Design and Implementation of a Scripting Language for Mobile Devices

version 1.1.589

This document is available via `http://lightscript.net/`, which also contains the most recent version of LightScript.
Notice: the thesis describes the first version, and major changes has happend since then.

Rasmus Jensen[1]

2009

[1]rasmus@lightscript.net

Abstract

This thesis creates two new languages: Yolan and LightScript, which run on top of Java Micro Edition and enable scripting on very low-end mobile devices.

The code footprint is the major limitation on low-end mobile devices, and both languages have a significantly smaller code footprint than existing scripting languages. The languages are comparable in speed to larger scripting language implementations, and an order of magnitude faster than most of the benchmarked scripting languages for mobile devices.

Both Yolan and LightScript have first-class functions, dynamic typing, built-in support for hashtables, stacks, etc., and they support interactive programming. They are also able to load and execute scripts presented in source form at run-time. The Java Micro Edition does not support dynamic loading of code, so both languages are interpreted.

Yolan is highly optimised for reducing the size of the implementation code footprint. It has dynamic scope, a Lisp-inspired syntax, and is interpreted by evaluating each node of the syntax tree.

LightScript is a subset of JavaScript; it has static scoping, and also includes support for closures, objects and exceptions. It is compiled to, and executed on, a stack-based virtual machine. One of things that makes the compact implementation possible is that the LightScript parser is an imperative optimised version of the top-down operator precedence parser.

Yolan has a code footprint less than half the size of LightScript, and they are similar in speed, even though they have very different evaluation strategies. Yolan served as a proof of concept and as a stepping-stone toward LightScript. LightScript is now a practical language for mobile development and will continue to be developed after this thesis.

Resumé

I dette speciale er der udviklet to nye scriptingsprog til mobiltelefoner: Yolan og LightScript. De er udviklet i Java Micro Edition, der er den programudviklingsplatform, der gør det muligt at køre på flest telefoner.

Sprogene er optimerede med hensyn til størrelsen af det eksekverbare program, da denne er den mest begrænsende faktor på Java Micro Edition. Sprogene udviklet i dette speciale er mindre end de eksisterende scriptingsprog. Udførselshastigheden er på niveau med større scriptingsprogsimplementationer og er betydeligt hurtigere end mange af de eksisterende små scriptingsprog.

Både Yolan og LightScript har funktioner som værdier, har dynamisk typesystem, har indbyggede hashtabeller, stakke, osv., og kan udføre programstumper, der indlæses som kildekode under kørslen. Begge sprog er fortolkede, da Java Micro Edition ikke understøtter indlæsning af ny kode under kørslen.

Yolan er optimeret med henblik på programstørrelsen. Sproget har dynamisk virtkefelt, Lisp-inspireret syntaks, og det bliver fortolket ved at gennemløbe syntakstræet og evaluere de enkelte knuder.

LightScript er kompatibelt med en delmængde af JavaScript, hvilket gør sproget lettere at gå i gang med at programmere i for andre. Det har statisk virkefelt, understøtter "closures", undtagelser og objektorienteret programmering. Det oversættes til, og udføres på, en stakbaseret virtuel maskine.

Yolan fylder cirka halvt så meget som LightScript, og de er omtrent lige hurtige, til trods for deres forskellige tilgange til programudførsel. Yolan afdækker mulighederne for scriptingsprogsudvikling indenfor mobiltelefonernes begrænsninger, og det tjener også som en grundsten for udviklingen af LightScript. LightScript er et praktisk værktøj til udvikling af software til mobiltelefoner, og det vil fortsat blive videreudviklet efter dette speciale.

Contents

Chapter 1

Introduction

The topic of the project is to design and implement a scripting language that runs on very low-end mobile devices. This is both to create a practical tool, and also a focus for exploration of programming language theory.

The educational goals are to learn about programming language design and implementation, and to learn about programming on mobile devices. Through the project, I should be able to evaluate and choose programming language features and implementation techniques, and design and implement a scripting language.

The focus of the language is that it should be portable, embeddable and have a low memory footprint. Portable implies that it should run on different devices, from very low-end mobile phones to high-end computers, possibly also within a web browser. Embeddable implies that it should be easy to include within and interface with other applications. Low memory footprint implies that it should be suitable for running on platforms where the available memory is measured in kilobytes rather than megabytes.

The approach will be pragmatic and favor simplicity.

1.1 Motivation

Scripting languages make it faster to write applications [Ous98]: The type systems allow for more reuse and compared to traditional system programming languages. They are also more expressive, with more instructions per statement and support for higher level language features such as first-class functions. In addition, scripting languages open up for code at run-time, allowing user based scripting and scriptable configurations.

On more powerful devices, ranging from high-end smartphones to personal computers, there are very good scripting languages available. Scripting language implementations, however, usually take a lot of resources, which is a problem on low-end mobile devices. On those devices, implementations may not be available, may be very slow, or may have limitations, such as not being able to be executed directly, but needing to be compiled on another machine, or they do not have basic data types. The focus on a better implementation of scripting languages for mobile devices is thus a niche, where the result may actually be

of practical use.

The focus on low-end devices also has another benefit: it broadens the number of devices on which the language may run. While very low-end devices are becoming uncommon in Denmark, they still live on in countries with less information technology penetration. Thus, by targeting very low-end devices, this project may make scripting, and thus easier content creation, more available, and that could be the beginning of a stepping stone towards more information and computing literacy. The restrictions of low-end mobile devices also imposes challenges that may lead to interesting solutions.

1.2 The structure and content of the thesis

This first chapter is an introduction to the project, defining the method and overall direction. The second chapter is a survey of topics related to the projects. This contains the background for decisions on the choice of platform, programming optimisations, language ideas gotten from other languages, and language implementation techniques. The third and the fourth chapters document two scripting languages developed through this thesis: Yolan and LightScript. These contain design choices for the languages, implementation details, descriptions of the languages themselves, and developer guides for embedding and using the languages. The fifth chapter benchmarks the developed languages against other scripting languages, and the sixth and seventh chapters are discussion and conclusion of the work.

The appendices contain the source code for the core Yolan and LightScript classes.

1.3 Approach

When making a project, the approach can be more or less bottom-up or top-down. A top-down approach could be like the waterfall model [Roy87] of software development, where we start with the specification, design and then implementation and test. The bottom-up approach is characterised by more agile [BBvB$^+$01] methods, starting out with a quick prototype, which gets more and more refined.

The purpose of this project is to learn about programming languages and mobile development, and create a scripting language. The bottom-up approach is chosen, as it opens more up for experimentation, and new ideas are easier to follow, than with the top-down approach.

The methodology of the project was to start out with a series of experimental prototypes related to mobile scripting language implementation, while surveying the field. The best parts of the prototypes were then expanded and built further upon, to create the actual scripting language. Discarded prototypes include a Forth-like language, several parser/compiler implementations on top of EcmaScript, a couple of virtual machines on the JVM, some experiments towards an implementation in C, experiments with mobile applications and their GUI, and drafts of parts of what became Yolan and LightScript.

Chapter 2

Survey

The survey first looks at different mobile platform to motivate the choice of the target platform for the language. We choose the Java Micro Edition platform, and section 2.2 then then looks into implementation methods for that platform. Section 2.3 then surveys other languages related to scripting and low-end devices, and the final section looks into implementation techniques relevant for the scripting language implementation on top of the chosen platform.

2.1 Mobile platforms

This section surveys the different development platforms for mobile devices, to determine a target platform for the language. Mobile devices ranges from low-end phones, which, if they are programmable at all, only support Java Micro Edition, up to advanced smartphones with performance resources similar to older desktop computers.

Most low-end phones support some kind of Java midlets.[1] Here there are various APIs and device profiles, but the basic execution and deployment model is the same for all devices that support midlets.

Low end devices rarely support loading of native code, and higher end devices may require various kinds of code signing to allow native programs to be distributed.

Besides Java and native code, JavaScript is becoming a potential language for applications for high-end phone. This is both due to its integration with the web, which means that it is available on the phones with advanced browsers, due to the increased amount of memory on high-end phones and due to recent major performance advances within the JavaScript implementations, which are propagating towards mobile devices.

2.1.1 Embedded systems

Low- to mid-end mobile phones are embedded devices, where high-end phones are getting closer to being personal computers. A characterising feature of em-

[1] A midlet is a small Java application targeting mobile devices, similar to applets, which are small Java applications targeting web browsers

bedded systems is that the software is shipped with the devices, and there are little or no support for software development. So while there is support for execution of midlets, native applications are not immediately possible to develop, unless except for the hardware developer.

There are ways to experiment with development for low-end devices nontheless: Some devices are hackable in the sense that somebody has found out how to upload and customise the firmware, usually without documentation and support from the manufacturer – examples here range from digital cameras [can09], mobile phones [**?**] and handheld gaming devices[nin08]. A few devices open up for development.

An interesting case is the Lego Mindstorm NXT, which has hardware similar to that of very low-end mobile phones, but without sound and GSM. The hardware contains an ARM cpu[ARM01], which is the same as that used in many mobile devices. It runs at 48MHz, and has 256KB of ROM, and 64KB of RAM, plus a coprocessor, a low resolution small display and a couple of buttons. This may be used for prototyping an implementation of a scripting language, targeting embedded devices.

2.1.2 The J2ME / Java Micro Edition

J2ME (Java Micro Edition[2]) is the most common platform for mobile applications, supported by more than a billion devices [Sun09]. Two thirds of mobile phones shipped today supports Java [Esm08]. Most mobile devices either require strict code signing, or do not allow native applications to be loaded at all, implying that J2ME is often the only option for mobile application development. J2ME is a trimmed down Java Virtual Machine (JVM) so it has most of the features and limitations of a standard Java JVM. It is a heterogenous platform: there are different APIs/extensions from different vendors, and different device profiles for different hardware capabilities, meaning that the applications need to be ported.

Applications for J2ME, are called Midlets and are distributed via JAR-files (Java ARchive). A JAR file is essentially a zip file containing the compiled Java classes, data files, and some meta information.

J2ME has two device configurations CLDC 1.0 (Compact Limited Device Configuration 1.0) [Sun00] and CLDC 1.1 [Sun03b]. The major difference between the two is that CLDC 1.0 is integer only, whereas CLDC 1.1 supports floating point numbers. Approximately $\frac{1}{6}$ of the mobile phones that support J2ME are limited to CLDC 1.0, whereas CLDC 1.1 is supported by the remaining $\frac{5}{6}$ of the phones [Get09]. A limitation of both CLDCs is that they do not support reflection nor dynamic code loading. The lack of a reflection API implies that a scripting language implementation cannot discover native functions itself. This means that foreign function interface between Java and a scripting language, cannot discover or work with Java dynamically, but Java functions that should be callable from the scripting language must be coded into the implementation before deployment. The lack of run-time class loading means

[2]Java Micro Edition is a recent rebranding of J2ME

that JIT compiling to the JVM is not possible. Nor is native code available, so the only way to execute scripts loaded at run-time is through interpretation, possibly via a virtual machine.

The CLDCs are based upon the Java JVM [LY99] with some instructions removed, and some metadata added to ensure stack discipline. In order to simplify the J2ME JVM implementation, Java class files have to be preverified, before they can be loaded. The preverification adds meta data about stack use, and removes certain instructions, such as local jump-and-save-register, to make it easier to implement a JVM that is safe against malicious code trying to overflow the stack. The reference implementation of CLDC 1.0, KVM, is a switch-based interpreter with a compacting mark-and-sweep garbage collector [Sun03a].

The JVM limits interpreter implementation: it does not support label references as values, nor does it support functions as values. This makes some of the optimisations discussed in Section 2.4 impossible. Instead, it does have a built in switch opcode as well as support for method-dispatch based on the type of an object, so an interpreter could be switch-based, or have a class for every opcode.

The resources available for J2ME applications on mobile phones start at 64KB for the size of the JAR-file and 200KB for the run-time memory on the lowest end devices [Nok04]. These numbers are for the full application, so the resources available for an embeddable scripting language could be significantly less than this, depending on the resource usage of the application in which the scripting language is embedded.

As Java Micro Edition is the most widespread platform for mobile devices, this will be the target platform for the scripting language during the thesis.

2.1.3 Smartphones

Many high-end phones – smartphones – allow loading of native code. This covers approximately 12% of the mobile phone market measured in number of units.[3] Smartphone development platforms give more access to the devices than J2ME, but they are at the same time more difficult to develop for, due to the large number of platforms and limits on distribution due to requirement on cryptographic signing of code.

The main operating systems for smartphones are: Symbian, RIM, Windows Mobile, Mac OS X and Linux, which are very different operating systems, and on top of the operating system, different phones also have different user interfaces and programming models. Development is not only possible as native applications, but many of the devices also support J2ME or other Java implementations, and on those devices that include a modern web browser, JavaScript is also a possible target for application development. If the scripting language is implemented on top of Java, then it will also run on many smartphones. It is important to note, however, that some devices only support Java dialects other than Java Micro Edition, so that should be kept in mind when developing

[3] In the fourth quarter of 2008 38.1 million smartphones where sold [Gar09], whereas the total number of mobile phones sold in the same period were 314.7 million devices [CN09].

the language. In addition, if the developed scripting language is a subset of JavaScript, it will already have an implementation deployed on those devices with advanced browsers.

2.2 Optimising Java for low-end mobile devices

The next two sections look into methods of reducing the memory usage and code footprint of Java midlets. Memory usage and code footprint are very limited resources on some mobile devices, especially the code footprint, and techniques for optimising these will be needed when implementing the language.

2.2.1 Reducing the memory usage of J2ME applications

Some devices only have little memory available, and an optimisation here is to be able to avoid having memory intensive parts of the program run at the same time. For example, with a scripting language, it is desirable to be able to garbage collect the executed parts of a long script when they are done, so that the memory becomes available for other computations. The usual size optimisations, such as finding compact representations, trimming dynamic data structures, avoiding sparse data, etc., are also applicable. There may be a tradeoff between the code footprint and run-time memory, as compact representations and other optimisations may require more code to be implemented.

2.2.2 Reducing the footprint of J2ME applications

Some optimisations to reduce the code or JAR-file footprint:

- Reduce the number of class-files.
- Write initialisation manually, where the automatically generated initialisation is inefficient.
- Use a JAR-optimiser/obscurifier.
- Put the classes in the unnamed package.

Another optimisation is to reduce the number of classes [Nok04, Kar08]. This may reduce the size of the JAR file significantly, even though it does not change the amount of code: Each class file has its own symbol table, which means that if classes are merged, then common symbols only need to be represented once, rather than once for each class. Furthermore JAR files are essentially zip-archives, and each file in a zip archive is compressed individually [Pkw07], which means that small files typically get compressed less than larger ones, due to the small compression context. The downside of reducing the number of classes is that it could go against the object-oriented design, and the implementation of the scripting language may become more difficult to read and edit.

Initial values are not supported directly by the Java class file format, but instead JVM-code is generated, that does the initialisation. This code is often inefficient. For example the initialisation:

```
byte[] bytes = { 1, 4, 3, 4, 5, 6, 2, 3, 1 } ;
```

generates code corresponding to

```
byte[] bytes = new byte[9];
bytes[0] = 1; bytes[1] = 4; bytes[2] = 3;
bytes[3] = 4; bytes[4] = 5; bytes[5] = 6;
bytes[6] = 2; bytes[7] = 3; bytes[8] = 1;
```

which for a larger initialisation is significantly more expensive than the following manually written initialiser:

```
byte[] bytes = "\001\u004\u003\u004\u005\u006\u002\u003\u001".getBytes();
```

Another thing to be aware of is that strings in Java class files are encoded such that only characters with unicode values between 1 and 127 (inclusive) use one byte per character. Binary data, which use the entire range in a byte (0-255), may therefor uses several bytes per byte if encoded as strings, and it may therefore make sense to place larger binary data objects, in external files, rather than within the class file.

JAR file optimisation/obfuscation can reduce code footprint because it may remove unused code, rename methods and classes, such that they use less space in the symbol table, make `static const`s work as `#define`s, and optimise the code, thus shortening it. This also allows one to write more readable code with less concern for the code footprint, knowing that some of the more verbose parts will be optimised away.

Using the unnamed package saves space in the symbol table, as class references become shorter.

The code footprint limit may be tighter that the run-time memory limit, and it may be possible to partition the execution such that different parts of the application do not need to use run-time memory at the same time. Therefore, in this project with the design of an embedded scripting language, the code footprint will have slightly higher priority than the run-time memory usage, although both are important.

2.3 Programming Languages

The following sections survey some exixting programming languages that have served as inspiration for this project:

- Forth is surveyed as it runs on very minimalistic systems, and it is also a stack-based language, thus relevant for the design of virtual machines.
- Hecl is the major scripting language for mobile devices
- JavaScript is the basis of the LightScript language implemented in this thesis.

- Lua has a focus on embedded devices, and solutions from their implementation can inspire the languages created in this thesis. There is also an implementation of the Lua virtual machine targeting mobile devices, which is why this language is also related to the benchmarks presented later on.
- Python is becoming relevant on mobile devices, as it runs on many smartphones. Some of its language features may be usable in the design of new languages.
- Lisp and Scheme are some of the inspirations for Yolan, especially with regard to the syntax.
- Self is highly relevant for the LightScript implementation, as LightScript's object system is based on this, via JavaScript.

2.3.1 Forth and other stack-based languages

Forth is a stack-based language. It is very interesting for this project, as it is an example of an interpreted minimalistic language running on even very low-end devices.

The syntax of Forth is different from that of most modern languages, due to the use of reverse polish notation. Functions work on a stack, which the programmer is explicitly aware of. Functions replace their arguments on the top of the stack with the result. Besides the stack for arguments, there is also a stack for return addresses.

Forth supports meta programming. Forth systems have explicit compilation, the system has two modes: compile mode and interpret mode. It is possible to create words that are executed at compile time, using the immediate keyword. Words – "functions" in Forth – are first-class data types.

An introduction to Forth is "Thinking Forth" [Bro84] which covers, not only Forth programming, but it also touches a lot programming and problem solving in general. This include details on implementation of stack-based languages. There is also a tutorial Forth implementation [Jon07], that shows how Forth, and thereby stack machines, can be implemented.

Forth is also interesting from the virtual machine point of view, as it is similar to the machine code of a stack-based virtual machines. So Forth implementation techniques share a lot with those of virtual machines. And research within this topic overlaps, an example is a dissertation on implementation of stack languages on register-based machine [Ert96], which uses Forth as the representative stack-based language.

Other more recent stack languages are Joy, Cat, and Factor [Pes08]. Interesting developments here are stronger typing, and also the use of anonymous code blocks to create the control structure. The idea of the anonymous code block is an inspiration for the Yolan scripting language developed in this thesis, as the concept of delayed computation is somewhat similar.

2.3.2 Hecl and Tcl

Hecl is relevant to investigate, because it seems to be the major mobile scripting language. It is a dialect of Tcl (Tool Command Language), which is an embeddable scripting language [Ous94] with a prefix notation, i.e. the syntax is similar to Logo, – and also similar to Lisp if we see each unquoted line-break as an end+begin parenthesis. Comparisons and mathematical expressions need to be written lisp-like in Hecl, whereas Tcl has an `expr` function, that evaluate an expression taking care of operator precedence etc. This indicate that a simple syntax need not be a hindrance for scripting languages on mobile platforms. The documentation and online materials seems well rounded, which may be a factor to it popularity.

2.3.3 JavaScript/EcmaScript

JavaScript was created as a scripting language for web browsers, and was later standardised under the name of EcmaScript [ECM99]. The reason it has become interesting is that it is included within most web browsers, and is thereby one of the most widely available platforms.

JavaScript is a dynamically typed scripting language with closures / first-class functions, and a prototype-based object system, similar to Self. While it has expressive features, it is also a very mainstream language, both through its presence within the web and its C-like syntax, and thus making it possibly less intimidating for new programmers.

JavaScript is very relevant for this thesis, as the LightScript language developed here, is a subset of JavaScript/EcmaScript, for several reasons: It already exists as a platform meaning that the new scripting language already has virtual machines deployed. As the EcmaScript syntax is known, or at least recognisable for many developers, even web-"developers" and users, it may be easier for them to transition, or start using the new scripting language on mobile devices. EcmaScript supports closures and first-class functions, which can thus be included in LightScript without breaking compatibility.

An issue with EcmaScript is that it has some unfortunate design decisions [Cro09a], which will be elaborated on in Section 4.1.

The following paragraphs describes the main open source implementations of JavaScript/EcmaScript.

JavaScriptCore JavaScriptCore [Web09] is the implementation of JavaScript in Webkit. The latest version is called SquirrelFish and is an optimised register-based virtual machine, where support for JIT compilation is also being developed.

The engine has been optimised a lot within the last year [Sta08]: the earlier versions were performing evaluation by walking through the abstract syntax tree, and were at that time similar in performance to SpiderMonkey JavaScript implementation. The new version – SquirrelFish – is much faster, which means that it is similar in performance to TraceMonkey and V8 JavaScript implementations.

QScript QScript [Nok09] is the EcmaScript implementation that is a part of QT.[4] This implementation is targeting application scripting rather that web scripting. An interesting part of QScript is its support for generation of bindings to native functions. As WebKit is being integrated with QT, QScript is likely to be replaced with or merged with the WebKit scripting engine in the long run.

Rhino Rhino [Moz07] is an implementation of JavaScript on top of Java. The implementation is mainly used as an embedded language for Java applications. The implementation transforms the JavaScript program to Java classes, which are then executed.

SpiderMonkey SpiderMonkey [Moz] is the JavaScript engine in Mozilla's browsers, and is also usable as an embeddable engine in other applications. Execution is based on a virtual machine, which either has a switch-based dispatch or direct threading, depending on what the compiler supports.

TraceMonkey TraceMonkey [Eic08] is a branch of SpiderMonkey, which incooperates JIT compilation, with a JIT compiler based on trace trees [GF06, CBYG07], which, rather than JIT-compiling the entire program, traces and then compiles the most executed paths through the program. This means that the JIT compiled code gets optimised to the actual execution, and less code needed to be compiled.

V8 V8 [Goo08] is a JavaScript implementation made by Google, and released in 2008 in conjunction with the release of their browser "Chrome". While it is not directly connected with the Android platform, a JIT compiler is already in place targeting the ARM CPU, so it seems likely that it will also target mobile devices in the long term.

A main focus and benefit of V8 is high execution speed. It is designed for JIT compilation from the beginning, and it also does some class inference to optimise methods and property accesses. Specifically, whenever a property is added to a JavaScript object this creates an implicit class. When code is JIT'ed, a property access is compiled to a type check followed by fast code for accessesing the property, similar to that of more static, class based languages. This is possible due to the type check and implicit class, and it is much faster than a traditional JavaScript object property lookup.

The garbage collector is fast, - it is a generational garbage collector with two generations. The young generation is collected with a copy-collector which is linear in the amount of live data, and thus heap allocation of activation records is almost free. This is combined with a mark and sweep collector when a full collection is needed.

[4] QT is a programming API and cross platform graphical toolkit. It is the base for KDE (the K Desktop Environment) as well as the GUI for several Motorola mobile phones. It was recently acquired by Nokia, who has released it under the LGPL licences, and is making it a part of their Symbian platform.

Targeting mobile devices

During the writing of this thesis, it has turned out that there are other projects working on JavaScript languages for mobile devices.

There is a standardised EcmaScript Mobile Profile, which removes some features from the language, but is not implementable in practice on top of CLDC 1.0 as it still relies on floating point numbers.

Mojax (Mobile Ajax) [Aig] is a properitary virtual machine for an EcmaScript-like language. The language is also a subset of of EcmaScript, integer only, and appears not to support exceptions nor regular expressions. The Mojax implementation appears to be a virtual machine that cannot execute scripts directly, but they need to be compiled before they can be loaded.

There is a closed source implementation of a subset of JavaScript, that targets embedded devices, written in C, and developed by Mbedthis [Mbe04]. It uses integers and does not have exceptions, labelled statements, switch, while, do-while, regular expressions, function literals, object literals, array literals, prototypes nor class methods. Recently that implementation seems to have been abandoned, in favor of a new, more compliant implementation: Ejscript [Sof09]. A C based implementation is available under the GPL, there is apparently a J2ME implemention that is not published yet.

2.3.4 Lua

Lua is the language that comes closet to the goals of this project: it is a scripting language with first-class functions that runs on embedded and mobile devices. In addition, it is a dynamically typed, statically scoped embeddable scripting language, which is characterized by only having one data structure: associative tables. It has both good performance and a relatively low code footprint.

On mobile devices there is an implementation of the Lua virtual machine called Kahlua [Kar09]. It is only a virtual machine, meaning that the Lua scripts need to be compiled on another platform, before they can be loaded and executed.

2.3.5 Python

As phones are becoming more powerful, Python is beginning to play a role as a mobile scripting language on high-end devices. This is probably due to that Pythin is already a popular and major scripting language on computer platforms.

Python has some nice features, that may be inspiring when implementing other languages. Comments/documentation are integrated in the language via the concept of docstrings, which are a special kind of comment that is also available for inspection during runtime. Unit testing is also integrated within the documentation framework, where a special syntax indicates that code in the documentation can also be executed when running tests. Required indentation leads to more easily readable programs.

The issue with Python on mobile devices is that its implementation is relatively resource intensive, and it is thus currently only available on high-end

phones.

2.3.6 Scheme and Lisp

Scheme and Lisp are especially interesting, due to their minimalism, which is inspiring in the design of a language with strong restrictions on the implementation environment. The syntax is mostly isomorphic with the abstract syntax tree, meaning that parsing is mostly trivial.

Early versions of JScheme[Nor98] have code footprints that are small enough for mobile devices, but would need to be ported, as they use reflection that is not available on mobile devices. JScheme is interesting as a benchmarking target, as it is a high level language with a small footprint.

2.3.7 Self

Self [US87] is in particular interesting due to the prototypical inheritance, that is the object model of EcmaScript and thus also LightScript. Traditional object oriented programming, is designed around classes and objects, where classes can inherit from other classes, and objects instantiate classes. Instead of classes and subclasses for inheritance, Self has a clone operator, which creates a new object using an existing object as blueprint. An object in Self contains a pointer to the parent (cloned) object, and a mapping from property names to values or methods. When a property is read, the mapping of the current object is first searched, and if the name is not found there, then the parent objects are searched for the property.

2.4 Interpreter implementation

As we are building on top of Java, several interpreter implementation issues become less relevant:

- On JVM platform, the dispatch can only be implemented with a switch statement or be based on class types. The latter is much more expensive, spacewise, and therefore not applicable. Threaded dispatch, and other faster dispatch methods are not possible due to lack of support for using pointers to code.

