Kai: A Dynamic Compiler

Samuel Grant Dawson Williams

October 14, 2010

Abstract

Kai is an experimental interpreter that provides explicit control over the compilation process. It can generate optimised code at run-time in order to exploit the nature of the underlying hardware. It is a unique exploration into world of dynamic code compilation and the interaction between high and low level semantics.

I am thankful to my supervisor Tadao Takaoka. His diverse knowledge and support have been exceedingly helpful throughout this research.

I dedicate this work to my much loved grandfather Ormiston Herbert Walker. After his passing late last year, he still had four science textbooks books in the process of being published. His commitment to the pursuit of science continues to inspire people around the world, and his devotion to both his family and his community are remarkable.

> 'Though my soul may set in darkness, it will rise in perfect light, I have loved the stars too fondly to be fearful of the night.' – Unknown – an old astronomer to his pupil Galileo.

Contents

1 Introduction

> A ship in port is safe; but that is not what ships are built for. Sail out to sea and do new things.
>
> Grace Hopper

Programming languages enable humans to communicate with computational machines at various levels of abstraction. Just like a spoken language, a programming language provides the basic constructs that shape the way we express our ideas. Unlike spoken languages, programming languages have precise semantic models that must be respected in order for a program to be executed correctly.

Humans are inherently creative, and in many ways the unbounded nature of the human mind creates many circumstances where language is insufficiently expressive. In these cases we utilise the human capacity beyond language to communicate our desires or feelings; but for computers this is not possible (at least, not yet!).

Thus, in order to satisfy humans expressive capabilities, programming languages provide varying levels of abstraction. The more capacity the language provides for abstract expression, the more convenient it is for solving problems by human minds, because we can deal with the concerns of the problem domain, rather than those of the implementation, and apply human intuition more naturally.

However, at some point a program needs to be executed to satisfy the users' computational needs. Theoretically, we can say that the distance between the physical hardware and the abstract execution model dictate the maximum efficiency of the abstract program. A program written directly in machine code will have its instructions executed as fast as is physically possible. However, an abstract program may need to have its instructions executed in a one-to-many fashion, which reduces performance.

For efficiency, many abstract programming languages provide constructs with well defined translation semantics. Modern optimising compilers can perform advanced transformations of the program to suit the underlying hardware architecture and because compilation is an offline process, a compiler can expend a large amount of effort to produce optimal code. Compiled programs can run quickly because the machine is executing a concrete implementation of the abstract program.

Another approach is to use an interpreter, which processes the program in whatever abstract form it takes, and executes the required behaviour on the processor directly. Because this translation happens at run time, the translation mechanism is sharing processor resources with the executing program, and this can cause interpreted programs to be less efficient than their compiled counterparts. However, in general, interpreted languages are far more flexible than compiled languages, because they have access to far more information about the program at run time.

This study is an investigation into both interpreted and compiled semantics, with a particular emphasis on programming language design and performance analysis of compiled and interpreted code.

2 Definitions

Compiler
A compiler is any kind of tool that processes input data to output data, typically without side effects. In this sense, the same input will typically produce the same output every time. Often, compilation process converts a verbose input format into a concise output format - some information may be discarded in the compilation process.

Execution Model
An execution model is a specific class of interpreters that defines how and when specific instructions are executed. This might include out-of-order execution, parallel execution, register and stack implementation, memory and device interaction, and so on.

Interpreter
An interpreter is a program that can execute a given input to produce behaviour. This program can either be a software program or a hardware processor. In this sense, an x86 CPU with microcode would be considered an interpreter.

Operating System
At a fundamental level, an operating system provides a set of primitives for software execution. In many cases, the primary purpose of an operating system is to share hardware resources amongst multiple processes efficiently and securely. In addition, many operating systems provide standard interfaces for a variety of different kinds of hardware such as network, display and audio.

Programming Language
A programming language consists of a well-defined syntax and a semantic behaviour. It may also include libraries of source code, supporting tools and infrastructure for execution.

Semantic Model
A semantic model is an abstract set of concepts which can be used to describe the behaviour of a system, and in this research will be used to refer to the behaviour of a computer programming language[1]. In theoretical computer science, formal semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation.

Syntax Model
A syntax model is a set of rules which can be used to validate and process textual input into a tree data structure, commonly referred to as a parse tree.

[1] For more information see `http://en.wikipedia.org/wiki/Formal_semantics_of_programming_languages`.

3 Background

> Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.
>
> Alan Kay

There are many programming environments in the world. They provide various combinations of semantics and syntax to facilitate the expression of computer programs; this allows us to solve many different kinds of problems. From crafting instructions to run directly on a processor to high level expressions of logic and behaviour, there are many systems for expressing that which is computable.

A syntax and semantic model (collectively known as a programming language) define the space in which a programmer creates a solution to a problem. A particular solution may require many compromises when expressed in a particular programming language, some which may create problems with the particular solution.

In the realm of software engineering, there are also additional concerns such as existing infrastructure (i.e. standard libraries, support libraries), development tools (i.e. source code aware editors, refactoring tools, debuggers) and scalability issues (i.e. deployment platforms, distributed execution, reliability and redundancy). Depending on the design of the programming language, these additional concerns may be minimised.

A complete programming environment includes some kind of execution model and typically an operating system. These factors may influence the performance of a solution due to architectural implementation. Several existing approaches exist, such as byte code interpretation and native code compilation (see table 1). Architectures that directly support specific parts of a semantic model are more likely efficiently execute a program.

Execution Model	Programming Language Implementation
Tree-based Interpreter	Perl (≤ 5), Ruby (MRI ≤ 1.8)
Bytecode Interpreter	Berkeley Pascal, Java (≤ 1.2), Python, Ruby (1.9), Smalltalk-80
Native Code Compilation	Ada, C & C++ (GCC), Haskell (GHC), Java (GJC)
Bytecode + Dynamic Compilation	Self, Smalltalk, Java (≥ 1.3)

Table 1: A variety of different execution models

A particular semantic model might be designed to match a particular execution model, or a particular execution model might be created to match a given semantic model[2]. This is normally a performance trade-off, but when the machine model dictates the semantic model, there can be serious issues with regards to machine-independence[3].

Modern processors provide advanced features such as multiple threads of execution, parallel execution of instructions, multiple layers of cache and sub-processors for specific tasks to name a few. Building a general semantic model that can be executed efficiently on multiple different machine models can be complex, because the underlying hardware may not necessarily support the required features. Making a programming language both abstract and efficient can be a challenging task.

[2]Several Java processors implement Java bytecode directly in hardware. This is an example of a semantic model dictating the structure of a hardware platform.

[3]Programming languages that rely on byte order or register size may create difficulties when executing on a machine model with different intrinsic properties.

There are many issues surrounding the development of an effective programming language, and while there are a large number of existing programming languages, there are still many possibilities that can be explored.

3.1 1800–1960: The Birth of the Compiler

The Jacquard loom was invented in 1801 and used punched cards to program sewing patterns automatically. Although it was incapable of performing computations, it is considered the conceptual precursor to modern punch-card based computers, because it allowed the operator to change the behaviour of the loom by providing a different program.

In 1837, Charles Babbage described the Analytical Engine, an essentially modern general purpose computational machine. During 1842–1983, Ada Lovelace was tasked with translating documents about Babbage's Analytical Engine, and she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the machine. This is generally accepted as the world's first computer program, even thought it was impossible to execute it at the time.

Over the next 100 years, computer science related technology and theory continued to advance. Alonzo Church created lambda calculus as a way to express function definition, application and recursion. Alan Turning developed the idea of the Turning Machine which can be used to explain what is and isn't computable. John von Neumann put forward the idea of the stored-program architecture, where a computer consists of memory, processors and input/output units.

It was in the 1940s that the first general purpose electric computers were developed[4]. The limitations imposed by these computers required programmers to write hand-tuned assembly language programs, and it was soon realised that such programs were difficult to produce and highly prone to errors.

In an attempt to avoid these problems, during 1943–1946, Konrad Zuse developed Plankalkül; a system for expressing algorithms widely regarded as the first attempt to create a programming language. However, due to the mindset of time time favouring assembly programming, '[it] never attained the consideration it deserved' according to Heinz Rutishauser.

It was also during this time that Grace Hopper (an American computer scientist and U.S. Naval officer) became a pioneering computer programmer, writing programs for the Harvard Mark I. During her time as a programmer, she conceptualised the idea of machine-independent programming languages, and is widely credited with popularising the term 'debugging' when she removed a dead moth from an electronic relay.

During the 1950s, the number of available computers and programming languages grew rapidly. John Backus developed FORTRAN[1] in 1955; a language designed for numerical and scientific computation. John McCarthy developed LISP[2] in 1958; a symbolic language based on Alonzo Church's lambda calculus. Grace Hopper heavily influenced the design of developed COBOL in 1959; a language designed for business, finance and administration. All three of these languages are still in use today in one form or another.

It was also around this time, the mid 1950s, that the ALGOL 60 Report[3] was published. It described the programming language ALGOL which represented a consolidation of many important programming language ideas. The language specification used Backus-Naur Form for describing the context-free portion of the language's syntax, and it introduced several important language concepts including lexical scoping of language declarations using nested blocks of code.

[4]For more details see http://en.wikipedia.org/wiki/History_of_computing_hardware

The first compiler was developed by Grace Hopper in 1952 for the A-0 programming language (a precursor by several steps to COBOL). Subsequent development led by John Backus at IBM led to a FORTRAN compiler in 1957. COBOL was one of the first languages to have compilers for multiple architectures in 1960. In 1962, the first self-hosting compiler[5] was created for LISP by Tim Hard and Mike Levin at MIT.

From this point onward, it was common to use a compiler for building executable binary code.

3.2 1960–1975: The Advancement of Abstraction

After the initial burst of blossom during the 1940s and 1950s, we begin to see programming languages bloom into full colour during the 1960s and 1970s. Many pivotal language paradigms were invented during this period.

One of the major changes was the shift away from unstructured programming using goto statements. In 1966 a paper was published[4], which showed that every computable function can be implemented using a structured approach. Thus, many computer scientists started to investigate more advanced methods of structured programming, such as object-oriented, imperative and functional programming. It was also during this time that Edsger Dijkstra published the somewhat humorous 'Go To Statement Considered Harmful'.

