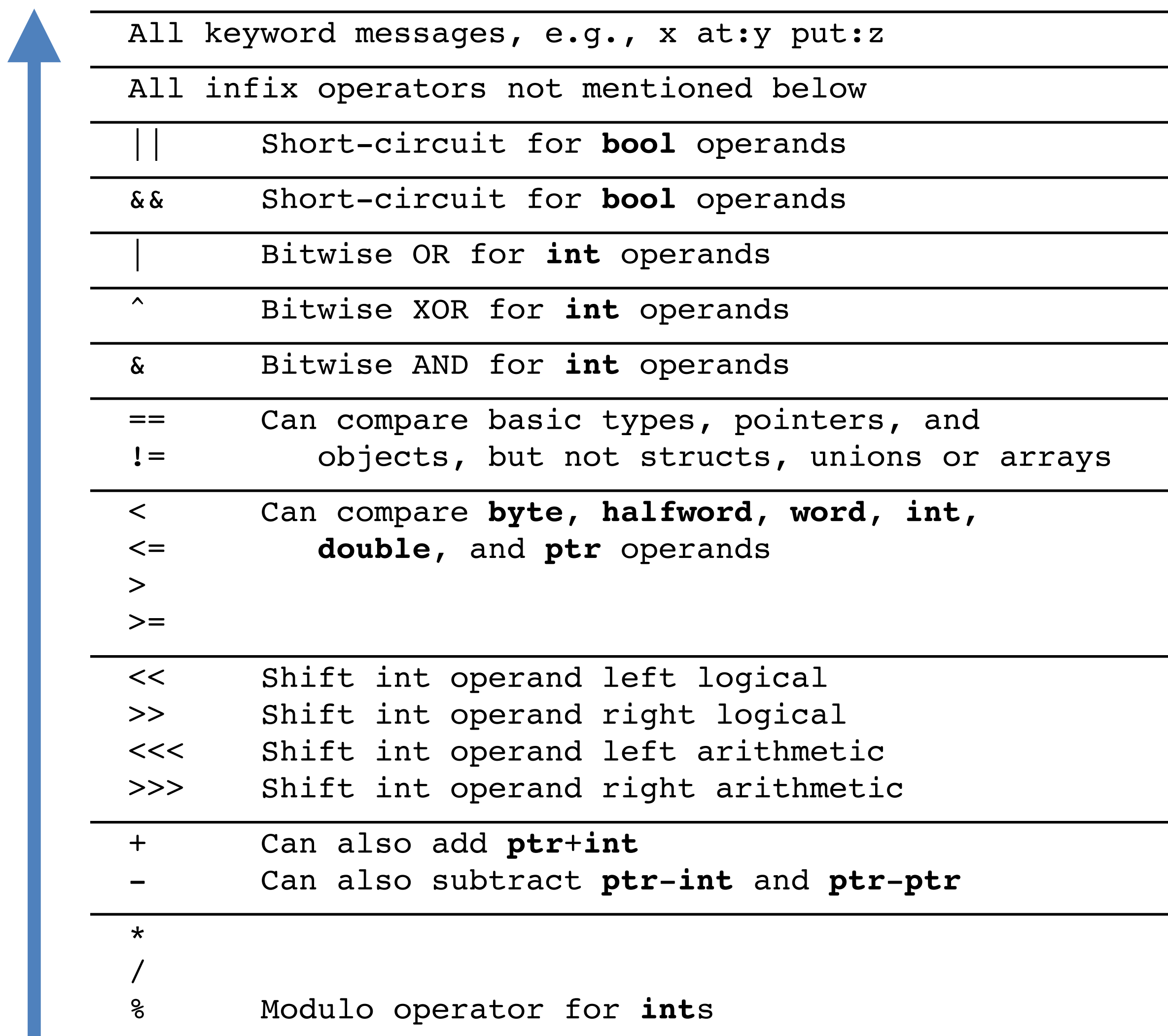
associated with the class. (The compiler computes a hash values based on the spelling of the class name and stores this at runtime as a constant in memory.) This hash value is then used to jump through a hash-based jump table, to go directly to the right code block. The result is that, even if there are 1,000 cases in the **switchOnClass** statement, the dispatch to the correct case block is quick and efficient.

# 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 operators are grouped and listed from lowest precedence to highest precedence. Operators with the same precedence are left-associative.

***Lowest Precedence***



<table>
<tr><td colspan="2">All keyword messages, e.g., x at:y put:z</td></tr>
<tr><td colspan="2">All infix operators not mentioned below</td></tr>
<tr><td>||</td><td>Short-circuit for bool operands</td></tr>
<tr><td>&&</td><td>Short-circuit for bool operands</td></tr>
<tr><td>|</td><td>Bitwise OR for int operands</td></tr>
<tr><td>^</td><td>Bitwise XOR for int operands</td></tr>
<tr><td>&</td><td>Bitwise AND for int operands</td></tr>
<tr><td>==<br>!=</td><td>Can compare basic types, pointers, and objects, but not structs, unions or arrays</td></tr>
<tr><td><<br><=<br>><br>>=</td><td>Can compare byte, halfword, word, int, double, and ptr operands</td></tr>
<tr><td><<<br>>><br><<<<br>>>></td><td>Shift int operand left logical<br>Shift int operand right logical<br>Shift int operand left arithmetic<br>Shift int operand right arithmetic</td></tr>
<tr><td>+<br>-</td><td>Can also add ptr+int<br>Can also subtract ptr-int and ptr-ptr</td></tr>
<tr><td>*<br>/<br>%</td><td><br><br>Modulo operator for ints</td></tr>
</table>

| | |
|---|---|
| Prefix - | For **int** and **double** operands |
| Prefix + | For **int** and **double** operands (nop) |
| Prefix ! | For **int** and **bool** operands |
| Prefix * | Pointer dereference |
| Prefix & | Address-of |
| All other prefix methods | |
| . | Message Sending: x.foo(y,z) |
| . | Field Accessing: x.name |
| [] | Array Accessing: a[i,j] |
| () | Parenthesized expressions: x*(y+z) |
| constants | e.g., 123, "hello", 34.998e-23 |
| keywords | e.g., **true, false, null, self, super** |
| nameless funct | e.g., **function** (...) ... **endFunction** |
| variables | e.g., x |
| function call | e.g., foo(4) |
| built-ins | e.g., **forceToDouble** (4) |
| function | e.g., **function** (...) ... **endFunction** |
| **new** | e.g., **new** Person { name="smith" } |
| **alloc** | e.g., **alloc** Person { name="smith" } |
| **sizeOf** | e.g., **sizeOf** (Person) ... in bytes |
| **asPtrTo** | e.g., **asPtrTo** (i, **double**) |
| **asInteger** | e.g., **asInteger** (ptr) |
| **arraySize** | e.g., **arraySize** (array/arrayPtr) |
| **arrayMaxSize** | e.g., **arrayMaxSize** (array/arrayPtr) |
| **isInstanceOf** | e.g., **isInstanceOf** (p,*ClassName*) |
| **isKindOf** | e.g., **isKindOf** (p,*ClassOrInterfaceName*) |

*Highest Precedence*

Operators that are grouped together at the same precedence level are parsed left-to-right with the normal left associativity.

For example:

```
x - y + z
(x - y) + z      -- Same meaning; parens are not needed.
```

and

```
x - y * z
x - (y * z)      -- Same meaning; parens are not needed.
```

## 64 Bit Signed Arithmetic

All integer arithmetic operations are performed using 64 bit signed arithmetic and result in a 64 bit signed result. Likewise, all logical operations are performed using 64 bits.

If the operands are smaller (e.g., **byte**, **halfword**, or **word**), they will automatically be upcast with sign extension to 64 bits. This sign-extension will occur for both arithmetic and logical operations.

```
var i, j: int, b: byte
...
i = j ^ -1       -- Flips all bits: "j XOR 0xffff_ffff_ffff_ffff”
i = j ^ 0xff     -- Flips 8 bits:   "j XOR 0x0000_0000_0000_00ff”
i = j ^ 255      -- Same as previous since 0xff = 255
b = asByte (0xff)        -- Error; 255 is not within -128...127
b = asByte (0xffff_ffff_ffff_ffdc)   -- No error; this is -36
```

If the result is to be stored in a smaller variable, the programmer must specify whether or not to ignore overflow errors.

```
var x, y, z: byte
...
x = y + z                       -- Compile-time error
x = asByte (y + z)              -- Throw error if problems
x = forceToByte (y + z)         -- Truncate if necessary
```

## Integer Division

Integer division and modulo (/ and %) will throw an “arithmetic exception” error if the divisor is 0. They also throw the error if overflow occurs. This can only happen if the dividend (top number) is the most negative value (-9,223,372,036,854,775,808) and the divisor (bottom number) is -1.

With integer division, there is a question about what happens when the operands are negative. If the top number is positive, the division is rounded to the nearest integer in the positive direction and the modulo result will always positive (or zero, of course).

```
7 /  3 =  2         7 %  3 =  1
7 / -3 = -2         7 % -3 =  1
```

If the top number is negative, the results are *implementation dependent*. Division might follow "truncated division", which is often the case in C, C++, and Java:

```
-7 /  3 = -2        -7 %  3 = -1
-7 / -3 =  2        -7 % -3 = -1
```

Or it might follow "floored division":

```
-7 /  3 = -3        -7 %  3 =  2
-7 / -3 =  2        -7 % -3 = -1
```

Or it might follow Euclidean division, which is perhaps more mathematically justifiable:

```
-7 /  3 = -3        -7 %  3 =  2
-7 / -3 =  3        -7 % -3 =  2
```

# Arithmetic Shifting

KPL uses the symbols << and >> for logical (i.e., bitwise) shifting. KPL uses the symbols <<< and >>> for arithmetic shifting.

Arithmetic right shift (>>>) will perform sign-extension. It can be used to perform division by a power of two (2, 4, 8, …). If the dividend is positive, it always works correctly. However, if the dividend is negative, care must be taken. Right shifting implements Euclidean and floored division correctly, but does not implement truncated division.

KPL introduces a new operator "left shift arithmetic" (<<<). It works exactly like a logical left shift except that it will throw an error if there is overflow, that is if significant bits are shifted off the left end. Thus, <<< can be used to correctly implement multiplication by any positive power of two, with exactly the same behavior in the event of overflow errors.

Mnemonic Hint  We use the symbols << and >> for "logical" because both are "simpler". We use <<< and >>> for "arithmetic" because this is a more "complex" operation. The extra character can be thought of as standing for the extra work of error checking. Furthermore, "left shift arithmetic" is an unusual, new operation. Likewise "<<<"  is an unusual symbol, not normally used.

## Syntax Exception Regarding '*'

The goal of making the syntax of KPL simple has not been entirely achieved. There is an ambiguity in the Context Free Grammar that requires an exceptional rule.

Any operator, such as "*", may be interpreted as infix or prefix. Here are two examples showing that an identical sequence of tokens can be interpreted two ways:

```
x = 123 * p.foo()              The "*" is a binary operator

x = 123
*p.foo()                       The "*" is a prefix operator
```

However, the last statement is not actually legal. Recall that prefix binds more loosely than message-send-dot, so it would be interpreted as

```
*(p.foo())
```

But this violates another rule: Prefix expressions always returns values, so they cannot be used at the statement level. If you want the second interpretation, you would have to add parentheses anyway, so it would look like this:

```
x = 123
(*p).foo()
```

And the parentheses would preclude the interpretation of "*" as a binary operator.

This applies to all operator symbols, such as "+", "-" and "*".

However, there is a second use of "*" that causes parsing problems. It is the use of "*" as the predefined dereference operator in an assignment statement. Consider this sequence of tokens; what is allowable next?

```
x = 123 * p ...
```

This could have two interpretations, as shown by these examples:

```
x = 123
*p = 456          The "*" is the beginning of an assignment statement
```

or

```
x = 123 * p
```
The “*” is a binary operator, continuing the expression

Since “p” could be an arbitrary expression, the parser must distinguish the cases before parsing “p”. This requires arbitrary lookahead, which the parser avoids.