- With regard to garbage collection, we can piggyback on the garbage collector from the JVM, which reduces the code footprint as we do not need to implement one ourselves. This has the issue that we can not rely on how it is implemented, meaning that the use of heap allocated activation records may be less of an option, as it is stongly affected by the performance of the garbage collector.

- JIT compilation to native code, or targeting the JVM, is not possible as dynamic loading of code is not supported at all on most Java Micro Edition implementations.

The next section looks at register vs. stack virtual machines, which is an important aspect at the design of virtual machines. This is followed by a look at parsing, more specifically top-down operator precedence parsing as this has low code footprint, which is important in our context. And finally we consider the issue of scope and the implementation of variables.

2.4.1 Stack and register based virtual machines

Virtual machines are usually either stack-based or register-based.

A stack-based virtual machine operates similar to the language Forth, where all operations work on the top of the stack. Operands are implicitly coded, such that for example the add instruction just pops the top two elements of the stack, and pushes the sum. Stack machines are easy to compile to, which can simply be done by emitting the opcodes of a post-order walk through of the abstract syntax tree. There are no issues of register allocation, spilling, etc. Stack machines are commonly used, the best known example being the Java Virtual Machine, and there are many other languages that turn out to be implemented on stack machines when looking under the hood, for example: Python, the SpiderMonkey JavaScript implementation, and the .NET Common Intermediate Language.

Register-based virtual machines are becoming more common. Usually they have a high number of registers, leading to longer opcodes than stack-based virtual machines. But on the other hand, they have fewer opcode per program, leading to faster execution [SGBE08, DBC$^+$02]. Examples of register-based virtual machines are the Dalvik [Bor08] virtual machine, LLVM [LA04], Parrot [Fag05], and the virtual machine of Lua 5.0 [IdFC05].

A third approach to implementing a virtual machine is just to use the abstract syntax tree for evaluation. This was for example used in earlier versions of WebKits JavaScript implementation, but has now been superseded by a stack-based virtual machine, which is currently being replaced by JIT compilation.

2.4.2 Parsing

A language implementation must parse the source text into the abstract syntax tree that is manipulated by the rest of the compiler.

Parsers often take quite a large amount of code, especially if they are generated by compiler-compilers. Generated lexers and LALR(1) parsers usually have quite large state tables. Recursive descent parsers seem to use a bit less code footprint that LALR parsers, but still require functions for all the grammar productions, which still takes quite a bit of space.. Grammar-based parser generation and implementation have been extensively studied and will not be discussed further.

Another approach, which is also very elegant, but has recieved less attention is the top-down operator precedence parser. This kind of parser has the benefit of a low code footprint, partly due to genericity of the parser code, for example: only one parsing function for the left associative infix operators, one function for the right associative infix operators, one function is needed for prefix opera-

tors, etc. So a top-down operator precedence parser will be used for parsing in LightScript.

Top-down operator precedence parsers

Top-down operator precedence parsing combines recursive descent parsing with operator precedence, which simplifies the implementation significantly. It was first described thirty years ago [Pra73], but has not received much attention until lately [OW07, Cro09b].

A token may have two functions and a precedence, which are used to build the syntax tree when the token is encountered. If the token stands first (is leftmost) in an expression, and the null denominator function is use to build the abstract syntax tree node.. The second function is the left denominator function is used to build the abstract syntax tree node, if we already have something to the left of the token within the expression. The precedence of the next token is used to determine whether we have finished parsing an expression or need to use the token to build another abstract syntax tree node by calling left denominator function of the token.

To be able to build the abstract syntax tree node, the left denominator function takes an argument, which is the parsed node to the left of the token. Additionally it is possible for the denominator functions to parse expressions to the right of the token by calling the parsing function recursively. Here it is necessary that the parsing function also take a priority as a parameter. This priority parameter ensures that the left denominator functions are not called for tokens with lower priority.

The parsing itself is then simply done by first calling the null denominator function of the first token, and then calling the left denominator functions of the next tokens, as long as the next token has higher priority than the priority passed to the parsing function. The denominator functions attached to each token are then responsible for building the syntax tree. So the core loop of the parser is implemented as follows:

```
define parse(int priority):
    syntax_tree = next_token.null_denominator()
    while next_token.priority > priority:
        syntax_tree = next_token.left_denominator(syntax_tree)
    return syntax_tree
```

Now we need to assign meaningful precedence and denominator functions to the tokens: Atoms, variable names, literals etc., have a null denominator function that returns their node. Unary operations such as minus, not, etc., have a null denominator function that calls parse once and creates a node that applies the operator to the parsed node. Binary (infix) operators have a left denominator function that creates a node with its parameter as the left hand side and calls parse to construct the right hand side. Binary operators can be made right associative by passing a priority to parse, that is one less than the priority of the operator itself. The parsing of lists calls parse until the end of the list is reached. Other constructions can be implemented similarly.

The limitation of this parser as described here is that only one left-hand side expression is passed to the left denominator function, making reverse polish notation languages difficult to implement. A non-recursive version with an explicit stack could solve that.

2.4.3 Scope

Defining the scope of variable declarations is important when designing a programming language. There are two major kind of scopes: static scoping and dynamic scoping. Static scoping is also called lexical scoping, and corresponds to the lexical structure of the source code. This is in contrast to dynamic scoping, where variables are accessible other places than in the blocks where they are defined. The difference between dynamic and static scoping is exemplified by the following:

```
function f() {
    x := 17
}
function g(x) {
    f()
    print(x)
}
g(42)
```

which would print 42 in a statically scoped language, and would print 17 in a dynamically scoped language.

Static scoping is more natural as the scope matches the structure of the code. Dynamic scoping requires more discipline, as it allows the programmer to tamper with local variables of other parts of the code, and counters good habits of information hiding. A very important feature is also that static scoping can be used to create closures for functions, which gives extra flexibility to the language. On the other hand, dynamic scoping can simply be implemented by a global mapping from name to value, and can thus use slightly less space in the code footprint. Dynamic scoping also does not suffer from the funarg problem discussed in Section 2.4.3. For more advanced language implementations static scoping has the benefit over dynamic scoping that code is easier to analyse and optimise due to locality, in the sense that the use of variables is limited to the local lexical scope.

Stacks and activation records

A typical way to implement local variables is by using a stack. In this section, we assume that we are on a full stack machine; in practical implementations on CPUs, the top of the stack is usually implemented in a number of registers instead.

Whenever a function is called, the arguments are pushed onto the stack. Then, when the function is entered, a return pointer is often pushed onto the same stack, and space is allocated for the local variables. The computation is

then done, and the function returns, jumping back to the call site and restoring the stack size to the original. The allocation on the stack allows for local variables, and functions can also be recursive. The part of the stack containing the local variables, etc., is called the activation record for the function.

Scope and first-class functions

When we have first-class functions and static scoping, we cannot just place the activation records on the stack. When functions are nested, inner functions may live on, and access local variables of the outer function, after the outer function has returned. Consider the following code:

```
function f(x) {
    function g(y) {
        return x + y
    }
    return g
}
```

In this code the function g lives on after f has returned, and at the same time g uses a value that lies in the activation record of f. Therefore, if the activation record is allocated on the call stack, and nothing further is done about it, the value of x will no longer be available when g is called. This is an example of the "funarg problem", which is the problem with scoping when a function is given as an argument or returned.

There are several solutions to this problem:

- Disallow/not support first-order nested functions or access to outer scope within nested functions. Or not having static scope.
- Disallow scope variables to be changed from the inner scope. Instead such variables are passed as immutable values at function/scope creation.
- Allocate the activation records on the heap, instead of on the stack, and let them be garbage collected.
- Keep track of which variables may live on after the function exits, and move those to the heap.

Not having nested functions, only having a single-function-local and a global scope, or not having static scope, simplifies the implementation and is a solution often taken in mostly imperative languages.

Copying the values of outer scope variables to an inner scope is possible in a purely functional language, as variables are not mutated. If the language is not purely functional, when a variable is mutated in an inner function, it is only the copy that is mutated, and the mutation is local to the function. A workaround for mutations in languages that copy variables from outer to inner scope is to use a reference to the mutated variable rather than the variable itself, such that when the reference is copied to the inner scope it is still possible to mutate the value, as we need not mutate the reference itself. Another approach is to let

the outer scope stay alive until the inner closure dies, and then just reference variables as usual. This can be done by allocating the activation records on a garbage-collected heap, which is not as expensive as it may seem, due to the high effiency of modern garbage collectors. Another approach is to keep track of which variables may stay alive at the exit of a function, and move them to the heap at that time. Or just identify variables that are used by in inner functions, and put those on the heap.

Chapter 3

Yolan

Yolan is an experiment in minimalism of language design and implementation. The goal is to minimise the code footprint, while still having a practical scripting language with first-class functions. I call the language Yolan as a contraction of yocto-language, where yocto- is the SI-unit of 10^{-24} and thus means very very small. Yolan also serves as a pilot system and stepping stone for further development.

We first describe the design choices that lie as a base for the language, then Section 3.2 elaborates on a detail in the implementation. Section 3.3 describes the language, and finally Section 3.4 describes how to embed the language within an application.

3.1 Design choices

The next paragraphs elaborate on the following design choices:

- The implementation must be able to run on CLDC 1.0
- It should use standard Java classes, for easier interaction with Java code
- Code should be loadable/executable as source code at run-time
- Functions should be first-class values, to make the language more expressive
- Variable look up should should be fast
- Only integers will be supported, - no floating point numbers
- Null is the false value
- Syntax should be Lisp-like, to reduce the size of the parser
- Scoping should be dynamic
- Arrays, hashtables, etc. should be built in in the language
- The interface with Java should be lazy, to allow implementation of custom control structures

- Execution is online / one statement at a time
- The implementation is single threaded/non-reentrant to reduce code size

CLDC 1.0 is the most limited device configuration and API for mobile devices. If the language is able to run this configuration, it will also be able to run on the other Java configurations, and then be able to run on most devices.

Use of standard Java classes instead of custom classes for builtin data types such as arrays or tables has three benefits: the code footprint is smaller, as these data types do not have to be reimplemented, it may be faster, as the builtins may be implemented natively, and it is more easily embeddable, as the developers already know the data types from standard Java and the host program may be using those data structures already. Nevertheless, using the builtins gives less flexibility to the design of the behaviour of the data types in Yolan.

Being able to load code at run-time is important for several reasons: It allows larger programs than would otherwise be possible within the devices, as the different parts of the program can be loaded and unloaded as needed. It allows adding updates and new features to the programs, without needing them to be reinstalled manually. It allows the language to be used for configuration files, and data initialisation. Being loaded as source code both gives more clarity, and make the scripts easier to edit and deploy, thereby making it more suitable for use as configuration language, and scripting by users.

Functions should be first-class values, both to enable functional programming and increase the expressiveness of the language.

Variable access happens very often, so if the scripting language does a full lookup of the variable name at each access, it will be a bottleneck for the performance. Instead the implementation must ensure that they are only resolved once for each place they are used in the source text.

Integer is the only number type supported on the platform. Floating point numbers would have to be emulated on CLDC-1.0 based devices, which would both give a performance penalty, and more importantly a huge increase in code footprint due to emulation code.

Null is also the false value. This simplifies the implementation slightly, thereby reducing the footprint.

Yolan has a Lisp-inspired syntax. This makes the implementation of the parser trivial, and thus reduces code footprint.

Dynamic scope removes the issue of the funarg problem, and thus simplifies the implementation of variables, thereby reducing the implementation footprint. So this is chosen, although dynamic scope is generally bad language design.

The language is more like a traditional scripting language, than a functional language, as it does not support tail recursion nor does it use linked list as the basic language structure, but instead it uses resizable arrays and hashtable as data structures, and has a more imperative programming model, with while- and foreach-loops, etc. The motivation for this is that it is simpler to implement efficiently on top of an already imperative/object-oriented virtual machine, and that functional languages often use more runtime memory.

Making the interface with Java lazy, allows new control-flow constructs to be added easily from Java by the embedder, which again allows the implementation to be a small core, with support for expansion.

Execution should be online, in the sense that whenever a full statement is read from the input, it should be executed, without needing to read the full file. This is both practical for interactive evaluation, and also has a benefit memorywise as the entire program never needs to be fully in memory, as executed code may be garbage collected.

By only having a single runtime, some classes can be made static and merged, leading to a smaller code footprint. This is a tradeoff leading to non-reentrant code and no support for threads.

3.2 Syntax tree rewriting

To make the interpreter as simple as possible, it interprets the syntax tree directly. Still this has some performance issues, as for example variable and function lookup must be done at each execution. To avoid the cost of this, some of the evaluation functions instead resolve the variables the first time they are executed and replace the node in the abstract syntax tree with a node of the resolved variable or function.

3.3 Language specification

3.3.1 Syntax

Function applications / lists

Function applications are written in Lisp-style as lists. The first element in the list is the function to be applied. Lists are enclosed within square brackets `[···]`, and may be nested. The reason to use square brackets, rather than parenthesis as in Lisp, is that the lists are not cons-lists, but instead arrays, which are usually written with square bracket notation in other scripting languages. This is also an indication that Yolan is not a proper Lisp-like functional language, but rather a scripting language. The notation is also similar to that of Tcl, except that there are not automatic expression breaks at newlines. The elements within the lists are separated by whitespace. As lists are the notation for function applications, every list must have at least one element, which is an expression evaluating to the function to be applied.

Variable names

A variable name is a sequence of characters. The possible characters are letters, numbers, the symbols `!#$'()*+-,-./:<=>?@\^_‘{|}~`, and any unicode symbol with a unicode value of 127 or higher. The first character in the name of a variable must be non-numeric.

Integer literals

Integers are written as a sequence of digits (`0123456789`). Only base 10 input is possible and only non-negative numbers can be written as literals. Negative integers must be generated by subtraction.

Comments and whitespaces

Characters with a unicode value of 32 or less are regarded as whitespaces. These include the usual space, tab, newline, and line-feed. Whitespace is used to separate list elements, and is discarded during parsing. A comment must be preceded by whitespace, it starts with a semicolon `;` and it continues until the end of the line. Comments are discarded during parsing.

3.3.2 Builtin functions

The builtin functions are listed below. As the language is designed for embedding in other applications, there are no standard functions for input/output, file access, network, etc. as these might not be present or differ significantly between target devices/platforms. This functionality can instead be added via Java functions added to the Yolan runtime.

In the following, function names are written with `fixed width` font, and their arguments are written in *c*ursive. An argument is named according to its type or functionality: *num*s are expressions that should evaluate to numbers, *exp*s are expressions that may be optionally evaluated (e.g. in `if`), *val*s are expressions that will be evaluated, *string*s are expressions that should evaluate to a string, *n*ame is the name of a variable, and so on.

Variables

`[set` *name value*`]`

Evaluate the expression *value* and let the variable *name* refer to the result.

`[locals [`$name_1 \cdots name_n$`]` $expr_1 \cdots expr_n$`]`

Let $name_1 \cdots name_n$ be local variables in $expr_1 \cdots expr_n$: First save the values corresponding to $name_1 \cdots name_n$, then evaluate the expressions $expr_1 \cdots expr_n$, next restore the values of $name_1 \cdots name_n$ and finally return the result of the evaluation of $expr_n$. The expressions are evaluated in order, with $expr_1$ as the first one, and $expr_n$ as the last one.

Conditionals and logic

`[if` *cond* $expr_1$ $expr_2$`]`

Evaluate *cond* and if the result is non-*null* then evaluate and return $expr_1$, else evaluate and return $expr_2$.

`[not` *cond*`]`

If the value of *cond* is *null* return *true* else return *null*.

`[and` $expr_1$ $expr_2$`]`

Evaluate $expr_1$ and if it is non-*null*, evaluate and return the value of $expr_2$, else return *null*.

`[or` $expr_1$ $expr_2$`]`

Evaluate $expr_1$ and if it is non-*null* return its value, else evaluate and return the value of $expr_2$.

Repetition and sequencing

`[repeat` *num* $expr_1 \cdots expr_n$`]`

Evaluate $expr_1 \cdots expr_n$ *num* number of times (*num* is evaluated once, and must evaluate to a number). The result is the last execution of $expr_n$, or *null* if no expressions were evaluated, i.e. $num \leq 0$.

`[foreach` *name iterator* $expr_1 \cdots expr_n$`]`

For every value from the *iterator*, bind it to the local *name* and evaluate $expr_1 \cdots expr_n$. The result of the evaluation is the last executed $expr_n$ or *null* if no expressions were evaluated. *name* is a local variable, and is thus saved before the loop, and restored afterwards.

`[while` *cond* $expr_1 \cdots expr_n$`]`

While *cond* evaluates to non-*null*, evaluate $expr_1 \cdots expr_n$, and return the value of the last $expr_n$ or *null* if no expressions were evaluated.

`[do` $expr_1 \cdots expr_n$`]`

Evaluate $expr_1 \cdots expr_n$ and return the result of $expr_n$.

Functions

`[lambda [`$name_1 \cdots name_n$`]` $expr_1 \cdots expr_n$`]`

Create a new anonymous function, with the parameters $name_1 \cdots name_n$. Application of the function will bind its arguments to local variables $name_1 \cdots name_n$, evaluate $expr_1 \cdots expr_n$ and return $expr_n$, saving and restoring $name_1 \cdots name_n$ when entering and exiting the function.

`[defun [`$name_{function}$ $name_1 \cdots name_n$`]` $expr_1 \cdots expr_n$`]`

Create a new function, and bind it to the variable $name_{function}$. The `defun` statement above is equivalent to `[set` $name_{function}$ `[lambda [`$name_1 \cdots name_n$`]` $expr_1 \cdots expr_n$`]]`.

`[apply` *function* $param_1 \cdots param_n$`]`

Apply the *function* to the parameters $param_1 \cdots param_n$. The difference between this and the usual function application `[`*function* $param_1 \cdots param_n$`]` is that that `apply` allows *function* to change between invocations, whereas the usual function application assumes that *function* is static, to be able to optimise it during runtime.

Integer operations

`[+` num_1 num_2`]`

Calculate the sum of two integers.

`[-` num_1 num_2`]`

Calculate the difference of two integers, the result is num_2 subtracted from num_1.

`[*` num_1 num_2`]`

Calculate the product of two integers.

`[/` num_1 num_2`]`

Calculate num_1 divided by num_2.

`[%` num_1 num_2`]`

Calculate the remainder of dividing num_1 by num_2.

Type predicates

`[is-integer` *val*`]`

Returns *true* if *val* is an integer.

`[is-string` *val*`]`

Returns *true* if *val* is a string.

`[is-list` *val*`]`

Returns *true* if *val* is a list.

`[is-dictionary` *val*`]`

Returns *true* if *val* is a dictionary.

`[is-iterator` *val*`]`

Returns *true* if *val* is a iterator.

Polymorphic functions

`[equals` val_1 val_2`]`

Compare val_1 to val_2 and return *true* if they are the same, or *null* if they are different. val_1 and val_2 must have the same type.

`[is-empty` *val*`]`

Returns *true* if a list, dictionary or iterator does not have any elements. Otherwise, it returns *null*.

`[put` *container position value*`]`

Store a value into a list or a dictionary. If the container is a list, the *position* must be an integer in the range $0, 1, \cdots,$`[size` *container*`]`-1. If the container is a dictionary, the position must be a string or an integer. An entry is deleted from a dictionary by storing *null* as the *value*.

`[get` *container position*`]`

Retrieve a value from a list or a dictionary. The same on *position* are the same as with `put`. Retrieving an uninitialised entry from a dictionary yields *null*.

`[random` *val*`]`

If *val* is an integer, return a random number in the range $0, 1, \cdots, val - 1$. If *val* is a list, pick a random value from the list.

`[size` *val*`]`

Return the length of a string, the number of values in a list, or the number of entries in a dictionary.

`[<` val_1 val_2`]`

Compare val_1 with val_2. If val_1 and val_2 are integers, return *true* if val_1 is strictly less than val_2 and otherwise *null*. If val_1 and val_2 are strings, do a lexicographic comparison and return *true* if val_1 comes strictly before val_2, and otherwise *null*.

`[<=` val_1 val_2`]`

Compare val_1 with val_2. If val_1 and val_2 are integers, return *true* if val_1 is less than or equal to val_2 and otherwise *null*. If val_1 and val_2 are strings, do a lexicographic comparison and return *true* if they are equal or val_1 comes before val_2, and otherwise *null*.

String functions

`[stringjoin` $val_1 \cdots val_n$`]`

Create a string by concatenating $val_1 \cdots val_n$. If val_i is an integer or a list, it is converted to a string. A list is converted to a string by concatenating its elements, as if `stringjoin` were called with the list elements as arguments.

`[substring` *string* num_{begin} num_{end}`]`

Create a substring from a string, starting inclusively at character position num_{begin} and ending exclusively at character position num_{end}. The positions starts counting at 0, so thus `[substring` *string* 0 `[size` *string*`]]` is the entire string.

List functions

The following functions work on lists. Notice that lists in Yolan are similar to lists in Python, and thus the functions below are imperative in that they alters the list parameter, which is different to what happens in functional languages, where lists are typically cons-cells. The end of a list is the element at the position equal to the length of the list minus one.

`[list` $val_1 \cdots val_n$`]`

Create a new list, containing the elements $val_1 \cdots val_n$.

`[resize` *list num*`]`

Change the size of the *list* to be *num* elements. If the new size is larger than the current size, new elements will be added to the end of the list, with the initial value of *null*. If the new size is smaller than the current size, then the list will be truncated at the end. The list is modified by the function and then returned.

`[push` *list val*`]`

Push the value *val* at the end of the *list*. The size of the list grows by one, and the last element is now *val*.

`[pop` *list*`]`

Remove the element at the end list. The function returns that element, and reduces the size of the list by one.

Dictionary functions

`[dict` key_1 val_1 $\cdots$ key_n val_n`]`

Create a new dictionary with n entries, where key_1 maps to val_1 and so forth.

Iterator functions

`[keys` *dictionary*`]`

Create a new iterator across the keys of a dictionary.

`[values` *container*`]`

Create a new iterator across the values of either a dictionary or a list.

`[get-next` *iterator*`]`

Get the next element from the iterator, or *null* if the iterator is empty.

3.4 Developer guide to embedding Yolan in Java

The implementation of Yolan consists of a single class, `Yolan`, containing the actual implementation, and an interface `Function`, which specifies what a class needs to implement in order to be callable from Yolan. While Yolan only has a single classfile for the implementation, in order to reduce the JAR file size, it consists of several logical classes: a parser implemented as static properties,[1] a runtime implemented as static properties, and delayed computations are the actual instantiated objects.

Having a single runtime reduces memory usage, but also limits applications to only executing a single script, and only having a single execution context at a time. The reduction of memory usage comes from the fact that refererences to the execution context can be hard coded, and thus the delayed computations do not have to carry a reference to the context. There is also a memory reduction because less code is needed and the class for the context can be joined into the main class file, as static properties. The delayed computation is the same as a node in the abstract syntax tree, as the scope is dynamic and execution context is global.

The `Function` interface consists of a function that takes an array of Yolan objects – delayed computations – as a parameter, and then returns a value. In this sense Java objects callable from Yolan are essentially lazy functions, and responsible themselves for evaluating their arguments. This makes it easy to add custom control function, similar to `if` and `while`, as it is left up to the

[1] A static property is a property pertaining to the class rather than to the object

called Java function whether, and how many times, each argument expression should be evaluated.

3.4.1 Getting Started

The core method of a Yolan object is the `value()` method which evaluates the code the Yolan object represents, and returns the result. This method may throw `Exeception`s as well as `Error`s if the code it represents has faults, so to make it robust against errors in scripts, the Yolan evaluation must be surrounded by a `catch(Throwable)`.

Yolan objects are created with the static `readExpression` method that parses the next Yolan expression from an input stream. So if we want to create a simple interactive interpreter, reading from the standard input stream `System.in`, we can implement it in Java as:

```
class Main {
    public static void main(String [] args) throws java.io.IOException {
        Yolan yl = Yolan.readExpression(System.in);
        while(yl != null) {
            try {
                System.out.println("Result: " + yl.value().toString());
            } catch(Throwable yolanError) {
                System.out.println("Error: " + yolanError.toString());
            }
            yl = Yolan.readExpression(System.in);
        }
    }
}
```

This code could be saved in a file called Main.java, placed in a directory with Yolan.class and Function.class, and then compiled and executed by executing `javac Main.java` and `java Main`.

Notice that the input stream `System.in` can be replaced with any input stream, so the same idea can be used for evaluating files, programs as strings within the application, or even as streams across the network, where Yolan could work as a shell for remote scripting/controlling an application.

If we want to execute an entire stream, there is a shorthand builtin method for doing that: `eval`. For example:

```
class Main {
    public static void main(String [] args) throws java.io.IOException {
        Yolan.eval(new FileInputStream(new File("script.yl")));
    }
}
```

This code opens the file "script.yl", and evaluates all the expressions within it. `eval` throws away the results of the individual expressions and does not print them, so the above code is only useful if we have added some user defined functions to Yolan that allow some kind of input or output.

3.4.2 Adding functions to the runtime

Yolan needs to call Java functions to do anything practical, and this section shows how to make Java code available for Yolan. While the builtin Yolan functions support basic data structures such as lists and dictionaries, there is no

built in way to do input/output from Yolan, as that is platform dependent: Java Standard Edition supports files, while Java Micro Edition has a record store, and user interfaces ranges between Midlets, Applets, graphical applications, and text standard-in/out. So when a script needs to communicate with the user, or work on the state of the host application, some functionality needs to be added. This is most easily done by adding functions to the runtime.

The `Function` interface is the way to do that. To implement the interface a single function, `apply`, needs to be implemented in the class. This function performs the actual application of the function within Yolan, and it takes an array of delayed computations (instances of the `Yolan`-class) as argument, and returns the result as a usual object. The delayed computations they are only evaluated when the called function chooses to evaluate them, by calling their `value()`-function. Execution of the delayed computations may also have side effects, and thus the number of times `value()` is called per argument matters.

In order to add a new Java function to be callable from the runtime, the method `Yolan.addFunction` takes a string name and a `Function` as parameters, and binds the name to the function. As an example the following code makes a new function, `println`, available to the runtime. This function takes one argument, which it prints to the standard output:

```
class PrintingFunction implements Function {
    Object apply(Yolan args[]) {
        System.out.println(args[0].value());
    }
}
class Main {
    public static void main(String [] args) throws java.io.IOException {
        Yolan.addFunction("println", new PrintingFunction);
        Yolan.eval(new FileInputStream(new File("script.yl")));
    }
}
```

The above program reads and evaluates the file script.yl, with an augmented runtime that also has the `println` function.

