Master thesis

Describing Live programming using program transformations and a callstack explicit interpreter

by

Olov Johansson

LITH-IDA-EX--06/045--SE

Supervisor : **Tobias Nurmiranta**
Dept. of Computer and Information Science
at Linköping University

Examiner : **Anders Haraldsson, Associate professor**
Dept. of Computer and Information Science
at Linköping University

Abstract

A formalization of how to apply incremental modifications to a running program while deterministically preserving state is presented. These are described using source code transformations converted to abstract syntax tree and callstack transformations. A rationale for why dynamically typed languages benefit most from the concept is given, as are explanations to the selected programming language and implementation using an explicit callstack.

Keywords : live programming, incremental, program transformations, explicit callstack, dynamic programming language, interpreted, javascript

Acknowledgements

Thanks to my examiner Anders Haraldsson and my supervisor Tobias Nurmiranta for being positive and encouraging, for interesting and helpful discussions and for letting me hack and write away.

Thanks to Brendan Eich and the Mozilla organization for writing fine software and releasing it as free software for others to build their work on.

Contents

Chapter 1

Introduction

1.1 Background

Traditional integrated development environments have often been centered around either the "Edit, Compile and Run" cycle or the "Read-Eval-Print Loop". Debuggers have become very powerful and are often integrated in the same interface as the editor, yet the "Edit, Compile and Run" cycle remains. The need to repeatedly quit running/debugging, modify the source code, re-compile and run/debug again has been deeply rooted as some kind of prerequisite, partly because the toolset tend to be separated, partly because it's often described that way in literature and perhaps partly because it is, indeed, needed.

There are many drawbacks of having to quit the running program for recompiling, as opposed to the "just works" approach where modified textual source code should result in an incrementally updated running program. One is the loss of context – perhaps it's not trivial to get the program into the same state (due to expensive data collection or event programming, for instance), another is loss of productivity due to lack of instant feedback. Silly mistakes such as misspelled variable names or off-by-one loops are easily detected with a good debugger but become extremely expensive to correct if the program needs to quit, recompile and restart instead of "apply

code changes and continue".

But that's just a scratch on the surface. Incremental modifications to a running program could just as well be used the opposite way around by applying in to the empty program and building up the solution from there (instead of using it to fix bugs in incorrect problem solutions). An example of such a session can be found in appendix A.

In order to support these modifications without a restart of the program and without any unnecessary rollback of the program counter/position at all, they must be expressed through *transformations* on the existing program. A program is typically represented as machine code, bytecode or an abstract syntax tree, together with a stack for intermediary values and return addresses (the callstack, from now on). Creating a new program representation (corresponding to the modified source code) and replacing the old one is what causes program restart or rollback, thus the need for transforming the *existing* program representation. Preferably as transparent as possible to the programmer, who modifies/transforms textual source code. A good development environment supporting this concept, "*Live programming*", should automatically convert those source code transformations to transformations on the internal program representation.

1.2 Purpose

The goals of my work is to motivate and define the Live programming concept a bit more precise than "modify running code and continue", outline what kind of typical source code transformations it implies, develop supporting operations for those using AST (abstract syntax tree) and callstack transformations and create a prototype implementation with a development environment for proof of concept. The chosen transformations are covered in this report, as are small parts of the source code (inserted throughout the text when appropriate). Chapter 2 elaborates further on the Live programming concept. This report presents the theoretical foundations for the prototype implementation, the prototype itself is complementary to the report and its source code is freely available, see section 1.4. A screenshot of the prototype follows at the end of chapter 2.

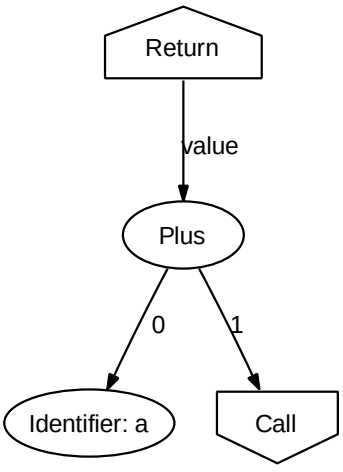
I focus on dynamically typed (from now on: *dynamic*) languages that

are interpreted using their AST representations. I will present some aids Live programming brings to both dynamic and statically typed (from now on: *static*) languages. The dynamic language chosen is JavaScript. The evaluator part of the interpreter must permit modifications to its callstack, an existing JavaScript interpreter is modified to facilitate this, see chapter 2 and 4 for more information.

1.3 Typographic conventions

All source code snippets are written using a `monospaced font`, so are parts of the source code whenever it's referenced in the text. Removed (as for readability) text segments are indicated with `....` A node of the abstract syntax tree are referred to as `<TYPE>` and can relate to either all or a specific instance of the type, depending on context.

AST's are visualised in some examples. A node in the shape of an ellipse indicates that no incoming or outgoing edges have been excluded, a house shape (arrow upwards) indicates that incoming edges and ancestor nodes have been excluded and an upside down house (arrow downwards) indicates excluded outgoing edges and descendants. The type names are written inside the nodes, although not in all capital letters for increased readability. In the example tree below ancestors to `<RETURN>` and descendants to `<CALL>` are excluded, `<PLUS>` and `<IDENTIFIER>` are both complete.



1.4 Report and source code availability

The full source code of the modified interpreter and the prototype environment implementing the program transformations, is available at http://liveprogramming.org together with an electronic version of this report. All source code are licensed under the free software licenses MPL[1], GPL[2] and LGPL[3], see the source code for more information. The interpreter is derived from Narcissus, written by Brendan Eich at Mozilla.

1.5 Structure

Chapter 2 elaborates on the Live programming concept, chapter 3 is a crash course in the JavaScript language and can be skipped but should rather be complemented with other material such as the recommended book. Chapter 4 discusses the inner workings of the interpreter and necessary changes to it, as well as some necessary terminology for the future chapters. Chapter 5 and 6 define incremental modifications to a program from the users as well as the language and environment implementers perspective. Related work follows in chapter 7 and a concluding discussion (with future work) in chapter 8. Appendix A describes a small Live programming session, appendix B has the source code and AST for an example program and can be useful to quickly get an understanding of the AST structure. The reader will likely benefit the most by reading from start to end, in order.

[1] http://www.mozilla.org/MPL/
[2] http://www.gnu.org/licenses/gpl.html
[3] http://www.gnu.org/licenses/lgpl.html

Chapter 2

Live programming justified

This chapter elaborates on the Live programming concept by presenting some aids and problems as well as reflecting on programming from an incremental perspective. Live programming is defined via program transformations and the programming language to implement a prototype with is chosen.

2.1 Type information as programmer aid

Static languages carry excessive (explicit or implicit) type information in its source code so it's natural to create an editor that take use of it to help out the programmer. Most programmers expect to see an argumentlist after typing the name of a function/method and left parentheses, as do they expect to see a list of object properties/variables/methods once they type the dereferencing operator on an object reference in the source code. Most (static language) development environments provide these aids. The editor has an integrated parser for the language to accommodate this.

There are limitations of how much information can be given about an object in edit time while the program is not yet running, though. The

type information for an object reference isn't always the actual type, but rather a base type higher up in the inheritance chain. More specialized information for those can't be determined until the program is actually run.

For dynamic languages, this is unfortunately the common case. No type information is available for references, at all. While built-in type initializers such as `3`, `'asdf'` or `['first-in-list', 2]` wear implicit type information, it's lost as soon as they are assigned to a reference. It's as if all references were typed as `Object` (or similar). One could use type inference algorithms onto the full program source code to deduce type information, while this would work on simple examples it would fail on many, example follows. The type information for a reference (or rather, type information for what it refers to) is thus only available at runtime.

```
function f() {
    return 'asdf';
}
z = f();
z.<display list of string methods here>

function g() {
    return (switch = !switch) ? 'asdf' : 101;
}
z = g();
z.<can't display a list here>
```

2.2 Identifier verification as programmer aid

A feature of static languages are that they won't compile code that includes erroneous identifiers such as misspelled local variables or function/method names. The typical interpreted dynamic language will indicate the error in runtime, instead – less helpful for the programmer and much more time consuming. Even worse, the erroneous code needs to actually execute which means that all possible code paths may need to run to expose the error.

As for type information, some identifier verification is possible to do at edit time for dynamic languages too, although it's not very common.

If the dynamic language is lexically scoped then it's indeed possible to determine what identifiers are visible at any position in the source code. This cover local variable names and (possibly nested) function names, but it doesn't cover object properties (methods, member variables) since they are typically allowed to be created/deleted ad hoc at runtime. The call `duck.quack()` may execute properly the first time but not the second, from the exact same spot in the source code, all depending on the dynamic nature on the program (assuming that the duck object may be altered in between). Verification of an object property is thus only possible at runtime.

2.3 REPL as programmer aid

The Read-Eval-Print Loop is used in Lisp and Scheme systems perhaps more than anywhere else. The typical arrangement is to have one "code buffer" containing the source code of the program. The buffer can at any time be re-evaluated, "sent to the interpreter". Then there is the "eval prompt", into which the programmer can type code snippets, often calling functions that has been defined in the code buffer. The code snippet gets evaluated and the result is printed, thus the "Read-Eval-Print Loop" name.

This tend to bring a different style of programming where the programmer is encouraged to experiment its way into solving the problem instead of knowing it beforehand. Functions are called and the result value are inspected by the programmer, who can then determine what to do next with the returned value. The programming technique has an informal "result driven" style to it.

Often the REPL belongs to the top-level of the program only but sometimes it can be spawned from inside a called function, enabling the programmer to inspect arguments or object properties and evaluate arbitrary expressions in the current scope of the paused program.

2.4 Debugging is a two way process

A good debugger is a (the) power tool for any experienced programmer. The debugger is most often used after a defect has been detected to pinpoint a troublesome part of the code. Once it has been found, the debugger is usually quit and the editor is entered once again, in the traditional Edit, Compile and Debug/Run manner. Most debuggers give the programmer tools to inspect and modify the current state of local variables and memory (with no restart), few debuggers does the same to code modifications. This can be especially frustrating for simple errors, such as wrong identifier names or off-by-one comparisons. Adding features (integrated into the debugger) to modify the current code as well as to add new in runtime would be beneficial for many kinds of bug fixes.

2.5 Three mantras

These three mantras all attest to programming as a highly iterative process of changes to a program.

"You aren't going to need it"

> The way YAGNI is usually described, it says that you shouldn't add any code today which will only be used by feature that is needed tomorrow. On the face of it this sounds simple. The issue comes with such things as frameworks, reusable components, and flexible design. Such things are complicated to build. You pay an extra up-front cost to build them, in the expectation that you will gain back that cost later. This idea of building flexibility up-front is seen as a key part of effective software design.
>
> However XP's advice is that you not build flexible components and frameworks for the first case that needs that functionality. Let these structures grow as they are needed. If I want a Money class today that handles addition but not multiplication then I build only addition into the Money class. Even

> if I'm sure I'll need multiplication in the next iteration, and understand how to do it easily, and think it'll be really quick to do, I'll still leave it till that next iteration.
>
> – *Martin Fowler, "Is Design Dead?"* [1]

"Do the simplest thing that could possibly work"

> It was a question: "Given what we're trying to do now, what is the simplest thing that could possibly work?" In other words, let's focus on the goal. The goal right now is to make this routine do this thing. Let's not worry about what somebody reading the code tomorrow is going to think. Let's not worry about whether it's efficient. Let's not even worry about whether it will work. Let's just write the simplest thing that could possibly work.
>
> Once we had written it, we could look at it. And we'd say, "Oh yeah, now we know what's going on," because the mere act of writing it organized our thoughts. Maybe it worked. Maybe it didn't. Maybe we had to code some more. But we had been blocked from making progress, and now we weren't. We had been thinking about too much at once, trying to achieve too complicated a goal, trying to code it too well. Maybe we had been trying to impress our friends with our knowledge of computer science, whatever. But we decided to try whatever is most simple: to write an if statement, return a constant, use a linear search. We would just write it and see it work. We knew that once it worked, we'd be in a better position to think of what we really wanted.
>
> – *"A Conversation with Ward Cunningham, Part V"* [2]

"Don't repeat yourself"

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. The alternative is to have the same thing expressed in two or more places. If

> you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn't a question on whether you'll remember: it's a question of when you'll forget.
>
> – *"The Pragmatic Programmers"* [3]

2.6 Levelling up tests and design by contract

Tests are used to verify functionality in a code unit, bigger module or full system. The Extreme Programming [4] crew, led by authorities such as Kent Beck, have been very pushing in formalizing tests[1] into code so that the tests can be reproduced and continuously executed to detect regressions as soon as possible. XP takes this as far as "design tests first". But then what, when the tests are run on the empty program? Instead of just using the tests to print out a "success" or "error" for each test case and then be done with it (evolving into the Edit, Compile and Run/Debug/Test cycle), the test could be aiding the programmer to implement to code faster, and better. The test is a program in itself, the programmer should be able to modify his code while the test suite is still running (it's down on the callstack, somewhere), that's "Test *driven* programming" for you.

The same applies to Design by Contract [3], a technique developed by Bertrand Meyer for the language Eiffel. The preconditions, postconditions and class invariants of DBC are verified at the entry and exit of a function, if they are not met an error is indicated, often raising an exception.

2.7 Delivering via Live programming

My stand is that to deliver these aids, as good as possible, the programmer needs to be given the appropriate tools to move parts of the programming from edit time to runtime. There are two extremes, one is doing only very small changes at runtime, the other is starting out with nothing and writing the whole program in runtime. The highest productivity (depending on

[1] make no mistake, XP has accomplished a lot more besides pushing tests

the programmer) should be somewhere between the two. Either way, Live programming should be about bringing the means without enforcing use of it.