For this situation, we add the following (non-CFG) restrictions:

(1) If t “*” is preceded by a newline, it is assumed to be a prefix operator.
(2) If a “*” occurs on the same line as the previous token, it is assumed to be an infix operator.

In the rare case when the programmer wants two statements on the same line, he or she would have to use parentheses to force the desired interpretation, as in:

```
x = 123   (*p) = 456
```

With a long infix expression involving “*” which must be spread over several lines, the programmer must code it like this:

```
x = aaaaaa *
       bbbbbb *
       cccccc
```

and not this:

```
x = aaaaaa
       * bbbbbb
       * cccccc
```

# Constants and Enumerations

## Constants

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
```

That example illustrated that **const** definitions may be grouped or may be independent. For example

```
const MY_CON = 500
const ANOTHER_CONSTANT = 30000
```

is equivalent to

```
const
  MY_CON = 500
  ANOTHER_CONSTANT = 30000
```

and the programmer can code it whichever way seems clearest.

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

```
int
double
bool
ptr to array of byte    -- 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
```

# Enumerations

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 = (183 % ONE_HUNDRED), B, C, D, E
const ONE_HUNDRED = 100
```

is equivalent to

```
const
  ONE_HUNDRED = 100
  A = 83
  B = 84
  C = 85
  D = 86
  E = 87
```

Both **const** and **enum** constructs may appear in either the header file or the code file. The placement of a **const** or **enum** determines the visibility of the names it defines. They follow the same visibility rules used by global variables, functions, type definitions, classes, interfaces, and error declarations.

To repeat the KPL visibility rule:

> *"If something is placed in a **code** file, then it is private and may only be used within that package. If it is placed in the **header** file, then it is shared and can be accessed by any package the uses the package where it was defined."*

Both **const** and **enum** constructs must appear at the top level in the header or code file. In other words, they cannot appear within a function, class, or method.

The recommendation is that the names defined **const** and **enum** definitions be fully capitalized, as in the above examples. They need not appear near the top of the file—although this is recommended—and their exact location in the header of code file is has no impact on the compiler.

# Errors and Try-Throw-Catch

## The Try and Throw Statements

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

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

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

Here is the grammar for the **try** statement. The body consists of sequence of statements, followed by one or more **catch** clauses. Each **catch** clause names an error—error names are simple identifiers—which is followed by a parameter list, which has the same syntax as a function or method parameter list. This header is followed by a colon and a sequence of statements known as the "catch clause statements".

```
try
  { Statement }*
  { catch ID ( Parameters ) : { Statement }* }+
endTry
```

Errors are said to be "thrown". When an error condition arises, some particular error will be thrown. For example, an attempt to divide by zero will result in an "arithmetic error being thrown. The precise name of this error is:

```
ERROR_ArithmeticException
```

An attempt to use a null pointer will cause this error to be thrown:

```
ERROR_NullAddress
```

As in Java, the body statements are executed first. If no error is thrown, none of the **catch** clauses is executed. Execution will continue with the statement following the **endTry** keyword.

However, 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. Whichever **catch** clause was selected will execute to completion and then execution will continue with the statement after the **endTry**.

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

The **try** statements may be lexically nested, like **while** or **for** loops. Also, code within the body of a **try** statement may call functions or invoke methods. The **catch** clauses that are active for the **try** statement will remain active until these functions and methods return and execution reaches the end of the body statements, or until an error is thrown.[23]

The only ways to leave the body statements of a **try** statement are:

---

[23] The best way to understand the try-throw-catch mechanism is to know that it is implemented using a stack. This "catch stack" is separate from the runtime stack of activation frames.

Each element on the catch stack represents an active catch clause. Whenever a **try** statement is executed, a record is pushed onto the stack for each catch clause, before the body statements are attempted. When the body statements complete without error, those records are popped off the catch stack.

There is one global catch stack (per thread), so if several **try** statements have begun but not finished, the catch stack will contain a number of records, with the records for the most recently entered **try** statement near at the stack top. When an error is thrown, the catch stack is search from top to bottom for the first catch clause that applies. Thus, the most recently entered **try** statement with a matching catch clause is the catch clause that is selected for execution. Just prior to executing the catch clause, both the catch stack and the runtime stack of activation frames are popped back to where they were when the original **try** statement was entered.

- Throw an error
- Complete execution and reach the end of the body statements
- Execute a **return** statement

The **break** and **continue** statements cannot be used to jump out of the body statements and their use is restricted accordingly.

The error handling statements in a **catch** clause may contain a **return** or **throw** statement, which will cause execution to leave the **try** statement. Since the catch stack will have been fully popped upon entry to the **catch** clause, the above-mentioned restrictions on the **break** and **continue** statements are lifted and they are usable as normal within the **catch** clause statements.

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

```
throw MY_ERROR (5)
```

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

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.

## Declaring Errors

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

```
errors
  MY_ERROR (i: int)
  error2 (a,b: String)
  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, and follow the KPL visibility rule. 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.

## Uncaught Errors and Debugging

It is possible for a method or function to throw an error that is uncaught. When this occurs, the runtime system will abandon the initial error and throw a second general-purpose error:

```
UncaughtThrowError
```

If this too is not caught, a fatal runtime error will occur and execution will halt.

This mechanism works quite well for handling runtime errors in KPL code. When an error condition occurs a specific error is thrown. If user code has chosen to catch that particular error, then the user code will continue running and perhaps recover in some fashion from the error condition. If the programmer has ignored that sort of error, then an “UncaughtThrowError” will occur. The user code made have a way to handle all errors and may catch handle this new error. But if the program again ignores the error, the default behavior is to halt program execution, print a descriptive error message, and invoke a debugger.

Most simple programs, where fault tolerance and high reliability are not great concerns, will not bother with catching errors.

Next, we show what happens when the “uncaught error” occurs. This was taken from a KPL program running on the Blitz-64 virtual machine emulator (with some minor editing).

You can see that the following information is included:

- The problem:
  "Arithmetic Exception" was thrown, but not caught.
- The current catch stack:
  Which errors are being actively caught.
- Where the error occurred:
  An ASSIGN statement on line 60 in function "main"   [examplePgm.c]
- Which functions and methods were in execution at the time
  (Highlighted in grey below)
- The machine instruction causing the exception:
  ADDI

```
====================  KPL PROGRAM STARTING  ====================
...
"System: ERROR_ArithmeticException" was thrown but not caught within thread "Main
Thread"

Here is the CATCH STACK:
  TestProgram: MY_ERROR
     address of catch record: 0x000000000FFFFF00
     catchCodeAddr:           0x000000000000BE8C
     prevSP:                  0x000000000FFFFDE8
     source file:             examplePgm.c (line 60)
==============================================================

*****  RUNTIME ERROR: An "ARITHMETIC EXCEPTION" has occurred!  *****
    Offending Instruction = 0x0000000001000177

*****  EMULATOR DEBUGGING: Type 'stack' for more info. *****

Execution is stopped at ASSIGN on line 60 in function "main"   [examplePgm.c]
      00000C574: 01000177       addi       r7,r7,1
Done!

Entering machine-level debugger...

Enter a command at the prompt.  Type 'quit' to exit or 'help' for info about
commands.
> stack
   Function/Method                  Execution at...     File
   ==============================   =================   ========================
   EmulatorDebuggingRequested                           KPL_lib/runtime.s
   invokeDebugger                   CALL    line 762    KPL_lib/System.c
   RuntimeErrorArithmeticExceptio   CALL    line 649    KPL_lib/System.c
   _runtimeErrorArithmeticExcepti                       KPL_lib/runtime.s
   AnotherFun                       ASSIGN  line 116    examplePgm.c
   MyClass.myMethod                                     examplePgm.c
   MyFunction                       SEND    line 110    examplePgm.c
   main                             CALL    line 58     examplePgm.c
   _kplEntry                                            examplePgm.c
   _entry                                               KPL_lib/runtime.s
```

Other systems may deal with uncaught errors differently. For example, an embedded system may simply turn on a red LED and freeze up.

# Naming and Scope Rules

## The Unique Name Rule

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 and everything must have a unique name.*

This includes:

- local variables
- parameters to functions, method, and errors
- constants
- types
- global variables
- functions
- classes
- interfaces
- errors

Even if two things are fundamentally different—such as a class and a parameter—they must be given different names.

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

```
type t = struct ... endStruct
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
```

# The Renaming Clause

One package may be "used" by other packages. In its header file, the package may define and export variables and other things for use in other packages.

A package's header file may define the following kinds of things. Any entity defined in the header file of one package is automatically exported for use in other packages that choose to use that package.

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

Within any single package, all of the above entities must have unique names, regardless of whether they are defined in that package or inherited from another package.

It doesn't matter whether any reference to the entity actually occurs in the package; every entity must have a unique name.

The **renaming** clause can be used to rename any of these, as necessary, to avoid name collisions.

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 }* ]
```

# Parameterized Classes

Classes may be parameterized.[24] 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".

[24] C++ has a similar construct, called "template classes". Java also has a similar construct, called "generic classes". KPL doesn't use the "template" terminology, since "template" implies copying and KPL avoids copying code. KPL doesn't use the "generic" terminology because "generic" implies lack of specificity. KPL's "parameterized" terminology seems more appropriate, since the type parameters in KPL are associated with constraints, which act to specify which types can be used in instantiations. Furthermore, this terminology makes it natural to talk about "type parameters" and "type arguments", which we need to differentiate. Also, the "parameterized type" terminology is analogous to function parameters and function arguments. We considered using the <> notation from C++ and Java or possibly «» or <<>>, but elected to go with the [] notation.

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)
  printf ("A Person with name = %s\n", p.name)
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.[25] 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

[25] The constraint type C for a type parameter (as in [T:C]) may be either (1) a class, (2) an interface, (3) the **anyType** type, or (4) a function type. The actual type provided when the class or interface is used must be a subtype of the constraint type. For example, the class "List [T: Person]" may be used, as in "var myStuLi: List [Student]" since Student is a subclass of Person. Likewise, if the class had been defined instead as "List [T: anyType]", then "var myIntLi: List [int]" would be allowable.

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

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.

# Conclusion

As its name reflects, the primary and initial application for KPL is writing OS kernel code. The features which have been included in the language have been selected with this in mind.

No programming language is perfect and no programming language can address all potential applications with equal facility. KPL is the embodiment of my opinions and biases about programming language design.

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-64 project is to create a practical and usable computer system. 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.

# Appendix 1: Predefined Functions

In this section we list and describe all the functions that are part of the KPL language core.

Some of these functions are recognized by the compiler either for efficiency reasons or because something about them requires special attention from the compiler. For example, consider the first argument to the function **asPtrTo**, which can be either an integer or a pointer. Since a normal function must have only a single type for each argument, the **asPtrTo** function must be handled specially by the compiler.

As another example, consider the second argument to **asPtrTo**, which is a Type. In KPL, types are not values and an attempt to pass a type (rather than a value) would be flagged by the compiler as a syntax error. So again, **asPtrTo** must be handled by the compiler.