3.4.3 Code-footprint-efficient addition of multiple functions to the runtime

The naive approach for adding functions to the runtime would be to create a new class implementing the `Function` interface for each function. This would add significantly to the size of the JAR-file. If the code size is critical, it can often be reduced by combining multiple functions into a single class, for example via a switch dispatch as shown below:

```
class ManyFunctions implements Function {
    int id;
    ManyFunctions(int id) {
        this.id = id;
    }
    public Object apply(Yolan args[]) {
        switch(id) {
            case 0: // first function
                    ....
                break;
            case 1: // second function
                    ....
```

```
                break;
            case 2: // third function
                    ....
                break;
            ....
            default:
                throw SomeKindOfError();
    }
    static void register() {
        Yolan.addFunction(new ManyFunction(0), "firstFunction");
        Yolan.addFunction(new ManyFunction(1), "secondFunction");
        Yolan.addFunction(new ManyFunction(2), "thirdFunction");
        ....
    }
}
```

When implementing functions, it is also possible to create control structures, due to the laziness of Yolan objects. This is done by not evaluating the arguments' `value`-functions exactly one time each, but zero or more times, depending on the purpose. The example below shows how the usual if-statement could be implemented:

```
class YolanIf implements Function {
    public Object apply(Yolan args[]) {
        // first evaluate the condition
        // and find out if it i true (not null)
        if(arg[0].value() != null) {
            // only evaluate if the condition yields true
            return arg[1].value();
        } else {
            // only evaluate if the condition yield false
            return arg[2].value();
        }
    }
}
```

3.4.4 Values and types

The builtin types in Yolan are mapped to Java classes for easier interoperability, so lists are implemented as java.util.Stack, dictionaries are implemented as java.util.Hashtable, strings are implemented as java.lang.String, null/false is implemented as the value null, integers are implemented as java.lang.Integer, and iterators are implemented as java.util.Enumeration. Operations on those data types are just as in Java.

Any Java object can be passed around within Yolan, so adding support for new data types is just a question of adding functions that work on values of those data types.

3.4.5 Functions defined within Yolan

When a user defines functions within Yolan, they are instances of the Yolan class. Before calling such a function, the number of arguments can be found using the `nargs` method. If the Yolan object is not a callable user-defined function, the result of `nargs()` is -1, and this is the only API method needed to check if a Yolan object is a callable function. This function is then applied with the `apply` method, which takes the arguments to the function as arguments, for example:

```
...
    // evaluate some Yolan object that yields a function
    Yolan function = yl.value();
    // ensure that it is a function and it takes two arguments
    if(function.nargs() == 2) {
        // apply the function
        result = function.apply(arg1, arg2);
    } else ...
....
```

The apply method is defined from zero up to three arguments. If there is a need for an apply method with more arguments, they are simple to add; see page 74 for the implementation details. There is also a general apply method, that takes an array of arguments as argument.

3.4.6 Modifying the runtime

In order for the scripting language to be practical, it should be able to work with and share data with the host application. Of course this can be done with functions, and evaluation, as described above, but an additional connection with the language can be added by accessing the variables defined, and used, by the running scripts. For this there are three functions: `Yolan.resolveVar`, `Yolan.getVar`, and `Yolan.setVar`.

When the value of a variable is accessed, this is done through a handle, which is found using `resolveVar`. This handle can then be used for reading and writing the variable. The motivation for the handle is that it takes time to look up a variable name, so this computation can be done once for each variable that needs to be accessed, and then additional accesses to the resolved variable are significantly faster. The `resolveVar` function takes the variable name as a string parameter, and returns the handle, which is an integer. If the variable does not exist in the runtime, space is allocated for it.

With a handle, it is then possible to set the value of a variable with `setVar`. For example setting the variable foo to 42 can be done with:

```
Yolan.setVar(Yolan.resolveVar("foo"), new Integer(42));
```

and similarly the variable can be read with `getVar`:

```
Object result = Yolan.getVar(Yolan.resolveVar("foo"));
```

If it the variable is commonly accessed, it saves time to cache the handle across calls, as follows:

```
class Class {
    int fooHandle;
    Class() {
        fooHandle = Yolan.resolveVar("foo");
    }

    int someMethod() {
        ... perhaps some scripts modifying foo are executed ...
        Object foo = Yolan.getVar(fooHandle);
        ...
    }

    void otherMethod() {
        ...
        Yolan.setVar(fooHandle, "A literal value or some variable");
```

```
        ...
    }
}
```

When defining functions, as described earlier, it is actually the same that is happening: the function is encapsulated in a Yolan object and added to the runtime as done by `setVar`.

3.4.7 Resetting the runtime and saving space

When the scripting language is only used in some parts of the application, it can be pratical to be able to unload its runtime data in order to save memory. For this there are two utility functions `Yolan.wipe()` and `Yolan.reset()`.

`Yolan.wipe()` sets all references in the runtime to zero, allowing data to be garbage collected. When the runtime has been wiped, Yolan expressions can no longer be evaluated, and trying to evaluate them yields errors.

`Yolan.reset()` resets the runtime: all variable handles are invalidated, all variables are removed from the runtime, and only the builtin functions remain. Existing Yolan expressions are invalid, and evaluation of them may lead to unexpected behavior. User defined functions and variables need to be re-added. Resetting is necessesary before scripts are executed after a `Yolan.wipe()`. It can also be practical when multiple scripts are run, one after another, and they must not modify the runtime for each other.

Chapter 4

LightScript

The purpose of LightScript is to make a practical scripting language for mobile devices. It should overcome some of the limitations of Yolan. In particular LightScript should support lexical scope to allow real closures, and it should not invent a new syntax, but build on a mainstream language in order to be readable and easy to learn for other programmers.

As for the name of the language, I chose to call it LightScript, as it is a lightweight scripting language, and a light (reduced) version of JavaScript. The name also has an aspect of illumination, and the "-Script" suffix also indicates the connection with JavaScript/EcmaScript whose dialects often have names with this suffix.

LightScript is a moving target, as I continually improve the code. Most of this chapter discusses LightScript version 1.0 as that was the language as it looked at the beginning of writing the report.

Section 4.1 gives an introduction to the design choices of the language. The next three sections, Section 4.2-4.4, look into some implementation details. Then, Section 4.5 describes the language, and Section 4.6 is a guide to how to embed, use, and integrate it with mobile Java applications. Section 4.7 looks into future directions, and outlines the current status of LightScript version 1.1.

4.1 Design choices

This section first lists the design choices of LightScript version 1.0, which are then elaborated on in the following paragraphs. The main design choices that distinguish LightScript from Yolan are:

- LightScript should be a subset of EcmaScript
- Scoping is lexical, and the language has closures
- Objects are supported, and have prototypical inheritance with a self-like `clone` function
- Exceptions are supported, and can be thrown to, from and across Java code

In addition to these, there are also some language design choices that are similar to those of Yolan:

- It must be able to run on CLDC 1.0
- Standard Java classes should be used for the builtin types, to make interaction with Java code easier
- Code should be loadable/executable as source code at run-time
- Functions are first-class values
- Parsing does not need to be concerned about error checking
- Only integers will be supported, - no floating point numbers
- Undefined, null and false, are joined into a single type/value. This joined value is the only false value
- Variable access should be fast

There are several reasons to make the language a subset of EcmaScript: This allows it to automatically run within most web browsers, including within smartphones where Java may not be installed. Being an EcmaScript subset also makes it easier for developers, that already know EcmaScript, or other languages with C-like syntax, to get started with. Nevertheless, being a subset of EcmaScript will also add more complexity to LightScript, and EcmaScript has some unfortunate design choices [Cro09a], for example: Variables are global by default. Scope is limited by functions rather than blocks, which is unusual. The usual equality operator `==` does type coercion leading to non-transitivity, so `===` should be used instead. Traversal of objects through for-each loops also traverse the prototype of the object which is usually not the intended semantic. Arrays and `null` are objects in typeof-operator, leading to more complex type test for those values as well as objects. The many reserved word adds unnecessary restrictions to naming of properties and variables. Automatic semicolon insertion encourages sloppy programming style. For compatibility with EcmaScript, some of these will be a part LightScript, although many can be omitted by being a subset and thus more strict.

LightScript should have a lexical scope similar to EcmaScript. This has the benefit of allowing closures which makes the use of functions much more expressive. The lexical scope in EcmaScript, and thus LightScript, is a bit different from most other lexical scoping, as the scope limit is the enclosing function rather than the block.

LightScript should support an object system similar to EcmaScript. Inheritance will be slightly different: while EcmaScript has prototypical inheritance, it mixes it with some Java/C++like syntax. The semantics will be the same for LightScript, but the inheritance will be done with a self-like `clone` clone, which seems more pure prototypical than EcmaScripts syntax. The `clone` function can easily be implemented in EcmaScript.

Exceptions should also be supported, and it should also be possible to throw between, and across Java-functions and LightScript functions.

Like Yolan, LightScript will run on CLDC 1.0, and will use standard classes for easier embedding and lower code footprint size. It will support run-time loading of source code, and will have functions as first-class values. See the Yolan design choices on page 21.

The parser may assume that the programs have valid syntax. Usually parsers both build a syntax tree of valid programs and also reject programs with invalid syntax. By removing the rejecting part, and only requiring that the parsers can build a syntax tree from valid programs, the parser may be optimised to run better on limited devices. The parser should still be guarenteed to terminate, even with invalid programs though the resulting syntax tree may be undefined.

LightScript only supports integers as number type, due to the platform. This is radically different from EcmaScript, which only supports floating point numbers. Scripts can still be compatible: addition, and multiplication, subtraction and modulo operations stay within the integer subset of floating point numbers, as long as there is no overflow and they start out with integers. The shifts and bitwise operators temporarily cast to integers in EcmaScript, so they are not an issue. The only problem of the operators is division, and the solution here is to omit the `/` binary operator from LightScript, and instead allow integer division to be implemented as a function `div`, which can also be implemented in EcmaScript with a combination of division and rounding.

Similar to Yolan, LightScript also only has one false value, which also joins the `undefined`, `null`, and `false` value of EcmaScript, which also removes the boolean type. This is an optimisation that makes truth test faster and slightly smaller, as they can be written as `obj != null`, rather than `!( obj == null || (obj instanceof Boolean && !((Boolean)obj).booleanValue()) || (obj instanceof Integer && ((Integer)obj).intvalue() == 0) || (obj instanceof String && obj.equals(""))`. It is still possible to preserve compatibility of scripts with EcmaScripts by requiring that conditions are always boolean values – as it is required in many static typed languages. Another issue with this choice is, that there is no distinction between `undefined` and `null`, which may be desirable.

Similarly to Yolan, LightScript should support fast variable access. This also means that another approach than looking through the chain of execution context objects, as described in the EcmaScript standard, must be used. Here LightScript should use a usual execution stack, boxing objects in the closure onto the heap.

4.2 Imperative implementation of top-down operator precedence parsers

This section looks at how top-down operator precedence parsers can be implemented efficiently in an imperative language.

As we have relaxed parsing requirements, such that only building the syntax tree, and not checking for error is required, several classes of tokens can be

joined. An example is to create an end-of-list token, instead of list termination such as `}`, `)`, `]`, which also allows a generalisation of parsing a list. Similarly a separator token can be introduced instead of "`,`", "`;`", and "`:`".

The top-down operator precedence parser described in Section 2.4.2 uses first-class functions and is thus better suited for being implemented in functional languages rather than imperative languages. For example, in Java a simple implementation of the parser uses a lot of space as the functions would use a class each, in order to be passed around as first-class values. Our solution is to use a dispatch function instead, and replace the denominator functions of the token, with integers. Actually it is simpler with two dispatch functions, one for the null denominator functions and one for the left denominator functions.

The token object contains information about the denominator functions, and corresponding abstract syntax tree node IDs, and also a priority/binding power. This is just five small integers, which, due to their limited range, easily can be represented within a single 32bit integer. Some tokens represents literal values or identifiers, where the value or identifier also has to be passed to the parser. So a token can be represented compactly as an integer and possibly an object for the value, which eliminates the need for an actual token object in the parse, thereby reducing the footprint by not needing a token class.

The token types can be encoded by the string representing the token followed by the five integers. The token name string is not needed by the parser, but is used by the tokeniser, to map the token to the IDs. So the tokens can be written as `"tokenname" + (char) binding_power + (char) null_denominator_function + (char) AST_ID_for_null_denominator + (char) left_denominator_function + (char) AST_ID_for_left_denominator`. The syntax is implemented as a list of the different token types, with the data described above. For this to be executed there also need to be a parse loop, and definitions of sensible denominator function bodies within the dispatch. The actual implementation can be seen on page 96.

4.2.1 Performance properties of the parser implementation

For each token, there is an instruction cost of 1-2 function calls, 1 switch-dispatch, 0-1 comparisons, extracting 2-3 parts of the token integer, and storing a copy of the token integer, plus the cost of building the actual AST node, and the cost of the tokenisation itself.

The size of the implementation can be kept very small, as the denominator functions can be reused across different token-types. For example: the binary operators share a single case in the left denominator dispatch, and adding a new binary operator only uses the length in characters of the operator plus 5 bytes.

4.3 Implementation of variables and scope

In the EcmaScript standard, identifier resolution is done by searching through the scope chain, which is a list of objects. An object in the EcmaScript context

is a mapping from property names to values. This approach to implementation would be very performance expensive.

Instead we want to resolve the variables at compile time, while preserving as much of the EcmaScript semantics as is practical. The main issue here is that when we resolve the variable at compile time, we do not keep information to be able to resolve it dynamically at run-time, which lead to limitations in `eval` like statements when compared to the EcmaScript semantics. Currently there is no `eval` statement in LightScript, but this may be an issue in future versions.

Of the different methods for implementing scope, discussed in Section 2.4.3, the only real possibility for a partly imperative scripting language with support for first-class functions is either to allocate the activation records on the heap, or to keep track of variables that could live on after exiting a function, and box those variable on the heap. The other options are ruled out, as we want to have closures, and we want the outer scope variables to be mutable. As we are implementing on top of the JVM, we inherit its garbage collector, which may vary between different devices, and which may be badly suited for activation record allocation on the heap. So the approach will be only to box variables onto the heap that could be alive after function exit. To simplify this every variable that is added to a closure of an inner function is boxed on the heap, and other local variables are just stack allocated in the usual way.

For the practical implementation of the stack on top of the JVM, there are two obvious possibilities: A `java.util.Stack` object could be used, or a stack could be implemented manually with an array and an index pointer. In order to select implementation strategy, a microbenchmark was done on the KVM, which indicated that the array approach is significantly faster than the `java.util.Stack`. The array grows dynamically when entering a function that uses more stack space than is available. The code footprint size is similar for both approaches, though probably a bit smaller for the `java.util.Stack`, as that one automatically grows, unlike the array approach, which needs a manual implementation. The actual access to a `Stack` requires a method call, which is 3 bytes plus 1-3 bytes for self and parameter loading, where reading from an array is a single byte opcode plus 2-4 bytes for self, index and parameter loading, and at the same time the array approach sometimes needs to adjust the index, costing 3 bytes.

4.4 Overview of the implementation

The core of the LightScript implementation consists of three parts: 1) a top-down operator precedence parser that is responsible for building the syntax tree, and for identifying which variables needs to be boxed on the heap for closures, 2) a compiler that translates the abstract syntax tree to a sequence of opcodes while keeping track of the stack depth, and 3) a stack-based virtual machine that executes the code. Besides these core elements, the implementation also contains an API for embedding, and a library of standard functions.

The parser is a concrete implementation of the top-down operator prece-

dence parser for imperative language, as described in Section 4.2, with the additional detail that during the parsing, the parser keeps track of which variables are defined and used in each function, such that this information can be used for scoping later on. The source code for the parser starts on page 96.

4.4.1 Compiler

The core of the compiler is a function that compiles a node of the syntax tree, possibly calling itself recursively for the children. As it is compiling to a stack-based virtual machine, the compilation itself can be quite simple/small, as a function or operator just needs to evaluate its parameters in a way that pushes the results to the stack, followed by doing the operation on the top elements of the stack.

As the activation records are allocated on the execution stack. The compiler keeps track of the stack depth of the emitted code, such that there is no need for a frame pointer at execution, but variables is referenced relative to the stack top.

The abstract syntax tree does not distinguise between statements and expressions. Instead the compiler takes a parameter that indicates whether the generated code is expected to push a result on the stack or not, and corresponding code will be generated.

By having the integer for the abstract syntax tree node type matching the virtual machine instruction, many of the compilation cases can be joined.

4.4.2 Virtual Machine

The virtual machine is stack-based in order to reduce the footprint of the compiler. It is implemented with a single larger switch statement as it is running on top of JVM and the JVM does not support for references to labels, etc. The instruction set is inspired by the JVM and calling conventions on i386, and it is also a product of the incremental development.

4.5 Language specification

The language is mostly a subset of EcmaScript [ECM99], and the description of the different parts of the languages is written in the same sequence as the EcmaScript standard, to make it easier to compare the two. The focus will be on where LightScript differs from EcmaScript. Scripts written for LightScript also run unaltered in EcmaScript compliant interpreters, with a couple of extra functions defined within EcmaScript. On the other hand, EcmaScript may or may not run within LightScript, as LightScript is only a subset of EcmaScript.

The specification is stricter than the actual implementation, for example: the implementation does not distinguish between statements and expressions, and it allows a statement everywhere an expression could be written, whereas the specification follows EcmaScript, and distinguishes the two.

Only a relatively small subset of EcmaScript is implemented: operators have been added as needed, meaning that rare operators have not yet been added.

On the other hand, more interesting language aspects, such as exceptions and prototypical inheritance, have been implemented and tested, though they may not have been needed for the example programs or benchmarks.

4.5.1 Lexical conventions

The lexical conventions are slightly different from EcmaScript. Whitespaces are space, tab, carriage return and line feed. LightScript does not distinguish between whitespace and line terminators, and does not have automatic semicolon insertion.

Comments starts with two consecutive slashes `//` and run to the next newline character.

Keywords reserved by EcmaScript are also reserved by LightScript. The keywords currently used by LightScript are: `catch do else for function if return this throw try var while`.

An identifier starts with a letter or an underscore and continues with any combination of letters, underscores and digits. LightScript does not support escaped unicode sequences, but does accept utf-8 encoded. Non-letter unicode symbols with value larger than 127 are not supported. This allows the parser to be implemented more easily as they can just treat any 8-bit character with a value larger than 127 as a part of a unicode letter.

The punctuators in the current version of LightScript are: `{ } ( ) [ ] . ; , < > <= >= === !== + - * % >> ! && || ? : = += -=`. Floating point division is omitted, but an integer `div` operator can be added, as discussed in Section 4.5.4 on page 41.

String literals are always enclosed in double qoutation `"`, and single quotation is not supported. It is possible to use backslash `\` to escape quotation marks, backslashes and newline (`"\n"` is the string containing a newline).

Integer literals are written as a sequence of digits, not starting with a zero, unless the number is zero. Only base 10 literals are supported.

Other literals are `true` which translate to a value that has a true truth value, and `false`, `null` and `undefined` which translates to a value that has a false truth value.

4.5.2 Types

LightScript has 5 builtin types: nil, string, number, array, and objects.

The nil type only has one value, which is referenced as `undefined`, `null` or `false`. A string is an immutable sequences of characters. A numbers is a 32 bit integer. An array is a growable and mutable sequence of values, where the values can have any type. An object is a mapping between names and values, where the name should be of string type, and the value can be of any type. An object can have another object as its prototype, such that if the lookup in the first object does not find a mapping, the name is then looked up in the prototype object. The order of traversal of the object names/values, may be implementation dependent in that implementations may choose to implement

them via hashtables or lists, depending on what makes most sense for the platform. This is different from EcmaScript, where the traversal is in the order the properties were added to the object.

4.5.3 Execution contexts

Variables defined with the `var` keyword are allocated in the current function's activation record. A new activation record is created at each function call. If a variable is not in the current function's activation record, it is looked up in the lexically enclosing function, or if there is no outer function, in the global scope. This is the usual static scoping, where functions are the only way to make a scope closure.

4.5.4 Expressions

When a function is called as a property on an object, evaluation of `this` within the function yields the object. Identifiers are evaluated in the scope to yield a value. Literal values are just their actual values.

Arrays are initialised with square brackets `[` $elem_1, \cdots, elem_n$ `]`. Elements can be any expression, and they are separated with commas (`,`). Objects can similarly be initialised with curly brackets: `{` $key_1 : value_1, \cdots key_n : value_n$ `}`, where the keys must be string literals, and values can be any expression.

Parenthesis can be used to group expressions together, for example: $2*(3+4)$ is different from $2 * 3 + 4$

Objects and arrays can be subscripted with the bracket notation: *object*`[`*key*`]`, where *key* can be any expression. The dot notation can be used as a shorthand for the case where *key* is a string literal, such that *object*`.`*name* is equivalent to *object*`["`*name*`"]`.

The `new` operator from JavaScript is not needed, and is therefor not supported in LightScript. A new object can be written as an object literal, created by constructor function, or be `clone`d for inheritance.

Function calls are written as an expression followed by, a possibly empty, parenthesised argument list, e.g. $func$`(`$arg_1, \cdots, arg_n$`)`, where the arguments are separated by commas. A LightScript function currently have a fixed number of arguments.

Postfix operators are not supported. Their implementation would add slightly more to the code space than the prefix versions.

The implemented prefix operators are `++`, `--`, `-`, and `!`, which work as usual. The `typeof` operator is omitted due to its unfortunate semantics [Cro09a], and instead a `gettype` function is supplied, which can easily also be implemented on top of EcmaScript.

The implemented binary arithmetic operations are: multiplication `*`, modulo `%`, addition `+`, subtraction `-` and right shift `>>`. Except for addition, these work only on numbers. For addition, if one of the arguments is not a number, it does string concatenation instead. Division is not implemented as `/`, due to the semantic differences in integer and floating point division, but can instead be implemented as an integer division function, `div(`a, b`)`.

Comparison operators are `<`, `<=`, `>`, `>=`, `!==`, `===`, which works as usual. The equality operators do an actual comparison[1] and not just a pointer comparison. The type coerced equality operators from JavaScript `!=` and `==`, are omitted as they have semantic issues, such as not being transitive [Cro08].

The current assignment operators supported by the language are: `=`, `+=` and `-=`. The remaining will be added in the next version. The conditional operator "`? :`" is supported. The comma operator "`,`" is not supported, as that is generally considered bad programming style [Cro09a].

4.5.5 Statements

Statements are terminated by semicolon `;`. A blocks consists of a sequence of statments within curly brackets `{···}`. A block may, for example, be used in conjunction with `if` or `function` declaration. Notice that in EcmaScript, and thereby also in LightScript, a block is not a scope limit. In LightScript, blocks are only allowed, where they could actually be needed, e.g. in conjunction with different conditional/iteration statements, function declaration and try/catch.

The conditional and iteration statements supported by LightScript are: `if`, `do···while`, `while`, and `for`, which work similarly to their EcmaScript counterparts.

Functions can be exited with the `return` statement, where the optional argument to the return statement is the value returned.

LightScript does not support the `with` statement,[2] as this is considered bad programming style.

`switch` statements, `break`, `continue` and labelled statements are not supported. In JavaScript a statement is mostly syntactic sugar for a sequence of `if`-statements combined with an anonymous variable, as the expressions of the `case`-clauses are evaluated sequentially and then compared to the result of the expression in the switch clause. These constructs may be added in later versions of LightScript.

Exceptions are thrown with the `throw` statements, which works as in JavaScript. The `try` statement is slightly different, as LightScript only supports the `try···catch` version, and does not currently support the `finally` option. LightScript also has slightly different scoping rules for the catch block, where the caught variable is an ordinary local scoped variable, whereas JavaScript creates a new object in the scope chain containing only this variable. The motivation for this change is that it is a more consistent approach to scoping, where this would otherwise be a special case, requiring more code space. At the same time, it is only in the case that the variable in the catch-statment shadows another variable, that JavaScript and LightScript will be different, and this case should be avoided anyhow, as it leads to more difficult-to-read code.

[1] This is equivalent with the Jave `.equals-` method.

[2] The `with` statement lets properties of an object behave as local variables.

4.5.6 Function definitions

Functions are defined using the `function` keyword, followed by an optional function name, a list of parameters and the function body. If the function name is specified, a variable with that name assigned to the function.

The LightScript currently does not support a variable numbers of arguments, nor can functions have properties. Support for variable number of arguments is added in the next version. Functions are first-class objects, and can be passed around, and used as such. A function applications is expressed with the usual $f(\cdots)$.

4.5.7 Native LightScript objects and functions

The current LightScript implementation only has a small standard library, with the purpose of being a proof of concept to which functions can easily be added. The following functions and methods are implemented:

print(value) Prints a value to the standard output.

gettype(value) Returns a string representation of the type of the parameter, either "object", "array", "number", "undefined" or "builtin". This can be used instead of the `typeof` operator, which in EcmaScript has some semantic issues.

parseint(str) Parses a string as a base 10 integer, and returns the integer result.

clone(parent) Creates a new object using the parameter as the prototype.

Array.push(elem) Pushes an element to the end of the array. This is a method of the array prototype.

Array.pop() Pops an element from the end of the array and returns it. This is a method of the array prototype.

Array.join(sep) Joins the elements of an array as strings, with separator between neighbours. This is a method of the array prototype.

Object.hasOwnProperty(name) Determines whether the object has a (non-inherited) property of the given name. This is a method of the object prototype.

***.length** A special property, indicating the length of an array or string, or the number of properties in an object.

4.6 Developer guide to embedding LightScript in Java

To evaluate code with LightScript, the developer first has to instantiate a LightScript object, which keeps track of global values, loaded libraries, and the internal compiler state. The constructor takes no parameters, and is used as:

```
LightScript lsContext = new LightScript();
```

The LightScript object is a context that can be used to evaluate LightScript code, using the `eval` method. This method either takes a string or an `java.io.InputStream` as parameter, which is then read and executed:

```
lsContext.eval("print(\"Hello world\" + 17 * 42);");
lsContext.eval(new FileInputStream(new File("myscript.js")));
```

Global variables of the context can be read and written with the `get` and `set` method, so for example:

```
lsContext.set("foo", new Integer(17));
lsContext.eval("bar = foo + 25;");
System.out.println(lsContext.get("bar");
```

would print `42`.

