COMP4 Project: Compiler

For a Custom, Educational CPU Architecture

Luke Wren 2142

Contents

Analysis

Introduction

Computing teachers need to teach students how processors work. Processors are complicated. This is a problem.

Modern processors are sophisticated and complicated to the extent that PhD theses can and have been written about the smallest implementation details – teaching a class of A level students about the inner workings of a modern x86 superscalar would be an exercise in madness.

Many efforts have been made to create “educational architectures” – designs for a computer processor that are easy to understand for students. These usually fail in one of three areas:

- They are not really very easy to understand.
- As a result of simplification, they are not useful for real world applications.
- The processor is often let down by the quality of the supporting software stack.

I’ve designed and implemented a simple architecture, based around the concept of an OISC (one instruction set computer). The small instruction set means the implementation is simple enough to scribble on the back of a napkin, but some interesting compiler techniques make everyday tasks such as programming, text editing and simple games perfectly possible.

The implementation of the processor is beyond the scope of this document – instead, it focuses on the implementation of a cross-compiler for a high-level language, targeted at this educational processor architecture and tailored to the needs of my school’s teaching staff.

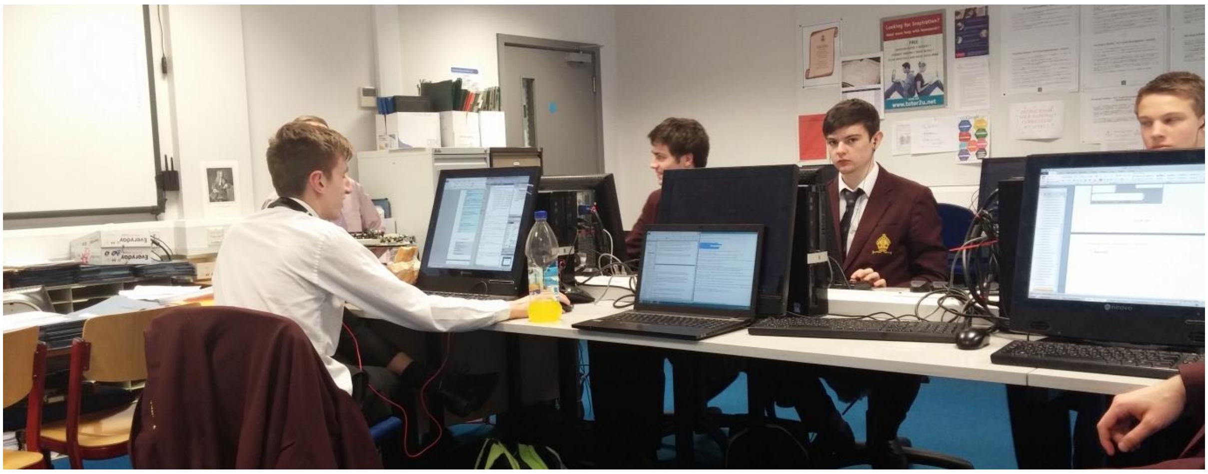
Research

My school would like a compiler targeting a simple CPU architecture, for use in A-level computing lessons. The school will be the intended client – students will be the chief users.

This software will need to be tailored to the needs of my staff: it should meet their existing performance requirements, and also solve any problems they may have with the currently employed solution.

In order to better scope out these needs, I drew up a list of questions around the topic, and interviewed the head of computing, Mr M Greenhalgh, with a view to better understanding what the department wants and needs from the new software. I also sought a more general view of computing lessons at my school, to more thoroughly understand how this software would fit into the flow of a lesson.

IT Lessons at the King's School – Identifying the Client



The King's School teaches both ICT and computing; depending on whether they are more interested in the theory behind computers, or in more practical things such as spreadsheets and pretty powerpoints, students can choose which of these two subjects they take.

They are interested in having some software developed for them, for use in IT lessons: a compiler for a new language, which compiles down to very simple machine code, running on an easy-to-understand computer architecture (they referred to existing model computers such as the Little Man Computer).

Lessons at the King's School tend to be a roughly even split between theory and practical. Theory is usually explained with the help of a good old-fashioned whiteboard (although visual aids and demonstrations on the projector/interactive whiteboard are used wherever available), and students work through the more practical side of programming and theory on their own independent workstations: reasonably powerful computers (2.6GHz Core 2 Duos with 2GiB of RAM) running Microsoft's Windows 7.

During the course of A-level computing (particularly modules COMP1 and COMP2) the students are taught about binary and hexadecimal numbers, representation of information and how computers process information, with the lowest level taught being the fetch-execute cycle, moving on upwards through machine code, assembly code and a cursory tip of the hat towards high-level languages.

Most of these topics (in particular the layers between machine code and high-level languages) are taugh with the use of copious worksheets; a functional but sufficiently simple compiler would ease student comprehension of the material.