This means integrating the editor and debugger into one. All the usual functions of an debugger are available but at any time the programmer can choose to modify the source code, either by adding new functions and modifying function that's not active on the callstack, or by modifying code that is actually running (but paused).

Looking into Smalltalk, much of this has been done and many Smalltalk programmers swear by it. Smalltalk environments are very much focused on incrementally extending the program by adding new objects and methods. Code can be replaced, in the Smalltalk case any method can be modified. The problem, however, is that upon doing this the callstack (and thus the active program position) will rollback to the first call of the modified method, at least that was my experience of the popular open source Squeak [5] implementation.

Visual Studio on the other hand is the complete opposite to Squeak regarding modifying active programs. Visual Studio was one of the first C++ development environment (that I am aware of, at least) to support the "edit and continue" operation, which is very handy for fixing small faults (such as off-by-one errors) but limited to such small changes. Visual Studio 2005 was released in November 2005 and recently brought to my attention. I was impressed with the improved edit and continue support they implemented in it, it's complete in some ways such as it transforms the callstack correctly, sadly it has severe limitations such as its inability to transform expressions and that new functions can't be defined without restarting.

Live programming delivers:

Reflecting on the previous sections Live programming can help a dynamic language with:

- Type information – the debugger can help with inspecting types and values, so can the editor since they are integrated into one. The reference to inspect just needs to be into scope somewhere on the

callstack.

- Identifier verification – if the lexical scope of the identifier is active then it's possible to tell whether it's defined or not. As for object properties the only way to really tell is by evaluating it, which can be done by adding the code and stepping into it.

- REPL – there is no reason not to bring an eval window to the programmer as a complement to the source window.

- Debugging – it is a two way process, indeed, and by providing means of actually fixing the defects, with no restart and no rollback, the programmer gains productivity.

- Three mantras – they boil down to advocating an iterative (often incremental) process of programming. "You aren't going to need it" highlights the importance of resisting the temptation of doing too much when you know too little – instead work yourself from inside the problem and out, adding features. "Do the simplest thing that could possibly work" is about solving the mental vacuum of staring into the problem by doing *something*, perhaps just to learn that it was wrong, still it was something (REPL is helpful here as well, encouraging the programmer to look at part solutions). "Don't repeat yourself" (XP has a similar "Once and only once") need support for refactoring in the form of breaking out a piece of code into a separate function, for instance.

2.8 How to deliver it

Live programming is code transformation

> *Live programming: Modifying source code for an active program through well defined transformations, while retaining and updating the callstack in a sane manner.*

Parsing a textfile or source code and then starting evaluating it is nothing new, neither is attaching debugging functionality for stepping, breakpoints

and variable inspection. The tricky part comes when the programmer modifies the source code. The modifications made to the source code must be detected and "injected" into the program, the environment should just do "the right thing" (the programmer is certainly confident in what the modifications meant, so there is in most cases a right thing to do). If the programmer inserted a new statement just after the current evaluating one inside of a function, the wrong thing to do would be to restart the function from the beginning (or even worse, quit the program), the right thing would be to finish evaluating the current statement and proceed with the inserted one, as if it would have been there to start with. If the program crashes on a misspelled identifier the right thing to do is to let the programmer change that identifier and continue (keeping the current program position). Similarly, a crash due to a call to an undefined function should be able to resolve by defining the function and continuing.

The key problem is that the running program needs to be transformed into the new version, it can't simply be replaced. I am confident that these transformations are possible for a compiled program as well as an interpreted one, but I do believe it's easier for interpreted ones. I settled on creating a prototype implementation of a dynamic programming language interpreter for proof of concept. The interpreter works directly on an AST using an explicit (modifiable) callstack. Mapping source code transformations into changes on the running program thus comes down to transforming the AST and transforming the callstack.

2.9 Proof of concept prototype

I wanted to use an existing programming language specification rather than creating a new one. I chose JavaScript, partly because it's a quite small language, commonly used and because there are multiple FLOSS[2] implementations available. It's syntactically and semantically neither trivial as Lisp, Scheme or Smalltalk[3], nor as complex as Perl, Python or Ruby. It is object oriented but with a prototype based object model instead of a class based, saving syntax by instead implementing object oriented fea-

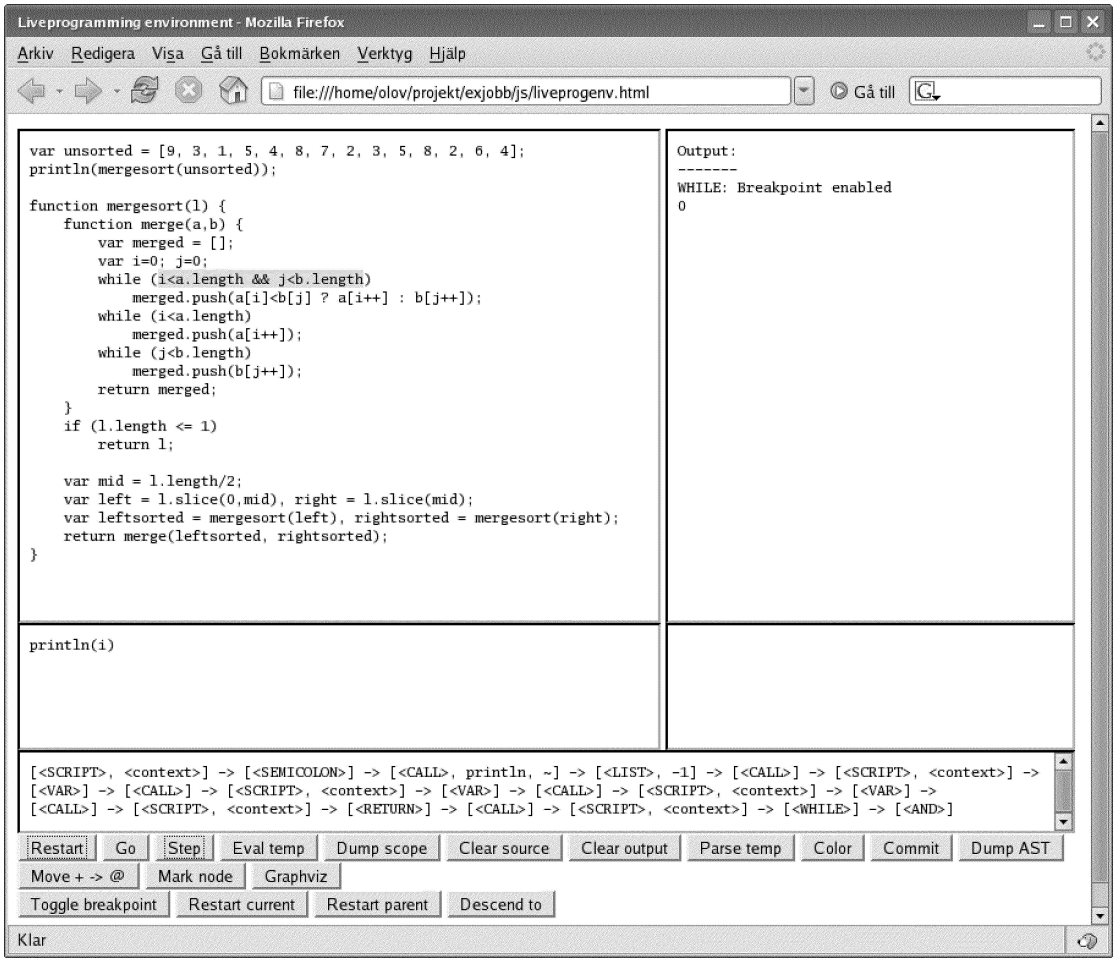
[2]Free/Libre and Open Source Software

[3]considering everything is done with messages

tures through the normal constructions of the language. The prototype based object model originate from the programming language Self [6].

Among the JavaScript implementations are SpiderMonkey (implemented in C), Rhino (implemented in Java), Narcissus (implemented in JavaScript itself) and KJS (implemented in C++), the first three of which are created by the Mozilla organization, KJS are created by the KDE project for their web browser Konqueror (and later adopted by Apple for their spinoff, Safari). SpiderMonkey is a high performance implementation running inside every Firefox browser, Rhino is object oriented in its design and more structured and Narcissus is very elegant although very slow. Narcissus is meta circular and borrows much of its functionality (for objects and arrays, for instance) from the underlying JavaScript host, that's how it can consist of only approximately 2000 lines of code, evenly divided between the parser and evaluator. Narcissus is written by Brendan Eich, the inventor of JavaScript and also the author of SpiderMonkey.

I chose Narcissus as a base for my work since the parser should be almost the same and the evaluator could be similar in style, although needing a rewrite for an explicit callstack. Using a JavaScript evaluator written in JavaScript itself meant that I could run it inside a web browser, giving free access to all the available GUI web elements there. A screenshot of the prototype running inside Firefox follows.


Liveprogramming environment - Mozilla Firefox
Arkiv Redigera Visa Gå till Bokmärken Verktyg Hjälp
file:///home/olov/projekt/exjobb/js/liveprogenv.html
Gå till
var unsorted = [9, 3, 1, 5, 4, 8, 7, 2, 3, 5, 8, 2, 6, 4];
println(mergesort(unsorted));
function mergesort(l) {
function merge(a,b) {
var merged = [];
var i=0; j=0;
while (i<a.length && j<b.length)
merged.push(a[i]<b[j] ? a[i++] : b[j++]);
while (i<a.length)
merged.push(a[i++]);
while (j<b.length)
merged.push(b[j++]);
return merged;
}
if (l.length <= 1)
return l;
var mid = l.length/2;
var left = l.slice(0,mid), right = l.slice(mid);
var leftsorted = mergesort(left), rightsorted = mergesort(right);
return merge(leftsorted, rightsorted);
}
Output:

WHILE: Breakpoint enabled
0
println(i)
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL>, println, ~] -> [<LIST>, -1] -> [<CALL>] -> [<SCRIPT>, <context>] ->
[<VAR>] -> [<CALL>] -> [<SCRIPT>, <context>] -> [<VAR>] -> [<CALL>] -> [<SCRIPT>, <context>] -> [<VAR>] ->
[<CALL>] -> [<SCRIPT>, <context>] -> [<RETURN>] -> [<CALL>] -> [<SCRIPT>, <context>] -> [<WHILE>] -> [<AND>]
Restart
Go
Step
Eval temp
Dump scope
Clear source
Clear output
Parse temp
Color
Commit
Dump AST
Move + -> @
Mark node
Graphviz
Toggle breakpoint
Restart current
Restart parent
Descend to
Klar

Chapter 3

JavaScript

JavaScript is a dynamically typed, lexically scoped programming language. Numerous books have been written about the JavaScript language, most of which are mainly targeting web designers. Stating that one book is better than another, or that one book is especially poorly written in a technical and/or educational way is highly subjective and perhaps of little academic value. I suffice to say that I only found one book "good enough" – namely JavaScript: The Definitive Guide [7]. Among other books I read were Dynamic HTML: The Definitive Reference [8] and JavaScript Bible [9]. The ECMAScript specification is freely available on the web [10] but can't be recommended to anyone except JavaScript language implementers.

I recommend the reader to borrow the nearest copy of JavaScript the Definitive Guide to get up to speed with JavaScript, most of the book is a reference for web programming and can be skipped for the non-interested reader. I will give a very brief description of the language here.

3.1 Syntax

JavaScript has a very C-like syntax, The C Programming Language [11] (commonly called K&R C) is a great introduction for the reader who is unfamiliar with it.

3.2 Logical operators

In C, the short-circuiting logical operators `&&` and `||` return `0` or `1`, in JavaScript they return the value of the evaluated left or right operand. The `&&` operator returns the left operand if it evaluates to `false`, `undefined`, `null`, `0` or the empty string `""`, otherwise the evaluated right operand. The `||` operator returns the left operand if it evaluates to non-false, otherwise the evaluated right operand. This makes some expressions shorter to write (the alternative would be an temporary assignment followed by an `if` or `?:`).

```
return o && o.f();
var v = keywords[id] || IDENTIFIER;
```

3.3 Definitions

Variables are defined with the `var` statement:

```
var x; // defaults to undefined
var y=square(3), z; // initializer expressions are optional
const pi=3.14142; // constants can't be reassigned
```

Functions can be defined in three ways:

```
function double1(x) { // named function
    return x*2;
}
var double2 = function(x) { // anonymous function, ``lambda''
    return x*2;
}
// eval-calling Function object
var double3 = new Function('x', 'return x*2');
```

3.4 Scoping

JavaScript is lexically scoped (as opposed to dynamically scoped) but has function local scope as opposed to block local scope, meaning that any

variable defined inside a function (no matter in what sub-block of the function the definition resides in) will become visible from anywhere inside that function. A side effect of this is that a variable can't be shadowed unless it's defined once again in a nested function.

```
function f(x) {
    if (true) {
        var x = 23; // not shadowed but initializer is run
    }
    function g() {
        var x = 42; // old x shadowed
    }
    g();
    return x; // returns 23, always
}
```

The rules for how variables and function names are created and initialized into scope is as follows: First, when a function call is made, as many of the functions parameters as possible are assigned to the callers arguments. If there are less arguments than parameters, the rest are assigned to `undefined`. Second, all the names of the inner named functions are assigned to those functions. Last, all local variables are assigned to `undefined`. The initializers (if any) for those local variables aren't evaluated until the `var` or `const` statements are entered inside the function. An example follows.

```
function f(x) {
    function printvars() {
        print(x + ' ' + y  + ' ' + z + ' ' + g());
    }
    printvars(); // prints 'arg undefined undefined inner'
    if (true) {
        var y = 'local';
    }
    var z;
    printvars(); // prints 'arg local undefined inner'
    function g() {
        return 'inner';
```

```
    }
    z = 'local2';
    printvars(); // prints 'arg local local2 inner'
}
f('arg');
```

3.5 Dynamic (and weak) type

JavaScript is dynamically and weakly typed:

```
function addone(x) {
    return x+1;
}
var a = addone(42); // a = 43
var b = addone('42'); // b = 421
```

Like Scheme, JavaScript evaluates left-sides of function calls:

```
function f(x) { .. }
function g(x) { .. }
(a ? f : g)(x);
```

3.6 Closures

JavaScript has proper closures, the typical bank account example from Structure and Interpretation of Computer Programs [12] translated into JavaScript from Scheme looks like:

```
function make_account(balance) {
    function withdraw(amount) {
        if (balance >= amount)
            return balance -= amount;
        else
            return 'Insufficient funds';
    }
    function deposit(amount) {
        return balance += amount;
```

```
    }
    return function(m) {
        if (m == 'withdraw')
            return withdraw;
        else if (m == 'deposit')
            return deposit;
        else
            throw 'Unknown request -- make_account';
    }
}
```

The function `make_account()` can be used as follows, where each call to `acc` or `acc2` returns the locally defined `deposit()` or `withdraw()` function, which is then applied to the specified amount. Each call to `make_account()` will produce a completely separate account "object", which maintains its own local balance.

```
var acc = make_account(100), acc2 = make_account(500);
acc('withdraw')(30); // returns: 70
acc2('deposit')(130); // returns: 630
acc('deposit')(130); // returns: 200
acc2('withdraw')(700); // returns: 'Insufficient funds'
```

3.7 Prototype-based object model

An object is created with the `new` operator or an object initializer. The object is a hashtable where the key is called a property, in JavaScript terminology.

```
var o = new Object();
o.key1 = 'one'; // identifier key indexing
o['key2'] = 2; // string key indexing
var o2 = {key1: 'one', key2: 2}; // object initializer
```

Any valid expression can be assigned to an object property, just like any other variable. Specifically, functions (any of the three forms) can be assigned. When a function reference is accessed from an object using the .