Simula[5], developed during the 1960s, was a direct superset of ALGOL 60. It was designed for performing large simulations and included language features specifically for this purpose. It was one of the first languages to provide an object-oriented semantic model. It allowed for the definition of classes, subclasses and virtual methods, and its execution model included garbage collection.

Pascal is a structured programming environment developed by Nikalaus Wirth between 1968–1970. It was based on ALGOL and was designed largely as a language for teaching students. The language definition included a complete standard library of functionality and this made it ideal as a teaching tool.

BASIC, an unstructured programming language, was developed in 1964 by John George Kemeny and Thomas Eugene Kurtz to provide computer access to non-science students. It was designed to be an easy interactive programming language, but still provide advanced features where possible. One of its main features was its interactive shell and the fact that it shields the user from the execution environment.

Smalltalk[6] was developed during the 1970s by Alan Kay and was a departure from typical programming environments of the time. It provided an encapsulated and self-hosted execution model built on top of a single messaging primitive for communication between different objects – in this sense it was completely dynamically typed and object-oriented.

C was developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories for use with the Unix operating system (which was also developed around this time). It was designed to be a simple and portable language with constructs that could be compiled easily to efficient assembly code. It requires minimal runtime support and provides a very simple standard library.

Prolog[7] was developed in the early 1970s and has its foundation in formal logic. It processes factual information, and can answer questions about the information it has been provided. It supports a variety of complex programming tasks such as natural language analysis, expert systems, theorem proving, and other sophisticated control systems that rely on relationships between data.

[5] A self hosted compiler is a compiler written in the language it compiles.

ML was developed in the late 1970s and was one of the first widely used functional programming languages. It has a number of important features such as advanced type inference, algebraic data types, parametric polymorphism and exception handling.

These major languages all represented different approaches to solving problems.

3.3 1975–1980: The Personal Computer

In 1975, the MITS Altair 8800 was released, a do-it-yourself microcomputer kit, and single-handedly sparked the beginning of the personal computer, which made programming widely available to people in a way that had previously been impossible.

Over the next few years, many different kinds of low cost personal computers were produced, including the Apple II, Commodore PET and TRS-80.

Many of these machines were equipped with variations of the BASIC interpreter. Previously all serious programming had been done on mainframe computers, however, this allowed many people to experience programming for the first time, and this ensured that BASIC became a popular and well known programming language.

3.4 1980–1990: Large Scale Software

During the 1980s, the progress of computer hardware and software was accelerating at a tremendous speed. Software applications were beginning to exist on a larger scale with the advent of ethernet, and as such, programming languages were being improved to solve problems spanning a wide variety of different concerns. In this sense, many of the languages developed during this period of time tried to augment and recombine existing ideas to improve their application to real world problems.

Modula-2, based on Pascal[6], developed by Niklaus Wirth between 1977–1980, was a pioneering language in this respect; it was designed to improve the development of large software projects by dividing source code into modules. Modules were compiled separately and explicitly listed their exported symbols; they could then be linked together to form a complete program.

C++, developed by Bjarne Stroustrup in 1983, was a combination of existing procedural C and object-orientated semantics borrowed from Simula. The object-oriented programming paradigm was widely respected as the best way to design large scale software, and thus C++ was an attempt add an object-oriented semantic model to a fast low level programming language in order to get the benefits of both a high level semantic model and a fast execution model.

Ada, developed by the United States Department of Defence from 1977–1983 was influenced by ALGOL 68 and Modula-2 and included similar mechanisms for modularity, but also included semantics to improve the reliability and verification of computer programs.

Eiffel[8], developed in 1986 by Bertrand Meyer, took many ideas from Ada and Simula and recombined them into a purely object-oriented programming language based around a design by contract approach.

Erlang[9], also developed in 1986 by Ericsson, built on many of the features of Scala, but included features for fault-tolerant distributed applications. It supports hot swapping code modules, so that running systems can be updated without downtime.

[6] Actually, a prototype existed called Modula, but it was never officially released.

3.5 1990–2000: The Information Era

During the 1990s, the Internet began to take off. Web browsers began to become a standard part of the desktop computer, and HTTP[7] and HTML[8] was widely adopted as a standard way of presenting information to the end user.

The web provided a unified front end for software to interact with users. Previously, software was used almost exclusively on the platform for which it was developed via some kind of platform specific user interface; but now it was possible to create software that interacted with users on different platforms via HTTP and HTML.

Due to the growth of platform and programming language independence, a wide variety of scripting languages were developed in an attempt to improve the speed at which applications were developed. Many of these languages were designed with different sets of priorities, but ultimately ended up with many similar features. This represented a convergence of features that were typically expected in a modern programming environment for high level programming tasks.

Python[10] was conceived of during the late 1980s and its first implementation was completed in 1991 by Guido van Rossum. Python was designed to have a clear and powerful syntax as well as a large standard library of functionality. It is a multi-paradigm language supporting a wide variety of different programming styles, primarily object-oriented and imperative semantics, but also including functional semantics where useful. It is a highly pragmatic language and despite its high level nature, has many similarities to minimalist languages such as LISP.

In 1993, Yukihiro Matsumoto developed Ruby[11], a language heavily influenced by Perl, Smalltalk, Eiffel and LISP. Like Python, it is a multi-paradigm language and provides a large standard library of functionality. In contrast, however, Ruby has a much more functional approach to programming, and coupled with its very flexible syntax, has provided a fertile ground for the development of domain specific languages.

Prior to 1995, HTML was a static medium for communicating with the end user, normally generated by some program on a server, and delivered to the end user via HTTP. In 1995 this changed with the advent of JavaScript, a language designed to run in the web browser and augment the delivery of HTML content to the end user. It was one of the first very popular (by virtue of its circumstances) prototype-based programming languages (the first being Self), and one of the first languages to have a distributed execution model, in the sense that code is sent to the web browser to be executed.

One of the most popular languages create at this time was PHP, a very simple object-oriented programming language. It did not bring anything new to the table in terms of programming paradigms, but it did provide a very low barrier to entry into web programming, and this made it very successful.

Also in 1995, the programming language Java was developed by James Gosling and representing a culmination of business programming practices and ideas. It was designed for very large scale software engineering challenges, and provided a rigorous, principled semantic model to simplify the development of interconnected software components. The execution model was specifically designed to allow cross-platform development, deployment and integration with minimal effort.

[7] Hyper Text Transport Protocol
[8] Hyper Text Markup Language

3.6 2000–2010: Refinement and Interoperability

During this period of time we see a large number of existing languages being refined and distilled into well engineered platforms for top to bottom software development. Investment in existing software platforms mandates that incremental enhancements be made to existing languages rather than a complete departure from existing infrastructure. In this sense, many developers preferred extensions to language functionality through the development of libraries rather than fundamental changes to languages; and one may say that the general foundations of computer programming languages settled during this time because of this change in thinking.

Objective-C, a language which appeared in the late 1980s, suddenly became popular due to the advent of Mac OS X which used it extensively in its fundamental systems frameworks. It provides a message based object model runtime expressed entirely with C, which provides an advantage over other object-oriented languages which often have bigger requirements. The language was revised in 2006 which provided improvements to syntax and advanced features such as garbage collection.

Also during this time, Microsoft released the .NET framework along with C# and Visual Basic .NET in 2001. These programming languages, despite having different semantics and syntax, built upon a single base runtime called the .NET Common Language Runtime, a virtual machine that provides generational garbage collection and just-in-time compilation and an extensive class library.

It is also during this time that we see a large number of applications and software libraries being released, often designed for integration with different programming languages. Software applications such as a database could be written in one language, and have bindings exported for integration with other languages. Leveraging existing technology reduces the complexity of developing new software and allows higher level problems to be solved more easily. This also brings issues of interoperability between languages, with some languages having sufficiently different semantic models as to make interoperability difficult.

One fundamental outcome of all these high level developments was the establishment of C as a common denominator across the majority of platforms. Because of its simple syntax and low complexity semantic model, it is a prime target for standard APIs which can then be shared across multiple different languages and adapted into more complex semantic models (i.e. object oriented or functional).

Even thought programming languages during this time generally improved in the sense that they started to become complete platforms for software engineering, there were still many new interesting languages developed.

Scala[12], released in 2003 by Martin Odersky, is an attempt to integrate both object-oriented and functional programming semantics. The name Scala stands for scalable language, and as such the language has been designed from the ground up to assist with the development of large scale programs using functional techniques. In order to leverage existing technologies, Scala runs directly on top of the Java Virtual Machine, and this allows it to utilise existing libraries and be integrated into existing applications seamlessly.

Factor[13], a language that has been developed in several iterations since 2003, is a dynamically typed stack based language with powerful meta-programming features. Like Scala, it combines various object-oriented and functional semantics to create a cross-platform, modern programming environment.

Clojure[14], developed in 2007 by Rich Hickey, is a modern dialect of LISP with a unique approach to parallel programming. Like Scala, it runs within the Java Virtual Machine which provides a large amount of existing functionality. Clojure concurrency is based on the idea of immutable data structures that can

be updated efficiently. It has well defined concept of time, and provides a semantic model to deal with changes to data structures over time.

Go[15], developed by Robert Griesemer, Rob Pike and Ken Thompson from 2007 and released in 2009, is a simple, fast, concurrent and safe language designed for network and systems programming. During design, a primary goal has been to keep the number of features to a minimum and provide strict type safety. Another important goal is to ensure fast compilation by reducing the dependencies between files and allowing the compiler to cache intermediate build steps more effectively. The language exposes interesting primitives for concurrent communication and distributed execution.

3.7 The Future: Scalability and Reliability

History has shown that success is often dictated by simplicity, despite the problems this may create in the fundamental properties of a language. Many programming languages have incrementally developed syntax and semantic models, and in this sense they are more familiar to existing programmers. Conversely, unfamiliar or complex languages are typically ignored. A programming language requires a significant critical mass before it becomes popular and is widely utilised.

Languages with complex semantic models naturally limit the amount of interoperability they provide with other environments. However, much has been gained from obscure languages too - new ideas and new concepts have often found their roots in programming languages which have ultimately proved to be unpopular.