This is an example of a worksheet currently used when teaching lessons on machine code:

King's School Computing – A-Level – Unit 2 – Machine Code/Assembler

Machine code is the set of binary instructions that a CPU has been designed to perform. All CPUs process instructions that are in machine code. They are coded in binary with tiny voltages used to represent 0s and 1s. (On most computers, 1.8 volts or above represents a 1; below that represents 0. Usually 5 volts is quoted.)

Each machine code instruction has two parts, an operation code and an operand:

	Operation Code (Op Code)	Operand (Data or Address)
e.g.	0001 0000	0000 00011

The op code is the actual machine code operation, e.g., ADD, MUL. The operand, if present, represents an item of data or an address of the data that is to be used by the operation. Some examples of machine code instructions with their assembler equivalents…

Machine Code	Assembler (Mnemonic)	Explanation
0001 0000 0000 0011	LOAD #3	Load the value 3 in the accumulator.
1000 0000 0000 1101	STORE 13	Store a copy of the accumulator contents into memory location 13.
0001 0000 0000 0110	LOAD #64	Load the value 64 into the accumulator.
0011 0000 0000 1101	ADD 13	Add the accumulator contents and the value at memory location 13; place the results in the accumulator
0111 0000 0000 0110	MUL #6	Multiply the accumulator with the value 6;store the results in the accumulator
0101 0000 0000 1010	SUB #12	Subtract 12 from the value in the accumulator; store the results in the accumulator
1001 0000 0000 1010	DIV 10	Divide the accumulator contents by 10, storing the result in the accumulator.
1011 0001 0000	ADD R1,#15	Add the value 15 to the contents in register R1.
1000 0000 0001 1101	STORE R2,30	Store the value in register R2 at memory address 30.

Exercises:

1) Write the assembler to perform 12 x 8 and store the result at memory location 400

2) Write the assembler to add up the 1st 3 prime numbers; store the result at memory location 200.

3) What value results from running the following code and where is it stored?

```
MOV #10
DIV #2
ADD 100
STORE 102
```

Address	Value
102	65
101	4
100	5
99	12
98	33

4) Write the assembler to add the values at memory addresses 100, 101 and 102 together. Then add the values at location 98 and 99 together and divide this by the 1st total.

Page 1 of 1

This is how the topics of machine code and computer architecture are currently taught: it works, but the school would like the students to be more involved and try something more hands-on.

From these observations we can immediately see that:

- The software should be able to run on Windows.
- It should be able to run interactively, so that teachers can use it for demonstrations in front of the class.

Client Interview – Identifying Needs

I talked to head of computing, Mr M Greenhalgh, to get an idea of problems currently faced when teaching programming and computer architecture, and their preference with regard to other features of the proposed language.

I don't have a word-for-word transcript of the discussion, but following are the questions and my rough notes of the answers:

1. What's the biggest problem when teaching students to program?
    - Don't understand variables very well
    - How to use variables
    - Being able to visualise RAM would help
    - Seeing workings of compiler, allocating memory
    - Conditional statements
    - Passing parameters (how it actually works under the hood)
    - Jumping within the code (more wrt function calls than gotos)
2. Would you prefer a simpler assembler or a more high-level language (e.g. C, Pascal, LISP)?
    - Start with simple assembly language
    - If machine code was incredibly simple (e.g. OISC model mentioned) could go straiht to high level concepts
3. Would you prefer a more traditional C-like syntax or something more like Python etc.?
    - More traditional C-like approach: more on level with things you'd want to teach such as structured programming, having a standard style and structure (methodology) for writing programs
    - If too free-form, too much scope to get confused
4. How important are things like pointers? What about recursion?
    - Quite fundamental, sometimes difficult to get across.
    - Getting idea across can be a challenge; understanding how they're implemented is also tricky. *(talking about pointers)*
    - Recursion is important too. They need to be aware of it at A level. Comparing recursive solutions to iterative solutions.
    - Wouldn't necessarily need to be able to program recursively, at least to any great length
5. How important is it that the machine code is easily readable? E.g. teaching how processors work
    - You want to be able to identify the individual instructions, and maybe draw mappings between the and the assembly language or higher level concepts
    - Understanding _what makes up_ a language
    - Looking at things like BNF

- Understanding that a compiler simply transforms high-level language into machine code by following basic rules

6. Would having an actual physical computer to run the code on be an advantage?
    - Makes it easier to visualise what happens. Physically linking everything together makes the concepts easier to understand.
7. Are you interested in being able to run the compiler on other systems than desktops, e.g. Raspberry Pi? Would Linux support be useful?
    - In theory would be nice. More scope for users.
    - Not currently important in our school setting (until we get raspberry pis!)
    - Don't write code in a way that makes it impossible to port easily in the future.