(dot) or `[]` (brackets) operator and then directly called, the special variable `this` will become bound to that specific object.

```
function doubleUp() {
    this.val *= 2;
}
var o = {val: 10};
o.doub = doubleUp;
var o2 = {val: 12};
o2.doub = doubleUp;

o.doub(); // o.val becomes 20
o2.doub(); // o.val becomes 24
o['doub'](); // o.val becomes 40
```

If an undefined property of an object is accessed and the `prototype` property refers to another object, then the lookup will continue in that object instead, continuing with that objects `prototype` until the key has been found or prototype is `null`.

The `new` operator allocates an object and calls a function, inside which the variable `this` gets bound to the allocated object. The `prototype` property of the created object is assigned the value of the `prototype` property for that function. The functions that are called through `new` are commonly called constructor functions, since they tend to initialize an object. The default `Object()` constructor function does nothing.

The translation of the closure based `make_account()` into a version using native JavaScript object functionality follows, where `Account()` is the constructor function (JavaScript lacks classes).

```
function Account(balance) {
    this.balance = balance;
}
Account.prototype.withdraw = function(amount) {
    if (this.balance >= amount)
        return this.balance -= amount;
    else
        return "Insufficient funds";
```

```
}
Account.prototype.deposit = function(amount) {
    return this.balance += amount;
}

var acc = new Account(100), acc2 = new Account(500);
acc.withdraw(30); // returns: 70
acc2.deposit(130); // returns: 630
acc.deposit(130); // returns: 200
acc2.withdraw(700); // returns: 'Insufficient funds'
```

3.8 Arrays

Arrays are heavily used in JavaScript code, an array is also an object, thus properties apart from indices can be used as well. Arrays have the property `length`.

```
var a = new Array(); // optional initial size argument, default 0
a[0] = 'test'; // grows as necessary
a[2] = 101;
a = ['test', undefined, 101]; // array initializer
```

Chapter 4

JavaScript interpreter implementation

This chapter describes the Narcissus JavaScript interpreter which is written in JavaScript itself. The structure of the abstract syntax tree produced by the parser along with the made modifications are described and motivated, as is the rewritten evaluator. The evaluator was originally written in an implicit callstack style which made (immediate) callstack transformations impossible, thus the rewrite to an explicit callstack.

4.1 Parser

The parser is written in the common recursive descent [13] style, consuming tokens (created from a stream of characters by the scanner) and yielding nodes resulting in an abstract syntax tree (AST). Sometimes one distinguishes between parse trees and AST's, where the former is more decorated than necessary while the latter is minimal. Two examples from Narcissus are groups (parentheses in expressions, used for overriding operator precedence) and semicolons (used to terminate expressions and turn them into statements, grammatically). Neither of these are normally of value for evaluating a tree (abstract or not) however Narcissus keeps them in the

AST. Since a richer AST should be easier to map onto (and update from) modified source code, this might be beneficial.

4.1.1 AST structure

The AST is made up by a series of nested JavaScript objects (each node being one distinct JavaScript object). Each node is augmented with type and corresponding source code (indices and tokenizer reference) as well as type specific information and references to its children, if any.

Consider the small greatest common divisor program that follows below, with its corresponding AST in JavaScript object notation and graphical visualization found in Appendix B. The AST has been stripped from some information for readability. Note that the AST has sufficient information to be able to recreate the source code in text format (apart from whitespace and comments) via deparsing, if so necessary. To facilitate bringing local variables and functions in scope, each `<SCRIPT>` node has a `funDecls` and `varDecls` property which are simply lists of references to all declarations inside the `<SCRIPT>`, where each function has a `<SCRIPT>` node as body (think of the obligatory curly braces for functions as being the `<SCRIPT>`) and the main program in itself being a `<SCRIPT>` too. The example shows that the top `<SCRIPT>` node has `a` and `divisor` as `varDecls` and `gcd` as `funDecls`. Note that the declarations can be on any block level (as for inside `if`-statements and such), it's only the parent function (if any) that matters. The function `gcd()` has no local variables or functions aside from the parameters, which can be found in the `params` list.

I extended Narcissus' parser giving all nodes two more properties: `parent` and `parentprop`. The first being a reference to the parent node and the second indicating from which property (such as: `'body'`, `'expression'`, `'0'` and `'1'`) in the parent the node is referenced. Thus, for any[1] node in a valid AST `node.parent[node.parentprop]` equals `node`. The effect of this is that the AST can be conveniently traversed upwards and a node can easily link itself out of the AST.

[1] the exception being the root node, whose parent is `null`

GCD source code

```
function gcd(i,j) {
    while (i!=j) {
        if (i > j)
            i = i-j;
        else
            j -= i; // note the AST difference compared to j = j-i
    }
    return i;
}

var a = 9, divisor;
println(divisor = gcd(a,12));
```

4.2 Evaluator

4.2.1 Implicit callstack

The core of the Narcissus evaluator is the `execute()` function, which basically is a switch-case for the different AST node types. The evaluation proceeds by further recursive calls to `execute()`, on different nodes (`n`) and possibly different execution contexts (`x`). The function `getValue()` is extensively used to convert identifier references to their bound values, by searching through the lexical scope chain. The underlying stack holds the current node and its necessary state for continued evaluation, often implicit in return addresses. Consider the tiny example below, presented as JavaScript source code along with the corresponding AST in object and graphical representation.

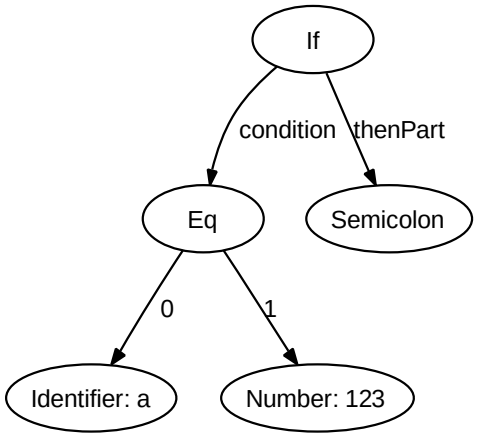
Example JavaScript source code

```
if (a==123)
    ;
```

AST object representation

```
{
    type: IF,
    condition: {
        type: EQ,
        0: {
            type: IDENTIFIER,
            value: a
        },
        1: {
            type: NUMBER,
            value: 123
        }
    },
    thenPart: {
        type: SEMICOLON,
        expression: null
    }
}
```

AST graphical representation



The relevant parts of `execute()` for evaluating the example follows.

```
function execute(n, x) {
    ...
    switch (n.type) {
        ....
      case EQ:
        v = getValue(execute(n[0], x)) == getValue(execute(n[1], x));
        break;
        ...

      case IF:
        if (getValue(execute(n.condition, x)))
            execute(n.thenPart, x);
        else if (n.elsePart)
            execute(n.elsePart, x);
        break;
        ...

      case NUMBER:
        v = n.value;
        break;
        ...

      case SEMICOLON:
        if (n.expression)
            x.result = getValue(execute(n.expression, x));
        break;
        ...
    }
    return v;
}
```

The trace of execution follows (operands are evaluated left to right).

```
execute(<IF>, x)
    execute(<EQ>, x)
        execute(<IDENTIFIER>, x)
            returns <reference>
        execute(<NUMBER>, x)
            returns 123
        returns true
    execute(<SEMICOLON>, x)
        returns undefined
    returns undefined
```

Passing the node as a formal parameter to `execute()` is necessary but passing the current execution context is a bit redundant as it's unlikely to change between each execution call. An explicit separate execution context stack could have been used to remedy that. Another source of stack usage are the the local variables inside `execute()`, since the lack of block local scope forces all variables to be visible in every case branch. Temporarily partial results from expressions will also be stacked up, such as the left hand side in the `<EQ>` case above. Some form of return address must also be saved.

A recursive evaluator will need to rely on some form of control flow breaking mechanism in the underlying implementation language to support early return from functions, break/continue from loops and throwing exceptions. Narcissus uses exceptions heavily for this, wrapping `try{} catch{}` around the call to `execute()` of any type that may break control flow. In a C implementation `setjmp()` and `longjmp()` would typically be used instead.

While the stack usage is a bit more excessive than necessary, that's not the main problem (Narcissus wasn't designed for raw speed). To support Live programming we need facilities to explicitly transform the callstack, and as explained most of that information is (explicit or implicit) buried on the underlying stack and impossible to reach. This means that the evaluator needs to be wrapped inside out, exposing every single element that has relevance to the program execution on an explicit callstack. The evaluator will turn iterative instead of recursive, which will cause some pleasant

side effects such as pause and resume of the program while retaining state. Breakpoints and singlestepping will thus come for free, independent of underlying support.

4.2.2 Explicit callstack

To create an explicit callstack instead of the underlying implicit one, the parameters and return values as shown in the trace above need to be lifted from the underlying implementation and explicitly put into a data structure, consisting of a list of successive stack frames (using a resizable list for each frame). Each frame corresponds to the explicit and implicit data for each recursive `execute()` call. Assuming that the plan is to put the current execution context on the stack only when it's needed, the basic structure of the callstack is:

```
[<NODE1>, arg1_1, arg1_2 .. arg1_n1] ->
    [<NODE2>, arg2_1, arg2_2 .. arg2_n2] ->
    .. ->
    [<NODEm>, argm_1, argm_2 .. argm_nm]
```

Each stack frame is enclosed in brackets, the entire callstack being the series of stack frames, separated by `->` for clarity. The first element is a reference to the evaluating node (corresponding to `n` in `execute()`) and the rest are arguments returned from evaluation of sub expressions or other necessary data for keeping state. The function call `execute(node,x)` is equivalent to creating the new frame `[<NODE>]` and pushing it on the callstack. Return (with value) from `execute()` is equivalent to popping the top frame (the callee) of the callstack and appending the return value (if any) to the new top frame (the caller). The evaluation continues iteratively on the top (current) frame.

A translation of the trace for the example program into the explicit callstack structure follows.

```
[<IF>]
[<IF>] -> [<EQ>]
[<IF>] -> [<EQ>], [<IDENTIFIER>]
[<IF>] -> [<EQ>, 123]
[<IF>] -> [<EQ>, 123] -> [<NUMBER>]
[<IF>] -> [<EQ>, 123, 123]
[<IF>, true]
[<IF>] -> [<SEMICOLON>]
[<IF>]
--empty callstack--
```

Apart from not sending the execution context into each frame, statements never return anything (thus `<SEMICOLON>` doesn't put `undefined` into `<IF>`'s frame. The implicit recursive `execute()` did return `undefined` for any statement, this was hardly by purpose but rather a side effect of the implementation language (JavaScript functions returns `undefined`, if nothing else is specified).

The callstack data structure has one more property besides the list of frames, a reference to the node inside the most recently removed frame, `prevn`. When `prevn` is `null` it indicates that the current frame *just got pushed*, otherwise it indicates which node the evaluator *just returned from*.