4.6.1 Adding native functions to the runtime

A method of a Java object can be called from LightScript if the object implements the `LightScriptFunction` interface, which defines an `apply` method. The interface is:

```
public interface LightScriptFunction {
    /**
     * Method callable from LightScript.
     *
     * @param thisPtr    The this LightScript object,
     *                   if apply was called as a method.
     *
     * @param args       An array that contains the parameters,
     *                   this is READ ONLY.
     *                   The first parameter is at args[argpos], and
     *                   the last parameter is at args[argpos + argcount - 1].
     *
     * @param argpos     The position of the first parameter.
     *
     * @param argcount   The number of parameters.
     *
     * @return           An object that is returned to LightScript
     */
    public Object apply(Object thisPtr, Object[] args, int argpos,
                        int argcount) throws LightScriptException;
}
```

So, for example, a function that returns the current number of milliseconds could be implemented as:

```
class MillisecondsFunction implements LightScriptFunction {
    public Object apply(Object thisPtr, Object[] args, int argpos,
                        int argcount) throws LightScriptException {
        return new Integer((int)System.currentTimeMillis());
    }
}
```

Adding a function to the runtime is just like adding any other variable, via the `put` method of the LightScript object. So the function above can be used to perform some timings in LightScript as follows:

```
lsContext.set("timer",new MillisecondsFunction());
lsContext.eval("begin = timer();"
               +"for(i=0;i<1000000;++i);"
               +"print(\"Time used: \" + (timer() - begin));");
```

When registering several functions, it is more compact to join them via a dispatch, so a class implementing several functions could be implemented as follows:

```
class FunctionLibrary implements LightScriptFunction {
    int id; // This tells which function the object represents
    public Object apply(Object thisPtr, Object[] args, int argpos,
                        int argcount) throws LightScriptException {
        switch(id) {
            case 0: // integer division
                return new Integer(((Integer)args[argpos]).intValue()
                                   /((Integer)args[argpos+1]).intValue());
            case 1: // increment property i, not of superclass
                int i = ((Integer)((Hashtable)thisPtr).get("i")).intValue();
                ((Hashtable)thisPtr).put("i", new Integer(i + 1));
        }
        return null;
    }
    private FunctionLibrary(int id) { this.id = id; }
    public static void register(LightScript lsContext) {
        lsContext.set("div", new FunctionLibrary(0));
        lsContext.set("propinc", new FunctionLibrary(1));
    }
}
```

which could be used like:

```
FunctionLibrary.register(lsContext);
lsContext.eval("obj = {}; obj.i = 1; obj.inc = propinc;"
               +"while(obj.i < 10) {"
               +"  print(div(42, obj.i));"
               +"  obj.inc();"
               +"}");
```

4.6.2 Datatypes

LightScript uses ordinary Java objects for most data. Strings, stacks, tables, are implemented using the standard Java classes. Boolean values are the constants `LightScript.TRUE` and `LightScript.FALSE`, instead of Java Booleans as that improves performance, as we can just do a pointer comparison, rather than first casting to the Boolean class followed by a method call to retrieve the boolean value.

LightScript objects are instances of `java.lang.Hashtable`. If a LightScript object is cloned from another object, it is an instance of the `LightScriptObject` class, which is subclass of `java.lang.Hashtable`. LightScript objects have a constructor that corresponds to clone in Self, described in Section 2.3.7, and the parameter to the constructor is a LightScript object, either as a `java.lang.Hashtable` or a `LightScriptObject`. `LightScriptObject` overloads the `get` operator of the hashtable superclass, such that when the key is not found in the current object, it does a lookup in the prototype from which the object was cloned.

Exceptions that can be thrown to/from LightScript are of the class `LightScriptException`. This exception has a property `value` that is the object that is thrown/caught within LightScript. The constructor just takes the value as an argument.

4.7 Versions and future directions

The LightScript version described above is version 1.0.426

The version number consist of a major version number, a minor version number and a revision number. The major version number is incremented at major rewrites and redesign. The minor version number is incremented with milestones and expansions of language/added functionality, it uses an even/odd strategy, such that even minor versions get bugfixes, and the odd minor versions are development versions where the new features are added, i.e. the features intended for version 1.2 are implemented in versions 1.1.*something* and when all of them are added, the minor version number increases to 1.2, where only bug fixes will be added. The revision number correspond to the svn version, and is incremented on each commit.

The following is the roadmap of scheduled changes to LightScript version 1:

1.0 Prototype for thesis and proof of concept. Version 1.0 is the one used for the benchmarks and described throughout this report, unless otherwise mentioned.

1.2 Expansion of language and standard library, more operators from EcmaScript are added, and an additional fixed point number representation will be added. More constructs from EcmaScript should be added, including distinction between false, null and undefined.

1.4 Extra optional libraries to make the language more useful for practical applications.

As the benchmarks in the next chapter show, version 1.0 fullfills the goal of a small code footprint and decent performance. Version 1.2 and 1.4 will add more functionality at a tradeoff of footprint and performance degradation, but should still be possible to keep it small the footprint reasonable small. Some functionality may optionally be left out at the compile time.

4.7.1 Roadmap towards version 1.2

Whereas the previous parts of this chapter describes LightScript version 1.0, this section describes the most recent changes and updates in the development branch. In the following is listed what have been done in version 1.1 and what needs to be done before version 1.2

Implemented changes

- Optimise the code footprint by reordering opcode IDs. The saving comes by a smaller dispatch table, is in the magnitude of hundred bytes code footprint, and has no other side effects

- Add support for fixed-point arithmetic. The cost in code footprint is in the magnitude of a few kilobytes.
- Distinguish between false, null and undefined.
- Replace the stack allocation instruction with automatic code to do this, based on metadata of the function object. This does not change to the footprint significantly
- Let the functions support a variable number of arguments
- Add optional clean-up code when exiting a function, to improve garbage collection. This degrade performance slightly for better reclaimation of runtime memory
- Add prototype object to functions

Unless otherwise mentioned, each of these changes addes in the magnitude of hundred of bytes or less to the code footprint.

A change toward more EcmaScript-like semantics by having cutom Java-classes for the different types in LightScript, was abandoned as the implementation turned out to be too expensive spacewise.

Remaining tasks

- Make the fixed-point arithmetic optional.
- Generalise the system for builtin objects, making it even easier to add support for using plain Java classes from within LightScript, i.e. LightScript-getter/setter interface
- Let the LightScript object implement getter/setter interface. The LightScript object should the represent the global object, and when we are not in a method call, it should be the content of "`this`"
- Support for the `new` operator and `.prototype` property to allow existing object oriented JavaScript code to run within LightScript
- Walk through EcmaScript specification, and add those operators and library functions that do not add too much to the footprint
- Maybe improve virtual machine by joining common instruction pairs to single instructions (adding superinstructions)
- Maybe add optional support for floating point numbers
- Maybe add breaks, continues and/or switches
- Maybe optimise compiler footprint, by changing code generation for some constructs to syntax tree rewriting

Chapter 5

Benchmarks

The purpose of this chapter is to measure the performance of the newly developed languages, compared to existing scripting languages. The first part of the benchmark looks at the code footprint, as this is the major limitation on very low end mobile devices, where it may be limited to 64KB for host application and embedded scripting language combined. The code footprint is compared to other scripting languages, in Section 5.1.1. In Section 5.1.2, we look at how the different parts of the LightScript implementation contribute to the code footprint, and finally, in Section 5.1.3, we look at the sizes of Yolan and LightScript when they are embedded in a mobile application where space optimisations are enabled. The second part of the benchmarking, Section 5.2, looks at the execution speed of the scripting languages.

5.1 Code footprint

This section looks at the code footprint: first footprint estimates are compared for different scripting languages, then details of the footprint of LightScript version 1.0 are investigated, and finally the actual size of the library embedded in a minimal application is found.

5.1.1 Comparison with other languages

Just comparing the JAR file size would yield an unfair comparison as some of the languages have large GUI libraries, which would be included in the size, and count against them. Similarly, some languages do not optimise/obscurify the JAR file. So to have a more fair comparison, the size is measured of zip archives[1] where the class files are non-obscurified, and extra libraries are removed, rather than the actual executable JAR.

Languages

The code size benchmark compares the languages implemented in this project, Yolan and LightScript, to other scripting languages for mobile devices, FScriptMe,

[1] A JAR file is a zip archive, possibly with some meta data included

Kahlua, Hecl, Simkin, and CellularBasic, and to some general scripting languages, JScheme, and Rhino.

FScriptME is the mobile edition of femto-script, which is "an extremely simple scripting language" [Mur04]. Out of the box, it only supports strings and integers as data types – no compound types – which limits use somewhat, and it is still in beta, since 2002.

Kahlua [Kar09] is an implementation of the Lua Virtual Machine for Java Micro Edition. The implementation requires CLDC-1.1 due to the use of floating point arithmetic, and therfore it does not run on the lowest-end mobile devices, which only support CLDC-1.0. Unlike the other languages, Kahlua is not a full language interpreter, but only a virtual machine, so the script cannot be executed directly on the device, but needs to be compiled on another computer before being executed.

Hecl [WK09] seems to be *the* major scripting language for mobile devices, or at least the one that keeps popping up at the top of most results, when searching for scripting languages for mobile devices. It is very portable with different editions running on CLDC-1.0, CLDC-1.1, Android, as applets and as standalone applications. It has many libraries targeting mobile devices, which in this comparison were removed from the zip file of class files, so as not to give languages with fewer libraries an advantage in the size comparison. The language is a dialect of Tcl, and is simplified such that arithmetic operators, and the like, are prefix operators, so expressions end up a bit Lisp-like.

Simkin [Whi09] is a scripting language embedded in XML, which also runs on mobile devices. It depends on kxml XML library, which is not included in the measured size.

CellularBasic [Els06] is a dialect of Qbasic, implemented for mobile devices. It includes a floating point support library which is not included in the measured size.

JScheme [Nor98] is a small Scheme implementation. We benchmark an early version as later versions have a vastly larger code footprint. It depends on reflection, and therefore it does not run on Java Micro Edition, but it is included because the implementation is compact, and may be changed to target mobile devices.

Rhino [Moz07] is a JavaScript implementation. This language is not designed for, nor does it run on, mobile devices. It is included as an example of an implementation of a usual non-mobile scripting language.

Results

The approximate JAR sizes of the scripting languages are:

Yolan	7K
LightScript	14K

FScriptME	17K
Jscheme	29K
Kahlua	39K
Hecl	54K
Simkin	81K
CellularBasic	83K
Rhino	397K

The languages developed in this thesis are smaller than other scripting languages for the platform. Yolan is approximately half the size of LightScript. One of the reasons for this can be that most of the other languages are implemented with nice object oriented designs on top of Java. Yolan and LightScript has been implemented with code space optimisations in mind from the very beginning, and with a focus on the generated JVM code rather than just using the abstractions of Java.

FScriptME is also quite small, and looking at its source, it has few Java classes, and it relies on the standard Java classes for types. Another cause of the small code size, may be size of the language which only has strings and integers as datatypes. Approximately 80% of the source code of FScriptME is its lexer and parser class.

5.1.2 Details on the footprint of LightScript

The sizes of the different parts of the LightScript class are listed below. "Reduction" is the reduction of the full class file when the mentioned part is left out. "Alone" is the size of the class containing only the mentioned part. These numbers are different as some things are shared and some things cannot be left out when compiling the class file.

	Reduction	Alone
Everything	15030	17706
Embedding API	645	3597
Tokeniser	1261	4048
Parser	2542	5850
Compiler	5324	8413
Virtual machine	4589	7650
Parser+Tokeniser	3825	7093

There is nothing particular surprising about these result. The code footprint is mainly split the three main parts: parser/tokeniser, compiler, and virtual machine, which are similar in size. Worth noticing is that the parser/tokeniser is the smallest of the three parts, indicating that a top-down-operator-precedence parser can be quite small.

5.1.3 Optimised JAR-file footprint

Previous sections have looked at comparable footprints for implementations by approximated JAR file size, and also at what parts contributes to the size of the LightScript class. From a practical view it is also interesting to consider the actual size of the optimised obscurified JAR-file of embedding the languages in a trivial host application. The host application is a simple Midlet that adds a print function to the language, and includes a hello-world program. The size of the entire application, including the embedded scripting language implementation is 5290 bytes for Yolan and 11342 bytes for LightScript.

5.2 Execution speed

This section benchmarks the execution speed of the languages. In the following subsections, the benchmarked languages, and the actual benchmark programs are described, and finally the results of running the benchmarks are tabulated.

5.2.1 Languages

The benchmarks are run on those languages from Section 5.1.1 that have an approximated JAR size of less than 64K. They also run on two JavaScript interpreters: Rhino 1.6r7 and SpiderMonkey 1.7.0, which are described in Section 2.3.3. Both are the default versions installed on Ubuntu Linux.

5.2.2 The benchmarks

The benchmarks are the following:

Fibonacci: Recursive calculation of the 30'th Fibonacci number

Loops: Nested loops with counters, 10.000.000 iterations

Recursion: Recursive benchmark, similar to the "controlflow-recursive" benchmark from [Web07, sho09]. It runs the ack, fib and tak function, which are highly recursive and uses a lot of stack space. Many of the scripting language runs out of stack space when executing this benchmark. On some of the languages where it fails, only the first part (the Ackermann function) was implemented.

Sieve: Simple implementation of Erasthones sieve - not implemented in languages that have already shown to be very slow in earlier benchmarks

For-in: Nested loops across keys of a dictionary, 1.000.000 iterations - not implemented in languages that have already shown to be very slow in earlier benchmarks

Primes: Simple primality test by looking at the remainders of division - not implemented in languages that have already shown to be very slow in earlier benchmarks

Exception: Throw/catch 500.000 exceptions. It is only implemented for LightScript/JavaScript as Yolan does not support exceptions.

Fannkuch: The access-fannkuch benchmark from [Web07, sho09], slightly modified to be able to run within LightScript – postfix increments replaced by prefix increments, and a `break` within a loop rewritten to a condition. This benchmark calculates and does a sequence of permutations of array elements. It is also an example of how existing JavaScript code can be ported to LightScript, and is thus only implemented in LightScript/JavaScript.

5.2.3 Results

The measurements of the performance of the different scripting languages are shown below. The timings are seconds per benchmark. $\perp$ indicates that the benchmark does not complete due to running out of stack space.

	Fibonacci		Recursion	For-in		Exceptions	
		Loops		Sieve	Primes		Fannkuch
Rhino	1.20	1.74	1.75	2.97	1.18	12.35	45.99	6.35
SpiderMonkey	1.28	1.71	$\perp$	2.08	1.14	11.03	0.45	5.10
LightScript	1.37	3.45	2.35	1.19	0.57	11.70	0.65	11.15
Yolan	1.47	2.23	$\perp$	1.95	0.32	9.20		
Kahlua	2.13	1.18	$\perp$	5.73	2.26	5.49		
JScheme	29.77	93.22	$\perp$					
FscriptME	176.27	112.68	$\perp$					
Hecl	207.96	47.21	$\perp$					

Rhino, SpiderMonkey, LightScript, Yolan and Kahlua are similar in speed, whereas JScheme, FscriptME and Hecl is a magnitude slower. Rhino and LightScript are the only ones which can handle the very recursive benchmark, whereas the others run out of stack space.

The results are discussed more in details in chapter 6.

Chapter 6

Discussion and future work

This chapter starts with discussion of the benchmark results, with a focus of the possible reasons for the execution speeds in Section 6.1, and the possible reasons for the code footprint sizes in Section 6.2. Section 6.3 then adds a discussion about the languages developed in the thesis, and finally Section 6.4 will look more into the future directions of LightScript

6.1 Performance

It is interesting to see that Rhino, SpiderMonkey, LightScript, Yolan, and Kahlua are all within the same magnitude of speed for all of the benchmarks, indicating that this is the speed obtainable with a simple interpreter for these kinds of scripting languages. Here it is also suprising, that Rhino and SpiderMonkey are not much faster, because Rhino compile to, and loads, Java classes at run-time, and SpiderMonkey is written in C rather than Java, which allows for using more advanced implementation techniques.

Other observation are that Rhino is slow with exceptions, and that Kahlua seems to be fast with loops and slow with arrays. The last is probably because Kahlua implement arrays as hashtables[1], and that the `for`-construct in Lua automatically increments and tests the counter variable, whereas this has to be executed as interpreted code in the other languages.

It is also interesting how similar in performance Yolan and LightScript are, even though they have very different evaluation strategies: Yolan is interpreted by traversing a syntax-tree data structure, whereas LightScript runs on a stack-based virtual machine.

Most of the scripting languages with a small code footprint are an order of magnitude slower than LightScript, Yolan, and Kahlua and the JavaScript implementations. This may be due to inefficient implementation of variable access in JScheme, FScriptME and Hecl. Looking at the source code, JScheme and FScriptME variables are looked up in a linked list or hashtable at each access. Hecl has a more complex lookup mechanism with a stack of hashtables, but it also employes some kind of caching.

[1] This is different from the C implementation of Lua where they are implemented as arrays.

6.2 Size

The scripting languages developed in the thesis has significantly smaller code footprint than other scripting languages for mobile Java devices. Two reasons for this are: 1) the implementations have focused on code footprint from start to end, using techniques for writing compact code to a much higher degree than existing scripting language implementations for mobile Java devices, and 2) the parsers, which usually would take tens, or more, of kilobytes, has been reduced significantly: in the case of Yolan this was done by having a trivial syntax, and in the case of LightScript by optimising and using top-down operator precedence parsing.

The size comparison is even more favorable, especially for LightScript, when taking the features of the languages into account: The third language in size, FScriptME, is a very minimalistic language, that does not have builtin support for arrays, objects, higher-order functions, or exceptions, which accounts for its small size. The next two languages, in fourth and fifth place in size, are JScheme and Kahlua, having a simple or no parser, due to having Lisp-like syntax or being just a virtual machine.

That the parser is likely to be a bottleneck can be seen in FScriptME, where the parser is 80% of the implementation. In CellularBasic the source code of the lexer and parser is more than 100KB. The cost of parser implementation may also have be an influence on Hecl not supporting Tcl-like `expr` with infix binary operators, but instead Hecl only has comparisons and arithemetic operators written in Lisp-like prefix notation which are simpler to write parsers for.

6.3 Developed languages

6.3.1 Yolan

Yolan does a good job of having a tiny code footprint and decent performance, while being a scripting language with first-class function, hashtables, easy embedding, etc. Nevertheless, it does a bad job trying to do anything else: the dynamic scoping makes it unsuitable for developing larger applications, and the Lisp-like syntax will probably scare away most developers. So the major use of this language is probably just to set a bar for how small a code footprint a high-level scripting language can have.

While it has an interesting result, code-size wise, there will probably not be any further developments in this language, other than serving as a stepping stone towards LightScript.

6.3.2 LightScript

LightScript is much more featureful and useful than Yolan. While this comes at the cost of having twice the footprint of Yolan, it is still small compared to other scripting languages targeting mobile devices. The JavaScript/EcmaScript-like syntax and semantics should make it much more approachable for other

developers, while still having expressive features like supporting closures and higher-order functions.

There are tradeoffs between EcmaScript compliance and size/performance. One of those is the joining of `false`, `null` and `undefined`, and not needing the boolean type. As the overall size and performance of LightScript were better than expected, that optimisation-tradeoff is probably worth undoing to get better semantics, but only an implementation and actual benchmark will show whether this is the case.

LightScript is definitely a language I will use for developing mobile applications in the future, so my project has been successful by making a tool in that direction.

6.4 Future development of LightScript

The further development with LightScript is toward version 1.2, which is described in the LightScript chapter. Another aspect is to make LightScript public available, and expanding the library as discussed below. On the long term, other possible milestones are optimisation of the byte code, porting LightScript to embedded C, and better development tools for prettyprinting and linting.

6.4.1 Making it public

LightScript is released as open source software. Just releasing the source, however, is not enough to make it really usable for others: documentation and examples are also very important, and there also has to be an awareness about the existence of the language, which entails some work on publishing information on the project, including an informative wikipedia page, and press release/posting of articles on news sites related to programming languages and mobile development. The website `http://www.lightscript.net/`, will serve as the main site, and a small dummy page is already put up.

6.4.2 User interface and other optional libraries

Currently LightScript needs custom Java functions to be added for doing anything useful, as there is no library for input/output. As the language implementation stabilises, the next step is to create a library which can optionally be included within the application. A lot of standard functionality, such as cryptography, http-networking, storage, etc., can relatively easily be merged from the platform, or existing open source libraries. Other libraries require further work: One major task here is integration of a graphical user interface with LightScript, as on this point, there are major differences between the platforms on which LightScript runs, both within the APIs on these platforms, and also between the actual platform capabilities.

User interface

Additional libraries for LightScript are scheduled for version 1.4, and the major design task here will be the user interface. Issues are variation and limitations in resolutions, ranging from screen size of 96x65 pixels, up to full desktops, and input methods from simple phone keypad, to touch-input-only, to multi touch, to regular computers.

A solution that I will pursue is in the direction of zooming interfaces, as these able to scale down to small screens. Prototypes within the thesis has also headed in the direction of zooming user interfaces, though they have yet to be integrated with the scripting language, and have not yet achived maturity to become a part of this report.

Other optional libraries

Simple function for getting and parsing web pages will be practical for enabling LightScript as a tool for mobile mashups. This will be a simple heuristic html parser mapping the XML-tree to JsonML[2] [McK], which can then easily be interacted with in LightScript. The access to http-connections are supported within CLDC/1.0 and CLDC/1.1, as well as larger Java platforms, and just needs to be wrapped.

6.4.3 Optimising the bytecode

A way to optimise the execution, and reduce the memeory usage of compiled programs, would be to optimise the byte codes, by joining common code sequences. Which byte codes to combine can be determined by using grammar-based compression techniques on generated code, followed by creating new codes from the found dictionary. This reduces the memory usage as the generated code becomes shorter, and it increases the performance, as the joined codes only need a single dispatch for the new code. The cost is a larger code footprint due to the new byte codes. If this is to be implemented, it probably has to wait until more code has been written using LightScript, as a larger amount of code is needed for better generation of dictionaries.

6.4.4 Embedded implementation

Another direction with the development of LightScript is the implementation on non-Java platforms. Embedded C is very different from Mobile Java: the run-time memory usage is more important than the code footprint, as embedded devices often have larger built-in flash ROM than working memory.

The first target platform for an embedded C implementation will be the Mindstorm NXT, which has 64K of RAM, and 256K of ROM. An optimisation here is to use 16 bit tagged value representing a pointer or small integers, instead of 32 bit words, in order to save memory.

An interesting implementation aspect to study is the effect of enforcing unique strings. This will cost approximately additional 3 bytes per string, and

[2] JsonML is an embedding of XML in JSON, somewhat similar to SXML in Scheme

string creation will be more expensive, but at the same time, there will be no copies of identical strings, which may save significant space, and string comparison is changed to pointer comparision, which is much faster.

6.4.5 Lint and prettyprinter

Another future target is to improve the development environment for LightScript. As the parser does not support error checking, currently the easiest way to develop LightScript scripts is to use an EcmaScript compliant development platform, and stay within the limits of the LightScript subset. A LightScript linting program would be a practical tool, as this would also be able to check, if the programmer goes outside the LightScript subset of EcmaScript. Currently much of this can be caught with debugging in the current version LightScript, but a real linter may also warn about bad usages, and would have more helpful messages to the developer.

Another development tool, possibly integrateable with the linter is a prettyprinter, which is both practical to enforce coding conventions and to keep source readable when several programmers are working on the same project. It may also go beyond just prettyprinting and further may transform some unsupported EcmaScript features into LightScript.

Chapter 7

Conclusion

I have created two new scripting languages targeting mobile devices.

Yolan is a minimalistic scripting language with first-class functions and Lisp-like syntax. This language is a proof of concept that the code footprint cost of an embeddable scripting language for Mobile Java applications, can be kept low, in this case approximately 5 KB.

LightScript is a more advanced language, that has a larger footprint than Yolan, but it still smaller than other languages. LightScript is more developer friendly, being a subset of JavaScript, and it has also more features, such as an object system with inheritance and support for exceptions. The language and its implementation can be of practical use. I have released it as open source and will develop it further after this thesis.

Yolan and LightScript are pushing the edge of small scripting language implementation on mobile Java enabled devices, by having a smaller code footprint, being fast, and having advanced language features, compared to other scripting language on mobile devices.

Bibliography

[Aig] Rodney Aiglstorfer. Milscript specification export - mojax - mfoundry wiki. `http://mfwiki.mfoundry.com/display/mojax/MILScript+Specification+Export`.

[ARM01] ARM. *ARM7TDMI (rev.3) Technical Reference Manual*, 2001.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. `http://agilemanifesto.org`, 2001.

[Bor08] Dan Bornstein. Dalvik vm internals. `http://sites.google.com/site/io/dalvik-vm-internals`, 2008.

[Bro84] Leo B. Brodie. *Thinking FORTH: a language and philosophy for solving problems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[can09] Chdk – a wiki dedicated to the chdk firmware add-on for canon's digic ii, digic iii and digic iv cameras. `http://chdk.wikia.com/`, 2009.

[CBYG07] Mason Chang, Michael Bebenita, Alexander Yermolovich, and Andreas Gal. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, 2007.

[CN09] Cellular-News. Worldwide mobile phone sales down by 5% in q4 '08. `http://www.cellular-news.com/story/36315.php`, March 2009.

[Cro08] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly, 2008.

[Cro09a] Douglas Crockford. Douglas crockford's javascript. `http://javascript.crockford.com/`, April 2009.

[Cro09b] Douglas Crockford. Jslint, the javascript verifier. `http://jslint.com/`, 2009.

[DBC+02] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *In Interpreters, Virtual Machines and Emulators (IVME'03)*, pages 41–49. ACM Press, 2002.

[ECM99] ECMA International. *Standard ECMA-262 – ECMAScript Language Specification*. 1999.

[Eic08] Brendan Eich. Brendan's roadmap updates: Tracemonkey: Jsvascript lightspeed. `http://weblogs.mozillazin.org/roadmap/archives/2008/08/tracemonkey_javascript_lightsp.html`, 2008.

[Els06] Mustafa M. H. Elsheikh. Cellularbasic: On-phone open source mobile basic interpreter for j2me. `http://cellbasic.sourceforge.net/`, 2006.

[Ert96] M. Anton Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, Austria, 1996.

[Esm08] Esmertec. Q2 2008: Esmertec reports 27.6 million jbed shipments and 146 million mobile subscribers. `http://www.myriadgroup.com/pdfs/esmertec/2008/Esmertec_reports_27.6_million_Jbed_shipments.pdf`, September 2008.

[Fag05] Fabian Fagerholm. Perl 6 and the parrot virtal machine. `www.parrot.org/files/Fagerholm-Parrot.pdf`, 2005.

[Gar09] Gartner Inc. Worldwide smartphone sales reached its lowest growth rate with 3.7 per cent increase in fourth quarter of 2008. `http://www.gartner.com/it/page.jsp?id=910112`, March 2009.

[Get09] GetJar. Cldc version market share (march 2009). `http://stats.getjar.com/statistics/world/gJavaCLDCVer`, April 2009.

[GF06] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical report, 2006.

[Goo08] Google. V8 javascript engine. `http://code.google.com/p/v8`, 2008.

[IdFC05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.

[Jon07] Richard W.M. Jones. A sometimes minimal forth compiler and tutorial for linux / i386 systems. `http://annexia.org/forth`, October 2007.