Other functions are not treated specially by the compiler and many are not even recognized by the compiler. Instead, these functions are implemented directly, either in assembly code or in KPL.

| Built-in Function | Argument Type | Result Type |
|---|---|---|
| **upcastToHalfword** | byte/halfword | halfword |
| **upcastToWord** | byte/halfword/word | word |
| **upcastToInt** | byte/halfword/word/int | int |
| **upcastToDouble** | byte/halfword/word | double |
| | | |
| **asByte** | int | byte |
| **asHalfword** | int | halfword |
| **asWord** | int | word |
| | | |
| **forceToByte** | int | byte |
| **forceToHalfword** | int | halfword |
| **forceToWord** | int | word |
| | | |
| **forceToDouble** | int | double |
| **forceToInt** | double | int |
| **copyBitsToDouble** | int | double |
| **copyBitsToInt** | double | int |

| **asInteger** | ptr | int |
|---|---|---|
| **asPtrTo** | int/ptr , *Type* | ptr to *Type* |
| **ptrToBool** | ptr | bool |
| **isKindOf** | x , *Type* | bool |
| **isInstanceOf** | x , *Type* | bool |
| **sizeOf** | *Type* | int |
| **initializeObject** | *ObjExpr* | < *Statement* > |
| | | |
| **initializeArray** | *ArrExpr* | < *Statement* > |
| **setArraySize** | *ArrExpr*, int | < *Statement* > |
| **arrayMaxSize** | *ArrExpr* | int |
| **arraySize** | *ArrExpr* | int |
| | | |
| **CPUControl** | int, constant | int |
| **CPUControlUserMode** | int, constant | int |
| | | |
| **cas** | ptr to int, int, int | int |
| **fence** | () | < *Statement* > |
| | | |
| **isnan** | double | bool |
| **isNegZero** | double | bool |
| | | |
| **unsignedAdd** | int, int | int |
| **unsignedSub** | int, int | int |
| **addOk** | int, int | bool |
| | | |
| **initializeThreadPtr** | ptr | < *Statement* > |
| **threadPtr** | () | int |
| **remainingStackSpace** | () | int |
| | | |
| **endianSwapH** | int | halfword |
| **endianSwapW** | int | word |
| **endianSwapD** | int | int |
| | | |
| **setFloatingRound** | (int) | < *Statement* > |
| **resetFloatingStatus** | () | < *Statement* > |
| **floatingInexact** | () | bool |
| **floatingUnderflow** | () | bool |
| **floatingOverflow** | () | bool |
| **floatingZeroDivide** | () | bool |
| **floatingInvalid** | () | bool |
| **floatingClass** | double | int |
| | | |
| **MemoryCopy8** | | |
| **MemoryZero8** | | |

Important Definitions (implemented in runtime.s or System/Utility):

```
ThreadData          < A class >
CatchRecord         < A struct >
```

## upcastToHalfword, upcastToWord, upcastToInt

These functions are not normally used by the programmer, but they can be used if it makes the program easier to understand. Instead, they are inserted implicitly and automatically as needed to change integer values from a smaller type to a larger type. The value is sign-extended as necessary. Therefore, there is never a change in the numerical value. There are no error conditions and errors will never be thrown.

## upcastToDouble

This function is not normally used by the programmer, but it can be used if it makes the program easier to understand. Instead, it is inserted implicitly and automatically as needed to change an integer value into a floating point value. There is never a change in the numerical value. Since all 32 bit integer values can be represented exactly as double precision floating point values, this function may be used to convert them without possibility of error.

```
var d: double
d = 123        -- Same as d = 123.0
```

However, there are some 64 bit integer values which cannot be represented exactly as a **double** value. Therefore, this function may not be applied to arbitrary **int**s. See **forceToDouble**.

```
var i: int
d = i                             -- Compile-time error
d = 9_223_372_036_854_775_807     -- Compile-time error
```

## asByte, asHalfword, asWord

These functions may be used to change an integer value from a larger representation to a smaller representation. For example, an **int** value (e.g., the result of a computation) can be “downsized”, as in:

```
var b: byte
b = asByte (i+j)    -- May cause a runtime error
```

The value will be checked at runtime and if it is not within range, i.e., if it cannot be represented by the target type, an “Arithmetic Exception” will occur. Here are the ranges that can be represented:

| Type | Size in bits | Range of values |
|---|---|---|
| **byte** | 8 | -128 … 127 |
| **halfword** | 16 | -32,768 … 32,767 |
| **word** | 32 | -2,147,483,648 … 2,147,483,647 |
| **int** | 64 | -9,223,372,036,854,775,808 … 9,223,372,036,854,775,807 |

## forceToByte, forceToHalfword, forceToWord

Similarly to **asByte**, **asHalfword**, and **asWord**, these functions may be used to change an integer value from a larger representation to a smaller representation. For example, an **int** value (e.g., the result of a computation) can be “downsized”, as in:

```
var b: byte
b = forceToByte (i+j)    -- Never an error, but value may change
```

The difference is how values that are out-of-range are treated. These functions will never throw an error; instead the upper bits will simply be discarded. This may result is a change in value.

```
var
  i: int
  b: byte
i = 1234567             -- 0x0000_0000_0012_D687
b = forceToByte (i)     -- 0x87 = 0xffff_ffff_ffff_ff87 = -121
b = -121                -- Equivalent
```

## forceToDouble, forceToInt

The **forceToDouble** function may be used to change an integer value into a **double** representation. The **forceToInt** function be used to change a **double** value into a 64 bit integer **int** representation. In many cases, the conversion will be exact and the value will be unchanged.

```
var
  i: int
  d: double
i = 1234567
d = forceToDouble (i)    -- No problem; yields 1.234567e+6
```

In the case of **forceToDouble** , if there is no **double** value that represents the integer value exactly, then **forceToDouble** will yield an approximate value. That is, the value will be rounded to a nearby value that can be represented.

Any integer within the following range can be represented as a **double** with no loss of accuracy:

```
-9,007,199,254,740,992 ... +9,007,199,254,740,992
```

In the case of **forceToInt** , if the **double** value is outside the range of values representable as a 64 bit signed integer, then **forceToInt** will set the floating point "overflow" (OV) flag. No error will be thrown. For positive values, the result of the function will be the largest **int** value, and for negative values, the result of the function will be the most negative **int** value. Any value outside the following range will cause this overflow behavior:

```
-9,223,372,036,854,775,808 ... +9,223,372,036,854,775,807
```

If the argument is not an integer within the above range, this function will return a nearby value and will set the "inexact" (NX) floating point status bit.

## copyBitsToDouble, copyBitsToInt

Like **forceToDouble** and **forceToInt**, these functions are used to copy data from **int** variables to **double** variables and vice versa. However these functions do no conversion. The bits are simply copied. This will result in a completely different numerical value (except in the case of the **double** value +0.0 and the **int** value 0,

which just happen to be represented identically as 0x0000000000000000). These functions will never throw an error.

```
i = copyBitToInt (1.75)
i = 0x3ffc_0000_0000_0000     -- Equivalent

d = copyBitToDouble (0x3ffc_0000_0000_0000)
d = 1.75                   -- Equivalent
```

# asInteger

The **asInteger** function takes a single argument of type **ptr to anyType** and returns an **int**. The bits are unchanged and, since every address is limited to 36 bits, the result will be in the range

```
0x0,0000,0000 ... 0xF,FFFF,FFFF
(decimal 0 ... 68,719,476,735)
```

This function is "safe". However, this function can be used to make a program implementation-dependent and it can cause different executions to produce different results since addresses may change from execution to execution for various reasons.

The identifier "**asInteger**" is actually a keyword and this function is included directly into the syntax of KPL.

# asPtrTo

The **asPtrTo** function takes two arguments. The first argument can be either an integer or a pointer. The second argument is a type "T". The function returns a result which is of type "**ptr to** T".

The result of this function is exactly the same as the first argument; the bits are identical. The only difference is that the type of the result is altered.

This function is "unsafe" since an error in using it can lead to a crash or unpredictable behavior in code that would otherwise be able to handle errors properly.

The identifier "**asPtrTo**" is actually a keyword and this function is included directly into the syntax of KPL.

## ptrToBool

This function takes a pointer as an argument and returns **true** if the pointer is not **null**. It return **false** if the pointer is **null**.

```
if p ...
if p != null ...    -- Equivalent
```

## isKindOf

This function takes two arguments, as in

```
if isKindOf (p, Person) ...
```

The first argument (p) must be either an object or a pointer to an object. Here are examples that would work:

```
var p: Student
var p: ptr to Student
var p: ptr to MyInterface
```

The second argument must name a class or interface.

If the second argument is a class and the object is an instance of that class, or an instance of a subclass of that class, or a sub-sub class (and so on) of that class, this function will return **true**. Otherwise, it returns **false**.

Likewise, if the second argument is an interface and the object is an instance of a class that implements that interface, then this function returns **true**. Otherwise, it returns **false**. The class may implement the interface directly or indirectly through the **extends** hierarchy that relates interfaces and the **subclass** hierarchy that relates classes.

The **isKindOf** function is implemented with an assembly coded function which performs a search. The compiler places data in the read-only memory segment which includes information about the subclass, implements, and extends

hierarchies. For each occurrence of **isKindOf**, the compiler generates a call to this function, which uses that information.[26]

## isInstanceOf

This function takes two arguments, as in

```
if isInstanceOf (p, Person) ...
```

The first argument (p) must be either an object or a pointer to an object. Here are examples that would work:

```
var p: Student
var p: ptr to Student
var p: ptr to MyInterface
```

The second argument must name a class.

If the object is an instance of that class, this function will return **true**. Otherwise, it returns **false**. This function does not look at superclasses.

If the pointer is **null**, then this function throws **ERROR_NullAddress**. If the object is uninitialized, this function returns **false**.

This function is implemented directly in assembly code produced by the compiler (no KPL function is called) so it is fast.

The class name must be given without any type arguments, even if the class itself is parameterized. For example, assume the "List" is a parameterized class in the following:

```
isInstanceOf (p, List [Person])          Error
```

[26] The compiler will insert a call to a separate function to perform this operation. Initially, this function was implemented as an upcall to a function in the **System** package. For efficiency, it was reimplemented in assembly code. The assembly function requires approximately 11 + 5×n instructions to perform the search, where n is the number of superclasses and super-interfaces of the object's class.

The two versions differ slightly in how errors are handled. The main errors are (1) the pointer is null and (2) the object pointed to is uninitialized. The assembly version will cause a **Null Pointer Exception** in both cases.

```
isInstanceOf (p, List)                    OK
```

## sizeOf

This function takes a single argument, which must be a type, and returns an integer:

```
var i: int
i = sizeOf (Person)
i = sizeOf (double)          -- Will set i to 8
i = sizeOf (ptr to Person)   -- Will set i to 8
```

Note that in KPL "**sizeOf**" has different capitalization from C / C++, which use "sizeof".

The compiler will process this function and no code is actually generated. Thus, there are no possible runtime errors.

## initializeArray

Every array must be initialized before use, as in:

```
var a: array [10] of int
initializeArray (a)
```

The argument may also be a pointer, as in:

```
var arrPtr: ptr to array [10] of int
initializeArray (arrPtr)
```

Recall that every array has two values stored in a hidden header word. Each array contains a 64 bit header, located directly before the first element. Any pointer to an array will point to this header word, not to the first element. This header word contains two values, and each is a 32 bit signed number. Thus, the maximum number of elements in any array is 2,147,483,647.

- The MAXIMUM number of elements
- The CURRENT number of elements

This function sets both the MAXIMUM and the CURRENT to N, where N is determined by the compiler based on the type of the argument.

If the programmer wants a different CURRENT size, he or she can use **setArraySize**. This is a common pattern:

```
initializeArray (a)  -- Set MAX size
setArraySize (a, 0)  -- Set CURRENT size; empty the array
```

If the array has been initialized previously and this operation is repeated, it is not a problem. Calling **initializeArray** is a good way to “maximize” the array, i.e., to increase the CURRENT size to the maximum value.

Since this function will set the CURRENT size to the maximum value, it will effectively pick up anything currently in memory. All variables are initialized to zero values so this is no problem when the array is first initialized. But whenever an array is initialized a second time, the programmer is responsible for clearing out old values.

The argument may not be a pointer to a dynamic array, since the compiler cannot determine the array size.

If the argument is a simple variable, then this is a **safe** operation.

However, if the argument is a pointer, then this is not **safe**. To understand why, consider this example.

```
var
  p: ptr to array [23] of int
  myArr: array [7] of int
p = & myArr
initializeArray (*p)  -- Will set size of myArr to 23!!!
```

This code will set the MAX size of myArr to an erroneous value, allowing the overwriting of whatever follows myArr. (Although the assignment to p would be flagged as an error, there are other ways to do the same thing.)

This function will throw **ERROR_InitializingArray** if the previous array MAX size was not 0 (uninitialized) or the correct MAX size (initialized). This function will also check that the previous CURRENT value was within range and throw this error if not.