The relevant parts (for the tiny example) of the rewritten evaluator follows. The data structure `opReferenceArg` is further explained in section 4.2.3.

```
function pushframe(node) {
    callstack.prevn = null;
    callstack.push([node]); // create and push new frame
}
function popframe() {
    callstack.prevn = callstack.pop()[0]; // popframe
}
function popframe_pusharg(value) {
    if (callstack.length>1) {
        var retframe = callstack[callstack.length-2];
        var refArg = opReferenceArg[retframe[0].type];
        if (refArg == undefined || retframe.length == refArg) {
```

```
            value = getValue(value);
        }
        else if (opValidateReference[retframe[0].type] != false) {
            validateReference(value); // may trigger an exception
        }
        retframe.push(value); // pusharg
    }
    callstack.prevn = callstack.pop()[0]; // popframe
}

while (callstack.length>0) {
    frame = callstack.top();
    n = frame[0];
    prevn = callstack.prevn;
    ...
    switch (n.type) {
        ....
      case EQ:
        if (prevn == n[1]) {
            popframe_pusharg(frame[1] == frame[2]);
        }
        else {
            pushframe(n[prevn == null ? 0 : 1]);
        }
        break;
        ...

      case IF:
        if (prevn == null) {
            pushframe(n.condition);
        }
        else if (prevn == n.condition) {
            if (frame.pop()) {
                pushframe(n.thenPart);
            }
            else if (n.elsePart) {
                pushframe(n.elsePart);
            }
            else {
```

```
                popframe();
            }
        }
        else {
            popframe();
        }
        break;
        ...

      case NUMBER:
        popframe_pusharg(n.value);
        break;
        ...

      case SEMICOLON:
        if (prevn == null && n.expression) {
            pushframe(n.expression);
        }
        else {
            popframe();
        }
        ...
    }
}
```

If the callstack data structure had not been extended with the `prevn` property, the example callstacks as translated from the sample trace could not have resulted in a terminating program. Consider the callstack when the `<IF>` node is the most current. These three occurrences correlate to the states "just got pushed" (first `[<IF>]`), "returned from condition" (`[<IF>, true]`) and "returned from then- or else part" (last `[<IF>]`). The first and last of these frames looks exactly the same, yet it's possible to distinguish between them thanks to the `prevn` property.

Without `prevn` the frame would need to be augmented with more data instead, where the "returned from then- or else part" state could correlate to (`[<IF>, null, null]`). The length could then be used to distinguish between the three states. Another option would be to insert a state counter

that the frame has (and updates) for its lifetime. The `prevn` solution is preferred since it is the least redundant and improves consistency between frames of different node types.

The evaluation of `<EQ>` consist of the states "just got pushed" (`[<EQ>]`), "returned from left expression" (`[<EQ>, 123]`) and "returned from right expression" (`[<EQ>, 123, 123]`). These three states can be distinguished using either `prevn` (as shown in the evaluator above) or the frame length (as shown below).

```
if (frame.length < 3) {
    pushframe(n[frame.length-1]);
}
else {
    popframe_pusharg(frame[1] == frame[2]);
}
```

4.2.3 Reference and value arguments

A modification to the usage of the `getValue()` procedure has been made so that a reference gets converted to its bound value (when it should) by the callee instead of the caller, this makes the callstack more accurately reflect the program and is convenient for human inspection, but it isn't strictly necessary otherwise. It does make the code shorter and less error-prone though, since most of the sprinkled `getValue()` calls gets replaced with the `opReferenceArg` data structure and some code inside `popframe_pusharg()`.

```
var opReferenceArg = {
    // i == -1: all arguments are reference arguments
    // i != -1: frame[i] is a value argument,
    //          others are reference arguments
    // i == undefined: (default) all arguments are value arguments

    NEW: -1, NEW_WITH_ARGS: -1, DOT: -1, CALL: -1, DELETE: -1,
    TYPEOF: -1, INCREMENT: -1, DECREMENT: -1, GROUP: -1,

    INDEX: 2, ASSIGN: 2
};
```

References are created by evaluation of <IDENTIFIER>, <DOT> or <INDEX> nodes, for instance to enable assignments with any of these as the lvalue. Most nodes want their arguments converted to values (if the argument came from a reference), such as <IF> and <EQ> in the example above. The exceptions can be found in opReferenceArg above. An example of how such nodes evaluate follows with the prefix and postfix <INCREMENT> and <DECREMENT> operators, note that the delaying of getValue() until the else clause is crucial, otherwise no reference would be available for assigning the result (via putValue()).

```
case INCREMENT:
case DECREMENT:
    if (prevn == null) {
        pushframe(n[0]);
    }
    else {
        t = frame.pop();
        u = Number(getValue(t));
        if (n.postfix)
            v = u;
        putValue(t, (n.type == INCREMENT) ? ++u : --u, n[0]);
        if (!n.postfix)
            v = u;
        popframe_pusharg(v);
    }
    break;
```

4.2.4 Function and constructor calls

Instead of pushing the execution context into every frame as the recursive execute() does, it gets pushed only for <SCRIPT> frames, which are created for the top script and each function call.

Evaluating <RETURN> is now possible without underlying exception support (thanks to the explicit callstack) by searching the callstack for the most recent <CALL> stack frame, and transferring the return value there (bypassing the <SCRIPT> frame in between). The saved execution context gets restored, as well.

An example on how the normal control flow is skipped by rollbacking parts of the callstack is shown below. The same concepts apply for all similar statements: `return`, `break`, `continue` as well as for exceptions (`try`, `catch` and `finally` paired with `throw` statements).

```
function double(x) {
    return 2*x;
}
double(42);
```

Callstack before and after return statement:

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL>] ->
    [<SCRIPT>, <context>] -> [<RETURN>, 84]
```

⇓

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL> 84]
```

4.3 Active/passive terminology

Any node in the AST is either active or passive. Consider the `gcd` program, where we interrupt just after `<GT>` pushed its second operand, the `<IDENTIFIER> j` for evaluation. The callstack is:

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL>, println] ->
    [<LIST>] -> [<ASSIGN>, divisor] -> [<CALL>] ->
    [<SCRIPT>, <context>] -> [<WHILE>] -> [<BLOCK>] -> [<IF>] ->
    [<GT>, 9] -> [<IDENTIFIER>]
```

Each frame in the callstack has a reference to the corresponding AST node and necessary state information. All these nodes are considered *active* since they have begun evaluating, but not yet finished. All the others are considered *passive*. For our interrupted example program `<NE>` is passive (though it was active recently, before `<WHILE>` pushed its body `<BLOCK>`) and `<GT>` is active (as are many other nodes). `<GT>` has two children, `<IDENTIFIER> i` and `j`, of which the first has already been evaluated (that's

where the 9 came from) and is passive, the second being pushed on top on the callstack right now, thus active.

An active node can have none, some or all of its children active simultaneously. Consider this example, where both the true (<MUL> node) and false (NUMBER node) expressions for the ternary ?: operator (CONDITIONAL node) are active, through recursion.

```
function faculty(n) {
    return n>1 ? n*‘faculty(n-1)‘ : ‘1‘;
}
‘faculty(2)‘;

[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL>] ->
    [<SCRIPT>, <context>] -> [<RETURN>] -> [<CONDITIONAL>] ->
    [<MUL>, 2] -> [<CALL>] -> [<SCRIPT>, <context>] -> [<RETURN>] ->
    [<CONDITIONAL>] -> [<NUMBER>]
```

With one exception all descendants to a passive node are passive and all ancestors to an active node are active. The parent to an active <SCRIPT> node is not necessarily active but the <CALL>[2] node from which the function was invoked is, or the other way around a <SCRIPT> node may be active even though its parent (a <FUNCTION> node typically) is passive. The <CALL> and <SCRIPT> nodes are runtime related (caller and callee) but typically not immediately related in the AST. If the <SCRIPT> node represents the top level script instead of a function body the parent is always null.

```
if (true) {
    function faculty(n) {
        ‘return n>1 ? n*faculty(n-1) : 1;‘
    }
}
‘faculty(2)‘;

[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<CALL>] ->
    [<SCRIPT>, <context>] -> [<RETURN>]
```

[2] can be <NEW> or <NEW_WITH_ARGS> also, actually

4.3.1 Source code correspondence

Surrounding backticks will sometimes be used to signify that a statement or an expression in the textual source code is active. All active statements or expressions won't be put inside backticks, for readability reasons. As stated before all ancestors to an active node are active as long as function boundaries aren't crossed, thus `(‘a‘*b-7/4)+2;` implies `(‘a*b‘-7/4)+2;`, `(‘a*b-7/4‘)+2;`, `‘(a*b-7/4)‘+2;`, `‘(a*b-7/4)+2‘;` and `‘(a*b-7/4)+2;‘`. Typically all active `<CALL>`[3] nodes and the topmost node on the callstack will be backticked, covering all function calls as well as the current position, as in the two previous `faculty` examples.

[3] `<NEW>` or `<NEW_WITH_ARGS>` applies here, too

Chapter 5

Source code transformations

Source code transformations are the basis of Live programming since they define how changes can be made to the program, in an incremental fashion. The six basic classes of these operations are insert, delete, replace, expand, reduce and swap.

The operations are shown below exemplified with JavaScript source code. Most of the transformations are likely to be applicable to any imperative or functional programming language, some would require less and others more.

5.1 Insertions

A fundamental source code transformation is the insertion of new expressions or statements into an active or passive sequence.

5.1.1 Insertion into an active block sequence

```
while (true) {
    ‘f(x)‘;
    x++;
}
```

⇓

```
while (true) {
    if (x<10)
        break;

    ‘f(x)‘;
    x = update(x);
    x++;
}
```

The active function call got prepended with an `if` statement and appended with an assignment. A `<BLOCK>` is a sequence of strictly ordered statements and has no return value, the block is a statement itself. Evaluation will continue with `x = update(x);` once `f(x);` returns and the prepended statement won't evaluate until next invoke of the `<BLOCK>`.

5.1.2 Insertion into an active comma sequence

```
x = 0.707, 3.14, ‘square(3)‘;
```

⇓

```
x = 0.707, sqrt(2), 3.14, ‘square(3)‘, 2>1;
```

Comma sequences are the equivalent of block sequence with expressions instead of statements. The comma sequence is an expression itself and returns the value of the last expression. An insertion is thus handled similarly, evaluation continues with the next expression in the updated sequence (`sqrt(2)` won't evaluate).

5.1.3 Insertion into an active variable declaration sequence

```
var x=1, ‘z=square(3)‘;
```

⇓

```
var x=1, y=sqrt(2), ‘z=square(3)‘, w=2>1;
```

As for comma sequences `y=sqrt(2)` won't run. The variable `y` will be brought into scope though, with value `undefined`. More about this in section 5.7.

5.1.4 Insertion into an active array initializer

```
x = [0.707, 3.14, ‘square(3)‘];
```

⇓

```
x = [0.707, sqrt(2), 3.14, ‘square(3)‘, 2>1];
```

An array initializer can't be handled equivalently to block and comma sequences, if it would the example would result in `x = [0.707, 3.14, 9, true]`. An slightly better variant is to skip evaluation of the prepended expressions (`sqrt(2)` in the example) but return the array in the correct length and with the evaluated expressions bound to the updated indices and `undefined` for the others, `x = [0.707, undefined, 3.14, 9, true]`. Better still, couldn't `sqrt(2)` run as soon as `square(3)` returns, `x = [0.707, 1.4142, 3.14, 9, true]`? This do change the strict left to right evaluation order of the expressions but is more likely to produce the result the programmer actually intended with the source code transformation.

5.1.5 Insertion into an active object initializer

```
x = {a: 0.707, c: 3.14, d: ‘square(3)‘];
```

⇓

```
x = {a: 0.707, b: sqrt(2), c: 3.14, d: ‘square(3)‘, e: 2>1};
```

Object initializers are handled just as array initializers, the result is `x = {a: 0.707, b: 1.4142, c: 3.14, d: 9, e: true}` if the evaluation order is relaxed or `x = {a: 0.707, b: undefined, c: 3.14, d: 9, e: true}` otherwise.

5.1.6 Insertion into an active arguments list

```
x = f(0.707, 3.14, ‘square(3)‘);
```

⇓

```
x = f(0.707, sqrt(2), 3.14, ‘square(3)‘, 2>1);
```

The resulting call is `f(0.707, 1.4142, 3.14, 9, true)`, same behaviour as for array initializers. Note that a function call that has already been dispatched (the function reference and all arguments have been evaluated and the body of the called function has started evaluating) isn't affected by changes to the argument list.

5.1.7 Insertion into a parameter list of an active function

```
function f(x, y) {
    return ‘x*y‘;
}
‘f(1, 2, 3, 4)‘;
```

⇓

```
function f(x, a, y, b) {
    `...`
}
`f(1, 2, 3, 4)`;
```

JavaScript binds as many evaluated arguments as possible to the parameters of the called function. The full list of arguments are available for the callee by the special `arguments` variable, an array like[1] object.

The initial call resulted in `x=1`, `y=2` and `arguments=[1,2,3,4]` (or rather `arguments={0:1, 1:2, 2:3, 3:4}` since it's an object, not an array). The source code transformation doesn't affect neither `x` and `y` (which may have been re-assigned since) nor `arguments` (which reflect the caller arguments, not the callee parameters). The parameters `a` and `b` will be brought into scope though, with value `undefined`. More about this is in section 5.7.

5.2 Deletions

Deletions are the direct opposite to insertions, having defined how insertions work deletions should be fairly obvious. Note that only passive statements or expressions can be deleted unless a rollbacking replace (see below) precedes the operation.

5.3 Replace

The replace source code transformation removes an active or passive statement and replaces it with another, while retaining as much callstack information as possible.

[1] Interested readers can investigate the Narcissus source code to find out how Brendan Eich, the father of JavaScript, feels about this decision. Hint: search for "curse ECMA".

5.3.1 Replace an active statement or expression

```
function f(x) {
    return ‘x>3‘ ? 10 : 0;
}
var c = g(a)+(5-a)/‘f(b)‘;
```

⇓

```
function f(x) {
    return x>3 ? 10 : 0;
}
var c = g(a)+‘10*b‘;
```

For replacing an active statement or expression a partial rollback (possibly following through function calls) of the callstack is necessary, discarding state and data corresponding to the removed parts of the source code. In the example the whole division expression got replaced (with a multiplication), but the rollback didn't need to erase the already evaluated value returned from `g(a)` (which return value is hanging on the callstack as the left operand to the addition). Evaluation will continue with the multiplication, directly after the transformation.

Common simple expression replacements are numbers or identifiers (for variable or function lookup), since they are either self or atomically evaluating no rollback (or rollback of one step, depending on approach) is ever needed. Three examples follow, the first exemplifying a misspelled identifier that triggered a runtime exception which the user has then corrected (while retaining callstack). The second and third exemplifies that whole statements can be replaced, as well.