The future of programming languages will be centred around standardisation and scalability. As part of this, domain specific languages will play an important role and help solve specific problems more effectively (where in the past a general purpose language would have been used). Performance in many cases will be less important as computer hardware continues to improve and hardware resources continue to expand.

Java is a prime example of standardisation and scalability. While Java has a well defined execution model that suits many software engineering problems, there are problems which are far more efficiently solved with specific sets of semantics. Because of this, Java as a platform has been the centre for many specific languages (e.g. Scala[12], Groovy[16], Jess[17], Clojure[14], Fantom[18]) which help to solve specific types of programming tasks.

As the scale of software engineering tasks continues to expand, serious scalability issues become apparent. Many performance issues are not those of individual programs, but that of large bodies of programs running together. Programming language semantics can facilitate the communication and sharing of data between programs, and this will be an important aspect of modern programming languages.

3.7.1 Distribution and Execution

How do we make programs faster by distributing the problem over multiple execution platforms? How do we deal with architectural differences? As programs are combined of many different pieces, how do we decide which pieces are distributed and in which way? Even with large scale software which is typically deployed on thousands of machines, can we test pieces in isolation and verify the behaviour of software as it is scaled up? Are there new language paradigms that can support programmers to create software with these types of concerns?

3.7.2 Reliability and Verification

How do we improve the reliability of software and verify its correctness? As programs get bigger, it becomes increasingly difficult for any one person to understand their characteristics. Are there semantic models and execution environments which improve the chance that software works as required? Are there efficient systems for ensuring the correct execution of computer programs and enabling fault-tolerance?

3.7.3 Software Engineering

How do we describe the solutions to problems in ways which allow us to avoid repetition? As software projects increase in size and scope, how do programming languages support the development of modular programs and are such solutions tied to a specific semantic model or can they be generalised?

3.7.4 Integration and Modularity

How can we integrate disparate technologies and platforms without increasing the burden on the developer and the end user? Platform agnostic technologies such as XML and Unicode have provided a good foundation for the exchange of information; but these are also very verbose formats that may affect performance.

What other options exist for modularity of source code and how can these be used as a net gain (i.e. a reduction of programming work required to solve a particular problem)? Existing solutions are often tied to specific platforms and languages. Is it even possible to have a net gain in all areas when writing different parts of a program in different languages?

3.7.5 Libraries vs Languages

Finally, which of the above issues should be solved at the language semantic level, in comparison to supporting libraries which provide functionality? This is perhaps, the most difficult question. Many programming languages have limited support for meta-programming and the ability to introduce new semantics into the programming language itself. Thus, this problem is not one of pure computer science, but also of software engineering – can we really seek to change the extensive foundations of existing computer programs?

4 Philosophy

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
>
> Sir Charles Antony Richard Hoare

In order to create a successful programming language, one must balance multiple, often competing, concerns and demands. The high level design of a programming language is not only a technical challenge, but a philosophical and creative theatre where many different perspectives must be considered.

4.1 The Art of Syntax

Most general purpose programming languages represent source code using structured text[9]. For this text to mean anything, a parser must process this text into a structured form. This is typically referred to as an intermediate representation[10].

Structured text can be processed in a variety of different ways. An interpreter might evaluate the value of a node directly, while a compiler might use this representation to make optimisations and then produce machine code.

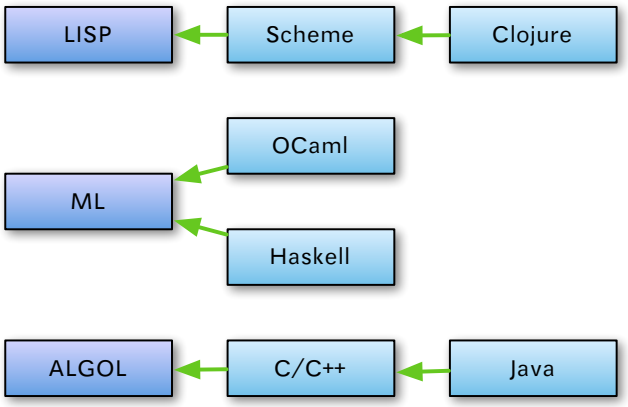

Figure 1: Three major roots of programming language syntax.

It is very difficult to categorise the roots of syntax, but in general there appear to be three main categories (see figure 1). Modern programming languages are harder to classify as they draw on many specific elements beyond any single categorisation. However, it is still useful to understand how these different approaches affect the design of a programming language.

Due to the expressive power of LISP, its fundamental syntax has not changed very much over time. However, both ML and ALGOL style syntax have changed significantly. This could be attributed to the changing semantics of modern languages, and the implicit connection between syntax and semantics in these languages.

[9] However, some languages have experimented with visual programming interfaces, such as Scratch (http://info.scratch.mit.edu/)
[10] For more details see http://en.wikipedia.org/wiki/Intermediate_representation

LISP style syntax can be hard to read because it has a sparse and repetitive visual context[11]. Humans have the capacity to recognise and process complex expressions, but LISP style syntax does not utilise this capability. Languages which are syntactically denser require less code to express the same ideas.

When programming languages have strict syntax, many types of errors can be detected easily. This may make a syntactically strict language less prone to errors, and better at recovery in the event of a programmer error. Different forms of syntax for specific types of expressions - while tedious - may reduce the chance that the program is invalid.

There are many different approaches to syntax in the design of a programming language. There are many qualities that one may seek to maximise:

- A syntax which builds on familiar syntactic structures helps new programmers understand a language quickly by utilising existing knowledge and expectations.
- A syntax which is consistent across multiple different syntactic structures allows new knowledge to be applied intuitively throughout the programming language.
- A syntax which has an intuitive structure allows programmers to use perceptual abilities and prior knowledge to understand source code without being familiar with the specific syntactic constructs.
- A simple structure which allows programs to be parsed by external tools easily and correctly. This can simplify processes such as refactoring, statistical analysis, and program visualisation.
- A recursive syntax which allows programmers to express nested expressions in a way that is more inline with typical thought patterns.

4.2 The Semantic Gap

Programming language semantics are vast and varied. There are a large number of issues to decide upon, and no simple answer.

One fascinating trend in the history of programming languages is the changes in the way designers have approached the semantic problem. Initially we see a very purest approach to semantics where a programming language is designed around one or two central pillars, often referred to as paradigms. As an example, we can look at LISP (pure lambda calculus), Prolog (pure logic based), Smalltalk (pure object oriented).

Many of these paradigms are very useful, but in order to make a more complex program we want better integration between different paradigms; this is very much a pragmatic approach to problem solving. Most modern languages are considered multi-paradigm and include a variety of orthogonal language concepts – this allows the programmer to combine syntax and semantics in the most appropriate way for a given problem.

Python, a multi-paradigm language, builds upon many existing ideas and supports a wide variety of programming styles including object-oriented programming, structured programming, functional programming, aspect-oriented programming (including meta-programming) and design by contract. The design philosophy[19] embodies many concepts that have been explored in prior programming languages.

[11] If LISP was black and white, C++ would be all the colours of the rainbow, including infrared and radio waves

4.3 Constant or Variable?

Many programming languages have the concept of immutable values and constant expressions. Immutability is a restriction that provides an invariant about a value or data structure. In general, things which are immutable will not change. By specifically marking something as immutable, a wide variety of different optimisations (e.g. constant propagation and inlining) become possible.

In many static languages, functions and types cannot be defined more than once – they are immutable. This is part of the semantic model for practical purposes because it would make the language ambiguous if different declarations could have the same identification. Static languages, due to immutability, provide many opportunities for code analysis and error checking.

Most basic data types are also immutable in the sense that expressions on basic data types produce new values rather than mutate existing values. However, in many programming languages these semantics are not consistent[12].

Beyond this basic definition, programming languages with shared state may require strong guarantees about when particular state can be changed; this is especially true in parallel execution environments. If two objects contain the same state, and this state is shared/aliased, a change to one object may inadvertently change the other.

4.4 Past, Present and Future

There are many different approaches to time in a computer program. Depending on the semantics of instruction execution, some programs can be easily distributed across multiple processing units.

In a typical procedural language, one instruction executes after the other in a well defined sequence. This kind of program is easy to follow and has a well defined behaviour. However, this approach does not lend itself well to parallel processing. It can be impossible to analyse the dependencies between instructions and make decisions about how to distribute execution in order to increase efficiency.

Expert systems which have rules and dependent behaviours (such as Prolog) deal with cause and effect and have a different approach to time. As the state of an expert system changes, behaviours may be invoked, which may in turn update state. This type of system decouples the sequence of instructions that are executed as in a procedural system, and exchanges this for well defined dependencies between behaviours.

Some languages support lazy evaluation which means that expressions may not be evaluated if the result is not required, even if it is part of an expression. This can be very convenient as it reduces the amount of explicit ordering required, but it can also introduces subtle semantic details which increase the chance of programmer error.

4.5 Evolutionary Ascent

Programming languages can be defined in a very abstract sense - a semantic model can be expressed using logic and mathematical notation, while syntax can be represented using state machines and other such mechanisms. However, such definition may not be very practical. Through the process of implementation, a new compiler or interpreter may find its place in history.

[12] Python's list mutation operator $x \mathrel{+}= y$ changes the original list x, and is not strictly equivalent to $x = x + y$

All compilers and by implication interpreters have ties to the birth of the first computer. Every modern compiler was given life by one that came before it, and assumptions and structures of prior technology influence the structure of that which comes next. Things as simple as data types and operations, to the structuring of memory and order of execution, may be dictated by prior technology.

One such example of this is the term bootstrapping which in this context refers to a compiler compiling a new compiler for a different execution model. In this case we are creating an entirely new compiler for a platform on which no compiler may have existed previously; this phenomenal achievement allows existing technology and advances to be applied to new hardware, despite the fact that previously there may have been few, if any, tools developed for the platform.

4.6 The Cost of Change

When developing a programming language, there are many issues to consider - one being change to fundamental semantic and syntactic models. It is implicitly important that programming languages are able to support a wide ecosystem of software and related tools. These tools often depend on assumptions built into the programming language, and if these change, significant problems are often caused.