From this transcript, I drew up the following table of features that the language and the compiler must/should/could have:

Feature	Must/Should/Could
Must support variables: • Variables of different types must exist. • Compiler must automatically allocate memory for each variable. • Programmer must be able to assign to and from variables within the code.	Absolutely must. This was a key issue identified by the client.
Conditional statements: • `if` and `while` as an absolute minimum. • Should function as in C. • This covers both loops and branching.	Must.
Parameter passing: • Necessitates functions and function calls. • Function calling convention should be clearly documented.	Must. This was another key issue that the client identified.
Arrays • Makes pointers twice as useful. Allows storing of (large amount of | indexed) data. • Only 1-dimensional necessary.	Must.
Traditional C-like syntax • Code should only differ from C syntax in the keywords and the semantics.	Should. The client identified this as preferable.
Pointers • Must work: programmer can read to/write from indirectly addressed locations. • Mechanics of pointers should be clearly exposed in machine code.	Must. Key need.
Recursion • Recursive data structures (requires use of pointers) should be implementable by programmer. • Recursive code/function calls should be usable.	Should.
Machine code output should have a limited number of instructions (less than 10) and a simple (fixed) format.	Should.
Syntax should be documentable (documented?) with BNF	Should.
Have a physical computer platform (custom homemade CPU) specifically designed to run the compiler's output.	Could. "Nice to have."
Support for multiple platforms: Linux etc.	Could. Nice to have in

(Essentially don't do anything to make it platform-dependent).	the future.

Follow Up Interview

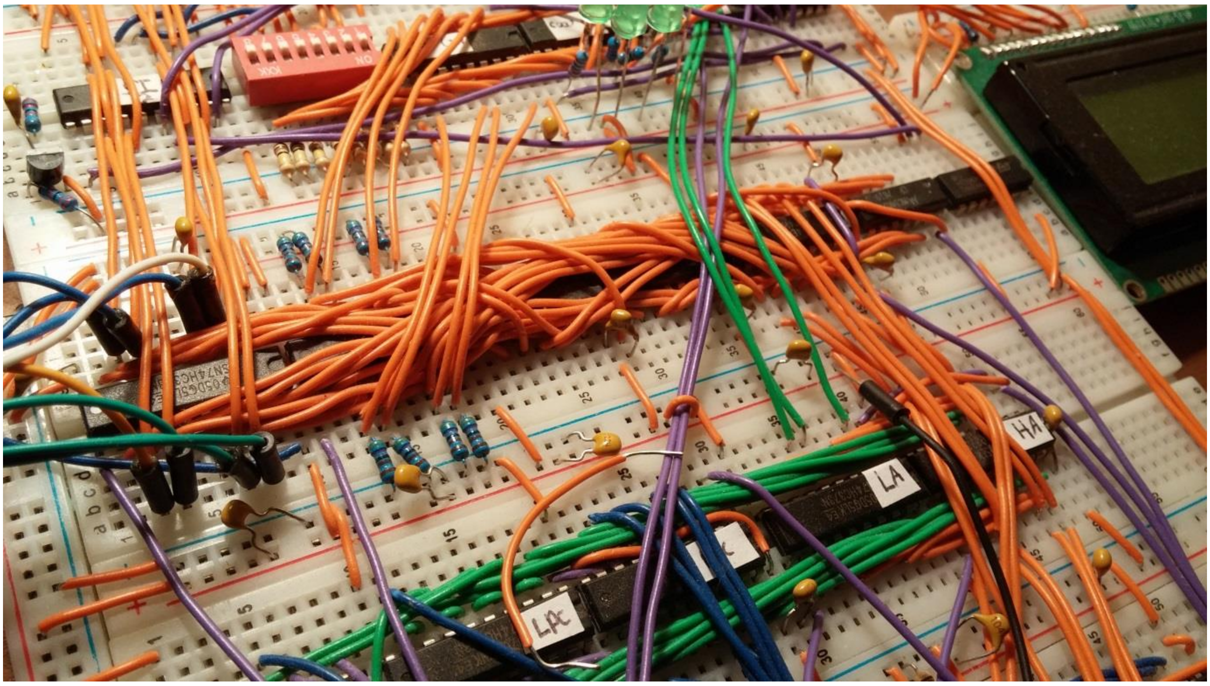
Having drawn up this table of needs, I saw my IT teacher again for a follow-up assessment of the requirements. We looked through the table, and I also asked a few more questions to flesh out ideas about how the compiler itself should function, from a user interface and user-experience (UI and UX) standpoint. We identified the following additional needs:

Feature	Must/Should/Could
Report syntax errors to user – code must be well formed. • State the type of error (what is wrong/must be corrected) • State the line and file in which the error occurs	Must.
Report missing/undeclared variables and type mismatches. • The names of undeclared but used variable names should be reported. • Type mismatches should be stated, along with the expected type, the received type, the line number and the file in which the error occurred.	Must.
Report if a file is missing/cannot be found. • State the name of the missing file.	Must.