```
var c = f(b)*‘valeu‘;
```

⇓

```
var c = f(b)*‘value‘;
```

```
if (a>b)
    return ‘a*3‘;
```

```
   ⇓

if (a>b)
    `break`;



if (a>b) {
    `c()`;
}

   ⇓

`a -= b`;
```

5.3.2 Compatible replace

```
var d = f(a) - g(b)*`h(c)`;

   ⇓

var d = f(a) - g(b)/`h(c)`;
```

The return value from `f(a)` and `g(b)` are both hanging on the callstack, waiting for `h(c)` to return. The multiplication is now replaced with a division. The normal replace transformation would include a callstack rollback leaving only the left operand of the substraction intact, however since multiplication and division are compatible (they are both binary operators) the operands can be transferred from the old operator into the new one, thus the value of `g(b)` remains and `h(c)` remains active.

5.3.3 Replace a passive statement or expression

Replacing passive statements or expressions is an easier subcase of replacing active, since the callstack never needs to be altered.

A special case that might be a bit confusing is the addition of a statement or expression that was previously empty, i.e. adding an `else`-clause to a `if` statement or an expression to a `return` statement (both of which

are done in the example that follows). They can be mistaken for insertion into sequence or replace active statement or expression but are both examples of replacing passive statement and expression.

```
if (a>=10)
    ‘return‘;

   ⇓

if (a>=10)
    ‘return a‘;
else
    a++;
```

5.4 Expand

Expanding is similar to replacing except we wish to retain and expand (instead of rollbacking) the callstack and internal program representation (AST, bytecode or whatever chosen).

5.4.1 Expand an active statement or expression

Examples follows.

Wrap parentheses around expression:

```
a = g(x)*‘f(x)‘+1;

   ⇓

a = (g(x)*‘f(x)‘)+1;
```

Note that the parentheses must comply to the already implicit evaluation order (due to operator precedence). Had the parentheses been placed as `a = g(x)*(f(x)+1);` instead, this would have corresponded to an reduction (see section 5.5) to `a = g(x)*‘f(x)‘;` followed by an expansion to `a = g(x)*(‘f(x)‘+1);`. Note that the left operand of the multiplication remains intact on the callstack during the reduction and expansion, keeping `f(x)` active.

Wrap block around statement:

```
if (b)
    a = f(c)-‘3*b‘;
```

⇓

```
if (b) {
    a = f(c)-‘3*b‘;
}
```

Wrapping a block around a single statement is commonly used to accommodate the insertion of more statements for the conditional or loop.

Wrap conditional or loop around statement:

```
a = f(c)-‘3*b‘;
```

⇓

```
while (a>10) {
    a = f(c)-‘3*b‘;
}
```

In this example both a while statement and a block statement got wrapped around the assignment.

Add unary operator to expression:

```
f(c)-(d*‘f(b)‘)
```

⇓

```
f(c)-~(d*‘f(b)‘)
```

The function call `f(b)` is currently evaluating and there are two values (left side operands to substraction and multiplication) hanging on the callstack. The multiplication got expanded with a bitwise negation, the intended result is to finish the active function call, multiply the two operands (nothing new here) but afterwards negate the result before subtracting.

Add binary operator to expression:

```
f(c)-‘3*b‘;
```

⇓

```
f(c)-‘3*b‘/d;
```

This example is similar to adding an unary operator. It's obvious that once `3*b` has finished evaluating, `d` should evaluate and the two results be divided, continuing on with the substraction.

Inserting a binary operator with a new left side value complicates the situation, as for `f(c)-d/‘3*b‘;`. The division can't resume as usual once `3*b` has finished, since the left side operand isn't available.

Some languages has unspecified evaluation order of operands. The C language is one of those [11] with three obvious exceptions (the operands to `&&`, `||` and `?:`). Others, such as Java [14] and JavaScript [10], specifies a strict left to right evaluation order, for good and bad[2].

If the left to right evaluation order is to be strictly respected, then the only choice is to either forbid this transformation (forcing a rollbacking replace instead of expand) or to consider the left operand evaluated with an `undefined` value. For most cases it's simply better to relax the evaluation order and allow operators to calculate their operands in arbitrary order[3], enabling the full use of the transformation.

5.4.2 Expand a passive statement or expression

Expanding passive statements or expressions is an easier subcase of expanding active, since the callstack never needs to be altered.

[2] making life harder for optimizers and specification compliant Live programmers
[3] excluding `&&`, `||` and `?:` just as C does, for obvious reasons

5.5 Reduce

Reducing is the opposite to expanding. It corresponds to throwing away information from the callstack and the internal program representation, though not more than necessary.

5.5.1 Reduce an active statement or expression

Examples follows.

Reduce expression:

```
a = f(c)-‘3*b‘;

   ⇓

a = ‘3*b‘;
```

The reduction of the substraction operator results in the already calculated return value from `f(c)` being obsolete and removed from the callstack. Reducing operators that have more than one operand generally results in exactly one of them remaining (the programmer needs to select which operand to reduce the operator into).

Reduce statement:

```
if (f(c)-‘3*b‘)
    g(a);

   ⇓

f(c)-‘3*b‘;
```

A statement can be reduced to any of its inner expressions or statements (an `if` statement has a `condition` expression, a `then` statement and an optional `else` statement). Note that it's the outer statement that is active, not necessarily one of the inner (the same applies for reducing expressions).

Note that a statement that gets reduced into an expression, as in the example, most likely need to be turned into an statement again by adding an extra semicolon. There is one exception to this in JavaScript, the initializing part of the `for` expression, which in most of the cases is an expression but is allowed to be a `var` statement instead. The reduction of `for (var z=‘f(x)‘ ...);` into `for (‘f(x)‘ ...);` is thus legal.

5.5.2 Reduce a passive statement or expression

Reducing passive statements or expressions is an easier subcase of reducing active, since the callstack never needs to be altered.

5.6 Swap

Expressions or statements should be movable within the same function (or top-level script). There are two obvious choices, either let the source overwrite the destination and put a placeholder in source's original place (this would force destination to be passive), or instead swap source and destination (any or both of source and destination can be active). Swap is a superset of move, thus explained further.

Swapping is similar to reduction followed by an expansion, for both the swapped statements/expressions (if active).

5.6.1 Swap active statements or expressions

Consider that one statement/expression is active and one passive, as in the example below. After the swap (between `-` and `1`) and once `sqrt(4)` has returned and the substraction been performed, the evaluation will continue in the array initializer, just as if the functions had been called from there originally. The function `f` will return `[33, 7, 2]`.

```
function f(z) {
    z += square(3)-‘sqrt(4)‘;
    if (z>100) {
        return [11*3, 1, 2];
    }
}
```

```
function f(z) {
    z += 1;
    if (z>100) {
        return [11*3, square(3)-‘sqrt(4)‘, 2];
    }
}
```

The swapped statements/expressions need not to be direct descendant to each other, blockwise. When an active expression/statement is to be moved inside conditional blocks such as `if` and `while`, it shall be though of as the branch/loop was taken based on the condition was true (though the condition was never checked). That means that the `while` loop in the following example will terminate upon its next iteration (when it will actually check its condition), the `if` branch will terminate without the `else` part being considered according to `if` semantics – the `then` part has been evaluated, after all.

```
function f(z) {
    if (false) {
        while (false) {
            0;
        }
    }
    else if (z>100) {
        z += ‘g(z/0.5, 101)‘;
    }
}
```

```
function f(z) {
    if (false) {
        while (false) {
            ‘g(z/0.5, 101)‘;
        }
    }
    else if (z>100) {
        z += 0;
    }
}
```

Some swaps would be illegal unless one of the swapped statements/expressions is wrapped into parentheses, due to operator precedence as discussed in section 5.4. Consider the added parentheses in the following example demonstrating a swap between the `+` and `y`.

```
c = f(a)+‘f(b)‘-1;
z = x*y;
```

⇓

```
c = y-1;
z = x*(f(a)+‘f(b)‘);
```

5.6.2 Swap passive statements or expressions

Swapping two statements/expressions that are both passive is an easier subcase of swapping where one or both are active, since the callstack never needs to be altered.

5.7 Updating scope

JavaScript is lexically scoped, the visible variables and functions can thus be determined for any location in the source code. JavaScript has block local scope, as discussed in section 3.4 so all variable (function parameters included) and named function declarations are visible anywhere from the function (or top level script) they are declared in.

As soon as a source code transformation results in new such declarations the scope must be updated for all running instances on the callstack. All transformations besides swap are potentially affected. These changes come from new `var` statements, insertions into existing `var` statements, new named `function` declarations and insertions into parameter lists of active functions. The new variables (and parameters) are created with value `undefined`, the new named function declarations are bound to the function itself.

How should removed variables and functions be handled? Most of the classes of operations described in this chapter can result in deleted variables, and they should be supportable by removing declarations from scope in the same way as bringing them in.

Closures can lead to some problems when it comes to modifying scope. Consider the bank account example from section 3.6. The returned closure from `make_account()` can be stuffed away in a variable and be kept unused for a while. Imagine that two source code transformation are made as shown below, the insertion of a `doubleup()` function and a corresponding inserted `if` statement inside the dispatcher. The modification will work just fine for new accounts, but that stuffed away account will unfortunately error out when `doubleup` is to be returned, with a `ReferenceError: doubleup is not defined`. This is because `balance`, `deposit` and `withdraw` were all captured at the time the dispatch function was created. Added declarations only get added to *the running instances (of the dispatcher) on the callstack* (and to those run after the transformation, obviously), thus the problem. There are two solutions to the problem, either keep track of all closures (function objects) created, by means of an list of weak[4] references for each function node. The alternative is to update the scope lazily instead, by having each function invoke verifying its current scope against the expected (by the possibly updated lexical scope of the AST).

[4] they would need to be weak to not hinder garbage collection

```
function make_account(balance) {
    function doubleup() { // inserted function declaration statement
        return balance *= 2;
    }
    ...
    return function(m) {
        if (m == 'doubleup') // inserted if statement
            return doubleup;
        ...
    }
}
```

Chapter 6

AST and callstack transformations

The source code transformations are mapped into transformations of the AST and the callstack. An intermediary AST is often parsed from the new source code, the current AST is then modified (as to linking in nodes from the intermediary AST) and the root of the intermediary AST is released.

These transformations are visualised with graphs showing in parallel the intermediary and transformed (relinked current) AST's. Dotted nodes and edges indicate that the edge or node got removed. Note that the initial AST before the transform (current) isn't visualised separately since it's easily derivable from the transformed by following the dotted edges.

Some transformations can be made directly on the current AST, without any need for an intermediary AST. The node properties `parent` and `parentprop` are updated as necessary for all transformations and involved nodes. The example transformations in the following sections correlate to the JavaScript source code for the same examples in chapter 5.

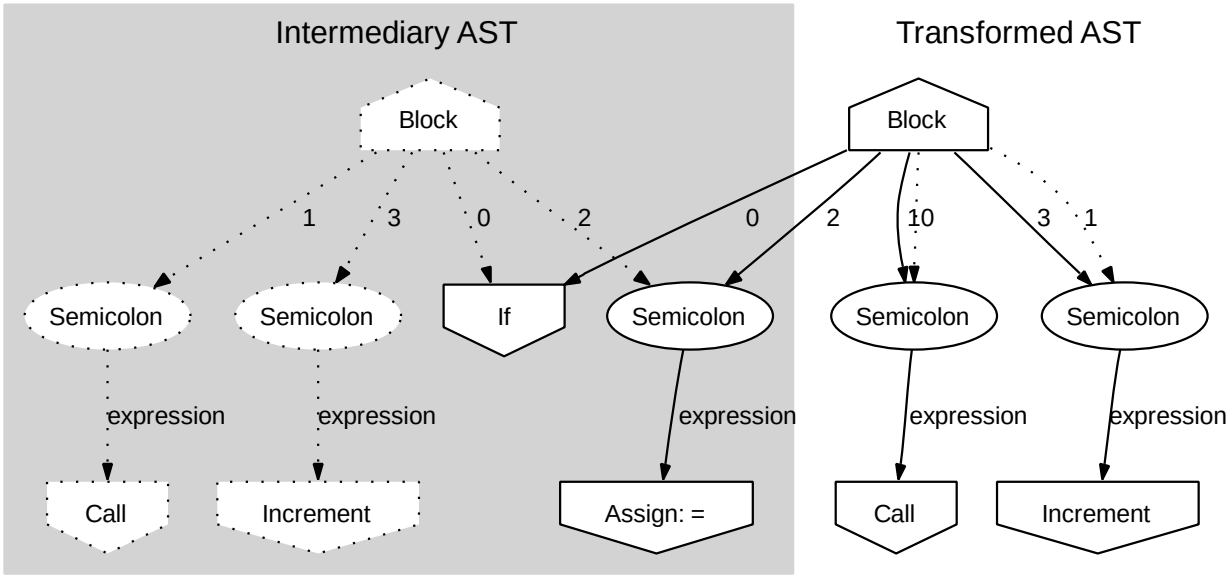
6.1 Insertions

6.1.1 Insertion into an active sequence

The same transformations apply for block, comma and variable declaration sequences. Insertion into an active block sequence are exemplified below.

AST transformation

The sequence holds a list of subnodes by the means of numbered edges, the insertion corresponds to creating new edges from the sequence node into nodes in the intermediary AST, as well as reordering the current edges as necessary.



Callstack transformation

The next node to evaluate from a block is determined by incrementing the previous node index (`parentprop`), thus no need to transform the call-stack even though `prevn` might have changed index upon return from it.

The code for evaluating `<BLOCK>` nodes follows, `<COMMA>` and `<VAR>` nodes follows the same pattern.