This function is predefined and built in to KPL. It is impossible to implement this function in KPL since there is no single type that will subsume all arrays.

## setArraySize

Every array has a CURRENT size, which can be adjusted with this function:

```
var
  a: array [10] of int = ...
setArraySize (a, 7)
```

The first argument is any expression which evaluates to either an array or a pointer to an array. The second argument must be an integer.

The array must be initialized before this function is called and, if not, the error **ERROR_SetArraySize** will be thrown.

Every array also has a MAXIMUM size. The new current size must be within the range

```
0 ... MAX-1
```

If not, **ERROR_SetArraySize** will be thrown.

## arrayMaxSize

Every array has a MAXIMUM size and a CURRENT size. This function returns the MAXIMUM size.

```
var
  a: array [10] of int = ...
i = arrayMaxSize (a)
```

This function returns the value stored in memory, not the compiler determined size.

The argument is any expression which evaluates to either an array or a pointer to an array.

If the array is uninitialized, this function will return 0. If the array is a pointer and that pointer is null, a **Null Pointer Exception** will occur.

## arraySize

Every array has a CURRENT size and a MAXIMUM size. This function returns the CURRENT size.

```
var
  a: array [10] of int = ...
i = arraySize (a)
```

The argument is any expression which evaluates to either an array or a pointer to an array.

If the array is uninitialized, this function will return 0. If the array is a pointer and that pointer is null, a **Null Pointer Exception** will occur.

## initializeObject

Every object must be initialized before use. Normally this is done with either:

```
var x: Person = new Person
```

or:

```
var p: ptr to Person = alloc Person
```

In a few situations, these methods cannot be used. An alternative is to use this function:

```
var p: ptr to Person
initializeObject (p)
```

The argument must be a pointer which must point to an area of memory large enough to accommodate the object. Furthermore, the memory really should be zero-filled, since all other code assumes that the object's fields will be initialized to their zero values.

This function will simply set the object's header word to point to the dispatch table. it will not modify any other fields.

**NOTE:** If the memory is the result of a call to **MemoryAlloc**, there is no guarantee that it will be zeroed. The programmer is warned: **initializeObject** will not zero the

object's fields. Since memory returned by **MemoryAlloc** will almost always contains zeros, this bug will be difficult to spot.

The argument is an address which must be doubleword aligned. The compiler will use the type of the argument to determine the class to use.

This function is predefined and built in to KPL. It is not safe.

## CPUControl and CPUControlUserMode

The Blitz-64 hardware contains two machine instructions which are intentionally left "undefined". Individual implementations of the Blitz-64 core will elect to use these instructions differently. In some implementations, these instructions may not be used at all.

Here is the form of the machine instructions:

```
CPUControl            RegDest,RegSource,Value16
CPUControlUserMode    RegDest,RegSource,Value16
```

The "Value16" field is a 16 bit operation code which indicates which operation is to be performed. Its precise meaning is left as "implementation dependent" and up to individual core designers. Consult

*Blitz-64 Instruction Set Architecture Reference Manual*

for details and examples of how these instructions might be used.

These **CPUControl** and **CPUControlUserMode** functions are provided in order to allow programmers to use these instructions. For example:

```
var i: int
  i = CPUControl (x, 23)
  i = CPUControlUserMode (x, 23)
```

The first argument may be any integer-valued expression. The second argument must be a constant integer expression. In other words, the compiler must be able to determine the value of the second argument; it may not be a variable. Furthermore,

the second argument must be within -32,768 … +32,767. This integer will be used as the “Value16” operation code in the instruction.

These functions are implemented by the compiler. The 16 bit operation code is not in a register: it must be statically determined at compile-time. Since the operation code must be known before runtime, it cannot be passed and plugged in at runtime. Therefore these functions cannot be implemented as external assembly routines.

## CAS: Compare and Swap

To access the CAS (compare-and-swap) machine instruction, a builtin function named **cas** is provided.

The **cas** function behaves as if it has the following definition, with the proviso that the entire operation is executed atomically.

```
function cas (p: ptr to int, old: int, new: int) returns bool
  if *p == old
    *p = new
    return true
  endIf
  return false
endFunction
```

When **cas** is used, the compiler inserts the CAS machine instruction inline, avoiding the procedure call overhead.

The **cas** function can be used to implement a spin lock.

For example, assume **myLock** is a doubleword which is used to represent a mutex lock where 0=unlocked and 1=locked. The following code spins if the lock is unavailable:

*Acquire the lock…*
```
  while !cas (&myLock, 0, 1) endWhile
```
*Execute critical section…*
```
  ...code...
```
*Release the lock…*
```
  myLock = 0
```

In KPL, the store operation is atomic since all variables are aligned. Thus, the lock can be released with a simple assignment statement.

## The “fence” Memory Barrier

To access the FENCE machine instruction, a builtin function named **fence** is provided, but the function does more than just that.

Programmers naturally and implicitly assume that expressions and statements in a programming language are evaluated and executed in the order they appear in the program.

To increase performance, modern processors often execute instructions out of sequence. Any “out-of-order” execution is considered acceptable and correct as long as the effect is identical to executing them in the sequence they appear.

The correctness definition just given makes the implicit assumption that there are no other processors. However, in a multiprocessor system with shared memory, this is not true and, in some situations, the results can be unexpected, counterintuitive, and incorrect. Programmers generally assume the operations in their code will be executed in the sequence written and this is normally not a problem. But with concurrent algorithms and out-of-order execution, subtle and difficult race bugs can arise if this issue is ignored.

For example, in the code snippet given above for the **cas** built-in function, we used a STORE operation to release a mutex lock. If this instruction is executed out-of-order and before some statements in the critical section are completed, a race bug occurs.

The built-in **fence** function does two things.

First, it causes the insertion of a FENCE machine instruction. The FENCE machine instruction forces critical instructions to complete before other instructions begin. Therefore, FENCE constrains and limits out-of-order execution in the processor

core.[27] In particular, the FENCE instruction affects instructions that LOAD from or STORE to memory, including the CAS instruction.

Second, the **fence** function tells the compiler to avoid keeping shared variables in registers. Instead, the function will cause the compiler to write all shared variables that are in registers back to memory before the **fence**. It will also cause the compiler to reload all shared variables it needs from memory after the **fence**. In other words, the **fence** prevents the compiler from keeping shared variables in registers across the invocation **fence**.

By "shared variables" we mean any global variable (i.e., variables that are declared outside any function or method) as well as any data pointed to. Local variables and parameters are not affected unless the address-of operator (&) is applied to them at some point.[28] Thus, it is permissible for the compiler to keep local variables and parameters in registers across the use of **fence**.[29]

Perhaps you might suggest that the **fence** operation could be implemented as an external function. The idea is that the function would be coded in assembly and would simply execute the FENCE machine instruction and return. However, the **fence** operation may be used in spin-locks and other performance-critical code so we do not use a separate function. Instead, the FENCE instruction is inserted inline.

[27] Using FENCE may introduce delays and pipeline bubbles but—since it is a question of correctness and is rarely used anyway—this is irrelevant.

[28] If the address-of operator is never applied to a local variable, the program will never have a pointer to that variable. This means the local variable can only be accessed within one invocation of the function in which it appears. It cannot be accessed by multiple concurrent threads.

[29] I believe the combination of these functions into one operation is my invention. In other languages multiple mechanisms are used, including "volatile" variables and incantations like "**__asm__ __volatile__ ("" ::: "memory");**" and "**void _mm_mfence(void)**".

# Appendix 2: Printing with “printf”

## Introduction

KPL provides an output system similar to C / C++. The following two functions are built in to the language:

```
printf
sprintf
```

In order to use these functions in a package, the package must use “PrintPackage”:[30]

```
header MyPack
  uses System, PrintPackage
  ...
endHeader
```

This section provides an overview of **printf** and **sprintf**. Their full functionality is described in:

*“Blitz-64: Software Reference Manual”*

## Examples

The following KPL statements show that many features familiar from C / C++ are the same in KPL:

```
printf ("int value = %d\n", i)
printf ("double = %g  %.6e   %.6f\n", d, d, d)
printf ("%s  %10d  \n\t  %016.16x\n", str, i, i)

sprintf (str, "Hello %s\n", userName)
```

[30] If a package doesn’t invoke **printf** or **sprintf**, then there is no need to use **PrintPackage**.

KPL does not use **fprintf**. Instead, the **printf** function can optionally begin with a **FileID**, as in:

```
printf (stdout, "int value = %d\n", i)
printf (stderr, "int value = %d\n", i)
printf (myFile, "int value = %d\n", i)
```

## Format Codes

The **printf** and **sprintf** functions are unusual in that they can take a variable number of arguments. All other KPL functions have a fixed number of arguments.

The two forms of the **printf** statements are:

```
printf (FileID, FormatString, arg1, arg2, arg3, ...)
printf (FormatString, arg1, arg2, arg3, ...)
```

The general form of **sprintf** is:

```
sprintf (ID, FormatString, arg1, arg2, arg3, ...)
```

The following format codes are recognized:

```
%d     print an int in decimal
%s     print a String
%c     print a single character
%x     print an int in hex
%e     print a double number
%f     print a double number
%g     print a double number
%b     print a bool value
%h     print a halfword value in binary
%w     print a word value in binary
%i     print a int value in binary
%o     print an object's class name
%%     print %
%(     print (
```

For details, see:

*"Blitz-64: Software Reference Manual"*

The compiler will check the *FormatString* and each of the format codes. For each, it will verify that the format code is specified correctly. The compiler will also verify that for each format code, there is a corresponding argument of the correct type.

The general form of a format code is:

```
%  [ Flags ]  [ Width  ]  [ . Precision ]  FormatCharacter
```

For example

```
printf ("  %d   %-#20.8x   ", i, j)
```

The following *Flag* characters are recognized:

```
-   0   #
```

The *Flag* characters are optional. They may be specified in any order, but there must be at most one occurrence of each. The meaning of "-" is "left-justify within the field". The meaning of the other characters depends on the *FormatCharacter*.

The *Width* is an integer. More precisely, the *Width* is a sequence of decimal digits and it may not begin with a 0. This integer specifies the **field width** in which the data will be printed. Generally speaking, padding bytes are added as necessary to fill out to the full field width.

The *Precision* is an integer, i.e., a sequence of decimal digits. The meaning of *Precision* is dependent on the *FormatCharacter*.

Just as in C / C++, for each format code (such as "**%d**") there must be exactly one argument. The arguments following the *FormatString* are matched up, in order, with the format code.

KPL also allows a second form, which may be unfamiliar to C / C++ programmers. If the argument is a simple identifier, then the argument may be embedded directly in the *FormatString*.[31] Parentheses are used for this.

The following two are equivalent:

---

[31] Between the parentheses only a simple ID is allowed; full expression syntax is not allowed.

```
printf ("val1 = %d  val2 = %d\n", xxx, yyy)
printf ("val1 = %d(xxx)  val2 = %d(yyy)\n")
```

## Differences With C / C++

The **#** flag can be used with **%d** to insert separators. For example:

```
printf ("val = %d = %#d", 1000000000, 1000000000)
```

will print

```
val = 1000000000 = 1,000,000,000
```

The character to be used defaults to comma (,) but this can be changed in **PrintPreferences**.

The **#** flag can be used with **%x** to prefix a hex number with "0x". For example:

```
printf ("hex = %x = %#x", 1000000000, 1000000000)
```

will print

```
hex = 3b9aca00 = 0x3b9aca00
```

With **%c**, the value to be printed is a character, and in KPL this includes any Unicode codepoint. For example:

```
var i: int
i = 'a'
printf ("%d = %c\n", i, i)
i = '€'
printf ("%d = %c\n", i, i)
```

will print

```
97 = a
8364 = €
```

With **%x**, the *Precision* value can be used to truncate a hex value. For example:

```
printf ("%0.4x", 0x0000000000007fff)
```

will print:

```
7fff
```

However, only sign extension bits can be removed this way. For example:

```
printf ("%0.4x", 0x0000000000008003)
```

will cause an error. However, the programmer might write this instead:

```
printf ("%0.4x", forceToHalfword (0x0000000000008003))
```

This will change the value to `0xffff_ffff_ffff_8003` and then print:

```
8003
```

With **%e**, **%f**, and **%g** there are some subtle differences with C/C++ but nothing too disturbing.