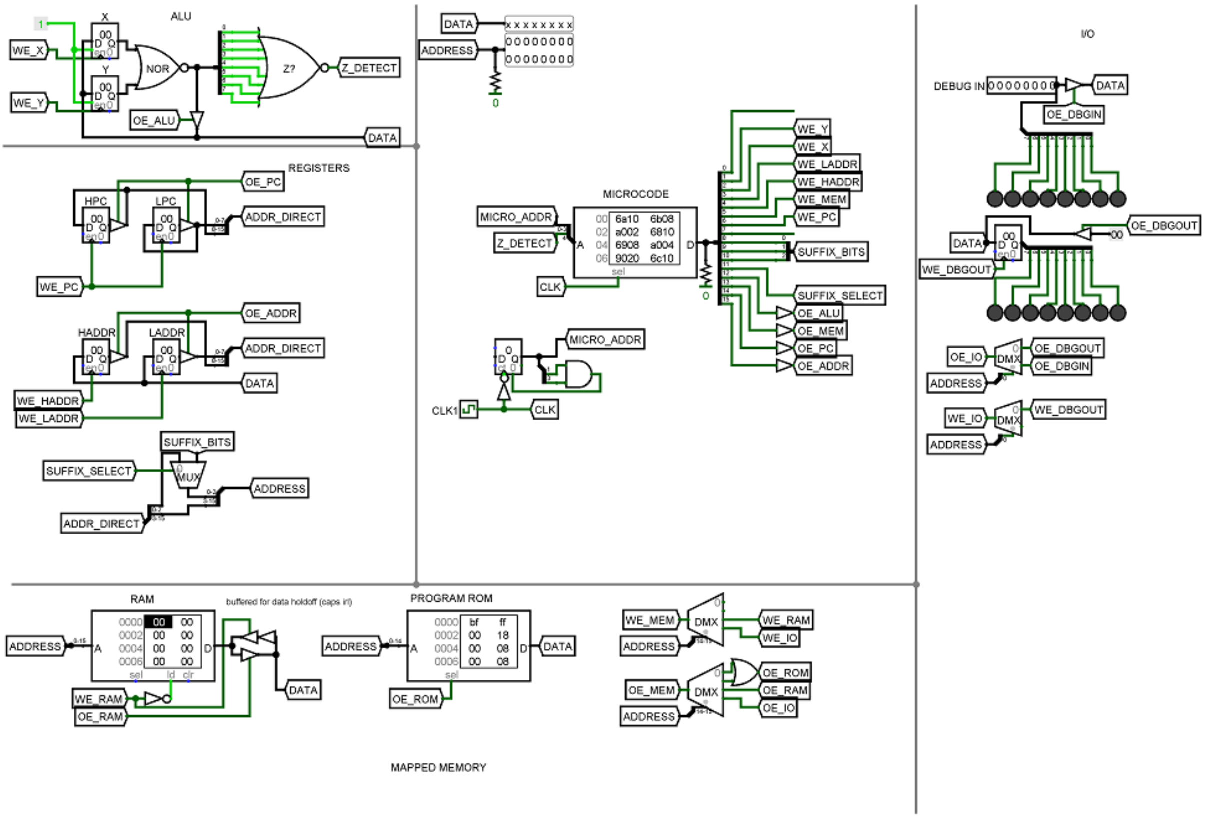
With these needs in mind, we can start looking at the more technical details and limitations.

Architecture

In order to generate machine code for our target architecture, we must first understand it.



Physical machine implementation (8MHz) – this is a working CPU, the program counter and address latches are visible at the bottom of the picture



Logical diagram of architecture

We will not delve into the lower-level operations of the CPU and peripherals[1] - these are not relevant to the compiler. The machine is capable of executing one type of instruction: Nor and Fork Conditionally, or NFC[2].

The processor lives inside a 16 bit memory space, and it addresses ROM, RAM and peripherals by use of a memory map. Each memory word is 8 bits – this gives a total addressable memory size of 64 KiB.

An instruction word is 64 bits long, and consists of 4 16-bit addresses.

The steps of execution are as follows:

1. Fetch the bytes from the first two addresses
2. NOR them together and write back to the first address
3. Branch based on the result:
    a. If non-zero, jump to the third address
    b. Else jump to the fourth

The execution of each instruction consists of 10 steps – these steps occur at a rate of 8MHz (million per second), yielding a throughput of 0.8 million instructions per second (MIPS).

[1] If you're interested: https://github.com/Wren6991/NorForkConditionally/tree/master/processor
[2] NFC comes from Jack Eisenman's DUO Compact CPU (although structurally the processors are very different).

Although each individual instruction is very simple, in these quantities they can be combined to create sophisticated software.

The system is memory mapped – the 16 bit address space allows 64 KiB of memory to be addressed (2^{16} bytes). The first 32 KiB is mapped into read only memory (ROM), which is immutable to programs and contains the program the machine runs on startup. The next 16KiB is random access memory (RAM), which can be used to store data and additional loaded programs. The final 16KiB sector is mapped to various peripherals.

Feasibility

When targeting a new architecture, feasibility is determined by one question: does the processor support all the operations needed for this language?

Furthermore, does it do so in an efficient way?

If an operation can be implemented in machine code, then a compiler can generate it. With that in mind, let's identify some operations and attempt to find an efficient machine code implementation.