```
case BLOCK:
    i = (prevn != null) ? prevn.parentprop+1 : 0;
    if (i < n.length) {
        pushframe(n[i]);
    }
    else {
        popframe();
    }
    break;
```

6.1.2 Insertion into an active array initializer

The same transformations apply for object initializers and arguments lists as well.

AST transformation

The AST is transformed following the same procedure as for insertion into sequences, section 6.1.1.

Callstack transformation

The callstack is searched for the inserted array initializer's active frames. Note that the node can be active simultaneously in many frames, due to recursion.

The frame holds a reference to the AST node (as usual), an index to the first *prepended* unevaluated expression or `-1` if none and the evaluated array values so far. If any of these values are `nullish` (an unique value that can't be referenced from the user program) this indicates that the corresponding expression hasn't been evaluated. The following code shows how the array initializer continues evaluation based on this. Thanks to the index pointing out the first unevaluated expression there is no performance hit for evaluation of untransformed array initializers.

```
case ARRAY_INIT:
    if (prevn == null) {
        if (n.length > 0) {
            frame.push(-1);
            pushframe(n[0]);
        }
        else {
            popframe_pusharg([]);
        }
    }
    else {
        i = (prevn.parentprop-0);
        s = frame[1];
        frame[(i++) + 2] = frame.pop();
        frame[1] = -1;

        if (s != -1 && s < i) {
            j = s+2;
        }
        else {
            j = i+2;
        }
        for (; j<frame.length; j++) {
            if (frame[j] === nullish) {
                break;
            }
        }
        if ((j -= 2) < n.length) {
            pushframe(n[j]);
        }
        else {
            popframe_pusharg(frame.slice(2));
        }
    }
    break;
```

The callstack transformation corresponds to inserting `nullish` values into the list of evaluated expressions, if the inserted expression is prior to the

active one. The "first `nullish`" index is updated unless it's already less than the index of the inserted expression (indicating a previous insertion that hasn't been evaluated yet).

The unique `nullish` value is printed as tilde, ~.

```
... [<ARRAY_INIT>, -1, 0.707, 3.14] -> [<CALL>, ...] ...
```

⇓

```
... [<ARRAY_INIT>, 1, 0.707, ~, 3.14] -> [<CALL>, ...] ...
```

6.1.3 Insertion into a parameter list of an active function

AST transformation

The `params` property of the corresponding `<FUNCTION>` node holds a list of the parameter names and gets updated to reflect the added parameters.

Callstack transformation

The callstack is searched for running all instances of the function, matching active `<SCRIPT>` nodes to the functions `body` property. The new parameters are then inserted into scope with value `undefined`. No other callstack modifications are made.

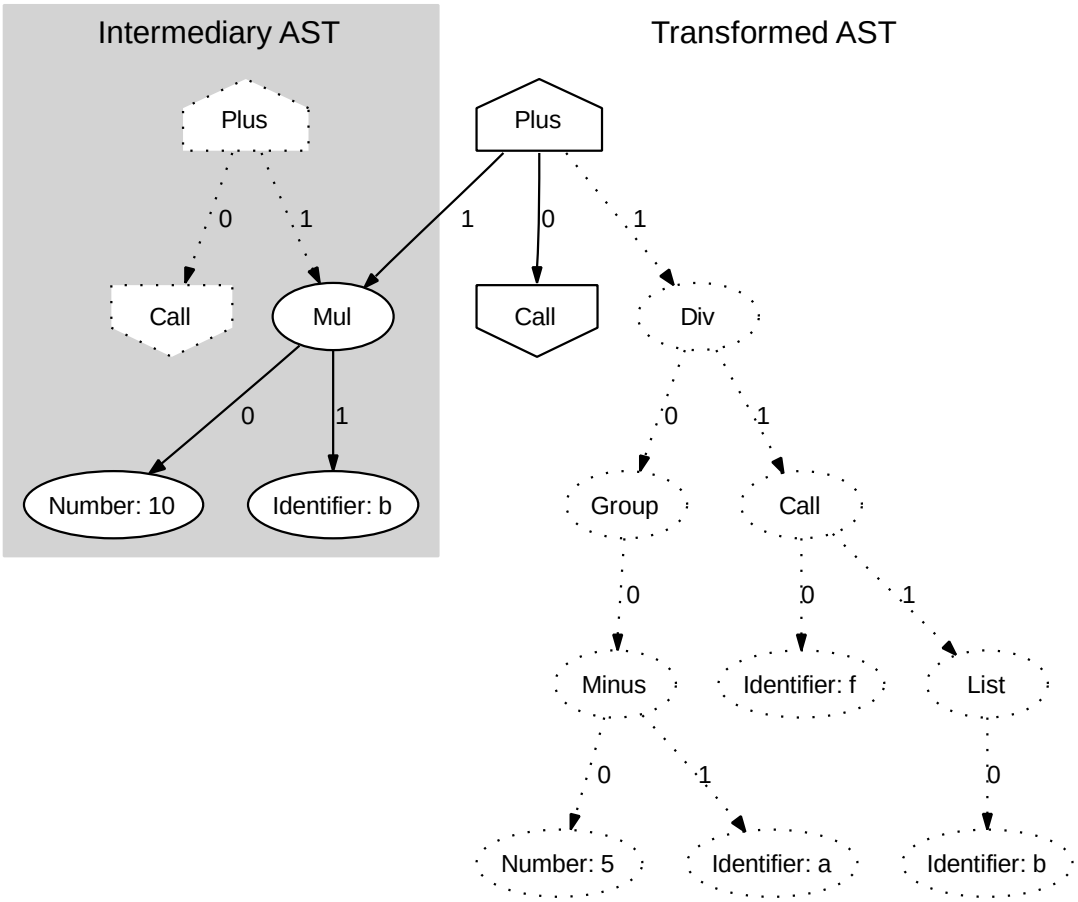
6.2 Deletions

Deletions are the direct opposite to insertions, having defined how insertions work deletions should be fairly obvious.

6.3 Replace

Replace corresponds to replacing an edge in the current AST with another pointing to a node in the intermediary AST.

AST transformation



Callstack transformation

The first frame that refers to the replaced node and all successive frames thereafter gets removed (a rollback). A new frame for the new node gets pushed instead.

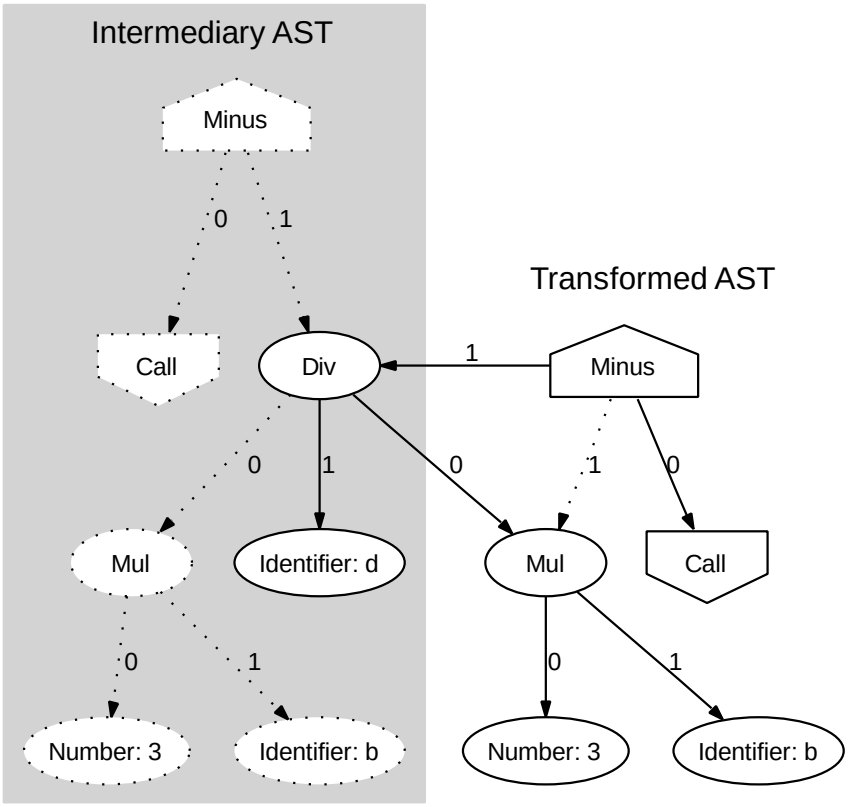
```
[<SCRIPT>, <context>] -> [<VAR>] -> [<PLUS>, 1, ~] ->
    [<DIV>, 4, ~] -> [<CALL>] -> [<SCRIPT>, <context>] ->
    [<RETURN>] -> [<CONDITIONAL>] -> [<GT>]
```

```
[<SCRIPT>, <context>] -> [<VAR>] -> [<PLUS>, 1, ~] -> [<MUL>]
```

6.4 Expand

Expand corresponds to replacing an edge in the current AST with another pointing to a node in the intermediary AST, similar to replace. The difference is that an edge inside the intermediary AST is replaced as well, pointing back into the transformed AST. No nodes are thus removed from the transformed AST. Exemplified below is adding a binary operator to an expression from section 5.4.

AST transformation



Callstack transformation

New generated frames need to be inserted into the framestack prior to the expanded node.

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<MINUS>, 2, ~] -> [<MUL>]
```

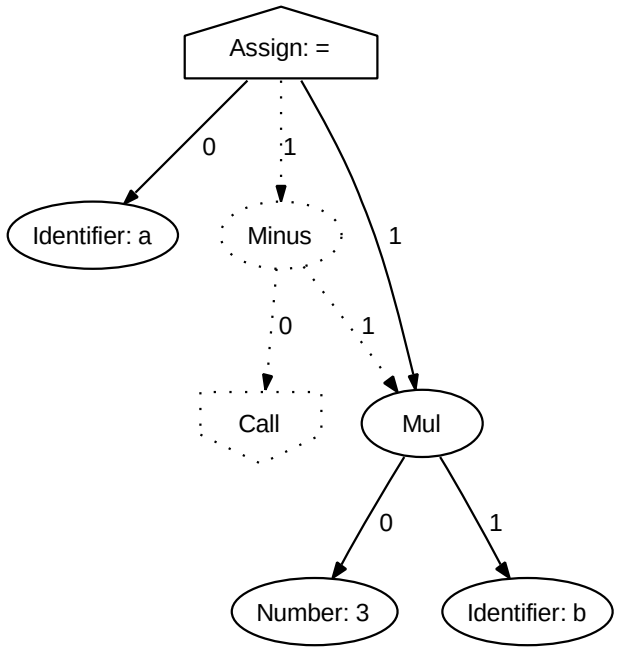
⇓

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<MINUS>, 2, ~] ->
    [<DIV>, ~, ~] -> [<MUL>]
```

6.5 Reduce

Reduce is performed directly on the current AST without need for an intermediary. An edge is replaced with another, pointing to an ancestor of the reduced node.

AST transformation



Callstack transformation

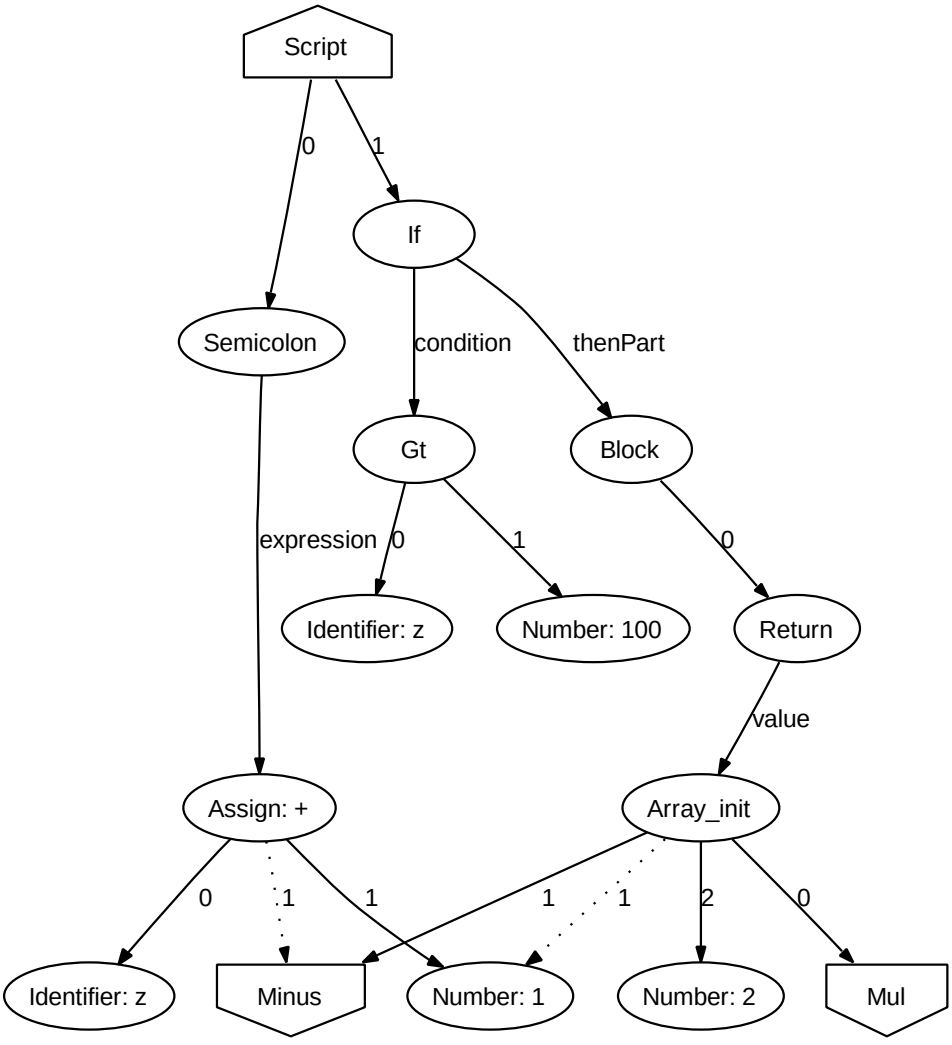
Frames corresponding to the removed nodes as a result of the reductions are removed from the callstack.