The following are controlled by **PrintPreferences**, and are used for **%e**, **%f**, and **%g**. These can be changed if desired.

| **value** | **default output** |
|---|---|
| +inf | `<pos infinity>` |
| -inf | `<neg infinity>` |
| nan | `<not-a-number>` |
| +0.0 | `0.0` |
| -0.0 | `-0.0` |
| positive numbers | *do not print "+"* |
| decimal point | *. (i.e., the period character)* |
| separator | *, (i.e., the comma character)* |

With **%f**, the value is always printed in the form

*digits* **.** *digits*

The *Precision* is the number of digits to the right of the decimal point. For example:

```
printf ("%f   %.6f", d, d)
```

might print:

```
1000.3   1000.333333
```

With **%e**, the value is always printed in "exponential form". The *Precision* is the number of significant digits to be printed, defaulting to 17. For example:

```
printf ("%e   %.6e", d, d)
```

might print:

```
1.0003333333333334e+3 (inexact)   1.00033e+3 (inexact)
```

The postfix of "(inexact)" is added whenever the value printed is not exactly equal to the **double** value.[32]

The **%g** format will choose to print in exponential form sometimes, and sometimes not, in an effort to display the value in the most human-friendly way. For example:

```
printf ("%g   %.6g\n", d, d)
```

might print:

```
1000.333333333333   1000.33
```

Here is an example that shows the benefit of using **%g** over **%e**:

```
printf ("e = %e\n", d)
printf ("g = %g\n", d)
```

Here is the output for some value of d:

```
e = 1.234567e+6
g = 1234567.0
```

And here is the output for a different value of d[33]:

```
e = 3.9999999999999996e+0 (inexact)
```

[32] This is the default, but it can be changed with **PrintPreferences**.

[33] Both lines are applied to the same value, and neither is a perfectly accurate representation.

```
g = 4.0
```

As illustrated here, **%g** does not add the "(inexact)" postfix.

With **%e**, **%f**, and **%g**, separators will be printed if the **#** flag is present. For example:

```
printf ("e = %#e\n", d)
printf ("f = %#f\n", d)
printf ("g = %#g\n", d)
```

might print:

```
e = 1.234_567_89e+8
f = 123,456,789.0
g = 123,456,789.0
```

By changing **PrintPreferences**, we can get the following output instead, which is more in the European style:

```
e = 1,234 567 89e+8
f = 123.456.789,0
g = 123.456.789,0
```

# Appendix 3: Alternate Method Syntax

## 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 character sequences may not be defined as methods:

```
/*   */   --   +=   -=   =
==   !=   &&   ||   +    -    *    &    !
```

## 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
```

This is functionally equivalent to invoking a method with the standard syntax, if function names were allowed to contain colons. However, it seems unusual because the method name is split apart after each colon.

```
x.atPut (y, z)   -- Same functionality, using traditional syntax
```

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.

# Appendix 4: Style Recommendations

This appendix includes recommendations on how to write KPL code in a standardized way. Following these recommendations makes your code easier to read by both you and others.

It is important to form style habits early in the use of any new programming language.

As with any style guide, we include the advice to follow these rules as much as possible but only when they make sense. When you can do better, do so and don't be a slave to the rules. But also remember that—for programs—clarity, simplicity, and readability are paramount; creativity and originality for their own sake are bad.

Write and edit your code using a fixed-width font, such as

```
Courier:     abcdeABCDEF012345
Courier New: abcdeABCDEF012345
Monaco:      abcdeABCDEF012345
Menlo:       abcdeABCDEF012345
```

Align all "**end**…" keywords directly under the corresponding opening keyword. Indent everything between the opening keyword and the "**end**…" keyword by two additional spaces.

```
if Condition
  if Condition
    Statements
  elseIf Condition
    Statements
  else
    Statements
  endIf
else
  Statements
  for (…)
    Statements
  endFor
  Statements
endIf
```

Always indent in increments of two spaces.[34]

Do not use tabs for indentation. In fact, do not use the tab "\t" any where in your code.

Place the following keywords in column 1:

```
header
endHeader
code
endCode
```

Indent two spaces and place the following keywords in column 3:

```
uses
const
type
var
functions
errors
class
endClass
interface
endInterface
function
endFunction
```

For example:

```
header MyPackage
  uses System
  const
    MAX = ...
  var
    x: ...
  functions
    foo ...
    bar ...
endHeader
```

[34] We prefer two spaces because it is enough to be clearly visible, yet small enough to prevent highly indented material from disappearing off the right side of the display.

Put every statement on a different line.

Indent the **switch** statement like this:

```
switch Expr
  case Value:
    Statements
  case Value:
    Statements
  default:
    Statements
endSwitch
```

Indent the **try** statement like this:

```
try
  Statements
catch ERROR (…):
  Statements
catch ERROR (…):
  Statements
endTry
```

Use "--" for all comments.

Do not "/* ... */" comments, except for disabling larger blocks of code.

Begin all variables with a lowercase letter. Use "camel case" for capitalization.

```
Right                         Wrong
=======================       =======================
anExampleVariableName         an_example
personCount                   personcount
```

For boolean variables, make the variable name resemble a true/false statement. Name boolean variables so that they make sense within an IF statement. Using the word "is" often works well.

```
Right                        Wrong
=======================      =======================
moreLeft                     terminate
fileIsEmpty                  fileStatus
isArrayFull                  arrayCondition
```

This makes your code more readable.

```
Easy to read                 Hard to read
=======================      =======================
while moreLeft               while terminate
  …                            …
endWhile                     endWhile
if fileIsEmpty               if fileStatus
  …                            …
endIf                        endIf
isArrayFull = true           arrayCondition = true
```

Choosing good names for things is critically important. Do not hesitate to rename your variables, functions, classes, and methods if you can think of a more descriptive name.

Do not worry about the length of names; long names are not a problem.

```
Difficult to read and understand:
=================================
  pcnt = numM + numW + numUn

Much better:
============
  personCount = numberOfMen + numberOfWomen + numberUnknown
```

Begin all type and constant names with an uppercase letter:

```
MyClass
MyType
```

Write all constants and error IDs in all uppercase, using underscore to separate words.

```
MAX_SIZE
```

Function names and method names can begin with an uppercase letter or not, as you choose.

```
ComputeTaxes
computeTaxes
```

Make all identifiers a sequence of properly spelled words.

```
Right
=======================
  sizeOfArray
  vehicleCount

Wrong
=======================
  x45j            -- Avoid non-words
  vehicCnt        -- Avoid abbreviations; Spell words out
```

Use small names for variables that are only used locally in one small region of code.

```
for i = 0 to MAX_SIZE-1
  a[i] = 999
endFor
```

For the local variables in a function or method, indent each line by two spaces.

```
var
  i, j: int
  sum, average: double
  personPtr: ptr to Person
```

Exception: If all variables can be put on one line, then it is okay to just use one line.

```
var i, j: int
```

Follow every comma by a single space.

```
Right                           Wrong
=======================         =======================
var i, j, k: int                var i,j,k: int
foo (x, y, z)                   foo (x,y,z)
```

Precede each opening parenthesis by a single space, unless it is another opening parenthesis. Do not follow the opening parenthesis with a space. Same for closing parentheses.

```
Right                           Wrong
=======================         =======================
foo ((x + y) * z))              foo( ( x+y )*z ) )
```

Surround every operator by a single space.

```
Right                           Wrong
=======================         =======================
foo ((x + y) * z))              foo ((x+y)*z))
i = 0                           i=0
```

Use the same style for braces { } and brackets [ ].

Concerning the dot operator, do not surround it with spaces.

```
Right                           Wrong
=======================         =======================
p.myMeth (x)                    p . myMeth (x)
p.myfield = x                   p . myfield = x
```

Concerning the dereference (*) and address-of (&) operators, add spaces at your discretion.

```
*p = &x                        -- Okay
* foo (i) = & bar (j)          -- Also okay
```

Each function should be commented in the same way. Same for methods. See the example below.

To aid scanning, insert three blank lines between functions. Then place a single line with the function name surrounded by a bunch of horizontal dashes like this:

```
    ------------------------  foo  ------------------------
```

This makes it much easier to find a function and to scan large blocks of code quickly.

Follow this with the function header. After the header, insert a “block comment” to describe the function.

The block comment is primarily meant to be read by someone wanting to use or call the function. It is intended for someone who is completely unfamiliar with the function. Write the block comment at a level of detail that makes sense for the function’s user. Begin with very basic information about the function. The block comment is a summary. Imagine that the reader wants to use the function but doesn’t want to read to function’s code.

This block comment should include this information:

- What the arguments are and what they mean / represent.
- What the function does; what it computes.
- If the function returns anything, describe what it returns.
- Discuss all error conditions that might arise.
- Describe any application-specific errors it might throw and why.

If the algorithm is complex, describe it. But put this description after the introductory material. Generally speaking, the function’s user wants to know what the function does, not how it does it. But some functions involve complex algorithms which need to be explained. Furthermore, the algorithm employed may be important to the user of the function.

Indent the block comment as shown, so the function header stands out. If the function is lengthy, add the function name to the **endFunction** line.

Here is an example of the recommended style:

```
code
…

-----------------------  foo  -----------------------

  function foo (x, y: int) returns int
    --
    -- This function is passed "x", which is… and "y"…
    -- It computes … and returns …
    -- In the case of the … error, it returns …
    -- If … is true, this function will throw error …
    --
    -- The algorithm used by this function is …
    --
    var
      i, j, …
    if …
    return …
  endFunction -- foo



-----------------------  bar  -----------------------

  function bar (...) …
    ...
  endFunction

endCode
```

Within a function or method, comments are used for two things. First, a comment may be necessary to include information that is important but not obvious from the code. Second, a comment is used to make the code easier to read and to help the reader to find whatever he or she is looking for.

These functions are very different and should be commented in different ways.

First, consider a comment that is needed to add important information. Generally, this is a short comment that can be added directly to the statement it applies to.

```
i = person.age()
j = person.computeTax (r)    -- r is the tax rate
sum = sum + j
```

The second function is to support the reader who is scanning the code looking for something, or trying to get an understanding of the algorithm it implements. To aid the reader, these comments need to be spread even throughout the entire function or method.

The recommendation is to add comments inline, indented exactly the same as the statements they apply to. Each comment should apply to one or a few statements following it.

```
-- Initialize the …
for i = 0 to MAX-1
  a [i] = -1
endFor
sum = 0

-- Perform the calculation …
for i = 0 to MAX-1
  a [i] = b [i] + sum
  sum += a [i]
endFor
```

Some functions and methods are so short and simple that they don't need inline comments. They still need a block comment.

```
------------------------  foo  ------------------------

function foo (x, y: int) returns int
  --
  -- Passed x = … and y = …
  -- Compute … and return it.
  --
  var i: int
  i = (x + 3) * (y + 4)
  return i * i
endFunction
```

However, if you are commenting blocks of statements, always follow these rules:

- Make the comment apply to all statements after it.
- Comment everything at the same degree of detail.

Below is an example of what not to do. The comment doesn't completely describe the statements that follow it. It only applies to the first part. A comment describing the second part is missing.

```
Wrong
=======================
  -- Initialize the …
  for i = 0 to MAX
    a [i] = -1
  endFor
  sum = 0
  for i = 0 to MAX
    a [i] = b [i] + sum
    sum += a [i]
  endFor
```

The important thing is the calculation. Since the initialization is really a part of that, it is acceptable to leave the comment about it out.

```
Right
=======================
  -- Calculate the …
  for i = 0 to MAX
    a [i] = -1
  endFor
  sum = 0
  for i = 0 to MAX
    a [i] = b [i] + sum
    sum += a [i]
  endFor
```

Don't comment what is obvious.