To help me with this feasibility study, I implemented a simple assembly language in Javascript, named Fork (Nor and Fork Conditionally… Get it? I need to have some fun while I write this up). It handles macros, static variable allocation, constants and not very much else.

This language is all documented but the user guide is rather long to include here – the syntax should be easy enough to pick up. The only important thing to understand is that there is a one-to-one mapping between command and machine code, i.e. the command `"x y"` (NOR `x` and `y` together then go to next instruction) is assembled into 64-bit hex `[xxxx] [yyyy] [next] [next]`, and the command `"x y a b"` becomes hex `[xxxx] [yyyy] [aaaa] [bbbb]` where x, y, a and b are either hexadecimal numbers or constants. (Hopefully this will become clear in the examples.)

Moving Bytes

Our high-level language will most certainly have to move bytes around at some point. The most efficient way to accomplish this is as such:

```
def clear(dest)
    dest 'ff
end
def invcopy(src, dest)
    clear(dest)
    dest src
end

def invert(dest)
    dest dest
end

def copy(src, dest)
    invcopy(src, dest)
    invert(dest)
end
```

Here we see a few interesting features of the NOR operation:

- We can clear by NORing with 0xff (all ones) – this takes one instruction.
- 0 NOR x is ¬x. We can do a copy-and-invert in two instructions (one to clear, one to copy).
- X NOR x is ¬x. We can invert in-place in one instruction.
- If we copy and invert, and then invert in-place, we will move x to the destination (law of involution).

So copying between static locations is easy, and the compiler can potentially optimise instances where the inverted value is needed (2 instructions instead of 4).

As an example, the following command:

```
copy(0001, 0002)
```

Results in the following machine code:

```
0000: 0002 0018 0008 0008
0008: 0002 0001 0010 0010
0010: 0002 0002 0018 0018
0018: ff
```

(The constant 0xff has been linked in at the end.)

Bitwise Operations

We have a NOR operation, so it follows that we should be able to implement any other bitwise operation using some quick boolean algebra.

Bitwise Operation	NOR Equivalent	Comment
NOT A	A NOR A	
A OR B	NOT(A NOR B)	
A AND B	NOT(A) NOR NOT(B)	De Morgan's Law
A NAND B	NOT(NOT(A) NOR NOT(B))	
A XOR B	NOT((NOT(A) NOR B) NOR (NOT(B) NOR A))	Equivalent to (A^¬B)v(B^¬A)
A ANDNOT B	A NOR NOT(B)	Useful for masking + XOR – two instructions only! It's also itself universal

These are all fairly compact and efficient, so it's quite reasonable for the compiler to generate these in-place.

Indirection

Indirection (pointers) is incredibly important for data structures like stacks and arrays, for implementing lookup tables, and for reducing code size (by allowing iteration instead of having to unroll loops). Educational architectures – in particular those with separate code and data stores – often aren't capable of this, which makes them fairly useless for general computing and as a target for a high-level language.

How can this be implemented on a single-instruction machine with direct addressing? There is no direct support, but we have already shown that we are able to manipulate memory, and the architecture is of the Von Neumann type (i.e. unified memory). This would suggest self-modifying code as a solution.

Self-modifying code is frowned upon for two main reasons:

- It's confusing to follow.
- It invalidates the instruction cache, so any modern pipelined processor will run like an absolute dog.

Whilst the machine code may be harder to understand it is still trivial to generate, and our processor is not pipelined, so there is no issue here.

This means that *pointers will be a runtime feature*, provided by the language (self modifying code in read only memory is going to need some runtime support). As long as it's seamless, it's still practical.

Branching

A high-level language lets you express ideas – these include decisions.

Branches may be conditional or unconditional. This is where the skip fields of the instructions come in. Each instruction is effectively followed by two `GO TO`s, one of which is followed according to the result – to quote a well known text, "Put that in Pascal's pipe and smoke it."

Unconditional branches are represented as such (in Fork assembly):

```
def jump(address)
    var x
    x x address address
    free x
end
```

This is a single instruction: NOR some unused location with itself (effectively making the NOR a no-op) and then jump to another address irrespective of the result.

If we put different addresses into the two skip fields, we get a conditional jump:

```
def branch(val, address_true, address_false)
    var x
    invcopy(val, x)
    x x address_true address_false
    free x
end
```

With conditional and unconditional branches you can implement while loops, for loops, ifs, elses and just about any other high-level procedural control construct such as short-circuit logic operators (`&&` and `||` in C).

Arithmetic

This is the only potential stumbling block. We can generate code that will move bytes around, make decisions and perform bitwise manipulation like nobody's business, but at some point a high-level language will have to perform some cold hard sums.

We might want to:

- Bit shift, to the left and the right (multiply and divide by two)

- Add and subtract two 8-bit numbers
- Increment/decrement a single 8 bit number
- Perform comparisons

In the course of a high-level program.

An addition operation can be broken down into:

- XOR A and B to get sum
- AND A and B to get carries
- Shift carries to the left, repeat addition until carry is 0.

This is as fast as we can expect – time is proportional to the word size, or O(log n) to the number size, while a naïve solution might be O(n). It does however depend on a fast implementation of a left shift.