```
[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<ASSIGN>, a, ~] ->
    [<MINUS>, 2, ~] -> [<MUL>]

   ⇓

[<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<ASSIGN>, a, ~] ->
    [<MUL>]
```

6.6 Swap

As for reduce, swap is performed directly on the current AST without need for an intermediary. The two swapped nodes switch positions in the AST by swapping their incoming edges. None of the nodes may be an ancestor of the other.

AST transformation



Callstack transformation

An ancestor chain is generated for each of the nodes by traversing their `parent` references until a `<SCRIPT>` node is found. The ancestor chains are used to find the nearest common ancestor for the swapped nodes. The `<SCRIPT>` node representing the body of the function is the nearest common ancestor for the example. A reduction and expansion follows. The procedure is repeated twice if both nodes are active.

```
... -> [<SCRIPT>, <context>] -> [<SEMICOLON>] -> [<ASSIGN>, z, ~] ->
    [<MINUS>, 9, ~] -> [<CALL>]
```

⇓

```
... -> [<SCRIPT>, <context>] -> [<MINUS>, 9, ~] -> [<CALL>]
```

⇓

```
... -> [<SCRIPT>, <context>] -> [<IF>] -> [<BLOCK>] -> [<RETURN>] ->
    [<ARRAY_INIT>, -1, 33] -> [<MINUS>, 9, ~] -> [<CALL>]
```

Chapter 7

Related work

This chapter discusses related work in the form of programming language implementations and integrated development environments.

7.1 Stackless Python

Stackless Python [15] is a variant of the Python interpreter, "an experimental implementation that supports continuations, generators, microthreads, and coroutines". The author does a very similar implicit to explicit callstack modification that I describe in chapter 4. Python is written in C instead of in itself, but the concepts apply precisely the same – turn a recursive evaluator iterative and get rid of the borrowed underlying implicit stack. Christian Tismer had a different reason than mine, to enable continuations, microthreads and coroutines. Interestingly, the explicit access to the callstack remains the key for both his and mine objectives. One of the results he present in his paper is that the performance difference between the implicit and explicit callstack versions is negligible.

7.2 The "Lisp machines"

I've heard numerous stories about the mythical Lisp machines that seem to have been able to do almost everything besides making coffee. Unfortunately I haven't been able try one out myself[1], I have been able to get my hands on the Interlisp reference manual [16] though. It's clear that this system was very much about modifying an active program, I haven't decrypted how much it was about keeping as much as possible of the callstack (instead of rollbacking it) though. It's kind of natural that Lisp has been a source of experimenting with the concept (just as Smalltalk) since it has a very exact and consequent syntax, the source code *is* the AST. I would very much like to try the system for real, to see what actual similarities there are and what features and good ideas can be borrowed over – I'm sure there are plenty.

7.3 Smalltalk

I've tried one Smalltalk implementation (Squeak) while there are plenty of others. More actual experience with other implementations would be needed to cherry pick all the nice Live programming'ish features[2] that are available in Smalltalk land.

7.4 Visual Studio 2005

As a big surprise to me, see section 2.7, the latest version of Microsoft's IDE has taken Live programming much further than previous versions, and in some aspects further than most others[3]. It is great in its integration into their editor/debugger, the environment guesses what kind of source code transformation is meant (see chapter 5) without hints (apart from modified source code) and tends to do the right thing most of the time. The tests I performed showed that they do correctly maintain the callstack when moving recursed statements around. Most transformations are only

[1] I came as close as "there is one, but no keyboard"
[2] apart from 1+2*3 being 9
[3] as far as I know

possible on statement level, not expression level. It's possible to wrap an `if` statement around an active statement but illegal to add a multiplication around an active function call, for instance. It's a pity that much of the previous work (available in Smalltalk for decades), such as adding functions and manipulating function signatures in runtime is entirely absent. So Visual Studio is both leading and lagging behind, at the same time. What's perhaps most interesting of all is that they support these transformations for compiled static languages, not dynamic interpreted ones. I tried it using both C# compiled for their managed .NET platform and C++ compiled to machine code.

Chapter 8

Discussion

How can Live programming, if seen as one additional method or technique in the toolbox of the experienced programmer, be evaluated? Is it "good enough"? Is it "better"[1]? Depends on who you ask. As long as it's implemented into an existing language and development environment without getting in the way of the programmer, it won't be yelled at. This means that there should be a negligible performance hit, if any, for normal evaluation and debugging. The actual transformation operations can be a bit more costly if necessary. Those outlined in this report for a dynamic interpreted language can be considered negligible in cost. For a compiled language the actual transformation may take a bit longer due to more passes and more difference between source code and actual program internal representation. As long as it's about interactive changes done by the programmer, as long as 100-500 ms should be tolerable, plentiful for even the heaviest compiler under those circumstances.

Performance wise it comes down to (at least for interpreted languages): Can the implementation of the language be done with an explicit callstack at similar memory and cycle cost. While this wasn't my main focus with the report, I believe that I've showed that an explicit callstack potentially saves memory, at least. Other reports tell that the cycle cost should be

[1] better than what? (it isn't a replacement)

comparable. Rewriting an existing evaluator (which is tuned for performance to start with) and benchmarking the two will tell.

How much productivity can be gained from heavily use of Live programming? No idea. Somewhere between "nothing" to "plenty" is reasonable to expect. An survey could be made where programming problems were given to "similar skilled" programmers, using either the normal editor and debugger combination or the described Live programming environment. One could look at the time it took, the completeness and efficiency of the solutions. Perhaps more valuable survey data are collected by letting the same programmer solve the same problems in both environments, and see if he/she felt more productive in any of them.

As long as there are no drawbacks, the question shouldn't be "why?" but rather "why not?".

8.1 The prototype environment

I built a simple environment hosted inside the web browser based on the work in this report. I had no decorations on the source code other than normal characters (actually, I color the source code corresponding to the top most active node, but I never rely on it to track changes). The programmer change the text and press "commit" to have the program updated. This means that some source code transformations can be mixed up, and the programmer explicitly needs to tell the environment by marking parts of the source code with another color, to tell that something was inserted for instance. This is a problem that is solvable by attaching node information to ranges of source code, making it easy to track which code was added, changed or moved.

Apart from the defined transformations, the environment handles the usual debugging facilities such as step into, step over, step out and breakpoints. The environment can also move the current program position anywhere inside a function (similar to the swap operation). The reader can try the prototype environment directly from http://liveprogramming.org. A screenshot of the prototype is shown at the end of chapter 2.

8.2 Future work

A list of proposals for future work is presented below, with brief descriptions.

- **Completeness of the defined transformations:** A difficult question to answer is if the defined transformations are incomplete or overlapping. Are they too tightly bound to the language, can they be expressed in more general terms?
- **Considering threaded programs:** All the work done in this report has considered single-threaded programs. Most should be applicable to multi-threaded programs as well, it's just more callstacks to handle, still it would need confirmation.
- **Explicit callstack benefits:** Having an explicit callstack opens up for microthreads, coroutines and continuations, which would be interesting to implement in the JavaScript interpreter.
- **Self modifying code:** For the situation when the interpreter is written in the same programming language as it implements (thus sharing the same data structures) exposing the AST to the code via a special variable, say `ast`, would enable the code to modify itself via the same operations that are normally used by the programmer in an environment. If the callstack is exposed too, a program would be able to implement functionality such as microthreads and coroutines itself, without any other language support.
- **Educational tool:** If serialization is added to the interpreter the state of a paused program could easily be transferred from one web browser to another, via a server. The teacher and students could exchange http links, and improve/debug the program together.
- **Live programming for other languages:** I would like to see Live programming enabled environments for some of the more common used dynamic languages such as Python, Ruby, Perl or perhaps PHP (with a web twist), as well as Smalltalk and those Lisp machines[2].

[2] if someone finds that missing keyboard

- **Performance evaluation:** An explicit callstack interpreter could be written to benchmark against an implicit callstack version, both implementing the same language and both tuned for performance.
- **Verifying the need of an explicit callstack:** More work could be put into trying to bring as many (any?) of the Live programming operations as possible into an implicit callstack evaluator. This would most likely result in some form of lazy transformations.
- **Implementation in C:** An implementation of both the evaluator and the Live programming operations could be written in a lower level language to verify that the Live programming functionalities isn't relying on the meta circular evaluator.

Appendix A

Sample Live programming session

Here follows a sample session to show how a series of different Live programming source code transformations can be combined to build a program from scratch (with instant feedback and verification) as well as to fix bugs. The program is an implementation of the merge sort [17] sorting algorithm. The reader is encouraged to visit http://liveprogramming.org and try it out from there.

An array and function call, to start with. Run until a reference error occurs.

```
var unsorted = [9, 3, 1, 5, 4, 8, 7, 2, 3, 5, 8, 2, 6, 4];
‘mergesort(unsorted)‘;
```

Create the function, with a minimal bi-partitioning for a start. *Insertion into an active block sequence.* Step until a reference error occurs.

```
var unsorted = [9, 3, 1, 5, 4, 8, 7, 2, 3, 5, 8, 2, 6, 4];
‘mergesort(unsorted)‘;

function mergesort(l) {
    var mid = ‘a‘.length/2;
```

```
    var left = a.slice(0,mid), right = a.slice(mid);
}
```

Correct `a` to `l`. *Replace an active expression.* Step over the declarations.

```
function mergesort(l) {
    var mid = l.length/2;
    ‘var left = l.slice(0,mid), right = l.slice(mid)‘;
}
```

Add a new variable and a recursive call for the left partition. Recurse until `l.length` is `3`, `mid` becomes `1.5`. How will slice handle this? Step and see that `left` has the first element `[9]` and `right` has the two other, a valid result. Continue into another recursive call with the list `[9]`.

```
function mergesort(l) {
    var mid = l.length/2;
    var left = l.slice(0,mid), right = l.slice(mid);
    var leftsorted = ‘mergesort(left)‘;
}
```

A list with only one (or none) element is sorted, return it as-is.

```
function mergesort(l) {
    ‘if (l.length < 1)
        return l;‘

    var mid = l.length/2;
    var left = l.slice(0,mid), right = l.slice(mid);
    var leftsorted = mergesort(left);
}
```

Step, the `if` condition evaluated false (off-by-one). *Compatible replace.* Change the operator and back up to the `if` statement, again, `return`.

```
function mergesort(l) {
    if (l.length <= 1)
```

```
        `return l`;

    var mid = l.length/2;
    var left = l.slice(0,mid), right = l.slice(mid);
    var leftsorted = mergesort(left);
}
```

The variable `leftsorted` has now been assigned. Add `rightsorted` as well to start the other recursion. *Insertion into an active variable declaration sequence.* Step over the declarations. Insert call to `merge`, run until a reference error occurs.

```
function mergesort(l) {
    if (l.length <= 1)
        return l;

    var mid = l.length/2;
    var left = l.slice(0,mid), right = l.slice(mid);
    var leftsorted = mergesort(left), rightsorted = mergesort(right);
    `merge(leftsorted,rightsorted)`;
}
```

Create a small `merge` function.

```
function mergesort(l) {
    function merge(a,b) {
    }
    ...
}
```

Add an empty return list and select one of the two start elements, step until another reference error occur.

```
    function merge(a,b) {
        var merged = [];
        merged.push(a[`i`]<b[j] ? a[i] : b[j]);
    }
```

Add variable declarations. Assign `i` and `j` to `0` in the eval window.

```
function merge(a,b) {
    var merged = [];
    var i=0; j=0;
    merged.push(a[‘i‘]<b[j] ? a[i] : b[j]);
}
```

Add a `while` statement around the selection. *Expand an active statement.*

```
function merge(a,b) {
    var merged = [];
    var i=0; j=0;
    while (i<a.length && j<b.length)
        merged.push(a[‘i‘]<b[j] ? a[i] : b[j]);
}
```

Increment for the indices was forgotten, thus `j` is wrong next iteration. Modify and run `j++` in the eval window.

```
function merge(a,b) {
    var merged = [];
    var i=0; j=0;
    while (‘i<a.length && j<b.length‘)
        merged.push(a[i]<b[j] ? a[i++] : b[j++]);
}
```

Push the remainders.

```
function merge(a,b) {
    var merged = [];
    var i=0; j=0;
    while (i<a.length && j<b.length)
        merged.push(a[i]<b[j] ? a[i++] : b[j++]);
    while (‘i<a.length‘)
        merged.push(a[i++]);
    while (j<b.length)
        merged.push(b[j++]);
}
```

Return the result.

```
    function merge(a,b) {
        var merged = [];
        var i=0; j=0;
        while (i<a.length && j<b.length)
            merged.push(a[i]<b[j] ? a[i++] : b[j++]);
        while (i<a.length)
            merged.push(a[i++]);
        while (j<b.length)
            merged.push(b[j++]);
        ‘return merged‘;
    }
```

The result from merge should be returned from `mergesort`, as well, add `return` statement around the function call. *Reduce an active statement* and *expand an active expression*. The reduction is done to strip the semicolon since statements never has semicolon as parent nodes. Could also be considered a *compatible replace*.

```
function mergesort(l) {
    ...
    ‘return merge(leftsorted,rightsorted)‘;
}
```

Add a print of the final sorted array.