We can see a very pertinent example of this with the evolution of Python from version 2.x to 3.0. Python 3.0 introduces changes to syntax and semantics that break existing code. Due to this, a large amount of effort has been expended updating standard libraries and improving support for already functioning 2.x code, and many people have voiced concerns about whether the benefits of the changes outweighs the cost in updating existing source code.

In many ways this is one of the strengths of C, in that it provides a baseline that has been well established in many different execution environments.

5 Kai

> Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution.
>
> Albert Einstein

With great dignity and respect for the history of programming languages, I now take the honour to discuss my exploration into this realm.

Kai started several years ago as a project to improve my understanding about computer programming languages, and also to make a real improvement to the way I write programs. Through this process I have expanded my understanding of the different kinds of language paradigms and learnt a great deal, and despite the investment of time on my part required for this journey, I have felt at every step it has been of great personal value.

Many different programming languages have contributed to my appreciation of computer science, and I feel that if there is some ideal programming language, it has yet to be discovered. I have a great deal of respect for the hardware we use as computer programmers, and I aim to avoid inefficiencies in both the way we express our ideas as programmers, as well as how they are executed.

Figure 2: The Kai logo incorporates a circle and the Japanese character Kai 会 which means ‘meeting’.

5.1 Meaning

The Japanese character 会 generally translates into ‘meeting’. There are many ways in which this is significant. In this particular instance its usage can be considered in terms of 弓道, which roughly translates into ‘Way of the Bow’ (Archery). The highest goals in 弓道 are truth 真, goodness 善 and beauty 美. Truth is the pursuit of the correct technique and to grasp the truth of things; Goodness represents inner calm and balance; beauty is to maintain a clear structure and composition. There are many formal steps in 弓道; Kai 会 is the moment before the arrow is released, when the energy of the bow and the body of the archer are in balance.

5.2 Overview

Kai is a simple language based on a LISP style syntax with several extensions. It is written in C++ and utilises several external libraries: the Boehm-Demers-Weiser conservative garbage collector[20] and LLVM[23] for dynamic compilation. It employs a prototype based object model (similar to JavaScript) and has several primary data types including Tables for associative storage (similar to Lua) and Cells for parse trees (similar to LISP). It has been designed for simplicity and extensibility; the core interpreter consists of several fundamental objects and for the most part is homoiconic[13].

The basic structure is outlined in figure 3.

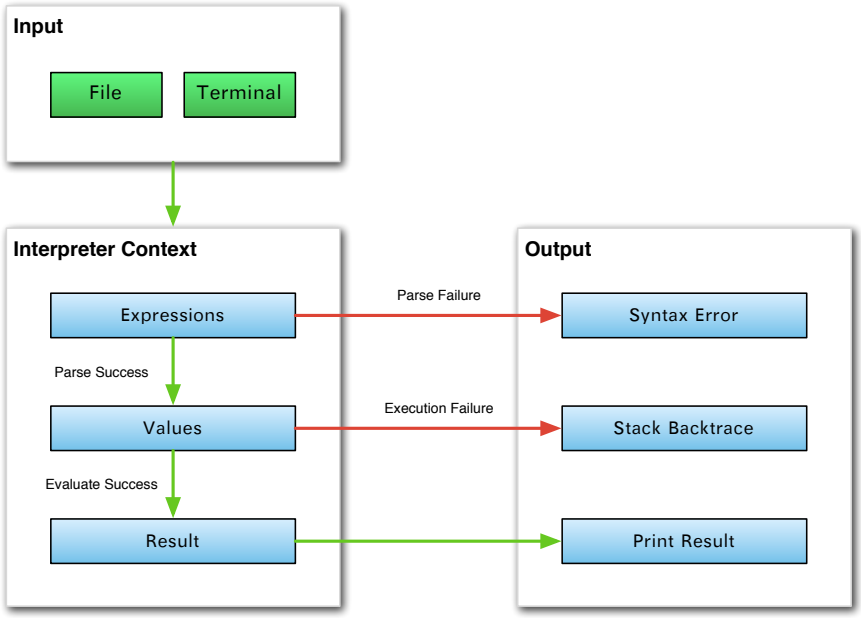

Figure 3: A high level overview of Kai.

5.3 Online Interpreter

During the discussion of Kai, several examples are given. These can be executed online using the Kai interpreter available at `http://kai.oriontransfer.co.nz/`.

[13] Homoiconicity is a property of some programming languages, in which the primary representation of programs is also a data structure in a primitive type of the language itself.

6 Syntax

Kai uses a syntax similar to that of LISP, with the facility to enhance the syntax easily with new expressions. Rather than implementing a complex grammar, Kai consists of a set of high level expressions. Input is tested against these expressions in turn, and if nothing matches it is considered an error. An expression is able to implement a grammar in any way that is convenient.

The Kai command line allows source code to be entered one line at a time. This is done by keeping track of individual expressions. For a given input i, we check against expressions $e_1, e_2, ..., e_n$, and if an expression parses completely then we consider the input valid. If the input i does not parse correctly, we consider two options: either the input $i + j$ may be successful, or input i can never be successful. We use this property to validate when we require more input from the user, or if the current input already contains a fatal syntax error.

6.1 Formal Grammar

```
       <alpha> = a-z A-Z ;
     <numeric> = 0-9 ;

(* A program is an expression *)
  <expression> = <integer> | <string> | <symbol> | <operator> |
                 <cell> | <value> | <call> | <block> ;

     <integer> = ('-' | '') <numeric> * ;

      <string> = '"' <character> * '"' ;
   <character> = ('\"' | [^ '"']) * ;

      <symbol> = <alpha> (<alpha> <numeric>) * ('?' | '!' | '=' | '') ;

(* This is an augmentation of <symbol> *)
    <operator> = '==' | '<=>' | '<=' | '>=' | '<<' | '>>' | '<' | '>' |
                 '+' | '-' | '*' | '/' | '%' | '=' ;

(* This is the primary form of nested expressions *)
        <cell> = '(' <expression> * ')' ;

(* These expressions are all for convenience. *)
       <value> = '`' <expression> ;

        <call> = '[' <expression> * ']' ;

       <block> = '{' <expression> * '}' ;
```

6.2 Strings

A string represents a sequence of characters. A character is one or more bytes as defined by the encoding. By default, Kai expects strings to be encoded using UTF-8, and does not have provisions for other encodings.

A string is represented literally by data enclosed between two quotation marks. Internal quotation marks can be escaped using a backslash, along with several other characters such as tab, newline and null characters. Bytes can be inserted using hexadecimal escape sequences.

Listing 1: Strings

```
kai> "Hello\nWorld!"
"Hello
World!"
kai> "Hello\t\tWorld!"
"Hello        World!"
kai> "Apples \x26 Oranges"
"Apples & Oranges"
```

6.3 Numbers

Basic numbers without a decimal point are considered integers. These can be both positive and negative.

Listing 2: Integers

```
kai> -55
-55
kai> 897
897
```

Numbers with a decimal point are current unsupported.

6.4 Symbols

Symbols are sequences of characters which represent an identifier. They are different from strings in the sense that they require no quotation marks, and internally a symbol also has an associated hash value.

Listing 3: Symbols

```
kai> `ThisIsASymbol
ThisIsASymbol
kai> [`ThisIsASymbol hash]
1291
kai> [`hash hash]
420
```

6.5 Cells

A Cell is a basic structure of recursion. A Cell consists of a head and a tail, and is, in traditional literature a single link in a singly linked list. The head of a cell is the value it contains, and the tail of the cell is the next cell in the list, or null.

A cell can represent a multitude of different data structures including lists, trees and graphs.

Listing 4: Cells

```
kai> `(10 apples and 20 oranges)
(10 apples and 20 oranges)
kai> [`(10 apples and 20 oranges) head]
10
kai> [`(10 apples and 20 oranges) tail]
(apples and 20 oranges)
```

6.6 Values

A value expression is a short-hand notation for giving the value of an operand rather than the result of its evaluation. It is done using the back-tick character before an expression.

Listing 5: Values

```
kai> ``bob
(value bob)
kai> `bob
bob
kai> bob
nil
```

6.7 Calls

A call expression is a short-hand notation for method dispatch on a given object. Because the semantics of this operation are non-trivial, having a syntactic expression for this type of function is preferable.

Listing 6: Calls

```
kai> [`orion hash]
551
kai> `[`orion hash]
(call (value orion) (value (hash)))
```

6.8 Blocks

Blocks are collections of code. They are are syntactic sugar to enhance reading code.

Listing 7: Blocks

```
kai> {[(this) set `x 10] [x return]}
10
kai> `{[(this) set `x 10] [x return]}
(block (call (this) (value (set (value x) 10))) (call x (value (return))))
```

7 Semantics

> The question of whether computers can think is like the question of whether submarines can swim.
>
> Edsger W. Dijkstra

Kai has been designed explicitly to have as few intrinsic semantics as possible. This makes it very easy to experiment with various different program flow controls and structures. It has also been important as it has provided insight into what basic structures are required to make a programming language sufficiently useful.

7.1 Evaluation

Kai has a single fundamental semantic: evaluation[14]. It is on this foundation that all other semantics are structured.

$$\textbf{evaluate}(\textit{value}, \textit{frame}) \rightarrow \textit{result}$$

A *result* is produced by evaluating a given expression *value* in an execution context *frame*. The importance of evaluation cannot be underestimated; without it there is simply the existence of the value, but nothing can be achieved.

The behaviour of evaluation is defined by the implementation of the various data structures exposed by Kai. Many built-in functions are provided by the current implementation and these provide additional high level semantics.