Subtraction can be achieved by inverting and incrementing B before performing addition (two's complement).

Left shift, right shift, increment and decrement all have a useful property – they only have one operand. As the operand is only 8 bits, it's actually quite feasible to use a lookup table for these operations – all four of them will take up only 1KiB in total. Indirection has already been shown to be possible, so table lookups should be fast if the tables are 8-bit aligned (this can be achieved by putting them at the end of ROM).

I originally tested the feasibility of an algorithmic approach to incrementation etc.:

```
def debugout c000
def debugin c001

def clear(dest)
  dest 'ff
end

def not(src, dest)
  clear(dest)
  dest src
end

def invert(dest)
  dest dest
end

def goto(address)
  var a
  a a address address
  free a
end

def copy(src, dest)
  not(src, dest)
  invert(dest)
```

```
end

def getBitAndFork(src, data, address)
  var a
  not(data, a)
  a src address next
  free a
end

def NFCPairAndJump(src1, src2, dest1, dest2, address)
  dest1 src1
  dest2 src2 address address
end


# Strategy: find the least-significant 0 bit, set that bit to 1, set
all bits below it to 0.
def increment(src, dest)
  var a b
  clear(a)
  clear(b)
  getBitAndFork(src, '1, carry1)
  getBitAndFork(src, '2, carry2)
  getBitAndFork(src, '4, carry3)
  getBitAndFork(src, '8, carry4)
  getBitAndFork(src, '16, carry5)
  getBitAndFork(src, '32, carry6)
  getBitAndFork(src, '64, carry7)
  getBitAndFork(src, '128, carry8)
  dest 'ff finish finish
carry1:
  NFCPairAndJump('fe, 'ff, a, b, skip)
carry2:
  NFCPairAndJump('fd, 'fe, a, b, skip)
carry3:
  NFCPairAndJump('fb, 'fc, a, b, skip)
carry4:
  NFCPairAndJump('f7, 'f8, a, b, skip)
carry5:
  NFCPairAndJump('ef, 'f0, a, b, skip)
carry6:
  NFCPairAndJump('df, 'e0, a, b, skip)
carry7:
  NFCPairAndJump('bf, 'c0, a, b, skip)
carry8:
  NFCPairAndJump('7f, '80, a, b, skip)
skip:
  clear(dest)
  dest src
  dest b
  dest a
  invert(dest)
  free a b
finish:
end
```

```
loopStart:
  increment(debugin, debugout)
  goto(loopStart)
```

This compiled perfectly and produced functional machine code, which ran on both the emulator and the logic-level hardware simulator, but in this one case the LUT-based solution was actually the more practical one (smaller *and* faster). Left and right shifts could not be implemented in this way either.

Multiplication can be implemented as a series of additions and shifts, or as a single addition in a counter loop – this depends on use case, but is entirely feasible either way.

Comparisons can be achieved with only two operations: equality, and either less than or greater than (whichever one isn't implemented can be created by swapping the operands). Equality is true if the result of an XOR is 0 – an XOR and conditional branch have already been demonstrated. Less than (A < B) could be implemented with the following algorithm:

- If most significant bit (A XOR B) is zero:
    - Left-shift A and B, return to top of loop.
- If MSB(A), return false.
- Otherwise return true.

This is fairly fast, returns early if the operands are not very similar, and uses only bitwise operations and two table lookups.

Feasibility Conclusion

It is possible to generate efficient machine code to copy bytes, perform pointer reads and writes, perform bitwise operations, branch both conditionally and unconditionally, and perform arithmetic including addition, subtraction, multiplication and comparison. Support for branching and indirect addressing allows support for function calls with little further effort.

A simple compiler/assembler was written that allowed instructions to be encapsulated in a more friendly way, and showed that static memory allocation is a suitable solution to language-level variable management.

With these operations in place, it is possible to write a compiler that can synthesize reasonably efficient machine code based upon high-level source code.

Choice of Language

Fork was written in Javascript, for speed of prototyping. This is an excerpt from its main function:

```
	var funcdefs = [];
	var constdefs = [];
	var literals = [];
	// strip out comments
	progstring = progstring.replace(/#[^\n]*/gm, "")
	// remove line continuations
	.replace(/&[\r\n\t ]+/, " ")
```

```
        // detect literals and push onto literal list
        .replace(/'([0-9a-fA-F]+)/g, function(match, name){
              if (literals.indexOf(name) < 0)
                    literals.push(name);
              return match;})
        // find and strip macro definitions.
        .replace(/def[\t ]+[a-zA-Z_]+[\t ]*\([^\)]*\)[\s\S]*?end/g,
function(match){funcdefs.push(match); return "";})
        // do the same for constants
        .replace(/def[\t ]+[a-zA-Z_]+[\t ]+[a-zA-Z0-9_]+/g,
function(match){constdefs.push(match); return "";});
```

There is an old quote by programmer Fredrik Lundh:

"Some people, when confronted with a problem, think
'I know, I'll use regular expressions.' Now they have two problems."