```
‘println(mergesort(unsorted))‘;
```

The finished program and output data:

```
var unsorted = [9, 3, 1, 5, 4, 8, 7, 2, 3, 5, 8, 2, 6, 4];
println(mergesort(unsorted));

function mergesort(l) {
    function merge(a,b) {
        var merged = [];
```

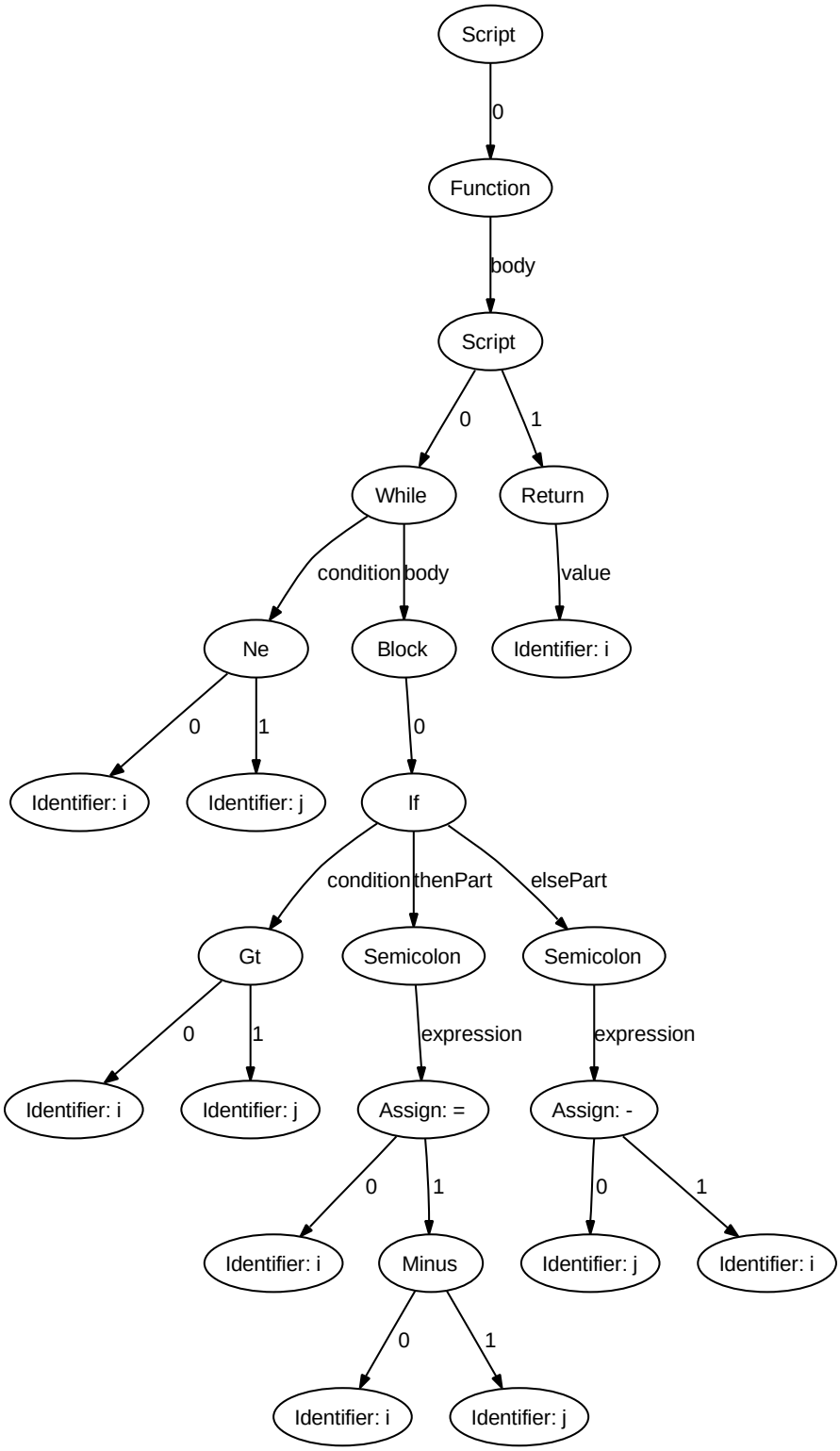
```
        var i=0; j=0;
        while (i<a.length && j<b.length)
            merged.push(a[i]<b[j] ? a[i++] : b[j++]);
        while (i<a.length)
            merged.push(a[i++]);
        while (j<b.length)
            merged.push(b[j++]);
        return merged;
    }
    if (l.length <= 1)
        return l;

    var mid = l.length/2;
    var left = l.slice(0,mid), right = l.slice(mid);
    var leftsorted = mergesort(left), rightsorted = mergesort(right);
    return merge(leftsorted, rightsorted);
}
```

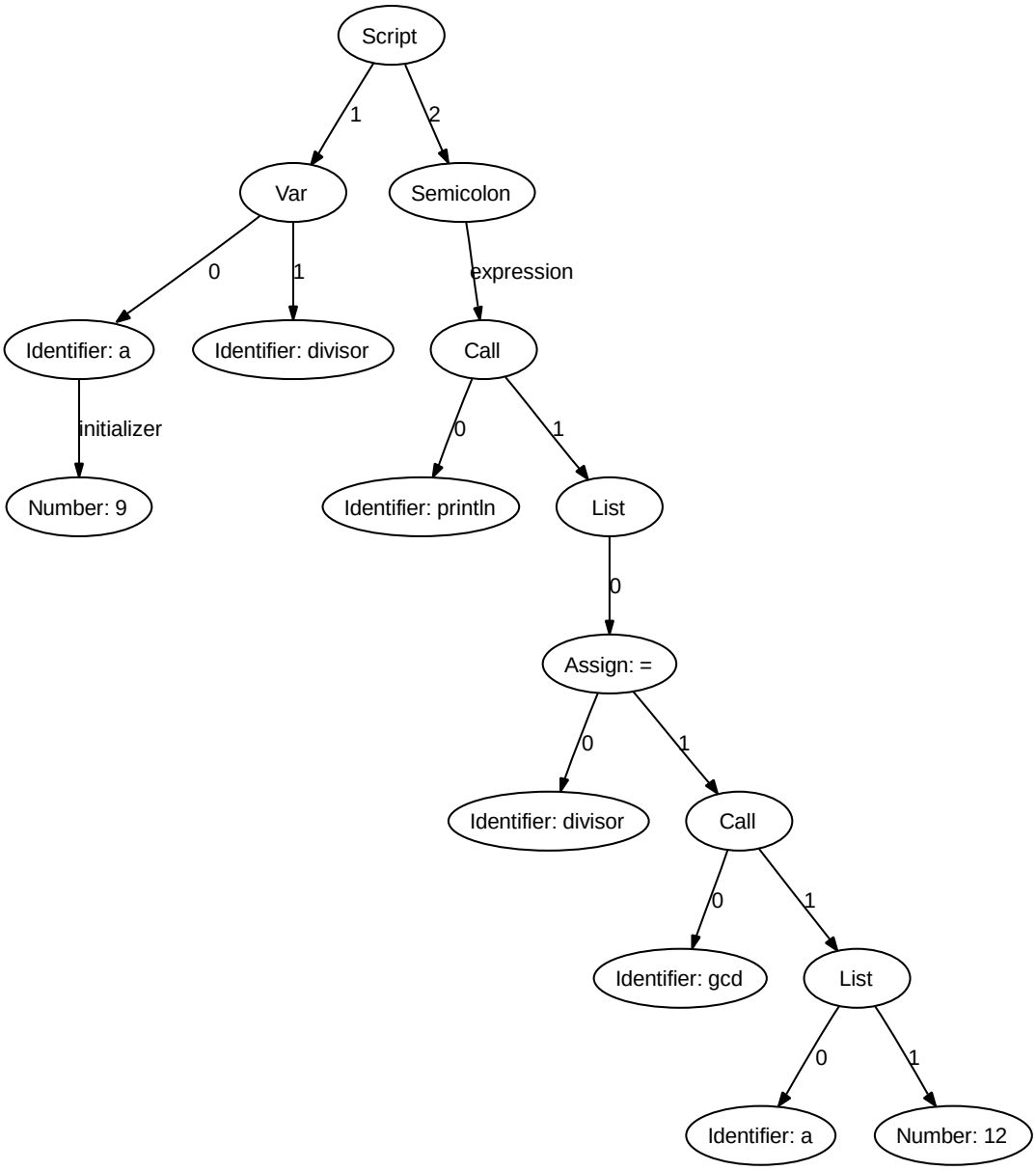
Prints: `1,2,2,3,3,4,4,5,5,6,7,8,8,9`

Appendix B

AST for GCD program

```
function gcd(i,j) {
    while (i!=j) {
        if (i > j)
            i = i-j;
        else
            j -= i;
    }
    return i;
}

var a = 9, divisor;
println(divisor = gcd(a,12));
```


Script
0
Function
body
Script
0
1
While
Return
condition
body
value
Ne
Block
Identifier: i
0
1
0
Identifier: i
Identifier: j
If
condition
thenPart
elsePart
Gt
Semicolon
Semicolon
0
1
expression
expression
Identifier: i
Identifier: j
Assign: =
Assign: -
0
1
0
1
Identifier: i
Minus
Identifier: j
Identifier: i
0
1
Identifier: i
Identifier: j


Script
1
2
Var
Semicolon
0
1
expression
Identifier: a
Identifier: divisor
Call
initializer
0
1
Number: 9
Identifier: println
List
0
Assign: =
0
1
Identifier: divisor
Call
0
1
Identifier: gcd
List
0
1
Identifier: a
Number: 12

```
{
    type: SCRIPT,
    0: {
        type: FUNCTION,
        body: {
            type: SCRIPT,
            0: {
                type: WHILE,
                body: {
                    type: BLOCK,
                    0: {
                        type: IF,
                        condition: {
                            type: GT,
                            0: {
                                type: IDENTIFIER,
                                value: i
                            },
                            1: {
                                type: IDENTIFIER,
                                value: j
                            },
                        },
                        elsePart: {
                            type: SEMICOLON,
                            expression: {
                                type: ASSIGN,
                                0: {
                                    type: IDENTIFIER,
                                    assignOp: 25,
                                    value: j
                                },
                                1: {
                                    type: IDENTIFIER,
                                    value: i
                                },
                            },
                        },
                        thenPart: {
```

```
                    type: SEMICOLON,
                    expression: {
                        type: ASSIGN,
                        0: {
                            type: IDENTIFIER,
                            assignOp: null,
                            value: i
                        },
                        1: {
                            type: MINUS,
                            0: {
                                type: IDENTIFIER,
                                value: i
                            },
                            1: {
                                type: IDENTIFIER,
                                value: j
                            },
                        },
                    },
                },
            },
            length: 1,
        },
        condition: {
            type: NE,
            0: {
                type: IDENTIFIER,
                value: i
            },
            1: {
                type: IDENTIFIER,
                value: j
            },
        },
    },
    1: {
        type: RETURN,
        value: {
```

```
                type: IDENTIFIER,
                value: i
            }
        },
        funDecls: ,
        length: 2,
        varDecls:
    },
    functionForm: 0,
    name: gcd,
    params: i,j,
},
1: {
    type: VAR,
    0: {
        type: IDENTIFIER,
        initializer: {
            type: NUMBER,
            value: 9
        },
        name: a,
        readOnly: false,
    },
    1: {
        type: IDENTIFIER,
        name: divisor,
        readOnly: false,
    },
    length: 2,
},
2: {
    type: SEMICOLON,
    expression: {
        type: CALL,
        0: {
            type: IDENTIFIER,
            value: println
        },
        1: {
```

```
                type: LIST,
                0: {
                    type: ASSIGN,
                    0: {
                        type: IDENTIFIER,
                        assignOp: null,
                        value: divisor
                    },
                    1: {
                        type: CALL,
                        0: {
                            type: IDENTIFIER,
                            value: gcd
                        },
                        1: {
                            type: LIST,
                            0: {
                                type: IDENTIFIER,
                                value: a
                            },
                            1: {
                                type: NUMBER,
                                value: 12
                            },
                            length: 2,
                        },
                    },
                },
                length: 1,
            },
        },
    },
    funDecls: <gcd>,
    length: 3,
    varDecls: <a>, <divisor>
}
```

Bibliography

[1] Martin Fowler. *Is Design Dead?* http://www.martinfowler.com/articles/designDead.html, 2004. Accessed 2006-06-06.

[2] Bill Venners. *A Conversation with Ward Cunningham, Part V.* http://www.artima.com/intv/simplestP.html, 2004. Accessed 2006-06-06.

[3] Andrew Hunt and David Thomas. *The Pragmatic Programmer.* Addison Wesley, 1st edition, 2000.

[4] Giancarlo Succi and Michele Marchesi. *Extreme Programming Examined.* Addison Wesley, 1st edition, 2001.

[5] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA '97 Conference Proceedings*, 1997.

[6] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87 Conference Proceedings*, 1987.

[7] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 4th edition, 2002.

[8] Danny Goodman. *Dynamic HTML: The Definitive Reference.* O'Reilly, 2nd edition, 2002.

[9] Danny Goodman and Michael Morrison. *JavaScript bible*. Wiley, 5th edition, 2004.

[10] ECMA, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. *ECMAScript Language Specification*, 3rd edition, 1998. Accessed 2006-06-06.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[12] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 2nd edition, 1996.

[13] Michael Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 1st edition, 2000.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[15] Christian Tismer. *Continuations and Stackless Python*. http://www.stackless.com/spcpaper.htm. Accessed 2006-06-06.

[16] Warren Teitelman. *Interlisp Reference Manual*. Xerox Palo Alto Research Center, 3rd edition, 1978.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1996.