7.2 Value Semantics

The most basic form of evaluation is simply to return the value itself.

$$\textbf{evaluate}(\textit{value}, \textit{frame}) \rightarrow \textit{value}$$

7.2.1 Example

In this case, evaluating a string or an integer returns the exact same value. This is the default behaviour for Kai datatypes.

```
kai> 19850402
19850402
kai> "Truth, Goodness and Beauty"
"Truth, Goodness and Beauty"
```

[14] In a sense, this corresponds to the release of the arrow, 離れ

7.3 Lookup Semantics

The second most basic form of evaluation is to correlate two things together. Given a symbol, we can find out what it referrers to, starting from the current frame.

$$\textbf{evaluate}(\textit{symbol}, \textit{frame}) \rightarrow \begin{cases} \textit{frame}[\textit{symbol}] & \text{if } \textit{frame} \text{ defines } \textit{symbol}, \\ \textbf{evaluate}(\textit{symbol}, \textbf{parent}(\textit{frame})) & \text{if } \textit{frame} \text{ has a parent}, \\ \textit{nil} & \text{otherwise}. \end{cases}$$

7.3.1 Example

The symbol x is not defined initially, so the result of **evaluate**(x) is *nil*. Once we define a value for x, evaluating it gives us a result.

```
kai> x
nil
kai> [(this) set 'x 10]
10
kai> x
10
```

7.4 Application Semantics

Possibly one of the most important low level semantics is application. This is similar to traditional function call and macro expansion semantics in other languages. The implementation in Kai is very similar to the **fexpr**[21] in LISP.

$$\textbf{evaluate}(\textit{cell}, \textit{frame}) \rightarrow \textbf{evaluate}(\textbf{evaluate}(\textbf{head}(\textit{cell})), \textbf{next}(\textit{frame}, \textit{cell}))$$

The evaluation of a cell looks at the head of the given cell (typically a symbol), and evaluates this. The resulting value is then evaluated in a new stack frame, which is constructed by $next$ using the given cell which provides the operands for the evaluation. Kai does not evaluate operands implicitly (like many traditional languages). Instead, the function is passed the unevaluated arguments and the local environment which can be used to evaluate the arguments.

7.4.1 Example

Given a cell containing a set of operands, evaluation of this cell will cause the first operand to be evaluated. The result of this will then be used to evaluate a new stack frame containing the given operands.

```
kai> list
(builtin-function Cell::list)
kai> (list a b c)
(a b c)
```

7.5 Built-in Functions

Kai provides a wide range of additional semantics through built-in functions and data types. These functions are written in C++ and are hooked directly into the interpreter.

```
kai> list
(builtin-function Cell::list)
kai> (list a b c)
(a b c)
```

7.6 Scope

Kai dynamically allocates stack frames, which provide scoped name lookup. A stack frame has a arbitrary value which represents its scope; in many cases this is a table, but not always. The function **this** returns the current scope.

7.6.1 Example

```
kai> [(this) set 'x 10]
nil
kai> (this)
(table{0x101d63db0} 'x 10)
kai> x
10
```

When defining functions, such as when using **lambda**, we capture the scope at the point of declaration. This is retained and used as the scope for variable lookup when calling the function. In a sense, a lambda is a closure over the current dynamic scope. We can use this to create functions that retain state, typically referred to as generators.

7.7 Prototypes

Kai exposes a prototype-based object model. It provides specific syntactic sugar to support this model (the call expression). As part of this object model, function lookup becomes powerful, because the name lookup uses the current scope as well as the object for evaluation.

7.7.1 Example

prototype is a free function that returns the prototype for its argument. When provided with the value 10, it returns the Integer prototype. However, we can also use the method call notation **[10 prototype]** which uses **lookup** to find the method **prototype** and execute it with 10 as the first argument.

```
kai> prototype
(builtin-function Value::prototype)
kai> (prototype 10)
(table{0x100bedc60}
'% (builtin-function Integer::modulus)
'* (builtin-function Integer::product)
'+ (builtin-function Integer::sum))
```

```
kai> [10 prototype]
(table{0x100bedc60}
'% (builtin-function Integer::modulus)
'* (builtin-function Integer::product)
'+ (builtin-function Integer::sum))
```

7.8 Tables

Tables provide a key-value storage semantic that maps directly to the lookup semantic. Tables also provide the ability to set values, check if a key is is set, and many other higher level operations.

7.8.1 Example

```
kai> [Table new 'name "Alice" 'age 30]
(table{0x100b98570}
'name "Alice"
'age 30)
kai> [_ lookup name]
"Alice"
```

7.9 Wrap and Unwrap

The default semantic for a given input is evaluation. When calling a function, arguments are evaluated. However, many functions, such as **if** only evaluate their arguments in certain conditions. We can control this behaviour using **wrap** and **unwrap** semantics[22] which change the way function arguments are evaluated.

7.9.1 Example

list is a macro that returns a cons-cell list of its arguments. We can use **wrap** and **unwrap** to change its behaviour:

Listing 8: Wrap and Unwrap

```
kai> [(this) set 'a 10]
10
kai> (list a b c)
(a b c)
kai> ((unwrap list) a b c)
('a 'b 'c)
kai> ((wrap list) a b c)
(10 nil nil)
```

This is an interesting concept because it unifies the semantics of 'macro expansion' with function dispatch. In the case of macro expansion, none of the arguments are evaluated (i.e. unwrapped), but we can take a macro and create a function where its arguments are evaluated (i.e. wrap).

Another benefit of structuring code around explicit evaluation functions (traditionally called fexprs)is that it allows both macros and functions to be represented as first class values, unlike many languages where

the two ideas are very different; because of this unification, the concept of a function is solidified without spurious limitations.

8 Data Types

The current implementation of Kai has a limited set of data types, including Integer, String, Cell, Symbol, Table and Lambda. These data types are allocated using memory managed by garbage collection[20]. This reduces the complexity of the implementation and reduces the chance for error due to memory management bugs.

8.1 Integer

Integer values store a 32-bit signed integral number and provide basic mathematical operations.

Listing 9: Integers

```
kai> Integer
(table{0x101a10c90}
'% (builtin-function Integer::modulus)
'* (builtin-function Integer::product)
'+ (builtin-function Integer::sum))
kai> [5 + 10 20 30]
65
kai> [10 * 10 2]
200
kai>  [12345678 % 9]
0
kai>
```

A more advanced implementation may add support for arbitrary-precision mathematics.

8.2 String

String values store a sequence of characters and provide basic concatenation and character extraction functions. A more advanced implementation might include support for regular expressions.

Listing 10: Strings

```
kai> String
(table{0x101a10870}
'at (builtin-function String::at)
'+ (builtin-function String::join)
'size (builtin-function String::size))
kai> ["Foo" + "Bar"]
"FooBar"
kai> [_ at 3]
"B"
```

8.3 Cell

Cell values store a head and tail value, equivalent to a LISP[2] cons cell. A list is represented by a chain of cells with values stored in head, and the remainder of list in tail (similar to a singly linked list).

Listing 11: Cells

```
kai> Cell
(table{0x101a10990}
‘each (builtin-function Cell::each)
‘head (builtin-function Cell::head)
‘new (builtin-function Cell::_new)
‘tail (builtin-function Cell::tail))
nil
kai> [‘(10 20 30) head]
10
kai> [‘(10 20 30) tail]
(20 30)
kai> [‘(10 20 30) each ‘trace]
(trace (10 20 30)) -> (10)
(trace (20 30)) -> (20)
(trace (30)) -> (30)
nil
```

All hierarchical syntax is represented using a set of nested cells.

8.4 Symbol

A Symbol value is similar to a string, however it is primarily used as a key. It is very important in the overall structure of Kai in the sense that most atoms are symbols. It is the only value that can currently be used for indexing into tables. Symbols are different from strings, because not only do they store the data that represents their name, they also cache their hash code for fast access into hash table.

Listing 12: Symbols

```
kai> Symbol
(table{0x101a10a20}
‘hash (builtin-function Symbol::hash))
kai> [‘Bob hash]
275
kai> [‘Bob toString]
"Bob"
```

8.5 Table

A Table value represents a key value data structure. It is the primary method of relational storage in Kai. Therefore, they must be fast and efficient. One of the primary concerns about Kai is startup time; if the interpreter must do many allocations at initialisation time, it will become slow. One option would be to statically build the hash table data structure and load this from disk, but exposes its own set of problems.

Tables may have a dynamically defined prototype. This is the core behaviour inheritance structure of Kai. The implementation of this is very simple: For key lookup k in table t, we check if k is defined in t. If so, we return the value $t[k] \rightarrow v$, otherwise we check to see if the table has a prototype p, and pass the key lookup $p[k] \rightarrow v$ in this case.

Listing 13: Tables

```
kai> Table
```

```
(table{0x101a10f90}
'get (builtin-function Table::lookup)
'each (builtin-function Table::each)
'prototype= (builtin-function Table::setPrototype)
'new (builtin-function Table::table)
'set (builtin-function Table::update))
kai> [(this) set 'Cat [Table new 'color "black"]]
(table{0x101a1bb10}
'color "black")
kai> [(this) set 'c [Table new 'name "Tiger"]]
(table{0x101a1b540}
'name "Tiger")
kai> [c prototype= Cat]
(table{0x101a1bb10}
'color "black")
kai> [c name]
"Tiger"
kai> [c color]
"black"
```

8.6 Lambda

Lambdas are the primary unit of functionality in Kai. They have a well-defined set of arguments which are evaluated before being passed to the function. Functions can either return a value explicitly, or the last evaluated value is returned.

Listing 14: Lambdas

```
kai> [(this) set 'x (lambda '(a b) '{(if [a == b] (return 'equal) (return '
    not_equal))})]
(lambda{0x101a1b6f0} '(a b) '(block (if (call a (value (== b))) (return (value
    equal)) (return (value not_equal)))))
kai> (x 10 10)
equal
kai> (x 10 20)
not_equal
```

9 Compiler

Compilation is a method of improving performance by making assumptions which allow the code to execute more efficiently on the computer processor (see figure 4). Compiled code can also interact directly with native libraries by using native data types and function calls.

In general, we cannot compile interpreted code unless we can make assumptions about its behaviour. Because Kai is a dynamic language, it is impossible to compile expressions directly.

Other programming languages also perform semantic checking during compilation process, including validation of program flow and data types. This is not required in Kai, because any errors in compilation will become run time errors during interpretation. This provides a rich feedback to the end user when there is a semantic error in the compiler.