[Kar08] Kristofer Karlsson. Methods for reducing distribution size for j2me applications. Master's thesis, KTH Computer Science and Education, 2008.

[Kar09] Kristofer Karlsson. Kahlua – a lua virtual machine for java. `http://code.google.com/p/kahlua/`, 2009.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Mar 2004.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[Mbe04] Mbedthis Software. Mbedthis appweb - embedded javascript. `http://www.appwebserver.org/products/appWeb/doc/guide/appweb/users/ejs/overview.html`, 2004.

[McK] Stephen M. McKamey. Jsonml (json markup language). `http://jsonml.org/`.

[Moz] Mozilla. Spidermonkey. (source code).

[Moz07] Mozilla. Rhino – javascript for java. `http://www.mozilla.org/rhino`, 2007.

[Mur04] Murlen. Fscript. `http://fscript.sourceforge.net`, 2004.

[nin08] Dslinux: the home of linux on the ds. `http://www.dslinux.org`, 2008.

[Nok04] Nokia. Designing midp applications for optimization, August 2004.

[Nok09] Nokia. Qt - a cross-platform application and ui framework. `http://www.qtsoftware.com/`, 2009.

[Nor98] Peter Norvig. Jscheme: Scheme implemented in java. `http://norvig.com/jscheme.html`, 1998.

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[OW07] Andy Oram and Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. 2007.

[Pes08] Slava Pestov. Factor: An extensible interactive language. Google Tech Talks, October 2008.

[Pkw07] Pkware. .zip application note support. `http://www.pkware.com/support/zip-application-note`, September 2007. Specification of the zip file format.

[Pra73] Vaughan R. Pratt. Top down operator precedence. pages 41–51, 1973.

[Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[SGBE08] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):1–36, 2008.

[sho09] The computer language benchmarks game. `http://shootout.alioth.debian.org/`, 2009.

[Sof09] Embedthis Software. Ejscript. `http://www.ejscript.org/products/ejs/index.html`, 2009.

[Sta08] Maciej Stachowiak. Introducing squirrelfish extreme. `http://webkit.org/blog/214/introducing-squirrelfish-extreme/`, 2008.

[Sun00] Sun Microsystems. Connected, limited device configuration. specification version 1.0. JSR-30, May 2000.

[Sun03a] Sun Microsystems. Cldc reference implementation version 1.1. (source code), March 2003.

[Sun03b] Sun Microsystems. Connected limited device configuration. specification version 1.1. JSR-139, March 2003.

[Sun09] Sun Microsystems. The java me platform. `http://java.sun.com/javame/`, 2009.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[Web07] WebKit.org. SunSpider JavaScript Benchmarks. `http://www2.webkit.org/perf/sunspirder-0.9/sunspider.html`, 2007.

[Web09] WebKit.org. The webkit open source project. `http://webkit.org/`, 2009.

[Whi09] Simon Whiteside. Simkin scripting language. `http://simkin.co.uk/simkin_language.html`, 2009.

[WK09] David N. Welton and Wolfgang Kechel. Hecl – the mobile scripting language. `http://www.hecl.org/`, 2009.

Appendix A

Thanks

Thanks to Julia for supervising this thesis, for giving me the freedom to go in the directions I chose, and for asking questions which saved me from choosing wrong roads. Thanks for the immense feedback which has made me a better computer scientist and a better writer. Thanks.

Thanks to MariAnne for hospitality and reading the thesis.
Thanks to Lotte for turning my world upside down.
Thanks to family, friends and colleagues
for bearing with me, while I worked on the thesis.
Thanks to the computer scientists, mathematicians, and other giants
on whose shoulders I try to climb to see further.
And thanks to God, to whom the honour belongs.

Appendix B

Grammar

This chapter describes the grammar for Yolan and LightScript. In order to reduce the code footprint of the parsers, the grammars are very lax, and are included to explain the implementation, and are not meant for learning how to program in the languages.

Notation: Literal characters are writen with single quotation mark. Ranges of possible characters are written within square brackets, such that `[a-c]` means one of the characters a, b or c. Ranges can also be negated, such that `[^a-cf]` means a character that is not a, b, c nor f. A range can also contain escaped characters: `\0` means the character with unicode value 0, and `\\` means the backslash character. `\[`, `\]`, and `\-` means `[`, `]` and `-`. `\n`, `\r`, and `\t` means new line, carriage return, and tab. Grammar productions are defined with `::=`, parentheses are used for grouping and `|` is means or. `*` is notation for zero or more occurences of the previous element.

B.1 Yolan

```
<number>            ::= [0-9][0-9]*

<identifier>        ::= [^\0- "0-9;\[\]] [^\0- \[\]]*

<comment>           ::= ';' [^\0-\n]* [\0-\n]

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

<white space>       ::= [\0- ]
                      | <white space> <comment>

<white spaces>      ::= <white space>
                      | <white space> <white spaces>

<optional ws>       ::= <white spaces> |

<expressions>       ::= <expression>
```

```
                      | <expression> <white spaces> <expressions>

<expression list>   ::= <optional ws> <expressions> <optional ws>

<expression>        ::= '[' <expression list> ']'
                      | <identifier>
                      | <string>
                      | <number>

<program>           ::= <expression list>
```

B.2 LightScript

The following sections describes the syntax accepted by LightScript 1.1. The language is very lax to minimise the code footprint of the parser. *When programming in LightScript, the code should be written such that it is compatible with EcmaScript, and the following grammar should not be used for learning LightScript.* This is only expository to give an impression of what the parser accepts, and thus how the syntax can be relaxed to reduce the code footprint. A example of how lax the syntax is, is demonstrated by the following EcmaScript/LightScript `for`-loop:

```
for(x = 0; x < 10; ++x) {
    print(x);
}
```

which also parses and executes in LightScript if written as "`for[x=0,x<10:++x]{ print(x])`" Correct LightScript code is the only that which is both accepted by the grammar of LightScript, and also is valid EcmaScript.

The next three sections describes the grammar: First there is the syntax of the tokens in Section B.2.1, then there is a tabulation of the tokens as in the top down operator precedence parser, in Section B.2.2, and the actual grammer is written in Section B.2.3.

B.2.1 Grammar for tokens

```
<number>            ::= [0-9] [0-9]*

<name>              ::= [_a-zA-Z] [_a-zA-Z0-9]*

<string>            ::= '"' ( [^"] | '\' [\\"n] )* '"'

<comment>           ::= '//' [^\n]* [\n]

<white space>       ::= [ \n\r\t]
                      | <comment>
                      | <white space> <white space>
```

```
<long symbol>       ::= [=!<&|+*/\->%]

<single symbol>     ::= [(),.:;?\[\\\]^{}~]

<symbol>            ::= <long symbol> <long symbol>*
                      | <single symbol>

<literal>           ::= <number> | <string>

<identifier>        ::= <name> | <symbol>

<token>             ::= <literal> | <identifier>

<program>           ::= ( <token> | <white space> )*
```

B.2.2 Top down operator precedence functions

`<literal>`s are mapped to a syntax tree node which represent a literal value. If an `<identifier>` is within the table below, the corresponding function are called in the top down operator precedence parser, otherwise the `<identifier>` is just viewed as a reference to a variable.

Tokens	Null denominator	Left denominator	Precedence
`.`		Infix	7
`( [`	List	Infix list	7
`/ * % >> /`		Infix	6
`+`		Infix	5
`-`	Prefix	Infix	5
`== === != !==`		Infix	4
`<= < > >=`		Infix	4
`&& || else in`		Right infix	3
`?`		Infix3	3
`= += -= *= /= %=`		Right infix	2
`function`	Function case		
`: ; ,`	Separator		
`] ) }`	List end		
`try catch do`	Prefix2		
`for if while`	Prefix2		
`++ -- !`	Prefix		
`return throw var`	Prefix		
`undefined null`	Single		
`true false this`	Single		
literal identifier	Single		
`{`	List		

"Infix" joins the previous and the next expression in a left skewed tree, whereas "Right infix" joins the previous and next expression in a right skewed tree.

“Infix3” creates a node which combines left expression and the next three right expressions. “InfixList” creates a node which takes the previous expression, and reads the next expressions until a “List end” is found.

“List“ reads the next expression until a “List end” is found. “Single” creates a leaf in the syntax tree. “Prefix” create a node with the next read expression. “Prefix2” create a node with the two next expressions read. “List end” is a special node used for terminate lists. “Separator”s are special nodes, that may be discarded during compilation. “Function case” handles `function`s which is a “Prefix2” or “Prefix3” depending on whether the first parsed expression is a list or identifier.

B.2.3 Grammar

The following is the grammar accepted by LightScript. This uses rules as defined in Section B.2.1: `<literal>` and `<identifier>`, with the exception that `<identifier>` should not include the tokens which is mentioned in the table in Section B.2.2.

```
<prefix name>       ::= '-' | '++' | '--' | '!'
                      | 'return' | 'throw' | 'var'

<prefix2 name>      ::= 'try' | 'catch' | 'do' | 'for' | 'if' | 'while'

<single>            ::= 'undefined' | 'null' | 'true' | 'false'
                      | 'this' | <identifier> | <literal>

<separator>         ::= ':' | ';' | ','

<list start>        ::= '[' | '{' | '('

<list end>          ::= ']' | '}' | ')'
                      | <expr> <list end>

<list>              ::= <list start> <list end>

<function>          ::= 'function' <list> <expr>
                      | 'function' <identifier> <list> <expr>

<expr8>             ::= <prefix> <expr>
                      | <prefix2> <expr> <expr>
                      | <single>
                      | <separator>
                      | <list>
                      | <function>

<expr7>             ::= <expr8>
                      | <expr7> '.' <expr8>
```

```
                      | <expr7> ('(' | '[') <list end>

<expr6 operator>    ::= '/' | '*' | '%' | '>>' | '/'

<expr6>             ::= <expr7>
                      | <expr6> <expr6 operator> <expr7>

<expr5>             ::= <expr6>
                      | <expr5> ('+' | '-') <expr6>

<expr4 operator>    ::= '==' | '===' | '!=' | '!=='
                      | '<=' | '<' | '>' | '>='

<expr4>             ::= <expr5>
                      | <expr4> <expr4 operator> <expr5>


<expr3 operator>    ::= '&&' | '||' | 'else' | 'in'

<expr3>             ::= <expr4>
                      | <expr4> <expr3 operator> <expr3>
                      | <expr3> '?' <expr3> <seperator> <expr3>

<expr2 operator>    ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='

<expr>              ::= <expr3>
                      | <expr3> <expr2 operator> <expr>

<program>           ::= <expr>*
```

Appendix C

Source code of the Yolan class

```
import java.io.InputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Random;
import java.util.Stack;

/**
 * Yocto-language (a very small programming language)
 * mainly targeted Java Micro Edition / J2ME.
 *
 * Objects of the class are delayed computations
 * / evaluatable expressions, which can be evaluated with the
 * value method.
 * The class itself also has a single virtual machine builtin
 * and the static methods typically manipulates that one,
 * - not reentrant nor threadsafe as a tradeof for less
 * memory and code footprint.
 */
public final class Yolan {

/*
```

C.1 Constants

```
*/

    //////////////////////
    // Internal functions
    ////
    private static final int FN_NATIVE_DUMMY = -12;
    private static final int FN_BUILTIN_DUMMY = -11;
    private static final int FN_LITERAL = -10;
    private static final int FN_RESOLVE_GET_VAR = -9;
    private static final int FN_RESOLVE_EVAL_LIST = -8;
    private static final int FN_CALL_USER_FUNCTION = -7;
    private static final int FN_USER_DEFINED_FUNCTION = -6;
    private static final int FN_GET_VAR = -5;
    private static final int FN_SET = -4;
    private static final int FN_NATIVE = -3;
    private static final int FN_FOREACH = -2;
    private static final int FN_LOCALS = -1;
    //////////////////////
    // Builtin functions
    ////
    // Variables
    private static final int FN_RESOLVE_SET = 0;
    private static final int FN_RESOLVE_LOCALS = 1;
```

```
    // Conditionals and logic
    private static final int FN_IF = 2;
    private static final int FN_NOT = 3;
    private static final int FN_AND = 4;
    private static final int FN_OR = 5;
    // Repetition
    private static final int FN_REPEAT = 6;
    private static final int FN_RESOLVE_FOREACH = 7;
    private static final int FN_WHILE = 8;

    // Functions and sequencing
    private static final int FN_LAMBDA = 9;
    private static final int FN_DEFUN = 10;
    private static final int FN_APPLY_FUNCTION = 11;
    private static final int FN_DO = 12;

    // Integer operations
    private static final int FN_ADD = 13;
    private static final int FN_SUB = 14;
    private static final int FN_MUL = 15;
    private static final int FN_DIV = 16;
    private static final int FN_REM = 17;
    private static final int FN_LESS = 18;
    private static final int FN_LESS_EQUAL = 19;

    // Type conditionals
    private static final int FN_IS_INTEGER = 20;
    private static final int FN_IS_STRING = 21;
    private static final int FN_IS_LIST = 22;
    private static final int FN_IS_DICT = 23;
    // Polymorphic functions
    private static final int FN_EQUALS = 24;
    private static final int FN_IS_EMPTY = 25;
    private static final int FN_PUT = 26;
    private static final int FN_GET = 27;
    private static final int FN_RANDOM = 28;
    private static final int FN_SIZE = 29;

    // String Functions
    private static final int FN_STRINGJOIN = 30;
    private static final int FN_SUBSTRING = 31;
    private static final int FN_STRINGORDER = 32;

    // List functions
    private static final int FN_LIST = 33;
    private static final int FN_RESIZE = 34;
    private static final int FN_PUSH = 35;
    private static final int FN_POP = 36;
    // Dictionary functions
    private static final int FN_DICT = 37;
    // Enumeration functions
    private static final int FN_KEYS = 38;
    private static final int FN_VALUES = 39;
    private static final int FN_GET_NEXT = 40;
    // Debugging
    private static final int FN_DEBUG_STRING = 41;


/*
```

C.2 The static execution context

```
*/

    /** Random source */
    private static Random random = new Random();
```

```
    /** Returnable true value */
    private static final Boolean TRUE = new Boolean(true);

    /** The variables. The array contains alternating variable names and values */
    private static Object vars[];

    /** The number of variables in vars */
    private static int varsize;

    /** Stack used for function calls */
    private static Stack stack;

/*
```

C.2.1 Make variables and native functions available

```
*/

    /**
     * Lookup the id of a variable. If the variable does not have an id,
     * allocate a new one for it. This lookup might be slow, so avoid
     * calling it in inner loops, but make lookups a priori and cache
     * the ids.
     * @param key the name of the variable
     * @return the id which maps into the vars-array
     */
    public static int resolveVar(String key) {
        int i = 0;
        while (i < varsize && !vars[i].equals(key)) {
            i += 2;
        }

        if (i == varsize) {
            if (varsize == vars.length) {
                // grow the var array exponential, and make sure
                // that the size is divisible by 2
                Object objs[] = new Object[(varsize * 5 / 4 + 4) & ~1];
                System.arraycopy(vars, 0, objs, 0, varsize);
                vars = objs;
            }
            vars[i] = key;
            varsize += 2;
        }
        return i + 1;
    }

    /**
     * Set the value of a variable
     * @param id the id of the variable, found with resolveVar
     * @param val the new value
     */
    public static void setVar(int id, Object val) {
        vars[id] = val;
    }

    /**
     * Set the value of a variable
     * @param id the id of the variable, found with resolveVar
     * @return the new value
     */
    public static Object getVar(int id) {
        return vars[id];
    }

    /**
     * Register a native function
     * @param name the name of the function
```

```
     * @param f the function itself
     */
    public static void addFunction(String name, Function f) {
        int id = resolveVar(name);
        vars[id] = new Yolan(FN_NATIVE_DUMMY, f);
    }

/*
```

C.2.2 Initialisation

```
*/

    /**
     * Register a builtin function
     * @param val the function id
     * @param name the name in the runtime
     */
    private static void addBuiltin(int val, String name) {
        int id = resolveVar(name);
        vars[id] = new Yolan(FN_BUILTIN_DUMMY, new Integer(val));

    }

    /** Initialisation */
    static {
        reset();
    }

    /**
     * resets every static element,
     * allowing them to be garbage collected
     * - futher evaluation of Yolan objects
     * will fail.
     */
    public static void wipe() {
        vars = null;
        stack = null;
    }

    /**
     * Resets the virtual machine.
     * After the virtual machine has been reset,
     * only new Yolan instances should be evaluated,
     * and the result of evaluating existing objects are undefined.
     */
    public static void reset() {
        vars = new Object[0];
        stack = new Stack();
        varsize = 0;

        /* Initialisation of builtin names
         * Commented out and replaced with optimised version below.

        // Variables
        addBuiltin(FN_RESOLVE_SET, "set");
        addBuiltin(FN_RESOLVE_LOCALS, "locals");

        // Conditionals and truth values
        addBuiltin(FN_IF, "if");
        addBuiltin(FN_NOT, "not");
        addBuiltin(FN_AND, "and");
        addBuiltin(FN_OR, "or");
        ...
         */

        // space optimised initialisation of builtin names
```

```
        // - having all the name in a single string
        // and then manually extracting them, is spacewise
        // significantly cheaper due to the design of the
        // class file format.

        String builtins = "set locals if not and or repeat foreach " +
                "while lambda defun apply do + - * / % < <= is-integer " +
                "is-string is-list is-dict equals is-empty put get " +
                "random size stringjoin substring stringorder list " +
                "resize push pop dict keys values get-next to-string ";

        int prevpos = 0;
        int pos = 0;
        int id = 0;
        while (pos < builtins.length()) {
            while (!builtins.substring(pos, pos + 1).equals(" ")) {
                pos++;
            }
            addBuiltin(id, builtins.substring(prevpos, pos));
            pos++;
            prevpos = pos;
            id++;
        }
    }

/*
```

C.3 The delayed computation

```
*/

    /** The closure */
    private Object c;

    /** The function id */
    private int fn;

    /**
     * Constructor for a new delayed computation
     * @param fn the function id
     * @param c the closure
     */
    private Yolan(int fn, Object c) {
        this.fn = fn;
        this.c = c;
    }

/*
```

C.3.1 Function application

```
*/

    /**
     * The number of arguments the function takes,
     * if the delayed computation is an applicable
     * user defined function. Notice that builtin functions
     * and added native functions does not have a number of arguments.
     * @return the number of parameters, or -1 if not a user defined function
     */
    public int nargs() {
        if (fn != FN_USER_DEFINED_FUNCTION) {
            return -1;
        }
        return ((Object[]) c).length - 1;
    }

    /**
```

```
     * apply as a function without arguments
     */
    public Object apply() {
        return doApply(0);
    }

    /**
     * apply as a function with one argument
     */
    public Object apply(Object arg1) {
        stack.push(arg1);
        return doApply(1);
    }

    /**
     * apply as a function with two arguments
     */
    public Object apply(Object arg1, Object arg2) {
        stack.push(arg1);
        stack.push(arg2);
        return doApply(2);
    }

    /**
     * apply as a function with three arguments
     */
    public Object apply(Object arg1, Object arg2, Object arg3) {
        stack.push(arg1);
        stack.push(arg2);
        stack.push(arg3);
        return doApply(3);
    }

    /**
     * apply as a function with the arguments given as an array
     */
    public Object apply(Object args[]) {
        for (int i = 0; i < args.length; i++) {
            stack.push(args[i]);
        }
        return doApply(args.length);
    }

/*
```

C.3.2 Conversion to string

```
*/

    /*
     * if the computation is a variable or literal,
     * return its string representation
     */
    public String string() {
        if (fn == FN_RESOLVE_GET_VAR) {
            return (String) c;
        } else if (fn == FN_GET_VAR) {
            return (String) vars[((Integer) c).intValue() - 1];
        } else if (fn == FN_LITERAL) {
            if (c instanceof Integer) {
                return c.toString();
            } else if (c instanceof String) {
                return "\"" + c + "\"";
            }
        }
        return null;
    }
```

```
    /**
     * Override the toString function
     */
    public String toString() {
        return to_string(new StringBuffer(), this).toString();
    }

/*
```

C.3.3 Evaluation of delayed computation

```
*/

    /**
     * Evaluate the delayed computation
     * @return the result
     * @throws java.lang.Throwable Errors with bounds automatically
     *                             thrown. Throws Error if evaluating
     *                             things that are not functions
     */
    public Object value() {
        switch (fn) {
            case FN_LITERAL: {
                return c;
            }

            case FN_RESOLVE_GET_VAR: {
                fn = FN_GET_VAR;
                c = new Integer(resolveVar((String) c));
                return value();
            }

            case FN_RESOLVE_EVAL_LIST: {
                Object o = val0();
                if (o instanceof Yolan) {
                    Yolan yl = (Yolan) o;

                    // builtin function
                    if (yl.fn == FN_BUILTIN_DUMMY) {
                        this.fn = ((Integer) yl.c).intValue();
                        Object args[] = (Object[]) c;
                        Object newargs[] = new Object[args.length - 1];
                        for (int i = 0; i < newargs.length; i++) {
                            newargs[i] = args[i + 1];
                        }
                        c = newargs;
                        return value();

                    // native function implementing Function-interface;
                    } else if (yl.fn == FN_NATIVE_DUMMY) {
                        this.fn = FN_NATIVE;
                        Object args[] = (Object[]) c;
                        Yolan params[] = new Yolan[args.length - 1];
                        for (int i = 0; i < params.length; i++) {
                            params[i] = (Yolan) args[i + 1];
                        }
                        args[0] = yl.c;
                        args[1] = params;
                        return value();

                    // user defined function
                    } else if (yl.fn == FN_USER_DEFINED_FUNCTION) {
                        Object args[] = (Object[]) c;
                        args[0] = yl;
                        this.fn = FN_CALL_USER_FUNCTION;
                        return value();
```

```
            }
        }
        throw new Error("Unknown function: "
                        + ((Yolan) ((Object[]) c)[0]).string());
    }

    case FN_CALL_USER_FUNCTION: {
        Object args[] = (Object[]) c;
        // evaluate arguments and push to stack
        for (int i = 1; i < args.length; i++) {
            stack.push(((Yolan) args[i]).value());
        }
        return ((Yolan) args[0]).value();
    }

    case FN_USER_DEFINED_FUNCTION: {
        Object args[] = (Object[]) c;

        // swap argument values on stack with local values
        int spos = stack.size();
        for (int i = 1; i < args.length; i++) {
            int pos = ((Integer) args[args.length - i]).intValue();
            spos--;
            Object t = stack.elementAt(spos);
            stack.setElementAt(vars[pos], spos);
            vars[pos] = t;
        }

        // evaluate the result
        Object result = ((Yolan) args[0]).value();

        // restore previous values
        for (int i = args.length - 1; i > 0; i--) {
            vars[((Integer) args[i]).intValue()] = stack.pop();
        }
        return result;

    }

    case FN_GET_VAR: {
        int id = ((Integer) c).intValue();
        Object x = vars[id];
        return x;
    }

    case FN_SET: {
        Object o = val(1);
        vars[((Integer) ((Object[]) c)[0]).intValue()] = o;
        return o;
    }

    case FN_NATIVE: {
        Object objs[] = (Object[]) c;
        return ((Function) objs[0]).apply((Yolan[]) objs[1]);

    }

    case FN_FOREACH: {
        Object os[] = (Object[]) c;
        int id = ((Integer) os[0]).intValue();
        Enumeration e = (Enumeration) val(1);
        Object result = null;
        stack.push(vars[id]);
        while (e.hasMoreElements()) {
            vars[id] = e.nextElement();
            result = doEm(2);
```

```
        }
        vars[id] = stack.pop();
        return result;
    }

    case FN_LOCALS: {
        int ids[] = (int[]) ((Object[]) c)[0];
        for (int i = 0; i < ids.length; i++) {
            stack.push(vars[ids[i]]);
        }
        Object result = doEm(1);
        for (int i = ids.length - 1; i >= 0; i--) {
            vars[ids[i]] = stack.pop();
        }
        return result;
    }

    case FN_RESOLVE_SET: {
        Object t[] = (Object[]) c;
        t[0] = new Integer(resolveVar((String) ((Yolan) t[0]).c));
        fn = FN_SET;
        return value();
    }

    case FN_RESOLVE_LOCALS: {
        Object[] locals = (Object[]) ((Yolan) ((Object[]) c)[0]).c;
        int locals_id[] = new int[locals.length];
        for (int i = 0; i < locals.length; i++) {
            locals_id[i] = resolveVar((String) ((Yolan) locals[i]).c);
        }
        ((Object[]) c)[0] = locals_id;
        fn = FN_LOCALS;
        return value();

    }

    case FN_IF: {
        return (val0() != null) ? val(1) : val(2);
    }

    case FN_NOT: {
        return val0() == null ? TRUE : null;
    }

    case FN_AND: {
        return val0() == null ? null : val(1);
    }

    case FN_OR: {
        Object o = val0();
        return o == null ? val(1) : o;
    }

    case FN_REPEAT: {
        int count = ival(0);
        Object result = null;
        for (int i = 0; i < count; i++) {
            result = doEm(1);
        }
        return result;
    }

    case FN_RESOLVE_FOREACH: {
        Object t[] = (Object[]) c;
        t[0] = new Integer(resolveVar((String) ((Yolan) t[0]).c));
        fn = FN_FOREACH;
```

```
        return value();
    }

    case FN_WHILE: {
        Object result = null;
        while (val0() != null) {
            result = doEm(1);
        }
        return result;
    }

    case FN_LAMBDA: {
        Object[] lambda_expr = (Object[]) c;
        Object[] arguments = (Object[]) ((Yolan) lambda_expr[0]).c;

        Object[] function = new Object[arguments.length + 1];
        function[0] = lambda_expr[1];
        for (int i = 0; i < arguments.length; i++) {
            function[i + 1] = num(resolveVar((String)
                                  ((Yolan) arguments[i]).c));
        }

        return new Yolan(FN_USER_DEFINED_FUNCTION, function);
    }

    case FN_DEFUN: {
        Object[] lambda_expr = (Object[]) c;
        Object[] arguments = (Object[]) ((Yolan) lambda_expr[0]).c;

        Object[] function = new Object[arguments.length];
        function[0] = lambda_expr[1];
        for (int i = 1; i < arguments.length; i++) {
            function[i] = num(resolveVar((String)
                                  ((Yolan) arguments[i]).c));
        }

        Yolan fnc = new Yolan(FN_USER_DEFINED_FUNCTION, function);
        vars[resolveVar((String) (((Yolan) arguments[0]).c))] = fnc;
        return fnc;
    }

    case FN_APPLY_FUNCTION: {
        Object args[] = (Object[]) c;
        // evaluate arguments and push to stack
        for (int i = 1; i < args.length; i++) {
            stack.push(((Yolan) args[i]).value());
        }
        return ((Yolan)val(0)).value();
    }


    case FN_DO: {
        return doEm(0);
    }

    case FN_ADD: {
        return num(ival(0) + ival(1));
    }

    case FN_SUB: {
        return num(ival(0) - ival(1));
    }

    case FN_MUL: {
        return num(ival(0) * ival(1));
    }
```

```
case FN_DIV: {
    return num(ival(0) / ival(1));
}

case FN_REM: {
    return num(ival(0) % ival(1));
}

case FN_LESS: {
    return ival(0) < ival(1) ? TRUE : null;
}

case FN_LESS_EQUAL: {
    return ival(0) <= ival(1) ? TRUE : null;
}

case FN_IS_INTEGER: {
    return val0() instanceof Integer ? TRUE : null;
}

case FN_IS_STRING: {
    return val0() instanceof String ? TRUE : null;
}

case FN_IS_LIST: {
    return val0() instanceof Stack ? TRUE : null;
}

case FN_IS_DICT: {
    return val0() instanceof Hashtable ? TRUE : null;
}

case FN_EQUALS: {
    Object o = val0();

    return o != null && o.equals(val(1)) ? TRUE : null;
}

case FN_IS_EMPTY: {
    Object o = val0();
    return (o instanceof Stack
                ? (((Stack) o).empty())
            : o instanceof Hashtable
                ? (((Hashtable) o).isEmpty())
            : o instanceof Enumeration
                ? (!((Enumeration) o).hasMoreElements())
            : false)
            ? TRUE : null;
}

case FN_PUT: {
    Object o = val0();
    if (o instanceof Stack) {
        ((Stack) o).setElementAt(val(2), ival(1));
    } else if (o instanceof Hashtable) {
        ((Hashtable) o).put(val(1), val(2));
    }
    return null;
}

case FN_GET: {
    Object o = val0();
    if (o instanceof Stack) {
        return ((Stack) o).elementAt(ival(1));
    } else if (o instanceof Hashtable) {
```

```
            return ((Hashtable) o).get(val(1));
        } else {
            return null;
        }
    }

    case FN_RANDOM: {
        Object o = val0();
        int rnd = random.nextInt() & 0x7fffffff;
        if (o instanceof Integer) {
            return num(rnd % ((Integer) o).intValue());
        } else if (o instanceof Stack) {
            Stack s = (Stack) o;
            return s.elementAt(rnd % s.size());
        } else {
            return null;
        }

    }

    case FN_SIZE: {
        Object o = val0();
        if (o instanceof String) {
            return num(((String) o).length());
        } else if (o instanceof Stack) {
            return num(((Stack) o).size());
        } else if (o instanceof Hashtable) {
            return num(((Hashtable) o).size());
        } else {
            return null;
        }
    }

    case FN_STRINGJOIN: {
        Object os[] = (Object[]) c;
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < os.length; i++) {
            stringjoin(sb, ((Yolan) os[i]).value());
        }
        return sb.toString();
    }

    case FN_SUBSTRING: {
        return ((String) val0()).substring(ival(1), ival(2));
    }

    case FN_STRINGORDER: {
        return ((String) val0()).compareTo((String) val(1)) <= 0
                ? TRUE : null;
    }

    case FN_LIST: {
        int len = ((Object[]) c).length;
        Stack result = new Stack();
        for (int i = 0; i < len; i++) {
            result.push(val(i));
        }
        result.trimToSize();
        return result;
    }

    case FN_RESIZE: {
        Stack s = (Stack) val0();
        s.setSize(ival(1));
        s.trimToSize();
        return s;
```

```
            }

            case FN_PUSH: {
                Stack s = (Stack) val0();
                s.push(val(1));
                return s;
            }

            case FN_POP: {
                return ((Stack) val0()).pop();
            }

            case FN_DICT: {
                int len = ((Object[]) c).length;
                Hashtable h = new Hashtable();
                for (int i = 0; i < len; i += 2) {
                    h.put(val(i), val(i + 1));
                }
                return h;
            }

            case FN_KEYS: {
                return ((Hashtable) val0()).keys();
            }

            case FN_VALUES: {
                Object o = val0();
                if (o instanceof Stack) {
                    return ((Stack) o).elements();
                } else if (o instanceof Hashtable) {
                    return ((Hashtable) o).elements();
                } else {
                    return null;
                }
            }

            case FN_GET_NEXT: {
                return ((Enumeration) val0()).nextElement();
            }

            case FN_DEBUG_STRING: {
                return to_string(new StringBuffer(), val0()).toString();
            }

            default: {
                throw new Error("Unexpected case " + fn);
            }
        }
    }

/*
```