```
Wrong
=======================
  -- Add 3 to i
  i = i + 3
```

Keep all source code lines short. By short, I mean under 80 characters. This is so every line will be displayed properly, regardless of the width of the screen window. Note that long strings can be broken into pieces.

```
Wrong
=======================
  printf ("Now is the time for all good men to come to the aid of
  their party.\n”)

Right
=======================
  printf ("Now is the time for all good men "
          "to come to the aid of their party.\n”)
```

Try to line up things so the addition of new elements will not require adjustment. Try to line up things so that the change of spelling will not require adjustment.

```
Good examples
=======================
  xxx = foo (xxx, yyy, zzz)
  xxx = foo (
    xxx,
    yyy)
  xxx =
    foo (
      xxx,
      yyy)
```

The following style is problematic:

```
Not recommended
=======================
  xxx = foo (xxx,
             yyy,
             zzz)
```

If we change the spelling of some identifiers, it destroys the alignment of the subsequent material:

```
Inconsistent alignment
=======================
  xxx_2 = foo_2 (xxx,
             yyy,
             zzz)
```

KPL does not require parentheses for statements such as **if**, **while**, and **switch**. Don't use unnecessary parentheses.

```
Right                          Wrong
=======================        =======================
while p                        while (p)
  …                              …
  p = p.next                     p = p.next
endWhile                       endWhile

if i == j                      if (i == j)
  …                              …
endIf                          endIf

return i + j                   return (i + j)
```

Within an expression involving several operators, it's okay to use extra parentheses to make it clear which operators are done first, even if they are not strictly necessary.

```
i = (*x.meth()) + y
i = *x.meth() + y                -- Equivalent; either is okay

if (a && b) || ((!c) && (!d))
if a && b || !c && !d            -- Equivalent; either is okay
```

# Appendix 5: Memory Management

## Stack Usage

Like many languages, KPL utilizes a runtime stack. Whenever a function or method[35] is invoked, a **stack frame** is allocated. Stack frames are sometimes called "**activation records**".

The stack frame is where the local variables and temporary variables of the function are stored. Often, parameters must be stored in memory and these are also placed in stack frames. The return address is stored in the stack frame as well.

A stack frame is created when a function is invoked (i.e., when the function is called). The stack frame is destroyed and deallocated when the function returns. Stack frames are allocated by **pushing** onto the stack and they are deallocated by **popping** off the stack.

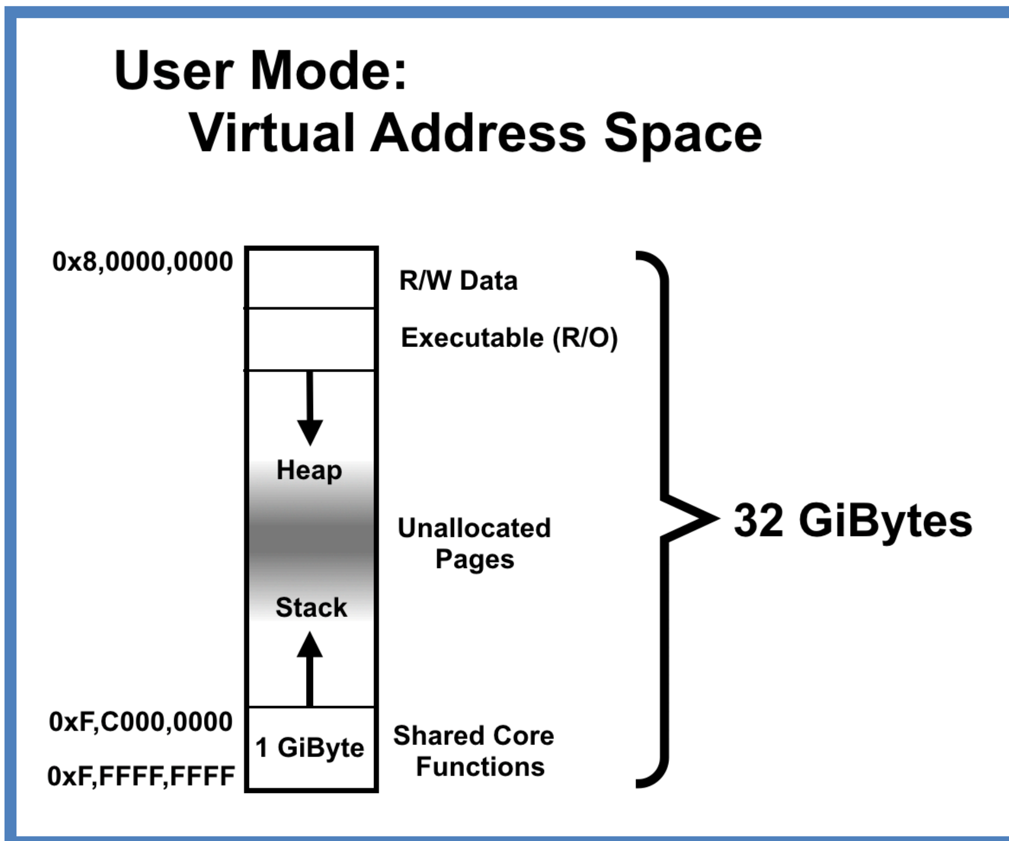
KPL and the Blitz-64 architecture have been carefully designed so that simple functions can often avoid allocating a stack frame. A **leaf function** is a function that does not invoke any other functions. A leaf function that requires no temporary storage can often avoid creating a stack frame and, in many cases, can avoid touching memory altogether. Since this discussion is about stack space, we will not mention leaf functions any further.

The stack frame is created by the initial code in the function, which is called the **function prologue**. This code will also zero out all the local variables, as required by KPL. The stack top is pointed to by the **stack top pointer**, which is kept in register r15 ("**sp**"). The stack grows downward from high memory so allocating a stack frame is achieved by subtracting the frame size from **sp**. Deallocating a frame is achieved by adding that same value back to **sp**.

Many programs also require **dynamically allocated memory on a heap**. We discuss heap strategy elsewhere, but in the simplest approach, the heap and the stack share a region of the virtual address space. The stack grows downward from

[35] In this discussion, functions and methods are treated the same. To keep it simple, we'll discuss functions, but everything we say also applies to methods.

the top of the space (i.e., from higher addresses) and the heap grows upward from the bottom of the space (i.e., from lower addresses). This way, the available space may be used for either stack or heap, as needed by the program. When the stack and the heap meet, then the program has run out of memory.




The memory space required for the stack will depend on

- How large the stack frames are
- The calling pattern of the functions

A function is said to be **recursive** if it can be invoked a second time before the previous invocation has returned.

For example, if a function "f" can be invoked recursively and there can be 1,000 invocations active at once, and the stack frame for "f" requires 200 bytes, the stack will need at least 200,000 bytes.

In general, the compiler cannot determine ahead of time how many bytes are required for stack space. This is not a human failure; it is a theoretical limitation that

cannot be overcome in all cases. For recursive programs, the amount of stack space required may be determined as a result of computation performed at runtime, which is impossible to compute at compile-time. For most recursive programs, any analysis of stack space usage is not practical and we will not discuss recursive programs here.

However, programs that are not recursive can be analyzed and a **maximum stack space usage** can be determined before runtime. This can be useful in programs requiring high-reliability, since we can preallocate a fixed amount of stack space while guaranteeing that the stack can never overflow this region. It it also required for programs that must run in severely limited memory systems.[36]

For a program that has **no recursion**, we can compute the stack frame sizes for each function[37]. By analyzing which functions call which functions, we can compute a total stack size limit. The program will never require any more storage than this, allowing us to put a bound on how much stack space is required for many useful programs. Many useful programs are not recursive and the stack bound for non-recursive programs is usually quite small.

## The Max_Stack_Usage Clause

KPL contains a syntactic mechanism by which the programmer can specify and limit the maximum stack usage for each function and method. For example:

```
function foo (x, y: int) returns int [ Max_Stack_Usage = 100 ]
  ...
endFunction
```

Unlike other keywords, **Max_Stack_Usage** is capitalized and contains underscores.

[36] At this date, we are putting Linux in toys and low cost appliances so it would seem that this concern is becoming outdated. But for systems with large numbers of cores, the space used for memory will necessarily reduce the space available for processing, so the prediction is that memory economy will remain important in the future.

[37] A clever compiler can easily determine whether a program contains recursion. A compiler can determine the size of every stack frame for KPL functions and methods, although this is not possible in some languages.

The brackets [ ] are literal, i.e., they are included, although the entire clause is optional. The general syntax is:

```
function ID (Args) [ returns Type ] [ '[' Max_Stack_Usage = Expr ']' ]
```

Normally the *Expr* is an integer constant. In any case, it must be possible for the compiler to evaluate it.

If the **Max_Stack_Usage** clause is used, it must appear on both the function prototype and the function itself and the values must be equal.

```
functions
  foo (x, y: int) returns int [ Max_Stack_Usage = 100 ]

function foo (x, y: int) returns int [ Max_Stack_Usage = 100 ]
  ...
endFunction
```

The **Max_Stack_Usage** clause may also be used on methods. If it is used on a method, the clause must appear only on the prototype, not the method itself:

```
class ...
  methods
    myMeth (x, y: int) returns int [ Max_Stack_Usage = 100 ]
  ...
behavior ...
  method myMeth (x, y: int) returns int
    ...
  endMethod
  ...
```

The **Max_Stack_Usage** clause may also appear on function types, as in:

```
ptr to function (int, int) returns int [ Max_Stack_Usage = 100 ]
```

If a **Max_Stack_Usage** clause appears, the compiler will enforce it. In other words, the compiler will ensure that the function meets the specified stack usage and cannot possibly exceed the limit. The compiler will print an error message if it cannot guarantee the limit. This requires that any method or function with a **Max_Stack_Usage** clause may only invoke methods and functions that also have **Max_Stack_Usage** clauses attached to them.

Stack management and the **Max_Stack_Usage** clause are discussed more fully in the document:

> *"Blitz-64: Software Reference Manual"*

## The Memory Heap

A **memory heap** is a region of memory that can be allocated dynamically at runtime, as needed. In KPL there are two ways to allocate memory on the heap.

First, the **alloc** clause can be executed, as in:

```
objPtr = alloc MyClass { ... }
arrPtr = alloc array of int { k of -1 }
```

The **alloc** clause can be used to allocate space on the heap for objects, arrays, structs, and unions.

Second, the program can explicitly call the function **MemoryAlloc**:

```
var p: ptr to byte
p = MemoryAlloc (n)   -- n = number of bytes to allocate
```

The **MemoryAlloc** simply returns a pointer to the new memory region. Any memory allocated on the heap is _not_ guaranteed to be zeroed before use. The **alloc** construct allows for additional type-specific initialization and the allocated data is guaranteed to be initialized with zero values if no initialization is present.

If the heap is full and no more memory remains, the error **ERROR_HeapFull** will be thrown.

To deallocate the space, the **free** statement can be used, or the function **MemoryFree** may be called:

```
free objPtr
free arrPtr
...
MemoryFree (p)
```

The **free** statement simply invokes **MemoryFree**. Either technique can be used to return memory to the heap.

Since prematurely freeing objects that are in use can cause bugs and behavior that is impossible to specify, the use of **free** statement requires the program to be compiled with the **-unsafe** option. Calling functions like **MemoryFree** does not require the **-unsafe** option, although it is exactly equivalent and can cause the same bugs.

The recommendation is to always use the **free** statement and avoid explicitly calling **MemoryFree**.

As in C / C++ there are no checks on pointers in KPL, beyond the ever-present checking for **null** and for proper alignment. As in C / C++, objects in the KPL heap will not be moved around in memory and the actual memory addresses are visible to the code.

For example, the following sort of thing is allowed in KPL, and can be useful, for example, in accessing memory mapped I/O registers.