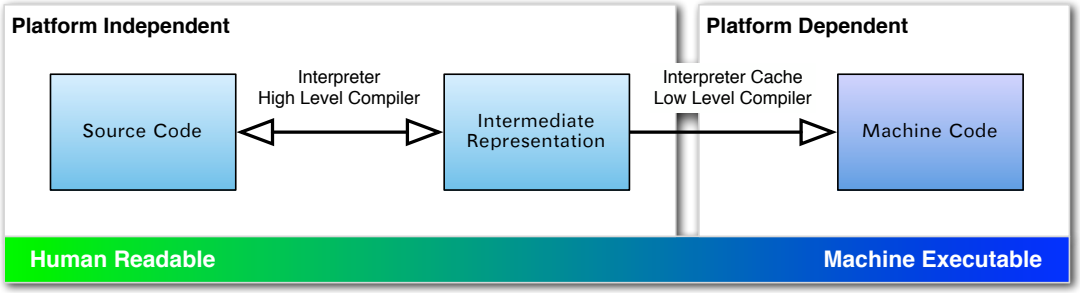

Figure 4: Code can be represented and executed in many different ways.

Kai integrates with LLVM[23], which provides cross-platform compilation, including whole program optimisation and just-in-time compilation. Because the compiler is a loadable module, it is conceivable that different compilers and compilation models could be supported in the same runtime.

Another benefit of the approach Kai takes to compilation, is that interpreted functions can be used to generate compiled code. This allows for a high level of meta-programming and code generation that is not possible in a traditional static language such as C.

9.1 Compiled Types

Kai provides support for all basic C data types, including integers, floating point numbers, structs and functions. The interface exposed for types allows for types to be explicitly defined in terms of size.

Listing 15: Lambdas

```
kai> (int 32)
(compiled-type i32)
kai> (int 44)
(compiled-type i44)
kai> (pointer (struct (int 32) (int 64) (float 64) (vector (int 32) 4)))
(compiled-type { i32, i64, double, <4 x i32> }*)
```

Kai currently does not have support for annotated composite types, and thus this would be a useful addition for the future.

9.2 Compiled Functions

The current implementation of Kai allows for the compilation of whole functions. It takes this approach because it aims to be an explicit dynamic compiler rather than providing transparent JIT features.

9.3 Trampoline Functions

Some compiled functions may depend on interpreted code, and vice versa. Because of this, we need support for bouncing between different kinds of execution models. Kai supports this by dynamically generating trampoline functions (see figure 5) which are 0-arity functions with embedded pointers to interpreter data structures.

```
llvm::Value * buildTrampoline(EvaluateFunctionT trampolineFunction, Frame * trampolineFrame)
{
    return llvm::Function::Create(
          Value * trampoline() {
              Frame * frame = trampolineFrame;
              EvaluateFunctionT function = trampolineFunction;
              return function(frame);
          }
    );
}
```

Figure 5: A compiled closure over interpreter state.

It is also important to jump from interpreter to compiled code, and because interpreter values are dynamically typed, they require unboxing (see figure 6). Support for this operation is currently under development, but the idea is that wrapper functions between the dynamic Value type and explicit compiled types can be either done implicitly or explicitly, depending on the nature of the compiled function.

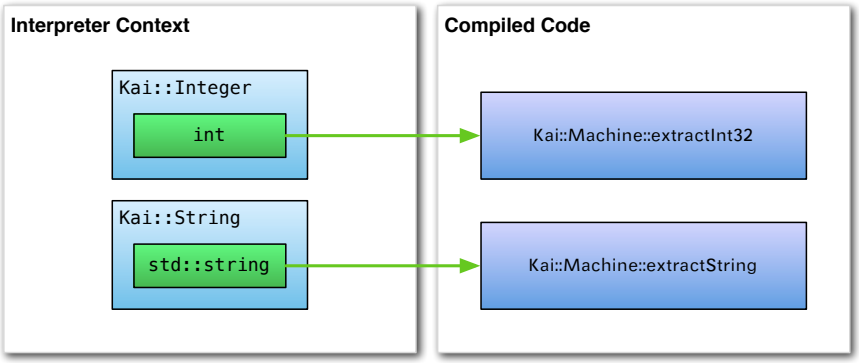

Figure 6: The compiled code can extract data types from dynamic values.

10 Performance Evaluation

There are several interesting questions one can consider when approaching an interpreter with compilation features.

- How long does it take to execute the interpreted function?
- How long does it take to compile the function?
- How long does it take to execute the compiled function?
- How can we improve overall performance by compiling functions?

For this evaluation, the greatest common divisor function will be programmed and executed (see appendix A for the source code and disassembly). The naive implementation of this function is recursive, so we have some opportunities for optimisation, and it is simple enough to be analysed by a human mind easily.

10.1 Method

Several different functions have been developed and executed across a wide variety of environments. There are two main performance considerations, the time it takes to define the function and the amortised run time. This is captured using the following benchmark:

$$\frac{\textit{initial definition} + (k \times \textit{execution})}{k}$$

Where *initial definition* is the time it takes to define the code for execution, and $k \times$ *execution* is the time it takes to execute the function k times. This represents the cost of using a function in a typical dynamic compiler environment, which includes both the time required to compile the function and the amortised cost to use it.

We also look at the *absolute time* required for execution, which is produced by ignoring the *initial definition* time and warming up the execution environment by running the function several times before taking a benchmark:

$$\textit{initial definition} \rightarrow (20 \times \textit{execution}) \rightarrow \frac{(k \times \textit{execution})}{k}$$

We measure the *absolute time* using the right most term. We choose a large k (in this case multiples of 1024) to reduce any $O(1)$ overheads. This metric exposes the fundamental performance of a particular implementation and is useful for comparison and analysis purposes.

Benchmark tests for different sized k are interleaved to improve the consistency of results.

10.1.1 High Resolution Timing

Function execution is typically very quick, especially for compiled code variants. A high resolution timer was designed specifically to improve accuracy.

10.2 Test Cases

10.2.1 Interpreted Code

High level expressions are directly executed by the Kai interpreter. This requires only a small amount of time for the *initial definition*, but the overall performance will be dictated by the performance of the interpreter, which will likely be an order of magnitude less efficient than direct execution on the processor.

10.2.2 Compiled Code

High level expressions are converted into machine executable assembly code. This requires a costly *initial definition* involving the conversion from high level expressions into LLVM IR and then compilation to machine-level assembly code. The execution of this code should be far faster than the interpreted code.

10.2.3 Optimised Code

Compiled code, which has been optimised using the LLVM optimiser before being converted to machine-level assembly code.

10.2.4 Pre-compiled Code

A C implementation is compiled inside the Kai interpreter with a direct link to the interpreter. The performance of pre-compiled code should be on the same order of magnitude as compiled code, but there is no initial cost because it is amortised with the compilation and loading of the main interpreter.

10.3 Results

The **GCD** function is used for testing with a fixed input of 892345 and 23426. This has a recursive depth of 8. All operations were repeated a minimum of 100 times and executed on a 2.5Ghz computer. The testing procedure was entirely automated to ensure repeatable results. The results are listed in table 2, and graphed together in figure 7.

k-Runs	Interpreted	Compiled	Optimised	Pre-compiled
Initial Definition	17.051μs	279.348μs	1248.327μs	0.000μs
1	244.651μs	2088.253μs	3005.646μs	1.150μs
2	213.653μs	1026.877μs	1527.386μs	0.817μs
3	180.981μs	686.143μs	1012.777μs	0.697μs
4	179.223μs	528.894μs	755.810μs	0.634μs
5	174.288μs	413.879μs	608.047μs	0.609μs
6	174.123μs	341.703μs	506.045μs	0.573μs
7	221.055μs	301.094μs	432.877μs	0.559μs
8	207.255μs	261.725μs	376.855μs	0.550μs
9	202.029μs	229.522μs	331.787μs	0.537μs
10	194.498μs	208.185μs	299.448μs	0.529μs
11	190.467μs	189.560μs	275.419μs	0.512μs
12	187.850μs	172.182μs	250.550μs	0.519μs
13	185.203μs	157.997μs	234.571μs	0.509μs
14	180.057μs	149.350μs	216.925μs	0.502μs
15	178.343μs	137.864μs	201.395μs	0.497μs
16	177.910μs	131.305μs	189.135μs	0.498μs
17	194.185μs	121.861μs	176.970μs	0.490μs
18	189.729μs	114.140μs	169.037μs	0.487μs
19	186.505μs	109.744μs	161.701μs	0.489μs
20	187.261μs	103.254μs	153.909μs	0.492μs
21	181.946μs	100.135μs	145.812μs	0.479μs
22	181.328μs	94.229μs	137.243μs	0.481μs
23	178.568μs	89.973μs	131.488μs	0.479μs
24	176.533μs	86.725μs	127.196μs	0.477μs
25	174.611μs	83.939μs	123.802μs	0.479μs
26	174.316μs	79.686μs	120.236μs	0.480μs
27	182.914μs	77.289μs	112.198μs	0.480μs
28	181.353μs	74.846μs	109.339μs	0.475μs
29	180.685μs	72.417μs	104.573μs	0.472μs
30	178.099μs	69.922μs	100.674μs	0.501μs
31	177.081μs	67.647μs	97.779μs	0.476μs
32	176.590μs	65.899μs	96.241μs	0.468μs
512	166.745μs	4.898μs	6.580μs	0.668μs
1024	165.683μs	3.025μs	3.915μs	0.974μs
Absolute Time	170.274μs	0.784μs	0.728μs	0.756μs

Table 2: A graph of absolute and amortised **GCD** performance.