C.3.4 Utility functions for evaluation

```
*/

    /**
     * Recursive implementation of the stringjoin builtin function.
     * @param sb string accumulator
     * @param o if o is a Stack, join its elements,
     *          else just append o as a string.
     */
    private static void stringjoin(StringBuffer sb, Object o) {
        if (o instanceof Stack) {
            for (int i = 0; i < ((Stack) o).size(); i++) {
                stringjoin(sb, ((Stack) o).elementAt(i));
            }
```

```
    } else {
        sb.append(o.toString());
    }
}

/**
 * utility function for function application
 */
private Object doApply(int n) {
    if (n != nargs()) {
        for (int i = 0; i < n; i++) {
            stack.pop();
        }
        return null;
    }
    return value();
}

/**
 * Assume the closure is a list, and evaluate
 * the last elements of it. This is used
 * for the implementation of body of
 * do, while, etc.
 * @param first the first element in the closure list to evaluate
 */
private Object doEm(int first) {
    Object result = null;
    int stmts = ((Object[]) c).length;
    while (first < stmts) {
        result = val(first);
        first++;
    }
    return result;
}

/**
 * Evaluate yolan list closure into an integer
 * @param i the index into the list
 * @return the integer result of evaluating the i'th element of c
 */
private int ival(int i) {
    return ((Integer) ((Yolan) ((Object[]) c)[i]).value()).intValue();
}

/**
 * Evaluate yolan list closure into an object
 * @param i the index into the list
 * @return the result of evaluating the i'th element of c
 */
private Object val(int i) {
    return ((Yolan) ((Object[]) c)[i]).value();
}

/**
 * Evaluate yolan list closure into an object
 * @return the result of evaluating the i'th element of c
 */
private Object val0() {
    return ((Yolan) ((Object[]) c)[0]).value();
}

/**
 * Shorthand for wrapping a number into an Integer object
 * @param i the number
 * @return the Integer
 */
```

```
    private static Object num(int i) {
        return new Integer(i);
    }

    /**
     * Convert an object to a string
     * @param sb the accumulator variable - the string i added to this buffer
     * @param o the object to convert
     * @return the accumulator (same as the paremeter)
     */
    private static StringBuffer to_string(StringBuffer sb, Object o) {
        if (o instanceof Object[]) {
            Object os[] = (Object[]) o;
            sb.append("[ ");
            for (int i = 0; i < os.length; i++) {
                to_string(sb, os[i]);
                sb.append(" ");
            }
            sb.append("]");
        } else if (o instanceof Yolan) {
            Yolan yl = (Yolan) o;
            sb.append("Yolan" + yl.fn + "(");
            to_string(sb, yl.c);
            sb.append(")");
        } else {
            sb.append(o.toString());
        }
        return sb;
    }

/*
```

C.4 The parser

```
*/

    /**
     * Parse the next Yolan expression from an input stream.
     * @param is the input stream to read from
     * @return a new Yolan expression
     * @throws java.io.IOException
     */
    public static Yolan readExpression(InputStream is) throws IOException {
        Stack stack = new Stack();
        int c = is.read();
        do {
            // end of file
            if (c == -1) {
                return null;

            // end of list
            } else if (c == ']') {
                Object result[];
                c = is.read();
                // find out how much of the stack
                // is a part of the terminated list.
                // null indicates a "["
                int pos = stack.search(null);
                // ] with no [ begun
                if (pos == -1) {
                    result = new Object[0];
                } else {
                    // stack search includes the null, which we want to skip
                    pos--;
                    // move the elements from the stack
                    result = new Object[pos];
                    while (pos > 0) {
```

```
                    pos--;
                    result[pos] = stack.pop();
                }
                // pop the null
                stack.pop();
            }
            // create the list obj
            stack.push(new Yolan(FN_RESOLVE_EVAL_LIST, result));


        // Whitespace
        } else if (c <= ' ') {
            c = is.read();

        // Comment
        } else if (c == ';') {
            do {
                c = is.read();
            } while (c > '\n');

        // List
        } else if (c == '[') {
            // null is a marker of "["
            stack.push(null);
            c = is.read();

        // Number
        } else if ('0' <= c && c <= '9') {
            int i = 0;
            do {
                i = i * 10 + c - '0';
                c = is.read();
            } while ('0' <= c && c <= '9');
            stack.push(new Yolan(FN_LITERAL, new Integer(i)));

        // String
        } else if (c == '"') { // (comment ends '"' when prettyprinting)

            StringBuffer sb = new StringBuffer();
            c = is.read();
            while (c != '"' && c != -1) { // (comment ends '"' when prettyprinting)

                if (c == '\\') {
                    c = is.read();
                }
                sb.append((char) c);
                c = is.read();
            }
            c = is.read();
            stack.push(new Yolan(FN_LITERAL, sb.toString()));

        // Identifier
        } else {
            StringBuffer sb = new StringBuffer();
            while (c > ' ' && c != '[' && c != ']') {
                sb.append((char) c);
                c = is.read();
            }
            stack.push(new Yolan(FN_RESOLVE_GET_VAR, sb.toString()));
        }
    } while (stack.empty() || stack.size() > 1 || stack.elementAt(0) == null);
    return (Yolan) stack.pop();
}

/**
 * Convenience function that reads and evaluates expressions from
```

```
     * an input stream until end of file, or error occurs
     * @param is
     * @throws java.io.IOException
     */
    public static void eval(InputStream is) throws IOException {
        Yolan yl;
        while ((yl = Yolan.readExpression(is)) != null) {
            yl.value();
        }
    }
}
```

Appendix D

Source code of the LightScript class

The following is the source code for the `LightScript.class` of version 1.1.509.

```
import java.io.InputStream;
import java.util.Enumeration;
import java.io.ByteArrayInputStream;
import java.util.Hashtable;
import java.util.Stack;

/*
```

D.1 Definitions, API, and utility functions

```
*/

/* If debugging is enabled, more tests are run during run-time,
 * and errors may be caught in a more readable way.
 * It also adds support for more readable printing of
 * id, etc.
 */
//#define __DEBUG__

/* If enabled, wipe the stack on function exit,
 * to kill dangling pointers on execution stack,
 * for better GC at performance price
 */
#define __CLEAR_STACK__

/* Identifiers, used both as node type,
 * and also used as opcode.
 */
#define ID_NONE 127
#define ID_TRUE 0
#define ID_FALSE 1
#define ID_UNDEFINED 2
#define ID_NULL 3
#define ID_PAREN 4
#define ID_LIST_LITERAL 5
#define ID_CURLY 6
#define ID_VAR 7
#define ID_BUILD_FUNCTION 8
#define ID_IF 9
#define ID_WHILE 10
#define ID_CALL_FUNCTION 11
```

```
#define ID_AND 12
#define ID_OR 13
#define ID_ELSE 14
#define ID_SET 15
#define ID_IDENT 16
#define ID_BLOCK 17
#define ID_SEP 18
#define ID_IN 19
#define ID_FOR 20
#define ID_END 21
#define ID_CATCH 22
#define ID_DO 23
#define ID_INC 24
#define ID_DEC 25
#define ID_ADD 26
#define ID_EQUALS 27
#define ID_LESS 29
#define ID_LESS_EQUALS 30
#define ID_LITERAL 31
#define ID_MUL 32
#define ID_NEG 33
#define ID_NOT 34
#define ID_NOT_EQUALS 35
#define ID_REM 37
#define ID_RETURN 38
#define ID_SHIFT_RIGHT 39
#define ID_SUB 40
#define ID_SUBSCRIPT 41
#define ID_THIS 42
#define ID_THROW 43
#define ID_TRY 44
#define ID_UNTRY 45
#define ID_BOX_IT 46
#define ID_BUILD_FN 47
#define ID_CALL_FN 48
#define ID_DROP 49
#define ID_DUP 50
#define ID_GET_BOXED 52
#define ID_GET_BOXED_CLOSURE 53
#define ID_GET_CLOSURE 54
#define ID_GET_LOCAL 55
#define ID_INC_SP 56
#define ID_JUMP 57
#define ID_JUMP_IF_FALSE 58
#define ID_JUMP_IF_TRUE 59
#define ID_NEW_DICT 60
#define ID_NEW_LIST 61
#define ID_NEXT 62
#define ID_POP 63
#define ID_PUSH 64
#define ID_PUT 65
#define ID_SAVE_PC 66
#define ID_SET_BOXED 67
#define ID_SET_CLOSURE 68
#define ID_SET_LOCAL 69
#define ID_SET_THIS 70
#define ID_SWAP 71

#define ID_DIV 72


/* The function id for the null denominator functions */
#define NUD_NONE 13
#define NUD_IDENT 1
#define NUD_LITERAL 2
#define NUD_END 3
```

```
#define NUD_SEP 4
#define NUD_LIST 5
#define NUD_PREFIX 6
#define NUD_PREFIX2 7
#define NUD_FUNCTION 8
#define NUD_VAR 9
#define NUD_ATOM 10
#define NUD_CATCH 11
#define NUD_CONST 12

/* The function id for the null denominator functions */
#define LED_NONE 8
#define LED_DOT 1
#define LED_INFIX 2
#define LED_INFIXR 3
#define LED_INFIX_LIST 4
#define LED_INFIX_IF 5
#define LED_OPASSIGN 6
#define LED_INFIX_SWAP 7

/* Tokens objects are encoded as integers */

/** The number of bits per denominator function */
#define SIZE_FN 4

/** The number of bits per id */
#define SIZE_ID 7

/* Masks for function/id */
#define MASK_ID ((1<<SIZE_ID) - 1)
#define MASK_FN ((1<<SIZE_FN) - 1)

/* Mask for the binding power / priority */
#define MASK_BP (-1 << (2*SIZE_ID + 2 *SIZE_FN))

/** The end token, encoded as an integer */
#define TOKEN_END (((((((\
                        0 << SIZE_FN)\
                      | NUD_END) << SIZE_ID)\
                      | ID_NONE) << SIZE_FN)\
                      | LED_NONE) << SIZE_ID)\
                      | ID_NONE)

/** The token used for literals, encoded as an integer */
#define TOKEN_LITERAL (((((((\
                        0 << SIZE_FN)\
                      | NUD_LITERAL) << SIZE_ID)\
                      | ID_NONE) << SIZE_FN)\
                      | LED_NONE) << SIZE_ID)\
                      | ID_NONE)

/** The token used for identifiers, encoded as an integer */
#define TOKEN_IDENT (((((((\
                        0 << SIZE_FN)\
                      | NUD_IDENT) << SIZE_ID)\
                      | ID_NONE) << SIZE_FN)\
                      | LED_NONE) << SIZE_ID)\
                      | ID_NONE)

/** Sizes of different kinds of stack frames */
#define RET_FRAME_SIZE 3
#define TRY_FRAME_SIZE 5
```

/*

D.1.1 Variables

```
*/
/** Instances of the LightScript object, is an execution context,
  * where code can be parsed, compiled, and executed. */
public final class LightScript implements LightScriptObject {

    /** Token used for separators (;,:), which are just discarded */
    private static final Object[] SEP_TOKEN = {new Integer(ID_SEP)};

    /** The true truth value of results
      * of tests/comparisons within LightScript */
    public static final Object TRUE = new StringBuffer("true");

    /** The false truth value of results
      * of tests/comparisons within LightScript */
    public static final Object NULL = new StringBuffer("null");
    public static final Object UNDEFINED = new StringBuffer("undefined");
    public static final Object FALSE = new StringBuffer("false");

    /** Token string when reaching end of file, it can only occur
      * at end of file, as it would otherwise be parsed as three
      * tokens: "(", "EOF", and ")". */
    private static final String EOF = "(EOF)";

    /** This stack is used during compilation to build the constant pool
      * of the compiled function. The constant pool contains the constants
      * that are used during execution of the function */
    private Stack constPool;

    /* Used to keep track of stack depth during compilation, to be able to
     * resolve variables */
    private int maxDepth;
    private int depth;

    /** This stringbuffer is used as a dynamically sized bytevector
      * where opcodes are added, during the compilation */
    private StringBuffer code;

    /** The stream which we are parsing */
    private InputStream is;

    /** the just read character */
    private int c;

    /** the buffer for building the tokens */
    private StringBuffer sb;

    /* Per function statistics
     * Sets of variable names for keeping track of which variables
     * are used where and how, in order to know whether they should
     * be boxed, and be placed on the stack or in closures. */

    /** The variables used within a function */
    private Stack varsUsed;

    /** The variables that needs to be boxed */
    private Stack varsBoxed;

    /** The local variables (arguments, and var-defined) */
    private Stack varsLocals;

    /** The variables in the closure */
    private Stack varsClosure;
```

```
    /** The number of arguments to the function, corresponds to the first
      * names in varsLocals */
    private int varsArgc;

    /** The value of the just read token.
      * used if the token is an identifier or literal.
      * possible types are String and Integer */
    private Object tokenVal;

    /** The integer encoded token object, including priority, IDs
      * and Function ids for null/left denominator functions */
    private int token;

    /** The globals variables in this execution context.
      * they are boxed, in such that they can be passed
      * to the closure of af function, which will then
      * be able to modify it without looking it up here */
    private Hashtable boxedGlobals;


/*
```

D.1.2 Public functions

```
*/

    /** Constructor, loading standard library */
    public LightScript() {
        boxedGlobals = new Hashtable();
        LightScriptStdLib.register(this);
    }

    /** Shorthand for evaluating a string that contains LightScript code */
    public void eval(String s) throws LightScriptException {
        eval(new ByteArrayInputStream(s.getBytes()));
    }

    /** Parse and execute LightScript code read from an input stream */
    public void eval( InputStream is) throws LightScriptException {
#ifndef __DEBUG__
        try {
#endif
            this.is = is;
            sb = new StringBuffer();
            c = ' ';
            varsArgc = 0;
            nextToken();
            while(tokenVal != EOF || token != TOKEN_END) {
                // parse with every var in closure
                varsUsed = varsLocals = varsBoxed = new Stack();
                varsBoxed.push("(ENV)");
                Object[] os = parse(0);
                varsClosure = varsUsed;

                // compile
                Code c = compile(os);

                // create closure from globals
                for(int i = 0; i < c.closure.length; i ++) {
                    Object box = boxedGlobals.get(c.closure[i]);
                    if(box == null) {
                        box = new Object[1];
                        boxedGlobals.put(c.closure[i], box);
                    }
                    c.closure[i] = box;
                }
                execute(c, new Object[0], 0, null);
```

```
            }
#ifndef __DEBUG__
        } catch(Error e) {
            throw new LightScriptException(e);
        }
#endif /*__DEBUG__*/
    }

    /** Set a global value for this execution context */
    public void set(Object key, Object value) {
        Object[] box = (Object[])boxedGlobals.get(key);
        if(box == null) {
            box = new Object[1];
            boxedGlobals.put(key, box);
        }
        box[0] = value;
    }

    /** Retrieve a global value from this execution context */
    public Object get(Object key) {
        Object[] box = (Object[])boxedGlobals.get(key);
        if(box == null) {
            return null;
        } else {
            return box[0];
        }

    }

    /** Prototypes for Objects */
    public Hashtable objectPrototype;

    /** Prototypes for Arrays*/
    public Hashtable arrayPrototype;

    /** Prototypes for Functions */
    public Hashtable functionPrototype;

    /** Prototypes for Strings*/
    public Hashtable stringPrototype;

    /** Default setter function, called when a property is set on
     * an object which is neither a Stack, Hashtable nor LightScriptObject
     *
     * The apply method of the setter gets the container as thisPtr,
     * and takes the key and value as arguments
     */
    public LightScriptFunction defaultSetter;

    /** Default getter function, called when subscripting an object
     * which is not a Stack, Hashtable, String nor LightScriptObject
     * or when the subscripting of any of those objects returns null.
     * (non-integer on stacks/strings, keys not found in Hashtable or
     * its prototypes, when LightScriptObject.get returns null)
     *
     * The apply method of the getter gets the container as thisPtr,
     * and takes the key as argument
     */
    public LightScriptFunction defaultGetter;

/*
```

D.1.3 Debugging

```
*/

#ifdef __DEBUG__
```

```
/** Mapping from ID to name of ID */
private static final String[] idNames = {
    "", "", "", "", "PAREN", "LIST_LITERAL", "CURLY", "VAR",
    "BUILD_FUNCTION", "IF", "WHILE", "CALL_FUNCTION", "AND",
    "OR", "ELSE", "SET", "IDENT", "BLOCK", "SEP", "IN", "FOR",
    "END", "CATCH", "DO", "INC", "DEC", "ADD", "EQUALS",
    "NOT_USED_ANYMORE", "LESS", "LESS_EQUALS", "LITERAL", "MUL", "NEG",
    "NOT", "NOT_EQUALS", "NOT_USED_ANYMORE", "REM", "RETURN", "SHIFT_RIGHT",
    "SUB", "SUBSCRIPT", "THIS", "THROW", "TRY", "UNTRY", "BOX_IT",
    "BUILD_FN", "CALL_FN", "DROP", "DUP", "NOT_USED_ANYMORE",
    "GET_BOXED", "GET_BOXED_CLOSURE", "GET_CLOSURE", "GET_LOCAL",
    "INC_SP", "JUMP", "JUMP_IF_FALSE", "JUMP_IF_TRUE", "NEW_DICT",
    "NEW_LIST", "NEXT", "POP", "PUSH", "PUT", "SAVE_PC",
    "SET_BOXED", "SET_CLOSURE", "SET_LOCAL", "SET_THIS", "SWAP",
    "DIV"
};

/** Function that maps from ID to a string representation of the ID,
  * robust for integers which is not IDs */
private static String idName(int id) {
    return "" + id + ((id > 0 && id < idNames.length) ? idNames[id] : "");
}

/** A toString, that also works nicely on arrays, and LightScript code */
private static String stringify(Object o) {
    if (o == null) {
        return "null";
    } else if (o instanceof Object[]) {
        StringBuffer sb = new StringBuffer();
        Object[] os = (Object[]) o;
        sb.append("[");
        if (os.length > 0 && os[0] instanceof Integer) {
            int id = ((Integer) os[0]).intValue();
            sb.append(idName(id));
        } else if (os.length > 0) {
            sb.append(os[0]);
        }
        for (int i = 1; i < os.length; i++) {
            sb.append(" " + stringify(os[i]));
        }
        sb.append("]");
        return sb.toString();
    } else if (o instanceof Code) {
        Code c = (Code) o;
        StringBuffer sb = new StringBuffer();
        sb.append("closure" + c.argc + "{\n\tcode:");
        for (int i = 0; i < c.code.length; i++) {
            sb.append(" ");
            sb.append(idName(c.code[i]));
        }
        sb.append("\n\tclosure:");
        for (int i = 0; i < c.closure.length; i++) {
            sb.append(" " + i + ":");
            sb.append(stringify(c.closure[i]));
        }
        sb.append("\n\tconstPool:");
        for (int i = 0; i < c.constPool.length; i++) {
            sb.append(" " + i + ":");
            sb.append(stringify(c.constPool[i]));
        }
        sb.append("\n}");
        return sb.toString();
    } else {
        return o.toString();
    }
```

```
    }
#else /* not debug */
#define idName(...) ""
    private static String stringify(Object o) {
        return o.toString();
    }
#endif

    /*
```

D.1.4 Utility functions

```
*/

    /* Constructors for nodes of the Abstract Syntax Tree.
     * Each node is an array containing an ID, followed by
     * its children or literal values */
    /** (id, o) -> (Object []) {new Integer(id), o} */
    private static Object[] v(int id, Object o) {
        Object[] result = {new Integer(id), o};
        return result;
    }

    /** (id, o1, o2) -> (Object []) {new Integer(id), o1, o2} */
    private static Object[] v(int id, Object o1, Object o2) {
        Object[] result = {new Integer(id), o1, o2};
        return result;
    }

    /** (id, o1, o2, o3) -> (Object []) {new Integer(id), o1, o2, o3} */
    private static Object[] v(int id, Object o1, Object o2, Object o3) {
        Object[] result = {new Integer(id), o1, o2, o3};
        return result;
    }

    /** Returns a new object array, with the seperator tokens removed */
    private Object[] stripSep(Object[] os) {
        Stack s = new Stack();
        for(int i = 0; i < os.length; i++) {
            if(os[i] != SEP_TOKEN) {
                s.push(os[i]);
            }
        }
        os = new Object[s.size()];
        s.copyInto(os);
        return os;
    }

    /** Push a value into a stack if it is not already there */
    private static void stackAdd(Stack s, Object val) {
        if(s == null) {
            return;
        }
        int pos = s.indexOf(val);
        if (pos == -1) {
            pos = s.size();
            s.push(val);
        }
    }

/*
```

D.1.5 Utility classes

```
*/

    /**
     * Analysis of variables in a function being compiled,
```

```
     * updated during the parsing.
     */
    private class Code implements LightScriptFunction {
        public Object apply(Object thisPtr, Object[] args,
                            int argpos, int argcount)
                                throws LightScriptException {
#ifdef __DEBUG__
            if(argcount == argc) {
#endif
                Object stack[];
                if(argpos != 0) {
                    stack = new Object[argcount];
                    for(int i = 0; i<argcount; i++) {
                        stack[i] = args[argpos + i];
                    }
                } else {
                    stack = args;
                }
                return execute(this, stack, argcount, thisPtr);
#ifdef __DEBUG__
            }
            else {
                throw new LightScriptException("Wrong number of arguments");
            }
#endif
        }
        public int argc;
        public byte[] code;
        public Object[] constPool;
        public Object[] closure;
        public int maxDepth;

        public int getArgc() {
            return argc;
        }

        public Code(int argc, byte[] code, Object[] constPool, Object[] closure, int maxDepth) {
            this.argc = argc;
            this.code = code;
            this.constPool = constPool;
            this.closure = closure;
            this.maxDepth = maxDepth;
        }

        public Code(Code cl) {
            this.argc = cl.argc;
            this.code = cl.code;
            this.constPool = cl.constPool;
            this.maxDepth = cl.maxDepth;
        }
    }

/*
```