```
i = * asPtrTo (0x4_0000_4f00, int)
```

In contrast, languages like Java hide the actual memory addresses from the programmer. In fact, the automatic garbage collector in languages like Java will move objects around in memory, adjusting all pointers as necessary, in such a way that the program's behavior will be unaffected.

Heap management and related algorithms are discussed more fully in the document:

*"Blitz-64: Software Reference Manual"*

# Appendix 6: KPL Syntax

This appendix provides a Context-Free Grammar (CFG) for the KPL language.[38] This grammar is meant to be exactly identical to the grammar the document

*"KPL Syntax"*

## The Notation Used in this Grammar

To make the grammar easier to read and understand, we use an extended CFG notation, which is described here.

*Non-terminal Symbols are shown like this:*

`HeaderFile     Type     Expr     Statement`     *etc...*

*Terminal Symbols:*

*Keywords are shown in boldface, like this:*

**`if     while     int     endWhile`**     *etc...*

*The following lexical tokens appear in the grammar:*

| | *Examples* | | |
|---|---|---|---|
| INTEGER | **`42`** | **`0x1234ABCD`** | **`10_000`** |
| DOUBLE | **`3.1415`** | **`6.022e23`** | **`1.000_001`** |
| CHAR | **`'a'`** | **`'\n'`** | |
| STRING | **`"hello"`** | **`"\t\n"`** | |
| ID | **`myName`** | **`MAX_SIZE`** | |
| OPERATOR | **`<=   <   >   >=   !=   +   -   *`** *etc...* | | |

[38] Technically, the KPL grammar is "LL(k)" and in most cases can be parsed with only a single token look-ahead, which makes it much easier for humans to understand than "LR(k)" grammars.

*Punctuation Symbols*

*The following characters are particularly important in KPL's grammar:*

**{ } [ ] | : , . = ( ) ;**

*Of these, the following punctuation symbols conflict with grammar meta-symbols:*

**{ } [ ] |**

*When used as grammar meta-symbols, they are shown without quotes:*

{ } [ ] |

*When used as terminals, i.e., when meant literally, they are quoted:*

'**{**' '**}**' '**[**' '**]**' '**|**'

*The remaining punctuation symbols are only used as terminals and are not quoted:*

**: , . = ( ) ;**

*Comments are not included in this grammar. There are two forms of commenting:*

**--** *through end-of-line*

/* *through* */

# A Context Free Grammar of KPL

```
HeaderFile        --> header ID
                        [ Uses ]
                        { Constants |
                          Errors |
                          VarDecls |
                          Enum |
                          TypeDefs |
                          FunctionProtos |
                          Interface |
                          Class }
                      endHeader
CodeFile          --> code ID
                        { Constants |
                          Errors |
                          VarDecls |
                          Enum |
                          TypeDefs |
                          Function |
                          Interface |
                          Class |
                          Behavior }
                      endCode
Interface         --> interface ID [ TypeParms ]
                        [ extends TypeList ]
                        [ messages { MethProto }+ ]
                      endInterface
Class             --> class ID [ TypeParms ]
                        [ implements TypeList ]
                        [ superclass NamedType ]
                        [ fields { Decl }+ ]
                        [ methods { MethProto }+ ]
                      endClass
Behavior          --> behavior ID
                        { Method }
                      endBehavior
Uses              --> uses OtherPackage { , OtherPackage }
OtherPackage      --> ID     [ renaming Rename { , Rename } ]
                  --> STRING [ renaming Rename { , Rename } ]
Rename            --> ID to ID
TypeParms         --> '[' ID : Type { , ID : Type } ']'
Constants         --> const { ID = Expr }+
Decl              --> ID { , ID } : Type
VarDecl           --> Decl [ = Expr2 ]
VarDecls          --> var { VarDecl }+
Errors            --> errors { ID ParmList }+
TypeDefs          --> type { ID = Type }+
Enum              --> enum ID [ = Expr ] { , ID }
IdList            --> ID { , ID }
```

```
ArgList          --> ( )
                 --> ( Expr { , Expr } )
ParmList         --> ( )
                 --> ( Decl { , Decl } )
FunctionProtos   --> functions { FunProto }+
FunProto         --> [ external ] ID ParmList [ returns Type ] [ StackUsage ]
Function         --> function ID ParmList [ returns Type ] [ StackUsage ]
                       [ VarDecls ]
                       StmtList
                     endFunction
StackUsage       --> '[' maxStackUsage = Expr ']'
NamelessFunction --> function    ParmList [ returns Type ]
                       [ VarDecls ]
                       StmtList
                     endFunction
MethProto        --> ID ParmList [ returns Type ] [ StackUsage ]
                 --> infix OPERATOR ( ID : Type ) returns Type
                 --> prefix OPERATOR ( ) returns Type
                 --> { ID : ( ID : Type ) }+ [ returns Type ]
Method           --> method MethProto
                       [ VarDecls ]
                       StmtList
                     endMethod
StmtList         --> { Statement }
Statement        --> if Expr StmtList
                         { elseIf Expr StmtList }
                         [ else StmtList ]
                     endIf
                 --> LValue = Expr
                 --> LValue += Expr
                 --> LValue -= Expr
                 --> ID ArgList
                 --> Expr { ID : Expr }+
                 --> Expr . ID ArgList
                 --> while Expr
                         StmtList
                     endWhile
                 --> do
                         StmtList
                     until Expr
                 --> break
                 --> continue
                 --> return [ Expr ]
                 --> for LValue = Expr to Expr [ by Expr ]
                         StmtList
                     endFor
                 --> for ( StmtList ; [ Expr ] ; StmtList )
                         StmtList
                     endFor
```

```
                  --> switch Expr
                          { case Expr : StmtList }
                          [ default : StmtList ]
                      endSwitch
                  --> switchOnClass Expr
                          { case Expr : StmtList }
                          [ default : StmtList ]
                      endSwitchOnClass
                  --> try StmtList
                          { catch ID ParmList : StmtList }+
                      endTry
                  --> throw ID ArgList
                  --> free Expr
                  --> debug [ STRING ]
                  --> printf ( [ ID , ] STRING { , Expr } )
                  --> sprintf ( ID , STRING { , Expr } )
                  --> initializeArray ( Expr )
                  --> setArraySize ( Expr , Expr )
Type              --> byte
                  --> halfword
                  --> word
                  --> int
                  --> double
                  --> bool
                  --> void
                  --> typeOfNull
                  --> anyType
                  --> ptr to Type
                  --> struct { Decl }+ endStruct
                  --> union { Decl }+ endUnion
                  --> array [ '[' Dimension { , Dimension } ']' ] of Type
                  --> function ( [ Type { , Type } ] )
                          [ returns Type ]
                  --> NamedType
NamedType         --> ID [ '[' Type { , Type } ']' ]
TypeList          --> NamedType { , NamedType }
Dimension         --> * | Expr
Constructor       --> Type ClassStructInit
                  --> Type ArrayInit
                  --> Type
ClassStructInit   --> ID '{' ID = Expr { , ID = Expr } '}'
ArrayInit         --> ID '{' [ Expr of ] Expr
                            { , [ Expr of ] Expr } '}'
LValue            --> Expr
Expr              --> Expr2  {   ID : Expr2 }
Expr2             --> Expr3  {   OPERATOR Expr3 }
Expr3             --> Expr5  {   '||' Expr5 }
Expr5             --> Expr6  {   && Expr6 }
Expr6             --> Expr7  {   '|' Expr7 }
Expr7             --> Expr8  {   ^ Expr8 }
Expr8             --> Expr9  {   & Expr9 }
```

```
Expr9           --> Expr10 {   == Expr10
                             | != Expr10 }
Expr10          --> Expr11 {   < Expr11
                             | <= Expr11
                             | > Expr11
                             | >= Expr11 }
Expr11          --> Expr12 {   << Expr12
                             | >> Expr12
                             | <<< Expr12
                             | >>> Expr12 }
Expr12          --> Expr13 {   + Expr13
                             | - Expr13 }
Expr13          --> Expr15 {   * Expr15
                             | / Expr15
                             | % Expr15 }
Expr15          --> OPERATOR Expr15
                --> Expr16
Expr16          --> Expr17 {   . ID ArgList
                             | . ID
                             | '[' Expr { , Expr } ']' }
Expr17          --> ( Expr )
                --> null
                --> true
                --> false
                --> self
                --> super
                --> INTEGER
                --> DOUBLE
                --> CHAR
                --> STRING
                --> NamelessFunction
                --> ID
                --> ID ArgList
                --> new Constructor
                --> alloc Constructor
                --> sizeOf ( Type )
                --> asPtrTo ( Expr , Type )
                --> asInteger ( Expr )
                --> arraySize ( Expr )
                --> arrayMaxSize ( Expr )
                --> isInstanceOf ( Expr , Type )
                --> isKindOf ( Expr , Type )
```

# Appendix 7: Lexical Details

## Source File Encoding

The KPL compiler expects the header and code files to be encoded in UTF-8. Since UTF-8 subsumes ASCII, any ASCII text file is acceptable.

## Comments

KPL supports two comments styles and they can be freely intermixed in any program.

In the first style, the comment begins with /* and ends with */. These comments may be nested.

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

In the second style, everything after two hyphens through end-of-line is a comment.

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

The second style also nests.

```
-- Disable this code...
--   x = y + 2   -- Adjust y a little
```

Comments are not restricted to ASCII; they may utilize the full Unicode character set.

## White Space

“White space” is defined as a sequence of one or more of these characters:

```
Space
Tab
Newline[39]
Byte-Order-Mark[40]
```

Newlines are not significant beyond being white space, except for comments. A “--” comment runs until the next newline.[41]

As usual, two tokens must be separated by white space if they would otherwise be interpreted as a single token.

| | |
|---|---|
| `myIdent 123` | White space required |
| `myIdent123` | … or else it becomes one long identifier |
| | |
| `124 e14` | White space required |
| `124e14` | … or else it becomes a floating point constant |

## Identifiers

An ID is a sequence of letters, digits, and underscores. It must begin with a letter. Only ASCII characters are allowed. Case is significant.

[39] To handle different OS conventions, the compiler will treat either \n (ASCII 0x0a) or \r (ASCII 0x0D) as a “newline” character.

[40] The Byte-Order-Mark (BOM) is a special Unicode character (codepoint = 0xFEFF) which is occasionally included in files that are encoded in UTF-16, where byte-order matters. However, the BOM is legal in any Unicode file and Unicode requires that it be ignored. If the BOM is present—and it usually is not—it should be the first character in the text file. The compiler treats the BOM as white space, effectively ignoring it.

[41] Also, see an earlier footnote concerning the disambiguation of function calls and pointers to functions for a minor exception.

# Integers

An INTEGER can be expressed either in decimal or in hex:

```
12345
0x01b5f3b
```

It makes no difference whether the integer is expressed in decimal or hex.

Every decimal integer token must be within the range

```
0 … 9,223,372,036,854,775,807
```

Expressed in hex, this range is 0x0 … 0x7FFF,FFFF,FFFF,FFFF, so every positive 64 bit value can be expressed in decimal.

Any integer token may be expressed in hex. There must be 16 or fewer hex characters. The hex characters (a, b, …, f) may be capitalized or lowercase or a mix, but the “0x” must be lowercase. Any 64 bit value can be expressed.

Values specified in hex with fewer than 16 digits are not sign-extended.

```
0xffff_ffff    -- Equal to +4,294,967,295
```

The negative sign is interpreted as a separate token. The following are legal *Expressions* within the grammar, and all representing the same value:

```
-1_793_851
-0x01b_5f3b
0xFFFF_FFFF_FFE4_A0C5
```

Notice that the most negative 64 bit value cannot be expressed in decimal, since 9,223,372,036,854,775,808 lies just beyond the range of legal decimal integer tokens.

```
-9_223_372_036_854_775_808      -- Compile-time error!
```

However, this value may be expressed in hex as follows, which is clearer and less error-prone.

```
0x8000_0000_0000_0000
```

The grammar also accommodates a unary plus sign, which is ignored.

To increase readability for numbers with many digits, humans often break them up with commas. In KPL code, the programmer may use the underscore as a separator within decimal and hex constants. This is recommended, but not mandatory.

For example, the following tokens are legal and all express the identical value:[42]

```
0x12345678_9abcdef0          two chunks of 32 bits
0x12_34_56_78_9a_bc_de_f0    as 8 bytes
1_311_768_467_463_790_320    decimal
```

## Floating Point Constants

A DOUBLE number must contain either a decimal point or the "e" for the exponent.