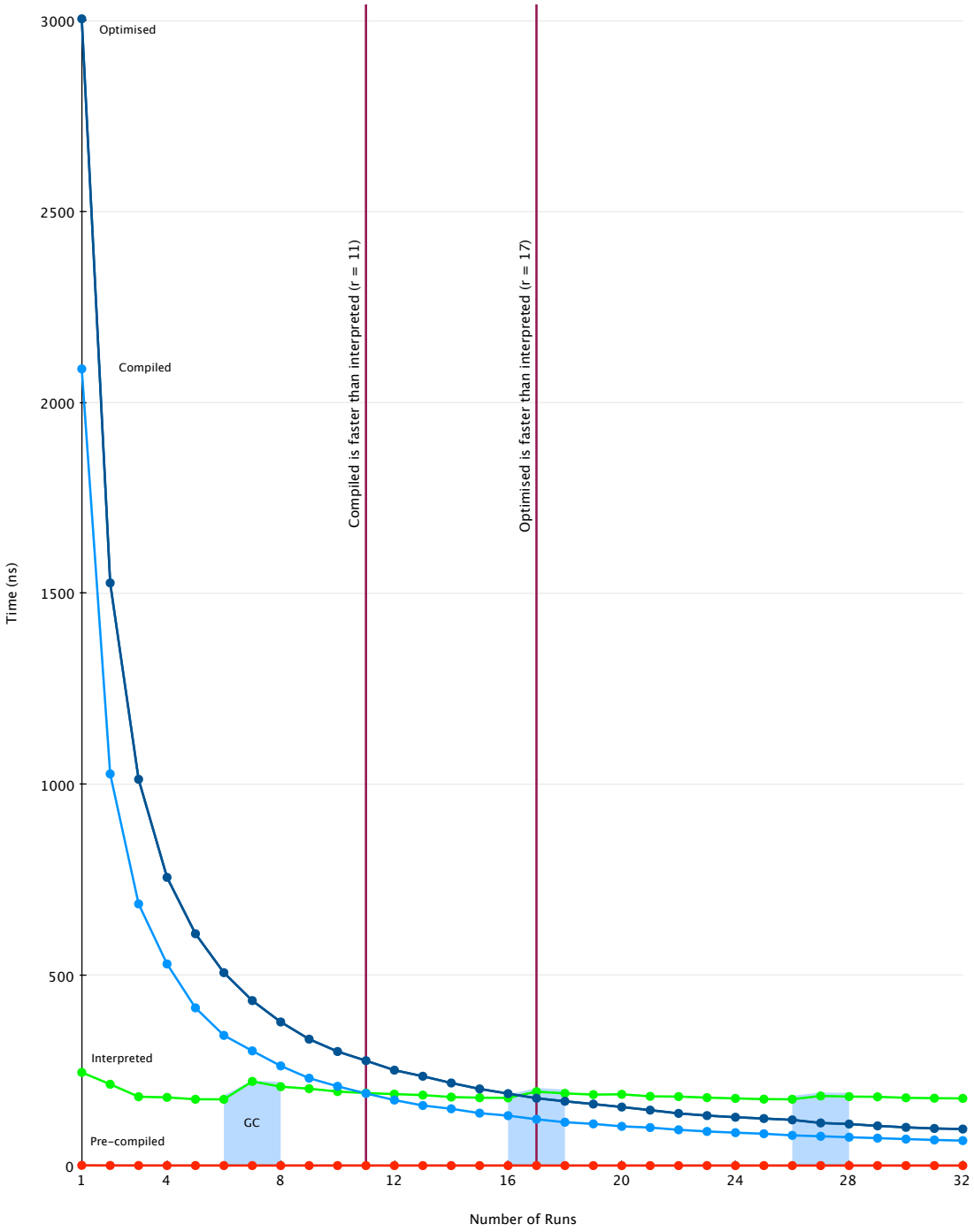

Figure 7: A graph of amortised GCD performance.

10.4 Discussion

10.4.1 Interpreted vs Compiled

The interpreted function has a high per-run cost, but it also has a very consistent amortised cost because the initial definition time is minimal. The compiled function has a vastly more expensive initial cost, and despite the fact that it is over 100 times faster, it takes 10 iterations for the amortised cost to dip below the cost of the interpreted function.

This clearly shows that that compilation of a function is only beneficial if it going to be used quite a few times *after* it has been compiled. This implies that some global statistical measurements may be required to improve performance, and that it is likely that not every decision to compile a function will improve performance.

One potential way to solve this problem is to distribute compilation to a different processor core. If we can move the compilation process out of the main program logic, the amortised cost could be vastly reduced. Another option would be to use some kind of compilation cache which is saved between executions of the program. However, behaviours like this lead to potentially unpredictable program performance which may not be desirable.

10.4.2 Compiled vs Optimised

The optimised function was very expensive to generate, yet overall its performance was worse than its non-optimised counterpart. The absolute performance of the optimised function is about 7% faster than the non-optimised version, but it would take a significant amount of time for the increased initial cost to pay off.

10.4.3 Optimised vs Pre-compiled

The pre-compiled function is about 4% slower than the optimised code generated by Kai/LLVM, but its amortised performance is very good because of its practically minimal initial cost. This shows that Kai/LLVM can generate better machine-level assembly than GCC in this instance.

The pre-compiled function does not represent dynamically defined program behaviour, like the other three test cases - its behaviour cannot be changed or redefined at run-time. In this sense, it is only useful as a performance comparison, and does not represent a functionally equivalent structure in Kai to any of the other defined functions.

10.4.4 Code Generation

Looking at the X86 code that is generated (see appendix A), we can get a good idea of the kind of code that is being generated by Kai and LLVM. The compiled code is 15 instructions, while the optimised code is 10 instructions. The pre-compiled code is 20 instructions, despite being optimised for small code size (GCC -Os).

The optimised version clearly has the most efficient implementation using a loop rather than recursion. The GCC pre-compiled code performs several redundant stack operations and is twice as long as the optimised code generated by Kai/LLVM, and is generally much harder to understand.

The reduced size of the optimised code may mean that the processor can fit the entire function in cache, which will increase the speed of the computation significantly.

10.4.5 Cache and Branch Prediction

We can see some unusual trends in the pre-compiled function performance, which might be attributed to cache behaviour. Initially, the cost is quite high ($r = 1...4$); then the cost decreases ($r = 14...32$). After that, when running higher benchmarks, performance cost increases again slightly. Because the **GCD** function is essentially a single branch at each step, the branch prediction unit may take some time to warm up, but it is not clear what is causing the slight decrease in performance for large r.

10.4.6 Garbage Collection

Garbage collection was noticeable in the results, and accounted for approximately 20% change in performance in the worst case for the interpreted function. As we increased the number of function executions, we see this reduced to around 3%. For a large program, garbage collection costs are likely to be minimal.

As an improvement, the garbage collection algorithm could be designed to run on a separate thread, which would reduce the performance impact it has on program performance.

10.4.7 Limitations

Because there was only a single function in this evaluation, the results are limited to a very specific set of operations.

The reason for this is simply due to the effort required to create a general purpose programming language that supports both compiled and interpreted semantics with the same behaviour. Recursive functions with no stack variables reduce the complexity of the implementation required.

Because the function is very simple, optimisations did not yield any overall performance benefits, and the overall amortised cost was higher by approximately 50% in the worst cases.

Despite this limitation, the results still provide useful insight into the behaviour of larger and more complex functions. The main difference would be that such functions would present more opportunities for optimisations, and this may increase the speed at which the amortised gains of optimisation outweigh its initial cost.

10.4.8 Further Work

It would be useful to do further evaluation involving more complex functions. Because Kai only supports a limited set of data types and functions at this time, more work would be required to expand the implementation of Kai.

It would be useful to do further analysis at the processor level to analyse the performance of specific kinds of optimisations. High level performance by wall-clock timing does not provide clarity on how the processor is being utilised internally.

11 Conclusion

An interpreter has been designed and developed. It exposes a very simple semantic model and builds a coherent programming language on top. A variety of data types have been implemented, including the support for dynamic code compilation and optimisation as part of the interpreter. Many kinds of syntactic and semantic models have been explored during this project, and the end result is a pragmatic balance of form and function.

LLVM was integrated successfully and was used to generate efficient compiled code. The Boehm-Demers-Weiser conservative garbage collector provides safe and moderately efficient memory management.

The amortised execution cost of interpreted code vs compiled code shows that the interpreter is efficient for functions which execute infrequently. The nature of program execution means that it may be impossible to predict in advance whether a function should be compiled, however the absolute performance of code generated by Kai/LLVM is on par with other mainstream compilers.

While the current interpreter is sufficiently powerful for the goals of this research, further work is required create a general purpose programming language.

References

[1] S. Best R. Goldberg L.M. Haibt H.L. Herrick R.A. Nelson D. Sayre P.B. Sheridan H.J. Stern I. Ziller R.A. Hughes J.W. Backus, R.J. Beeber and R. Nutt. The Fortran Automatic Coding System. pages 188–198, Los Angeles, California, February 1957. Available from: `http://www.bitsavers.org/pdf/ibm/704/FORTRAN_paper_1957.pdf`.

[2] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. volume 3, pages 184–195, New York, NY, USA, 1960. ACM.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. volume 6, pages 1–17, New York, NY, USA, 1963. ACM. Available from: `http://www.fh-jena.de/~kleine/history/languages/Algol60-RevisedReport.pdf`.

[4] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. volume 9, pages 366–371, New York, NY, USA, 1966. ACM.

[5] Ole-Johan Dahl and Kristen Nygaard. *Simula: A language for programming and description of discrete event systems. Introduction and user's manual*. Norwegian Computing Center, Forskningsveien 1 B, OSLO, Norway, 1967. Available from: `http://www.edelweb.fr/Simula/`.

[6] Alan C. Kay. The early history of Smalltalk. volume 28, pages 69–95, New York, NY, USA, 1993. ACM.

[7] Alain Colmerauer and Philippe Roussel. The birth of Prolog. pages 331–367, New York, NY, USA, 1996. ACM.

[8] Eiffel. Available from: `http://www.eiffel.com/`.

[9] Erlang. Available from: `http://www.erlang.org/`.

[10] Python. Available from: `http://www.python.org/`.

[11] Ruby. Available from: `http://ruby-lang.org/`.

[12] Scala. Available from: `http://www.scala-lang.org/`.

[13] Factor: a practical stack language. Available from: `http://factorcode.org/`.

[14] Clojure. Available from: `http://clojure.org/`.

[15] Go. Available from: `http://golang.org/`.

[16] Groovy. Available from: `http://groovy.codehaus.org/`.

[17] Jess: the rule engine for the Java platform. Available from: `http://www.jessrules.com/`.

[18] Fantom. Available from: `http://fantom.org/`.

[19] The zen of python. Available from: `http://www.python.org/dev/peps/pep-0020/`.

[20] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: framework and implementations. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–269, New York, NY, USA, 1990. ACM.