D.2 Tokeniser

```
\
index
{
Tokeniser
}
*/

    /** Read the next character from the input stream */
    private void nextc() {
        try {
            c = is.read();
```

```
    } catch (Exception e) {
        c = -1;
    }
}

/** Append a character to the token buffer */
private void pushc() {
    sb.append((char) c);
    nextc();
}

/** Test if the current character is a number */
private boolean isNum() {
    return '0' <= c && c <= '9';
}

/** Test if the current character is alphanumeric */
private boolean isAlphaNum() {
    return isNum() || c == '_' || ('a' <= c && c <= 'z')
                               || ('A' <= c && c <= 'Z');
}

/** Test if the current character could be a part of a multi character
  * symbol, such as &&, ||, += and the like. */
private boolean isSymb() {
    return c == '=' || c == '!' || c == '<' || c == '&' || c == '/' || c == '*'
        || c == '%' || c == '|' || c == '+' || c == '-' || c == '>';
}

/** Read the next toke from the input stream.
  * The token is stored in the token and tokenVal property. */
private void nextToken() {
    sb.setLength(0);

    // skip whitespaces
    while (c == ' ' || c == '\n' || c == '\t' || c == '\r' || c == '/') {
        // comments
        if (c == '/') {
            nextc();
            if (c == '/') {
                while (c != '\n' && c != -1) {
                    nextc();
                }
            } else {
                resolveToken("/");
                return;
            }
        }
        nextc();
    }

    // End of file
    if (c == -1) {
        token = TOKEN_END;
        tokenVal = EOF;
        return;

    // String
    } else if (c == '"') {
        nextc();
        while (c != -1 && c != '"') {
            if (c == '\\') {
                nextc();
                if (c == 'n') {
                    c = '\n';
                }
```

```
                }
                pushc();
            }
            nextc();
            token = TOKEN_LITERAL;
            tokenVal = sb.toString();
            return;

        // Number
        } else if (isNum()) {
            do {
                pushc();
            } while (isNum());
            token = TOKEN_LITERAL;
            tokenVal = Integer.valueOf(sb.toString());
            return;

        // Identifier
        } else if (isAlphaNum()) {
            do {
                pushc();
            } while (isAlphaNum());

        // FixedPoint symbol !== , ===, <= , &&, ...
        } else if (isSymb()) {
            do {
                pushc();
            } while (isSymb());

        // Single symbol
        } else {
            pushc();
        }
        resolveToken(sb.toString());
        return;
    }

/*
```

D.3 Parser

```
*/


    /** Parse the next expression from the input stream
      * @param rbp right binding power
      */
    private Object[] parse(int rbp) {
        Object[] left = nud(token);

        // token & MASK_BP extract the binding power/priority of the token
        while (rbp < (token & MASK_BP)) {
            left = led(token, left);
        }

        return left;
    }

    /** Read expressions until an end-token is reached.
      * @param s an accumulator stack where the expressions are appended
      * @returns an array of parsed expressions, with s prepended
      */
    private Object[] readList(Stack s) {
        while (token != TOKEN_END) {
            Object[] p = parse(0);
            s.push(p);
        }
```

```
        nextToken();

        Object[] result = new Object[s.size()];
        s.copyInto(result);
        return result;
    }

    /** Call the null denominator function for a given token
      * and also read the next token. */
    private Object[] nud(int tok) {
        Object val = tokenVal;
        nextToken();
        int nudId = (tok >> (SIZE_ID + SIZE_FN)) & ((1 << SIZE_ID) - 1);
        // extract the token function id from tok
        switch ((tok >> (SIZE_ID * 2 + SIZE_FN)) & ((1 << SIZE_FN) - 1)) {
            case NUD_IDENT:
                stackAdd(varsUsed, val);
                return v(ID_IDENT, val);
            case NUD_LITERAL:
                return v(ID_LITERAL, val);
            case NUD_CONST:
                return v(ID_LITERAL,
                            nudId == ID_TRUE ? TRUE
                          : nudId == ID_FALSE ? FALSE
                          : nudId == ID_NULL ? NULL
                          : UNDEFINED
                );
            case NUD_END:
                return null;// result does not matter,
                            // as this is removed during parsing
            case NUD_SEP:
                return SEP_TOKEN;
            case NUD_LIST: {
                Stack s = new Stack();
                s.push(new Integer(nudId));
                return readList(s);
            }
            case NUD_ATOM: {
                Object[] result = { new Integer(nudId) };
                return result;
            }
            case NUD_PREFIX:
                return v(nudId, parse(0));
            case NUD_PREFIX2:
                return v(nudId, parse(0), parse(0));
            case NUD_CATCH: {
                Object[] o = parse(0);
                stackAdd(varsLocals, ((Object[])o[1])[1]);
                return v(nudId, o, parse(0));
            }
            case NUD_FUNCTION: {
                // The functio nud is a bit more complex than
                // the others because variable-use-analysis is done
                // during the parsing.

                // save statistics for previous function
                Stack prevUsed = varsUsed;
                Stack prevBoxed = varsBoxed;
                Stack prevLocals = varsLocals;
                int prevArgc = varsArgc;

                // create new statistics
                varsUsed = new Stack();
                varsBoxed = new Stack();
                varsBoxed.push("(ENV)");
                varsLocals = new Stack();
```

```
                // parse arguments
                Object[] args = stripSep(parse(0));

                boolean isNamed = false;
                String fnName = null;
                // add function arguments to statistics
                varsArgc = args.length - 1;
                if (((Integer) args[0]).intValue() == ID_PAREN) {
                    for (int i = 1; i < args.length; i++) {
                        Object[] os = (Object[]) args[i];
#ifdef __DEBUG__
                        if (((Integer) os[0]).intValue() != ID_IDENT) {
                            throw new Error("parameter not variable name"
                                            + stringify(args));
                        }
#endif
                        varsLocals.push(os[1]);
                    }
                } else {
#ifdef __DEBUG__
                    if (((Integer) args[0]).intValue() != ID_CALL_FUNCTION) {
                        throw new Error("parameter not variable name"
                                        + stringify(args));
                    }
#endif
                    varsArgc--;
                    fnName = (String)varsUsed.elementAt(0);
                    varsUsed.removeElementAt(0);
                    isNamed = true;
                    for(int i = 0; i < varsUsed.size(); i++) {
                        varsLocals.push(varsUsed.elementAt(i));
                    }

                }

                // parse the body of the function
                // notice that this may update vars{Used|Boxed|Locals}
                Object[] body = parse(0);

                // non-local variables are boxed into the closure
                for (int i = 0; i < varsUsed.size(); i++) {
                    Object o = varsUsed.elementAt(i);
                    if (!varsLocals.contains(o)) {
                        stackAdd(varsBoxed, o);
                    }
                }

                //  find the variables in the closure
                // and add that they need to be boxed at parent.
                varsClosure = new Stack();
                for (int i = 0; i < varsBoxed.size(); i++) {
                    Object o = varsBoxed.elementAt(i);
                    if (!varsLocals.contains(o)) {
                        stackAdd(prevBoxed, o);
                        stackAdd(varsClosure, o);
                    }
                }
                Object[] result = v(nudId, compile(body));
                if(isNamed) {
                    result = v(ID_SET, v(ID_IDENT, fnName), result);
                    stackAdd(prevUsed, fnName);
                }
                varsClosure = null;

                // restore variable statistics
```

```
                // notice that varsClosure is not needed,
                // as it is calculated before the compile,
                // and not updated/used other places
                varsUsed = prevUsed;
                varsBoxed = prevBoxed;
                varsLocals = prevLocals;
                varsArgc = prevArgc;
                return result;
            }
            case NUD_VAR:
                Object[] expr = parse(0);
                int type = ((Integer) expr[0]).intValue();
                if (type == ID_IDENT) {
                    stackAdd(varsLocals, expr[1]);
                } else {
                    Object[] expr2 = (Object[]) expr[1];
#ifdef __DEBUG__
                    if (type == ID_SET
                    && ((Integer) expr2[0]).intValue() == ID_IDENT) {
#endif
                        stackAdd(varsLocals, expr2[1]);
#ifdef __DEBUG__
                    }
                    else {
                        throw new Error("Error in var");
                    }
#endif
                }
                return v(nudId, expr);
            default:
#ifdef __DEBUG__
                throw new Error("Unknown token: " + token+ ", val: " + val);
#else
                return null;
#endif
        }
    }

    /** Call the left denominator function for a given token
      * and also read the next token. */
    private Object[] led(int tok, Object left) {
        nextToken();
        int bp = tok & MASK_BP;
        int ledId = tok & ((1 << SIZE_ID) - 1);
        // extract led function id from token
        switch ((tok >> SIZE_ID) & ((1 << SIZE_FN) - 1)) {
            case LED_INFIX:
                return v(ledId, left, parse(bp));
            case LED_INFIX_SWAP:
                return v(ledId, parse(bp), left);
            case LED_OPASSIGN:
                return v(ID_SET, left, v(ledId, left, parse(bp - 1)));
            case LED_INFIXR:
                return v(ledId, left, parse(bp - 1));
            case LED_INFIX_LIST: {
                Stack s = new Stack();
                s.push(new Integer(ledId));
                s.push(left);
                return readList(s);
            }
            case LED_DOT: {
                Stack t = varsUsed;
                varsUsed= null;
                Object[] right = parse(bp);
                varsUsed = t;
#ifdef __DEBUG__
```

```
                if(((Integer)right[0]).intValue() != ID_IDENT) {
                    throw new Error("right side of dot not a string: "
                                    + stringify(right));
                }
#endif

                right[0] = new Integer(ID_LITERAL);
                return v(ID_SUBSCRIPT, left, right);
            }
            case LED_INFIX_IF: {
                Object branch1 = parse(0);
#ifdef __DEBUG__
                if(parse(0) != SEP_TOKEN) {
                    throw new Error("infix if error");
                }
#else
                parse(0);
#endif
                Object branch2 = parse(0);
                return v(ID_IF, left, v(ID_ELSE, branch1, branch2));
            }
            default:
#ifdef __DEBUG__
                throw new Error("Unknown led token: " + token);
#else
                return null;
#endif
        }
    }
    private static Hashtable idMapping;

    // initialise idMappings
    static {
        /** This string is used for initialising the idMapping.
          * Each token type has five properties:
          * First there is the string of the token,
          *   then ther is the bindingpower,
          *   followed by the null denominator function,
          *     and its corresponding id
          *   and finally the left denominator function,
          *     and its corresponding id
          */
        String identifiers = ""
        + "(EOF)"
            + (char) 1
            + (char) NUD_END
            + (char) ID_NONE
            + (char) LED_NONE
            + (char) ID_NONE
        + "]"
            + (char) 1
            + (char) NUD_END
            + (char) ID_NONE
            + (char) LED_NONE
            + (char) ID_NONE
        + ")"
            + (char) 1
            + (char) NUD_END
            + (char) ID_NONE
            + (char) LED_NONE
            + (char) ID_NONE
        + "}"
            + (char) 1
            + (char) NUD_END
            + (char) ID_NONE
            + (char) LED_NONE
```

```
    + (char) ID_NONE
+ ". "
    + (char) 8
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_DOT
    + (char) ID_SUBSCRIPT
+ "("
    + (char) 7
    + (char) NUD_LIST
    + (char) ID_PAREN
    + (char) LED_INFIX_LIST
    + (char) ID_CALL_FUNCTION
+ "["
    + (char) 7
    + (char) NUD_LIST
    + (char) ID_LIST_LITERAL
    + (char) LED_INFIX_LIST
    + (char) ID_SUBSCRIPT
+ ">>"
    + (char) 6
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_SHIFT_RIGHT
+ "/"
    + (char) 6
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_DIV
+ "*"
    + (char) 6
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_MUL
+ "%"
    + (char) 6
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_REM
+ "+"
    + (char) 5
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_ADD
+ "-"
    + (char) 5
    + (char) NUD_PREFIX
    + (char) ID_NEG
    + (char) LED_INFIX
    + (char) ID_SUB
+ "=="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_EQUALS
+ "==="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
```

```
    + (char) ID_EQUALS
+ "!="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_NOT_EQUALS
+ "!=="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_NOT_EQUALS
+ "<="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_LESS_EQUALS
+ "<"
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX
    + (char) ID_LESS
+ ">="
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX_SWAP
    + (char) ID_LESS_EQUALS
+ ">"
    + (char) 4
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX_SWAP
    + (char) ID_LESS
+ "&&"
    + (char) 3
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIXR
    + (char) ID_AND
+ "||"
    + (char) 3
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIXR
    + (char) ID_OR
+ "else"
    + (char) 3
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIXR
    + (char) ID_ELSE
+ "in"
    + (char) 3
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIXR
    + (char) ID_IN
+ "?"
    + (char) 3
    + (char) NUD_NONE
    + (char) ID_NONE
    + (char) LED_INFIX_IF
```

```
      + (char) ID_NONE
+ "="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_INFIXR
      + (char) ID_SET
+ "+="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_OPASSIGN
      + (char) ID_ADD
+ "-="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_OPASSIGN
      + (char) ID_SUB
+ "*="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_OPASSIGN
      + (char) ID_MUL
+ "/="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_OPASSIGN
      + (char) ID_DIV
+ "%="
      + (char) 2
      + (char) NUD_NONE
      + (char) ID_NONE
      + (char) LED_OPASSIGN
      + (char) ID_REM
+ "++"
      + (char) 1
      + (char) NUD_PREFIX
      + (char) ID_INC
      + (char) LED_NONE
      + (char) ID_NONE
+ "--"
      + (char) 1
      + (char) NUD_PREFIX
      + (char) ID_DEC
      + (char) LED_NONE
      + (char) ID_NONE
+ ":"
      + (char) 1
      + (char) NUD_SEP
      + (char) ID_NONE
      + (char) LED_NONE
      + (char) ID_NONE
+ ";"
      + (char) 1
      + (char) NUD_SEP
      + (char) ID_NONE
      + (char) LED_NONE
      + (char) ID_NONE
+ ","
      + (char) 1
      + (char) NUD_SEP
      + (char) ID_NONE
      + (char) LED_NONE
```

```
    + (char) ID_NONE
+ "{"
    + (char) 1
    + (char) NUD_LIST
    + (char) ID_CURLY
    + (char) LED_NONE
    + (char) ID_NONE
+ "var"
    + (char) 1
    + (char) NUD_VAR
    + (char) ID_VAR
    + (char) LED_NONE
    + (char) ID_NONE
+ "return"
    + (char) 1
    + (char) NUD_PREFIX
    + (char) ID_RETURN
    + (char) LED_NONE
    + (char) ID_NONE
+ "!"
    + (char) 1
    + (char) NUD_PREFIX
    + (char) ID_NOT
    + (char) LED_NONE
    + (char) ID_NONE
+ "throw"
    + (char) 1
    + (char) NUD_PREFIX
    + (char) ID_THROW
    + (char) LED_NONE
    + (char) ID_NONE
+ "try"
    + (char) 1
    + (char) NUD_PREFIX2
    + (char) ID_TRY
    + (char) LED_NONE
    + (char) ID_NONE
+ "catch"
    + (char) 1
    + (char) NUD_CATCH
    + (char) ID_CATCH
    + (char) LED_NONE
    + (char) ID_NONE
+ "function"
    + (char) 1
    + (char) NUD_FUNCTION
    + (char) ID_BUILD_FUNCTION
    + (char) LED_NONE
    + (char) ID_NONE
+ "do"
    + (char) 1
    + (char) NUD_PREFIX2
    + (char) ID_DO
    + (char) LED_NONE
    + (char) ID_NONE
+ "for"
    + (char) 1
    + (char) NUD_PREFIX2
    + (char) ID_FOR
    + (char) LED_NONE
    + (char) ID_NONE
+ "if"
    + (char) 1
    + (char) NUD_PREFIX2
    + (char) ID_IF
    + (char) LED_NONE
```

```
            + (char) ID_NONE
        + "while"
            + (char) 1
            + (char) NUD_PREFIX2
            + (char) ID_WHILE
            + (char) LED_NONE
            + (char) ID_NONE
        + "undefined"
            + (char) 1
            + (char) NUD_CONST
            + (char) ID_UNDEFINED
            + (char) LED_NONE
            + (char) ID_NONE
        + "null"
            + (char) 1
            + (char) NUD_CONST
            + (char) ID_NULL
            + (char) LED_NONE
            + (char) ID_NONE
        + "false"
            + (char) 1
            + (char) NUD_CONST
            + (char) ID_FALSE
            + (char) LED_NONE
            + (char) ID_NONE
        + "this"
            + (char) 1
            + (char) NUD_ATOM
            + (char) ID_THIS
            + (char) LED_NONE
            + (char) ID_NONE
        + "true"
            + (char) 1
            + (char) NUD_CONST
            + (char) ID_TRUE
            + (char) LED_NONE
            + (char) ID_NONE
        ;
        idMapping = new Hashtable();
        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < identifiers.length(); i++) {
            int result = identifiers.charAt(i);
            // result is the binding power
            // next in string is encoded nud/led-function/id object
            if(result < 32) {
                // read nud function
                result = (result << SIZE_FN) + identifiers.charAt(++i);
                // read nud identifier
                result = (result << SIZE_ID) + identifiers.charAt(++i);
                // read led function
                result = (result << SIZE_FN) + identifiers.charAt(++i);
                // read led identifier
                result = (result << SIZE_ID) + identifiers.charAt(++i);
                // save to mapping, and start next string.
                idMapping.put(sb.toString(), new Integer(result));
                sb.setLength(0);

            // result is a char to be addded to the string
            } else {
                sb.append((char)result);
            }
        }
    }

    private void resolveToken(Object val) {
        tokenVal = val;
```

```
        Object o = idMapping.get(val);
        if(o == null) {
            token = TOKEN_IDENT;
        } else {
            token = ((Integer)o).intValue() + MASK_BP;
        }
    }

/*
```

D.4 Compiler

```
*/

    private void pushShort(int i) {
        emit(((i >> 8) & 0xff));
        emit((i & 0xff));
    }

    private void setShort(int pos, int i) {
        code.setCharAt(pos - 2, (char) ((i >> 8) & 0xff));
        code.setCharAt(pos - 1, (char) (i & 0xff));
    }

    private void addDepth(int i) {
        depth += i;
        if (depth > maxDepth) {
            maxDepth = depth;
        }
    }

#ifdef __DEBUG__
    private void assertLength(Object[] list, int len) {
        if (list.length != len) {
            throw new Error("Wrong number of parameters:" + stringify(list));
        }
    }
#else
#define assertLength(...)
#endif

    private void emit(int opcode) {
        code.append((char) opcode);
    }

    private int constPoolId(Object o) {
        int pos = constPool.indexOf(o);
        if (pos < 0) {
            pos = constPool.size();
            constPool.push(o);
        }
        return pos;
    }

    private void curlyToBlock(Object oexpr) {
        Object[] expr = (Object[]) oexpr;
        if (((Integer) expr[0]).intValue() == ID_CURLY) {
            expr[0] = new Integer(ID_BLOCK);
        }
    }

    private Code compile(Object[] body) {
        constPool = new Stack();
        code = new StringBuffer();

        // allocate space for local vars
```

```
        maxDepth = depth = varsLocals.size();
        int framesize = depth - varsArgc;
        while (framesize >= 127) {
            emit(ID_INC_SP);
            emit(127);
            framesize -= 127;
        }
        if (framesize > 0) {
            emit(ID_INC_SP);
            emit(framesize);
        }

        // box boxed values in frame
        for (int i = 0; i < varsBoxed.size(); i++) {
            int pos = varsLocals.indexOf(varsBoxed.elementAt(i));
            if (pos != -1) {
                emit(ID_BOX_IT);
                pushShort(depth - pos - 1);
            }
        }

        // compile
        curlyToBlock(body);
        compile(body, true);

        // emit return code, including current stack depth to drop
        emit(ID_RETURN);
        pushShort(depth);

        // patch amount of stack space needed
        maxDepth -= varsArgc;

        // create a new code object;
        Code result = new Code(varsArgc, new byte[code.length()],
                new Object[constPool.size()], new Object[varsClosure.size()], maxDepth);

        // copy values into the code object
        constPool.copyInto(result.constPool);
        varsClosure.copyInto(result.closure);
        for (int i = 0; i < result.code.length; i++) {
            result.code[i] = (byte) code.charAt(i);
        }

        //System.out.println(stringify(body));
        //System.out.println(varsLocals);
        //System.out.println(varsBoxed);
        //System.out.println(varsClosure);
        //System.out.println(result);
        return result;
    }

    private int childType(Object[] expr, int i) {
        return ((Integer) ((Object[]) expr[i])[0]).intValue();
    }

    /**
     * Generates code that sets the variable to the value of the
     * top of the stack, not altering the stack
     * @param name the name of the variable.
     */
    private void compileSet(Object name) {
                    int pos = varsClosure.indexOf(name);
                    if (pos >= 0) {
                        emit(ID_SET_CLOSURE);
                        pushShort(pos);
                    } else {
```

```
                    pos = varsLocals.indexOf(name);
                    if (varsBoxed.contains(name)) {
                        emit(ID_SET_BOXED);
                    } else {
                        emit(ID_SET_LOCAL);
                    }
                    pushShort(depth - pos - 1);
                }
}

private void compile(Object rawexpr, boolean yieldResult) {
    boolean hasResult;
    Object[] expr = (Object[]) rawexpr;
    int id = ((Integer) expr[0]).intValue();
    switch (id) {
        case ID_ADD:
        case ID_MUL:
        case ID_DIV:
        case ID_SHIFT_RIGHT:
        case ID_REM:
        case ID_SUB:
        case ID_EQUALS:
        case ID_NOT_EQUALS:
        case ID_SUBSCRIPT:
        case ID_LESS_EQUALS:
        case ID_LESS: {
            compile(expr[1], true);
            compile(expr[2], true);
            emit(id);
            addDepth(-1);
            hasResult = true;
            break;
        }
        case ID_NOT:
        case ID_NEG: {
            compile(expr[1], true);
            emit(id);
            hasResult = true;
            break;
        }
        case ID_THIS: {
            emit(id);
            addDepth(1);
            hasResult = true;
            break;
        }
        case ID_LITERAL: {
            emit(id);
            pushShort(constPoolId(expr[1]));
            hasResult = true;
            addDepth(1);
            break;
        }
        case ID_BLOCK: {
            for (int i = 1; i < expr.length; i++) {
                compile(expr[i], false);
            }
            hasResult = false;
            break;
        }
        case ID_RETURN: {
            compile(expr[1], true);
            emit(ID_RETURN);
            pushShort(depth);
            addDepth(-1);
            hasResult = false;
```

```
                break;
            }
            case ID_IDENT: {
                String name = (String) expr[1];
                int pos = varsClosure.indexOf(name);
                if (pos >= 0) {
                    emit(ID_GET_CLOSURE);
                    pushShort(pos);
                } else {
                    pos = varsLocals.indexOf(name);
#ifdef __DEBUG__
                    if (pos == -1) {
                        throw new Error("Unfound var: " + stringify(expr));
                    }
#endif
                    if (varsBoxed.contains(name)) {
                        emit(ID_GET_BOXED);
                    } else {
                        emit(ID_GET_LOCAL);
                    }
                    pushShort(depth - pos - 1);
                }
                addDepth(1);
                hasResult = true;
                break;
            }
            case ID_VAR: {
                int id2 = childType(expr, 1);
                if (id2 == ID_IDENT) {
                    hasResult = false;
                } else if (id2 == ID_SET) {
                    compile(expr[1], yieldResult);
                    hasResult = yieldResult;
                } else {
#ifdef __DEBUG__
                    throw new Error("Error in var statement: "
                                    + stringify(expr));
#else
                    return;
#endif
                }
                break;
            }
            case ID_SET: {
                assertLength(expr, 3);
                int targetType = childType(expr, 1);
                hasResult = true;
                if (targetType == ID_IDENT) {
                    String name = (String) ((Object[]) expr[1])[1];
                    compile(expr[2], true);
                    compileSet(name);
                } else if(targetType == ID_SUBSCRIPT) {
                    Object[] subs = (Object[]) expr[1];
                    compile(subs[1], true);
                    compile(subs[2], true);
                    compile(expr[2], true);
                    emit(ID_PUT);
                    addDepth(-2);
                }
#ifdef __DEBUG__
                else {
                    throw new Error("Uncompilable assignment operator: "
                                    + stringify(expr));
                }
#endif
                break;
```

```
            }
            case ID_PAREN: {
#ifdef __DEBUG__
                if (expr.length != 2) {
                    throw new Error("Unexpected content of parenthesis: "
                                    + stringify(expr));
                }
#endif
                compile(expr[1], yieldResult);
                hasResult = yieldResult;
                break;
            }
            case ID_CALL_FUNCTION: {
                expr = stripSep(expr);
                boolean methodcall = (childType(expr, 1) == ID_SUBSCRIPT);

                if(methodcall) {
                    emit(ID_THIS);
                    addDepth(1);
                }

                // save program counter
                emit(ID_SAVE_PC);
                addDepth(RET_FRAME_SIZE);

                // find the method/function
                if(methodcall) {
                    Object[] subs = (Object[]) expr[1];
                    compile(subs[1], true);
                    emit(ID_DUP);
                    addDepth(1);
                    emit(ID_SET_THIS);
                    addDepth(-1);
                    compile(subs[2], true);
                    emit(ID_SUBSCRIPT);
                    addDepth(-1);
                } else {
                    compile(expr[1], true);
                }

                // evaluate parameters
                for (int i = 2; i < expr.length; i++) {
                    compile(expr[i], true);
                }

                // call the function
                emit(ID_CALL_FN);
#ifdef __DEBUG__
                if (expr.length > 129) {
                    throw new Error("too many parameters");
                }
#endif
                emit(expr.length - 2);
                addDepth(2 - expr.length - RET_FRAME_SIZE);
                if(methodcall) {
                    emit(ID_SWAP);
                    emit(ID_SET_THIS);
                    addDepth(-1);
                }
                hasResult = true;
                break;
            }
            case ID_BUILD_FUNCTION: {
                Object[] vars = ((Code) expr[1]).closure;
                for (int i = 0; i < vars.length; i++) {
                    String name = (String) vars[i];
```

```
            if (varsClosure.contains(name)) {
                emit(ID_GET_BOXED_CLOSURE);
                pushShort(varsClosure.indexOf(name));
            } else {
                emit(ID_GET_LOCAL);
                pushShort(depth - varsLocals.indexOf(name) - 1);
            }
            addDepth(1);
        }
        emit(ID_LITERAL);
        pushShort(constPoolId(expr[1]));
        addDepth(1);
        emit(ID_BUILD_FN);
        pushShort(vars.length);
        addDepth(-vars.length);
        hasResult = true;
        break;

    }
    case ID_IF: {
        int subtype = childType(expr, 2);

        if (subtype == ID_ELSE) {
            Object[] branch = (Object[]) expr[2];

            //    code for condition
            //    jump_if_true -> label1
            //    code for branch2
            //    jump -> label2
            // label1:
            //    code for branch1
            // label2:

            int pos0, pos1, len;
            // compile condition
            compile(expr[1], true);

            emit(ID_JUMP_IF_TRUE);
            pushShort(0);
            pos0 = code.length();
            addDepth(-1);

            curlyToBlock(branch[2]);
            compile(branch[2], yieldResult);

            emit(ID_JUMP);
            pushShort(0);
            pos1 = code.length();
            len = pos1 - pos0;
            setShort(pos0, len);

            addDepth(yieldResult ? -1 : 0);

            curlyToBlock(branch[1]);
            compile(branch[1], yieldResult);

            len = code.length() - pos1;
            setShort(pos1, len);

            hasResult = yieldResult;
            break;
        } else {
            int pos0, len;

            compile(expr[1], true);
```

```
                    emit(ID_JUMP_IF_FALSE);
                    pushShort(0);
                    pos0 = code.length();
                    addDepth(-1);

                    curlyToBlock(expr[2]);
                    compile(expr[2], false);

                    len = code.length() - pos0;
                    setShort(pos0, len);

                    hasResult = false;
                    break;
                }
            }
            case ID_FOR: {
                Object[] args = (Object[]) expr[1];
                Object init, cond, step;
                if(args.length > 2) {
                    //for(..;..;..)
                    int pos = 1;
                    init = args[pos];
                    pos += (init == SEP_TOKEN) ? 1 : 2;
                    cond = args[pos];
                    pos += (cond == SEP_TOKEN) ? 1 : 2;
                    step = (pos < args.length) ? args[pos] : SEP_TOKEN;
                    curlyToBlock(expr[2]);
                    compile(v(ID_BLOCK, init, v(ID_WHILE, cond,
                                v(ID_BLOCK, expr[2], step))), yieldResult);
                    hasResult = yieldResult;
                    break;
                } else {
                    // for(a in b) c
                    //
                    //   evalute b
                    // labelTop:
                    //   getNextElement
                    //   save -> a
                    //   if no element jump to labelEnd
                    //   c
                    //   jump -> labelTop
                    // labelEnd:
                    //   drop iterator.
                    int pos0, pos1;

                    // find the name
                    Object[] in = (Object[])((Object[])expr[1])[1];
                    Object name = ((Object[])in[1])[1];
                    if(!(name instanceof String)) {
                        // var name
                        name = ((Object[])name)[1];
                    }
#ifdef __DEBUG__
                    if(!(name instanceof String)) {
                        throw new Error("for-in has no var");
                    }
#endif

                    // evaluate b
                    compile(in[2], true);

                    pos0 = code.length();
                    // get next
                    emit(ID_NEXT);
                    addDepth(1);
```

```
            // store value in variable
            compileSet(name);

            // exit if done
            emit(ID_JUMP_IF_FALSE);
            pushShort(0);
            pos1 = code.length();
            addDepth(-1);

            // compile block
            curlyToBlock(expr[2]);
            compile(expr[2], false);

            emit(ID_JUMP);
            pushShort(0);

            setShort(pos1, code.length() - pos1);
            setShort(code.length(), pos0 - code.length());

            emit(ID_DROP);
            addDepth(-1);
            hasResult = false;

            break;
        }

    }
    case ID_SEP: {
        hasResult = false;
        break;
    }
    case ID_AND: {
        assertLength(expr, 3);
        int pos0, pos1, len;

        compile(expr[1], true);

        emit(ID_JUMP_IF_TRUE);
        pushShort(0);
        pos0 = code.length();
        addDepth(-1);

        compile(v(ID_LITERAL, UNDEFINED), true);

        emit(ID_JUMP);
        pushShort(0);
        pos1 = code.length();
        len = pos1 - pos0;
        setShort(pos0, len);
        addDepth(-1);

        compile(expr[2], true);
        len = code.length() - pos1;
        setShort(pos1, len);
        hasResult = true;
        break;
    }
    case ID_OR: {
        assertLength(expr, 3);
        int pos0, pos1, len;

        compile(expr[1], true);

        emit(ID_DUP);
        addDepth(1);
```

```
        emit(ID_JUMP_IF_TRUE);
        pushShort(0);
        pos0 = code.length();
        addDepth(-1);

        emit(ID_DROP);
        addDepth(-1);

        compile(expr[2], true);

        pos1 = code.length();
        len = pos1 - pos0;
        setShort(pos0, len);

        hasResult = true;

        break;
    }
    case ID_LIST_LITERAL: {
        emit(ID_NEW_LIST);
        addDepth(1);

        for (int i = 1; i < expr.length; i++) {
            if (childType(expr, i) != ID_SEP) {
                compile(expr[i], true);
                emit(ID_PUSH);
                addDepth(-1);
            }
        }
        hasResult = true;

        break;
    }
    case ID_CURLY: {
        emit(ID_NEW_DICT);
        addDepth(1);

        int i = 0;
        while (i < expr.length - 3) {
            do {
                ++i;
            } while (childType(expr, i) == ID_SEP);
            compile(expr[i], true);
            do {
                ++i;
            } while (childType(expr, i) == ID_SEP);
            compile(expr[i], true);
            emit(ID_PUT);
            addDepth(-2);
        }
        hasResult = true;

        break;
    }
    case ID_THROW: {
        compile(expr[1], true);
        emit(ID_THROW);
        addDepth(-1);
        hasResult = false;
        break;
    }
    case ID_TRY: {
        //    try -> labelHandle;
        //    ... body
        //    untry
        //    jump -> labelExit;
```

```
        // labelHandle:
        //   set var <- Exception
        //   handleExpr
        // labelExit:
        int pos0, pos1, len;

        Object[] catchExpr = (Object[])expr[2];

        emit(ID_TRY);
        pushShort(0);
        pos0 = code.length();
        addDepth(TRY_FRAME_SIZE);

        curlyToBlock(expr[1]);
        compile(expr[1], false);

        emit(ID_UNTRY);
        addDepth(-TRY_FRAME_SIZE);

        emit(ID_JUMP);
        pushShort(0);
        pos1 = code.length();

        // lableHandle:
        setShort(pos0, code.length() - pos0);

        addDepth(1);
        Object name = ((Object[])((Object[])catchExpr[1])[1])[1];

        compileSet(name);
        emit(ID_DROP);
        addDepth(-1);

        curlyToBlock(catchExpr[2]);
        compile(catchExpr[2], false);

        setShort(pos1, code.length() - pos1);

        hasResult = false;

        break;

    }
    case ID_WHILE: {
        // top:
        //   code for condition
        //   jump if false -> labelExit
        //   code for stmt1
        //   drop
        //   ...
        //   code for stmtn
        //   drop
        //   jump -> labelTop;
        // labelExit:

        int pos0, pos1, len;
        pos0 = code.length();

        compile(expr[1], true);
        emit(ID_JUMP_IF_FALSE);
        pushShort(0);
        pos1 = code.length();
        addDepth(-1);


        curlyToBlock(expr[2]);
```

```
                compile(expr[2], false);

                emit(ID_JUMP);
                pushShort(pos0 - code.length() - 2);

                setShort(pos1, code.length() - pos1);
                hasResult = false;
                break;
            }
            case ID_DO: {
                int pos0;
                pos0 = code.length();

                curlyToBlock(expr[1]);
                compile(expr[1], false);

                compile(((Object[])expr[2])[1], true);
                emit(ID_JUMP_IF_TRUE);
                pushShort(pos0 - code.length() - 2);
                addDepth(-1);
                hasResult = false;
                break;
            }
            case ID_DEC: {
                compile(v(ID_SET, expr[1], v(ID_SUB, expr[1],
                            v(ID_LITERAL, new Integer(1)))), yieldResult);
                return;
            }
            case ID_INC: {
                compile(v(ID_SET, expr[1], v(ID_ADD, expr[1],
                            v(ID_LITERAL, new Integer(1)))), yieldResult);
                return;
            }
            default:
#ifdef __DEBUG__
                throw new Error("Uncompilable expression: " + stringify(expr));
#else
                return;
#endif
        }

        if (hasResult && !yieldResult) {
            emit(ID_DROP);
            addDepth(-1);
        } else if (yieldResult && !hasResult) {
            compile(v(ID_LITERAL, UNDEFINED), true);
        }
    }

/*
```