```
12.345
123e-45
```

For both INTEGERs and DOUBLEs, a leading minus/negative sign "-" will be parsed as a separate token and used to form an expression, such as -(123). The compiler will evaluate such expressions at compile-time, so effectively any INTEGER or DOUBLE may be negated.

```
-1.5          -- Preferred
-(1.5)        -- Equivalent
```

To increase readability, the programmer may use the underscore as a separator either before or after the decimal point.[43] For example:

```
57.000_000_001
1.234_56
```

[42] An underscore may not appear before the first digit or after the last digit. If the value is given in decimal and underscores are used, they must follow the standard placement at every third place. With hex values, there are no constraints; "0x123___4" is legal.

[43] If used in a double constant, underscores must be spaced at every third digit. Using underscores before the decimal point does not mandate their use after the decimal point. Likewise, using underscores after the decimal point does not mandate their use before. So "12345.000_001" and "12_345.000001" are both acceptable, although "12_345.000_001" is preferred.

```
12_000.5e-23
```

The following KPL keywords can be used:

| | |
|---|---|
| **inf** | Positive infinity |
| **nan** | The canonical representation of not-a-number (NaN) |

Positive infinity is represented as 0x7FF0,0000,0000,0000.

```
d = inf
d = copyBitsToDouble (0x7FF0_0000_0000_0000)  -- Equivalent
```

To indicate negative infinity, use the expression "**-inf**".

Negative infinity is represented as 0xFFF0,0000,0000,0000.

```
d = -inf
d = copyBitsToDouble (0xFFF0_0000_0000_0000)  -- Equivalent
```

It is safe to test "**if** d == **inf**…" or "**if** d == **-inf**…".

Be aware that there are multiple representations of "not-a-number". Also, any value tested against NaN will false, so testing "if d == nan…" will always be false. For these reasons, you should use the built-in function **isnan()** to perform this test.

There are two different zero values, which can be written as:

```
+0.0        -- Equivalent to "0.0".
-0.0
```

Note that for the purposes of testing floating point values, these values — although distinct — will test as equal.

| | |
|---|---|
| `if d == 0.0 ...` | Tests for either value |
| `if (d == 0.0) && !isNegZero(d) ...` | Test for +0.0 only |

# Character Constants

A CHAR constant is enclosed in single quotes. Any Unicode character may be included, with a couple of provisos.

```
'A'
'∉'
'😀'      -- This happens to be a legal Unicode character
```

The newline (0x0a) and return (0x0d) characters may not appear and must be escaped instead, for example:

```
'\n'     -- Use this for NEWLINE 0x0a
```

Other common control character may be escaped. Here is the list of the escape sequences that may be used:

```
            Hex    Decimal                 ASCII Code Name
           =====   =======                ========================
   \0        00        0      control-@    NUL   null
   \a        07        7      control-G    BEL   alert
   \b        08        8      control-H    BS    backspace
   \t        09        9      control-I    HT    tab
   \n        0A       10      control-J    NL/LF newline/linefeed
   \v        0B       11      control-K    VT    vertical tab
   \f        0C       12      control-L    FF    form feed
   \r        0D       13      control-M    CR    return
   \e        1B       27      control-[    ESC   escape
   \d        7f      127                   DEL   delete
   \"        22       34                   "     double quote
   \'        27       39                   '     single quote
   \\        5C       92                   \     backslash
   \xHH      HH         < any hex value >
```

Except for \, \n, and \r, the ASCII control characters do not need to be escaped:

```
'''            -- The single quote character is allowed
'      '       -- The \t character
''             -- The \0 character! See it?
```

Of course, the recommendation is to use the escape sequence:

```
'/''           -- We prefer these
'/t'
'/0'
```

In the case of individual bytes, there must be exactly 2 hex characters. The byte value is sign extended.

```
var ch: int

ch = '\xff'
ch = -1                      -- Equivalent
ch = 0xffffffffffffffff      -- Equivalent
```

The source code file is assumed to be encoded in UTF-8 and any Unicode character may appear between the quotes. This allows for any codepoint up to the maximum of $2^{21}$-1 = 2,097,151.

Note that the value -1 = 0xFFFFFFFFFFFFFFFF is not a valid Unicode codepoint. This value is the preferred representation of conditions such as end-of-file (EOF).

## String Constants

A string constant consists of a sequence of zero of more characters enclosed in double quotes:

```
"Here is an example string constant\n"
```

The string constant may use the same escape sequences as character constants, which were listed directly above.

String constants may contain any byte, but there are limitations on the way a string must be written.

Character constants may not contain a newline directly, but this is allowed within a string constant. String constant may contain newline (0x0a) and return (0x0d) characters directly.

```
ch = '
'                -- Not allowed

str = "hello
world"           -- But this is okay
```

Other than the newline (0x0a) and return (0x0d) characters, string constants may not contain ASCII control codes directly. Instead you must use the escape sequences.

```
str = "hello    world"     -- TAB is not allowed
str = “hello\tworld"       -- You must use this instead
```

Otherwise, a string constant may contain any Unicode character.

```
str = "Café of the Näive: ∞ ∉ 😀  £ ⇒ €   π ≠ ÷"
```

A string constant may contain any byte. In particular, there is nothing special about the NUL byte (0x00). This is a significant difference with C / C++.

```
var str: String
...
str = "All bytes: \x00\x01\x02 ... \xfe \xff"
```

Each string constant will be translated into an array of bytes. In an assignment such as this, the variable will be set to a pointer to this array. In other words, the type of a string constant is "String = **ptr to array of byte**".

The size of this array (both the maximum and the current size) will be initialized to the number of bytes in the constant.

This array will be stored in writable memory. Bytes in the array may be altered and the array may be shortened if the programmer chooses to write such code. Each string constant will result in the allocation of a new and unique array; string constants are not shared.

```
str1 = "hello"
str2 = "hello"
str2 [3] = 'm'
printf ("%s %s", str1, str2)     -- Prints "hello helmo"
```

String constants are stored using the UTF-8 encoding. If the string contains only ASCII characters, the number of bytes will be identical to the number of characters. However, for Unicode code points above 0x7f, the UTF-8 encoding will require two or more bytes per character. So for strings containing unusual characters, the length of the array will be different from the number of bytes.

A particular UTF-8 encoding can be entered as a sequence of bytes, as the examples below will show.

A codepoint may also be entered directly in hex, using the \U escape sequence. Note that the "U" is capitalized. The "\U" must be followed by exactly 8 hex characters giving the Unicode codepoint. (The relationship between UTF-8 encoding and codepoint is complex and described elsewhere.)

The following are equivalent and create the exact same string.

```
str = "𝜋"                      -- MATHEMATICAL ITALIC SMALL PI
str = "\xf0\x9d\x9c\x8b"       -- The UTF-8 encoding
str = "\U0001d70b"             -- The codepoint (decimal: 120,587)
```

Note that many characters may look the same (or similar, depending on font) but have significantly different encodings:

```
str = "π"                      -- GREEK SMALL LETTER PI
str = "\xcf\x80"               -- Equivalent to above
str = "\U000003c0"             -- The codepoint (decimal: 960)
```

If the codepoint is 0xFFFF or less, then the code point can be entered in hex with the “\u” escape. Note that “u” is lowercase. The hex characters can be upper or lower case, or any mix. These are all equivalent:

```
str = "\U000003c0"
str = "\u03c0"
str = "\u03C0"                 -- Hex digits can be upper or lower
```

String constants are often have may characters. Long lines reduce program readability. To make programs more readable, a string constant may be broken into pieces. For example, this:

```
str = "A long string example: hello world"
```

is exactly equivalent to:

```
str = "A long string example: "    "hello "    "world"
```

and to this:

```
str = "A long string example: "
      "hello "
      "world"
```

Breaking long string constants into smaller pieces and spreading them over several lines can make programs easier to read. Whenever two strings appear together, the compiler will immediately concatenate them into a single string constant whose length is the sum of the original pieces.

At least one character of white space is required between the pieces:

```
str = "hello ""world"          -- Not allowed
```

A string constant may not include a double quote or a backslash. They must be escaped.

```
str = "abc"def\ghi"           -- Not allowed
str = "abc\"def\\ghi"         -- Use this instead
```

Single quotes are allowed.

```
str = "3 o'clock"
```

## Regular Expressions for Token

In the regular expressions below, the following notation is used:

| | |
|---|---|
| \| | Alternation |
| {}* | Repetition of zero or more |
| {}+ | Repetition of one or more |
| ( ) | Grouping |
| - | Set difference, e.g., `Letter - ( x | y | z )` |
| $\varepsilon$ | Epsilon |

The KPL grammar makes use of the following lexical tokens:

```
ID, CHAR, STRING, INTEGER, DOUBLE, OPERATOR
```

These are defined as follows:

```
ID            =  Letter  {  Letter | Digit | Underscore  }*
CHAR          =  Apostrophe  OneChar  Apostrophe
STRING        =  { StringPart }+
INTEGER[44]   =  { Digit }+  |  0 x { Hex }+
DOUBLE[45]    =  { Digit }+ . { Digit }+ [ Exponent ]
              |  { Digit }+ Exponent
OPERATOR      =  {  OpChar  }+
```

where

```
StringPart    =  DoubleQuote  { StringChar }*  DoubleQuote
OneChar       =  (Unicode - Forbidden1 )  |  EscapeSeq | CodePoint
StringChar    =  (Unicode - Forbidden2 )  |  EscapeSeq | CodePoint
EscapeSeq     =  \ EscCode  |  \ x Hex Hex
Exponent      =  (e | E) ( Plus | Minus | ε ) { Digit }+
Letter        =  a | b | … | z | A | B | … | Z
Digit         =  0 | 1 | … | 9
HexLetter     =  a | b | c | d | e | f | A | B | C | D | E | F
Hex           =  Digit | HexLetter
EscCode       =  \ | ' | " | 0 | a | b | t | n | f | r | e | d
OpChar        =  \ | / | ! | @ | # | $ | % | ^ | & | ~ | ` |
                 ? | < | > | = | Plus | Minus | Bar | Star
Unicode       =  < any single Unicode character >
Forbidden1    =  \  |  NewLine
Forbidden2    =  \  |  ControlCode  |  "
CodePoint     =  CodePoint4 | CodePoint8
CodePoint4    =  \ u Hex Hex Hex Hex
CodePoint8    =  \ U Hex Hex Hex Hex Hex Hex Hex Hex
NewLine       =  < character 0x0A (NL) >  |  < character 0x0D (CR) >
ControlCode   =  < characters in the range 0x00 … 0x1F >
Underscore    =  "_"
Apostrophe    =  "'"
DoubleQuote   =  """
Plus          =  "+"
```

[44] This rule is incomplete; underscore characters may also appear with the sequence of Digits or Hex digits.

[45] This rule is incomplete; underscore characters may also appear with the sequence of Digits.

```
Minus          =  "-"
Bar            =  "|"
Star           =  "*"
```

# Appendix 8: Recent Changes

This appendix lists recent changes to this document.

**20 October 2020**

Section titled “Syntax Exception Regarding ‘*’” was added.

**8 March 2021**

Built-in function **isNaN** was renamed to **isnan**. Built-in functions **posInf**, **negInf**, **NaN**, and **negZero** were removed. New built-in function **isNegZero** is added. New keywords introduced: **inf**, **nan**.

# About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s, which has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. His technical focus and research interests have included AI and neural networks; parsing and natural language processing; logic, object-oriented, and functional programming; compilers, operating systems, interpreters, and system software; and discrete math and computational theory. He has programmed in many high-level languages and written assembly code for a variety of machines, dating back to the IBM 360/67 and Intel 8080.

Porter lives in Portland, Oregon. When not trying to figure out how his computer actually works, he skis, hikes, travels, and spends time with his children building things.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.