The practical upshot of this is that, while regexes are useful tools, they are not in themselves a viable design pattern. Javascript is very useful for hacking together quick kludges ("rapid prototyping" is the approved term) but it is also very good for reinforcing bad habits and taking shortcuts, neither of which are conducive to maintainability. Something lending itself to more structured programming, with a rich standard library and a variety of data structures, would be preferable.

I considered several languages (slightly subjectively):

Language	Pros	Cons
C	• Naturally good for structured and procedural programming • Just about the best syntax of any language for manipulating bits and bytes (except Verilog, VHDL)	• Poor string processing capabilities • Slightly "DIY" approach to the standard library with regard to data structures • `free()`
C++	• Well suited for procedural and object oriented programming, suitable for a compiler • Same bit-level manipulation as C, useful for machine code generation • Very rich and general standard library (`STL`), lots of data structures	• String processing is comprehensive but cumbersome (however, still easy to address individual bytes) • `delete` (although RAII mitigates this)
Java	• Same pros as C++, apart from that one quirk where you have to cast bytes to ints to store them in bytes • "write once, run anywhere" (VM based), although in practice it is rarely quite this simple • Managed memory (garbage collection)	• Standard library is verbose to the point of absurdity
PHP	• Good for string processing	• Generally inconsistent, a fractal of poor design • Traditionally more suited to slinging

		strings around web back ends, not so suited to working with large chunks of binary data
Haskell	• Functional paradigm is very natural for expressing the process of compilation (transformation of source code -> machine code)	• Not too brilliant for bit level operations
Python	• Very expressive • Large community with lots of example code, enormous range of community libraries in addition to standard one • Very good for text processing as well as more complex data structures • Managed memory	• Performance (less of a problem these days, and perhaps less of an issue for a simple compiler) • Very dynamic, promotes some bad habits

I eventually went for C++: the wide selection of data structures should make the implementation simple, and the fact that it is so widely “spoken” should make it easier for other programmer to maintain the compiler later on during its lifecycle, which in itself may outweigh the potential benefits of other languages.

Design

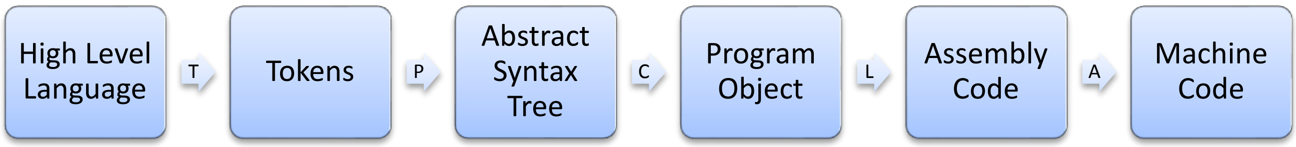
Top-level Design

Although a compiler may be internally very complex, its overall structure is highly linear, which lends itself to a very tidy compartmentalised approach; a solid structure makes the job of implementation much faster and less error prone, as opposed to making things up as you go along, which usually results in an unmaintainable mess.

A compiler's purpose is to translate the high-level language input into machine code – these languages are known as the input and output representations.

To achieve this, the compiler will be implemented as five stages, with four intermediate representations (IRs). Each of the five stages is like part of an assembly line in a factory, from raw materials to finished product.

The processes are shown above; below, each intermediate representation is shown as a box, with the process arrows in between.



In terms of implementation, object oriented programming (OOP) will be used to structure the code.

Each stage will be implemented as an object, with a well-defined interface to pass data in and out. The procedural code and any data structures that relate only to the internal operation of each stage are contained within – this is an example of information hiding, and keeps the program neat and modular. C++'s `class` construct would be ideal for this.

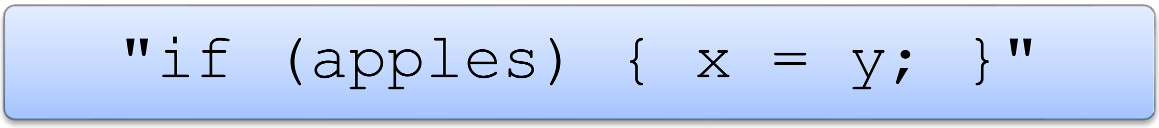
The intermediate representations are formats for transferring chunks of data between the stages – in C++, the `struct` language feature is typically the tool for the job, in conjunction with the container types of the Standard Template Library such as `vector`s (an implementation of a dynamically allocated array).

Processes

Tokenizer

The tokenizer’s purpose is to analyse a stream of characters and break it down into recognisable symbols and keywords – this process is also known as lexing, or scanning. Tokens are also known as lexemes, and represent one indivisible “atom” of the language.