D.5 Virtual Machine

```
*/

    private static int readShort(int pc, byte[] code) {
        return (short) (((code[++pc] & 0xff) << 8) | (code[++pc] & 0xff));
    }

    /**
     * Transform a object to a fixed point number,
     * represented as a long with 32bit integer part
     * and 32 bit fractional part.
     */
    private static long toFixed(Object o) {
        if(o instanceof FixedPoint) {
            return ((FixedPoint) o).val;
```

```
        }

#ifdef DEBUG
        if(o instanceof Integer) {
#endif
            return (long)((Integer) o).intValue() << (long)32;
#ifdef DEBUG
        }
        throw new Error("object " + o + " not an integer");
#endif
    }
    private static int toInt(Object o) {
        if(o instanceof Integer) {
            return ((Integer) o).intValue();
        }
#ifdef DEBUG
        if(o instanceof FixedPoint) {
#endif
            return (int)((((FixedPoint) o).val + 0x8000000) >> 32);
#ifdef DEBUG
        }
        throw new Error("object " + o + " not an integer");
#endif
    }
    /**
     * Wrap a number in either an Integer object
     * if int, or a FixedPoint object, if fixed point
     */
    private static Object toNumObj(long l) {
        if((l & 0xffffffff) == 0) {
            return new Integer((int)(l >> 32));
        } else {
            return new FixedPoint(l);
        }
    }
    private static class FixedPoint {
        public long val;
        public FixedPoint(long l) {
            val = l;
        }
        public String toString() {
            long rounded = val + ((long) 1 << 32) / 2000;
            String result = Integer.toString((int)(rounded >> 32));
            result += ".";
            int t = (int) rounded;
            t >>>= 16;
            for(int i = 0; i<3;i++) {
                t *= 10;
                result += Integer.toString(t >> 16);
                t &= 0xffff;
            }
            return result;
        }
        public boolean equals(Object o) {
            if(o instanceof Integer) {
                return ((Integer)o).intValue() == (val >> 32);
            }
            if(o instanceof FixedPoint) {
                return ((FixedPoint)o).val == val;
            }
            return false;
        }
    }
    private static boolean toBool(Object o) {
        if(o == TRUE) {
            return true;
```

```
        }
        if(o == FALSE || o == NULL || o == UNDEFINED) {
            return false;
        }
        if(o instanceof String) {
            return !((String)o).equals("");
        }
#ifdef DEBUG
        if(o instanceof Integer) {
#endif
            return ((Integer)o).intValue() != 0;
#ifdef DEBUG
        }
        throw new Error("unhandled toBool case for:" + o.toString());
#endif
    }
    private static Object[] ensureSpace(Object[] stack, int sp, int maxDepth) {
        if (stack.length <= maxDepth + sp + 1) {
            // Currently keep the allocate stack tight to max,
            // to catch errors;
            // Possibly change this to grow exponential
            // for better performance later on.
            Object[] newstack = new Object[maxDepth + sp + 1];
            System.arraycopy(stack, 0, newstack, 0, sp + 1);
            return newstack;
        }
        return stack;
    }
    /**
     * evaluate some bytecode
     */
    private Object execute(Code cl, Object[] stack, int argcount, Object thisPtr) throws LightScript
        int sp = argcount - 1;

        //System.out.println(stringify(cl));
        int pc = -1;
        byte[] code = cl.code;
        Object[] constPool = cl.constPool;
        Object[] closure = cl.closure;
        int exceptionHandler = - 1;
        stack = ensureSpace(stack, sp, cl.maxDepth);
#ifdef __CLEAR_STACK__
        int usedStack = sp + cl.maxDepth;
#endif

        for (;;) {
            ++pc;
            /*
            System.out.println("pc:" + pc + " op:"  + idName(code[pc])
                               + " sp:" + sp + " stack.length:" + stack.length
                               + " int:" + readShort(pc, code));
            */
            switch (code[pc]) {
                case ID_INC_SP: {
                    sp += code[++pc];
                    break;
                }
                case ID_RETURN: {
                    int arg = readShort(pc, code);
                    pc += 2;
                    Object result = stack[sp];
                    sp -= arg - 1;
#ifdef __CLEAR_STACK__
                    for(int i = sp; i <= usedStack; i++) {
                        stack[i] = null;
                    }
```

```
#endif
                    if (sp <= 0) {
#ifdef __DEBUG__
                        if(sp < 0) {
                            throw new Error("Wrong stack discipline"
                                + sp);
                        }
#endif
                        return result;
                    }
                    pc = ((Integer) stack[--sp]).intValue();
                    code = (byte[]) stack[--sp];
                    constPool = (Object[]) stack[--sp];
                    closure = (Object[]) stack[--sp];
                    stack[sp] = result;
                    break;
                }
                case ID_SAVE_PC: {
                    stack[++sp] = closure;
                    stack[++sp] = constPool;
                    stack[++sp] = code;
                    break;
                }
                case ID_CALL_FN: {
                    int argc = code[++pc];
                    Object o = stack[sp - argc];
                    if(o instanceof Code) {
                        Code fn = (Code) o;

                        int deltaSp = fn.argc - argc;
                        stack = ensureSpace(stack, sp, fn.maxDepth + deltaSp);
#ifdef __CLEAR_STACK__
                        usedStack = sp + fn.maxDepth + deltaSp;
#endif
                        sp += deltaSp;
                        argc = fn.argc;

                        for(int i = 0; i < deltaSp; i++) {
                            stack[sp - i] = UNDEFINED;
                        }

                        stack[sp - argc] = new Integer(pc);
                        pc = -1;
                        code = fn.code;
                        constPool = fn.constPool;
                        closure = fn.closure;
                    } else if (o instanceof LightScriptFunction) {
                        try {
                            Object result = ((LightScriptFunction)o
                                ).apply(thisPtr, stack, sp - argc + 1, argc);
                            sp -= argc + RET_FRAME_SIZE;
                            stack[sp] = result;
                        } catch(LightScriptException e) {
                            if(exceptionHandler < 0) {
                                throw e;
                            } else {
                                //System.out.println(stringify(stack));
                                sp = exceptionHandler;
                                exceptionHandler = ((Integer) stack[sp]).intValue();
                                pc = ((Integer) stack[--sp]).intValue();
                                code = (byte[]) stack[--sp];
                                constPool = (Object[]) stack[--sp];
                                closure = (Object[]) stack[--sp];
                                stack[sp] = e.value;
                            }
                            break;
```

```
                    }
#ifdef __DEBUG__
                } else {
                    throw new Error("Unknown function:" + o);
#endif
                }
                break;
            }
            case ID_BUILD_FN: {
                int arg = readShort(pc, code);
                pc += 2;
                Code fn = new Code((Code) stack[sp]);
                Object[] clos = new Object[arg];
                for (int i = arg - 1; i >= 0; i--) {
                    --sp;
                    clos[i] = stack[sp];
                }
                fn.closure = clos;
                stack[sp] = fn;
                break;
            }
            case ID_SET_BOXED: {
                int arg = readShort(pc, code);
                pc += 2;
                ((Object[]) stack[sp - arg])[0] = stack[sp];
                break;
            }
            case ID_SET_LOCAL: {
                int arg = readShort(pc, code);
                pc += 2;
                stack[sp - arg] = stack[sp];
                break;
            }
            case ID_SET_CLOSURE: {
                int arg = readShort(pc, code);
                pc += 2;
                ((Object[]) closure[arg])[0] = stack[sp];
                break;
            }
            case ID_GET_BOXED: {
                int arg = readShort(pc, code);
                pc += 2;
                Object o = ((Object[]) stack[sp - arg])[0];
                stack[++sp] = o;
                break;
            }
            case ID_GET_LOCAL: {
                int arg = readShort(pc, code);
                pc += 2;
                Object o = stack[sp - arg];
                stack[++sp] = o;
                break;
            }
            case ID_GET_CLOSURE: {
                int arg = readShort(pc, code);
                pc += 2;
                stack[++sp] = ((Object[]) closure[arg])[0];
                break;
            }
            case ID_GET_BOXED_CLOSURE: {
                int arg = readShort(pc, code);
                pc += 2;
                stack[++sp] = closure[arg];
                break;
            }
            case ID_LITERAL: {
```

```
        int arg = readShort(pc, code);
        pc += 2;
        stack[++sp] = constPool[arg];
        break;
    }
    case ID_BOX_IT: {
        int arg = readShort(pc, code);
        pc += 2;
        Object[] box = {stack[sp - arg]};
        stack[sp - arg] = box;
        break;
    }
    case ID_DROP: {
        --sp;
        break;
    }
    case ID_NOT: {
        stack[sp] = toBool(stack[sp]) ? FALSE : TRUE;
        break;
    }
    case ID_NEG: {
        stack[sp] = toNumObj(-toFixed(stack[sp]));
        break;
    }
    case ID_ADD: {
        Object o2 = stack[sp];
        --sp;
        Object o = stack[sp];
        if(o instanceof Integer && o2 instanceof Integer) {
            int result = ((Integer) o).intValue();
            result += ((Integer) o2).intValue();
            stack[sp] = new Integer(result);
        } else if( (o instanceof Integer || o instanceof FixedPoint)
              && (o2 instanceof Integer || o2 instanceof FixedPoint) ) {
            stack[sp] = toNumObj(toFixed(o) + toFixed(o2) );
        } else {
            stack[sp] = String.valueOf(o) + String.valueOf(o2);
        }
        break;
    }
    case ID_SUB: {
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if(o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = new Integer(((Integer)o1).intValue()
                        - ((Integer)o2).intValue());
        } else {
            stack[sp] = toNumObj(toFixed(o1) - toFixed(o2) );
        }
        break;
    }
    case ID_SHIFT_RIGHT: {
        int result = toInt(stack[sp]);
        result = toInt(stack[--sp]) >> result;
        stack[sp] = new Integer(result);
        break;
    }
    case ID_MUL: {
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if(o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = new Integer(((Integer)o1).intValue()
                        * ((Integer)o2).intValue());
        } else {
            stack[sp] = toNumObj((toFixed(o1) >> (long)16) * (toFixed(o2) >> (long)16));
        }
```

```
        break;
    }
    case ID_DIV: {
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if(o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = toNumObj(
                        ((long)((Integer)o1).intValue() << (long) 32)
                        / ((Integer)o2).intValue());
        } else if(o2 instanceof Integer) {
            stack[sp] = toNumObj(toFixed(o1) / ((Integer)o2).intValue());
        } else {
            long l1 = toFixed(o1);
            long l2 = toFixed(o2);
            boolean negative = false;
            if(l1 < 0) {
                l1 = -l1;
                negative = !negative;
            }
            if(l2 < 0) {
                l2 = -l2;
                negative = !negative;
            }

            while(l1 > 0 && l2 > 0) {
                l1 <<= 1;
                l2 <<= 1;
            }
            l1 >>>= 1;
            l2 >>>= 33;
            stack[sp] = toNumObj(l1/l2);
        }
        break;
    }
    case ID_REM: {
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if(o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = new Integer(((Integer)o1).intValue()
                        % ((Integer)o2).intValue());
        } else {
            stack[sp] = toNumObj(toFixed(o1) % (toFixed(o2)));
        }
        break;
    }
    case ID_NOT_EQUALS: {
        Object o = stack[sp];
        --sp;
        stack[sp] = (o == null)
                ? (stack[sp] == null ? FALSE : TRUE )
                : (o.equals(stack[sp]) ? FALSE : TRUE );
        break;
    }
    case ID_EQUALS: {
        Object o = stack[sp];
        --sp;
        stack[sp] = (o == null)
                ? (stack[sp] == null ? TRUE : FALSE)
                : (o.equals(stack[sp]) ? TRUE : FALSE);
        break;
    }
    case ID_PUT: {
        Object val = stack[sp];
        Object key = stack[--sp];
        Object container = stack[--sp];
```

```
        if (container instanceof LightScriptObject) {
            ((LightScriptObject)container).set(key, val);

        } else if (container instanceof Stack) {
            int pos = toInt(key);
            Stack s = (Stack) container;
            if(pos >= s.size()) {
                s.setSize(pos + 1);
            }
            s.setElementAt(val, pos);

        } else if (container instanceof Hashtable) {
            if (val == null) {
                ((Hashtable) container).remove(key);
            } else {
                ((Hashtable) container).put(key, val);
            }
        } else {
            defaultSetter.apply(container, stack, sp + 1, 2);
        }
        break;
    }
    case ID_SUBSCRIPT: {
        Object key = stack[sp];
        Object container = stack[--sp];
        Object result = null;


        // "Object"
        if (container instanceof LightScriptObject) {
            result = ((LightScriptObject)container).get(key);
        }  else if (container instanceof Hashtable) {
            result = ((Hashtable) container).get(key);
            if(result == null) {
                Object prototype = ((Hashtable)container).get("prototype");
                // repeat case ID_SUBSCRIPT with prototype as container
                if(prototype != null) {
                    stack[sp] = prototype;
                    sp += 1;
                    pc -= 1;
                    break;
                }
            }

        // "Array"
        } else if (container instanceof Stack) {
            if(key instanceof Integer) {
                int pos = ((Integer) key).intValue();
                Stack s = (Stack) container;
                result = 0 <= pos && pos < s.size()
                            ? s.elementAt(pos)
                            : null;
            } else if("length".equals(key)) {
                result = new Integer(((Stack)container).size());
            } else {
                result = arrayPrototype.get(key);
            }

        // "String"
        } else if (container instanceof String) {
            if(key instanceof Integer) {
                int pos = ((Integer) key).intValue();
                String s = (String)container;
                result = 0 <= pos && pos < s.length()
                            ? s.substring(pos, pos+1)
                            : null;
```

```
            } else if("length".equals(key)) {
                result = new Integer(((String)container).length());
            } else {
                result = stringPrototype.get(key);
            }

        // "Function"
        } else if (container instanceof LightScriptFunction) {
            if("length".equals(key)) {
                result = new Integer(((LightScriptFunction)container).getArgc());
            } else {
                result = functionPrototype.get(key);
            }

        // Other builtin types, by calling userdefined default getter
        } else {
            result = defaultGetter.apply(container, stack, sp + 1, 1);
        }

        // prototype property or element within (super-)prototype
        if(result == null) {
            if("prototype".equals(key)) {
                if(container instanceof LightScriptFunction) {
                    result = functionPrototype;
                } else if(container instanceof Stack) {
                    result = arrayPrototype;
                } else if(container instanceof String) {
                    result = stringPrototype;
                } else {
                    result = objectPrototype;
                }
            } else {
                result = objectPrototype.get(key);
                if(result == null) {
                    result = UNDEFINED;
                }
            }
        }
        stack[sp] = result;
        break;
    }
    case ID_PUSH: {
        Object o = stack[sp];
        ((Stack) stack[--sp]).push(o);
        break;
    }
    case ID_POP: {
        stack[sp] = ((Stack) stack[sp]).pop();
        break;
    }
    case ID_LESS: {
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if (o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = ((Integer) o1).intValue()
                < ((Integer) o2).intValue() ? TRUE : FALSE;
        } else if( (o1 instanceof Integer || o1 instanceof FixedPoint)
               &&  (o2 instanceof Integer || o2 instanceof FixedPoint) ) {
            stack[sp] = toFixed(o1) < toFixed(o2) ? TRUE : FALSE;
        } else {
            stack[sp] = o1.toString().compareTo(o2.toString())
                < 0 ? TRUE : FALSE;
        }
        break;
    }
    case ID_LESS_EQUALS: {
```

```
        Object o2 = stack[sp];
        Object o1 = stack[--sp];
        if (o1 instanceof Integer && o2 instanceof Integer) {
            stack[sp] = ((Integer) o1).intValue()
                <= ((Integer) o2).intValue() ? TRUE : FALSE;
        } else if( (o1 instanceof Integer || o1 instanceof FixedPoint)
               &&  (o2 instanceof Integer || o2 instanceof FixedPoint) ) {
            stack[sp] = toFixed(o1) <= toFixed(o2) ? TRUE : FALSE;
        } else {
            stack[sp] = o1.toString().compareTo(o2.toString())
                <= 0 ? TRUE : FALSE;
        }
        break;
    }
    case ID_JUMP: {
        pc += readShort(pc, code) + 2;
        break;
    }
    case ID_JUMP_IF_FALSE: {
        if (toBool(stack[sp])) {
            pc += 2;
        } else {
            pc += readShort(pc, code) + 2;
        }
        --sp;
        break;
    }
    case ID_JUMP_IF_TRUE: {
        if (toBool(stack[sp])) {
            pc += readShort(pc, code) + 2;
        } else {
            pc += 2;
        }
        --sp;
        break;
    }
    case ID_DUP: {
        Object o = stack[sp];
        stack[++sp] = o;
        break;
    }
    case ID_NEW_LIST: {
        stack[++sp] = new Stack();
        break;
    }
    case ID_NEW_DICT: {
        stack[++sp] = new Hashtable();
        break;
    }
    case ID_SET_THIS: {
        thisPtr = stack[sp];
        --sp;
        break;
    }
    case ID_THIS: {
        stack[++sp] = thisPtr;
        break;
    }
    case ID_SWAP: {
        Object t = stack[sp];
        stack[sp] = stack[sp - 1];
        stack[sp - 1] = t;
        break;
    }
    case ID_THROW: {
        Object result = stack[sp];
```

```
                    if(exceptionHandler < 0) {
                        throw new LightScriptException(result);
                    } else {
                        //System.out.println(stringify(stack));
                        sp = exceptionHandler;
                        exceptionHandler = ((Integer) stack[sp]).intValue();
                        pc = ((Integer) stack[--sp]).intValue();
                        code = (byte[]) stack[--sp];
                        constPool = (Object[]) stack[--sp];
                        closure = (Object[]) stack[--sp];
                        stack[sp] = result;
                    }
                    break;
                }
                case ID_TRY: {
                    stack[++sp] = closure;
                    stack[++sp] = constPool;
                    stack[++sp] = code;
                    stack[++sp] = new Integer(pc + readShort(pc, code) + 2);
                    stack[++sp] = new Integer(exceptionHandler);
                    exceptionHandler = sp;
                    pc += 2;
                    break;
                }
                case ID_UNTRY: {
                    exceptionHandler = ((Integer) stack[sp]).intValue();
                    sp -= TRY_FRAME_SIZE;
                    break;
                }
                case ID_NEXT: {
                    Object o = stack[sp];
                    if(o instanceof Hashtable) {
                        o = ((Hashtable)o).keys();
                        stack[sp] = o;
                    }

                    Enumeration e = (Enumeration)o;
                    if(e.hasMoreElements()) {
                        stack[++sp] = e.nextElement();
                    } else {
                        stack[++sp] = UNDEFINED;
                    }
                    break;
                }
                default: {
#ifdef __DEBUG__
                    throw new Error("Unknown opcode: " + code[pc]);
#endif
                }
            }
        }
    }
}
```