[21] John N. Shutt. Fexprs as the basis of Lisp function application or $vau : the ultimate abstraction, 2010. Available from: `http://www.wpi.edu/Pubs/ETD/Available/etd-090110-124904/unrestricted/j%shutt.pdf`.

[22] John N. Shutt. Revised Report on the Kernel Programming Language, 2009. Available from: `ftp://ftp.cs.wpi.edu/pub/techreports/pdf/05-07.pdf`.

[23] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Pixel. Available from: `http://merd.sourceforge.net/pixel/language-study/diagram.html`.

A Kai Source Code

A.1 Interpreted GCD

Here is the interpreted **GCD** function implemented using a lambda:

Listing 16: Interpreted GCD Function

```
(lambda `(a b) `{
    (if [b == 0]
        (return a)
        (return (gcd b [a % b]))
    )
})
```

A.2 Compiled GCD

Here is compiled **GCD** function implemented using compiler:

Listing 17: Compiled GCD Function

```
(compiler `gcd (function (int 32) (int 32) (int 32)) `(a b) `{
    (if [b == 0]
        (return a)
        (return (gcd b [a % b]))
    )
})
```

Compiled to LLVM IR using LLVM 2.7:

Listing 18: Compiled GCD Function : LLVM Assembler

```
define i32 @gcd(i32 %a, i32 %b) {
entry:
  %0 = icmp eq i32 %b, 0                          ; <i1> [#uses=1]
  br i1 %0, label %true, label %false

true:                                             ; preds = %entry
  ret i32 %a

false:                                            ; preds = %entry
  %1 = urem i32 %a, %b                            ; <i32> [#uses=1]
  %2 = call i32 @gcd(i32 %b, i32 %1)              ; <i32> [#uses=1]
  ret i32 %2
}
```

The X86 assembly code for the procedure is as follows:

Listing 19: Compiled GCD Function : X86 Assembler

```
0x0000000100d20010: sub    $0x8,%rsp
0x0000000100d20014: test   %esi,%esi
0x0000000100d20016: jne    0x100d20023
0x0000000100d2001c: mov    %edi,%eax
0x0000000100d2001e: add    $0x8,%rsp
0x0000000100d20022: retq
0x0000000100d20023: xor    %edx,%edx
```

```
0x0000000100d20025: mov     %edi,%eax
0x0000000100d20027: div     %esi
0x0000000100d20029: mov     $0x100d20010,%rax
0x0000000100d20033: mov     %esi,%edi
0x0000000100d20035: mov     %edx,%esi
0x0000000100d20037: callq   *%rax
0x0000000100d20039: add     $0x8,%rsp
0x0000000100d2003d: retq
```

A.3 Optimised GCD

The above compiled code can be further optimised by using tail call optimisation to remove recursion:

Listing 20: Optimised GCD Function : LLVM Assembler

```
define i32 @gcd(i32 %a, i32 %b) nounwind readnone {
entry:
  %0 = icmp eq i32 %b, 0                          ; <i1> [#uses=1]
  br i1 %0, label %true, label %tailrecurse

tailrecurse:                                      ; preds = %entry, %tailrecurse
  %b.tr2 = phi i32 [ %1, %tailrecurse ], [ %b, %entry ] ; <i32> [#uses=3]
  %a.tr1 = phi i32 [ %b.tr2, %tailrecurse ], [ %a, %entry ] ; <i32> [#uses=1]
  %1 = urem i32 %a.tr1, %b.tr2                    ; <i32> [#uses=2]
  %2 = icmp eq i32 %1, 0                          ; <i1> [#uses=1]
  br i1 %2, label %true, label %tailrecurse

true:                                             ; preds = %tailrecurse, %entry
  %a.tr.lcssa = phi i32 [ %a, %entry ], [ %b.tr2, %tailrecurse ] ; <i32> [#uses
      =1]
  ret i32 %a.tr.lcssa
}
```

The X86 assembly code for the procedure is as follows:

Listing 21: Optimised GCD Function : X86 Assembler

```
0x0000000100d20010: test    %esi,%esi
0x0000000100d20012: je      0x100d2002a
0x0000000100d20018: mov     %esi,%edx
0x0000000100d2001a: mov     %edi,%eax
0x0000000100d2001c: mov     %edx,%edi
0x0000000100d2001e: xor     %edx,%edx
0x0000000100d20020: div     %edi
0x0000000100d20022: test    %edx,%edx
0x0000000100d20024: jne     0x100d2001a
0x0000000100d2002a: mov     %edi,%eax
0x0000000100d2002c: retq
```

A.4 Pre-compiled GCD

A version of the **GCD** function was written in C and compiled directly inside Kai. Because of this, there is no compiler overhead.

Listing 22: Pre-compiled GCD Function : C++ Code

```
int gcd (int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}

Value * precompiledGCD (Frame * frame) {
    Integer * a = NULL, * b = NULL;

    frame->extract()(a)(b);

    int result = gcd(a->value(), b->value());

    return new Integer(result);
}
```

The X86 assembly code for the procedure is as follows, compiled using 'g++ -Os':

Listing 23: Pre-compiled GCD Function : X86 Assembler

```
0x0000000100013020: push   %rbp
0x0000000100013021: mov    %rsp,%rbp
0x0000000100013024: sub    $0x10,%rsp
0x0000000100013028: mov    %edi,-0x4(%rbp)
0x000000010001302b: mov    %esi,-0x8(%rbp)
0x000000010001302e: cmpl   $0x0,-0x8(%rbp)
0x0000000100013032: jne    0x10001303c <_ZN12_GLOBAL__N_13gcdEii+28>
0x0000000100013034: mov    -0x4(%rbp),%eax
0x0000000100013037: mov    %eax,-0xc(%rbp)
0x000000010001303a: jmp    0x100013054 <_ZN12_GLOBAL__N_13gcdEii+52>
0x000000010001303c: mov    -0x4(%rbp),%edx
0x000000010001303f: mov    %edx,%eax
0x0000000100013041: sar    $0x1f,%edx
0x0000000100013044: idivl  -0x8(%rbp)
0x0000000100013047: mov    %edx,%esi
0x0000000100013049: mov    -0x8(%rbp),%edi
0x000000010001304c: callq  0x100013020 <_ZN12_GLOBAL__N_13gcdEii>
0x0000000100013051: mov    %eax,-0xc(%rbp)
0x0000000100013054: mov    -0xc(%rbp),%eax
0x0000000100013057: leaveq
0x0000000100013058: retq
```

B A Brief Programming Language History

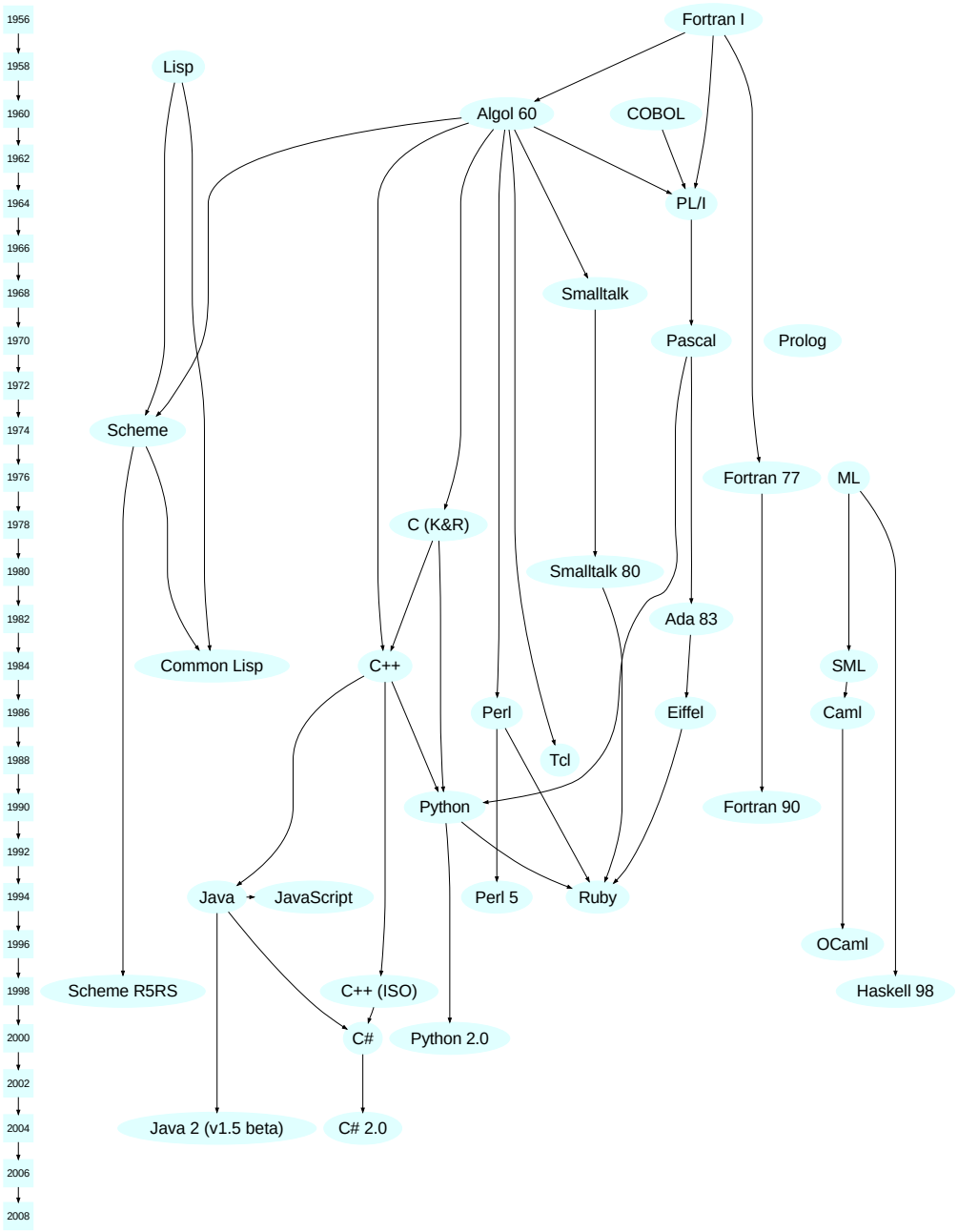

Figure 8: A simplified programming language history[24].