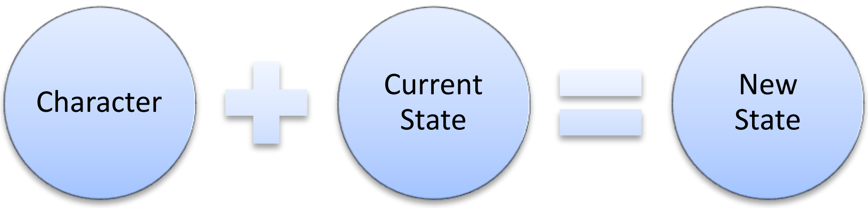
For example:



This is a string of characters that represents high-level ideas. A state machine will read through the character list and output a series of tokens, or raise an error if it detects an invalid state/character combination (e.g. a letter while parsing a number).

Each token will be tagged with an enumerated type, identifying it as a keyword, variable name, or whatever else is appropriate. A string of characters – ‘a’, ‘p’, ‘p’, ‘l’, ‘e’, ‘s’ – will become a single language object (`“apples”, variable_name`), making it easier for the next stage to process. Whitespace and line return characters will also be ignored, as they are not significant to the syntax.

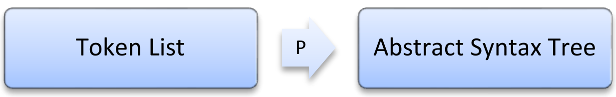
The state machine is the classic implementation of a lexer/scanner: the state variable represents the type of token currently being processed, and an enormous `switch`/`case` statement in a loop implements the transition function:



The tokenizer outputs a list of tokens for further processing.

Parser

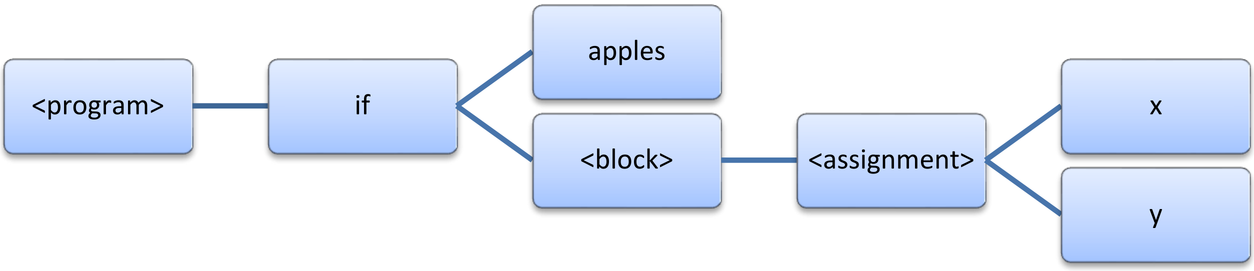
The parser will implement the language’s syntax.



It will accept a list of tokens from the tokenizer, and produce an abstract syntax tree for semantic processing. The tree is abstract in that it does not contain any extraneous details – an if node, for example, doesn't contain any brackets. A recursive tree structure is the natural way of representing language syntax – BNF is itself a tree representation, if one considers the <identifiers> as references.

The example token list:

Would be processed into the following tree:



Note that this tree can be recursive – e.g. an if statement may contain another if statement within its block. This would suggest *recursive descent parsing* as a suitable algorithm – each node type would have an associated function within the parser object, and the structural recursion would be mirrored by the functional recursion of the code. For example, the "if" function calls the "expression" function which returns the "apples" variable. The if function then calls the "block" function which recognises the statement type and calls the "assignment" function; it would continue as such.

For example, for the following BNF (for a mathematical expression):

```
<expr> ::= <term> {`+` | `-` <term>}
<term> ::= <factor> {`*` | `/` <factor}
<factor> ::= <number> | <name> | ( `(` <expr> `)` )
```

We could write the following pseudocode (with appropriate implementations of `gettoken()` and `error()`):

```
function accept(type)
  if current_symbol is type then
    gettoken()
    return true
  else
    return false
  end
end

function expect(type)
```

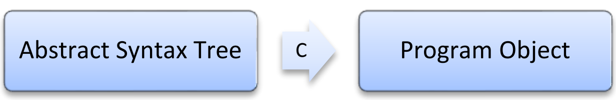
```
  if not accept(type) then
    error("Error: expected " + type + " but got " current_symbol)
  end
end

function expr()
  exp = table containing term()
  while accept("+") or accept("-") do
    insert last_symbol into exp
    insert term() into exp
  end
  return exp
end

function term()
  trm = table containing factor()
  while accept("+") or accept("-") do
    insert last_symbol into trm
    insert factor() into trm
  end
  return exp
end

function factor()
  if accept("name") then
    return new_name(last_symbol)
  else if accept("(") then
    exp = expr()
    expect(")")
    return exp
  else
    expect("number")
    return new_number(last_symbol)
  end
end
```

By tracing the execution of this code, you can see how the algorithm correctly implements the specified syntax for expressions such as `2 * (3 + x)` and returns an easily-processable syntax tree, and you can also see how elegantly it rejects malformed input such as `"1 + 2 +"` ("Error: expected number") or `"1 * (2 + 3"` ("Error: expected right bracket").

It is also evident how this algorithm yields a high degree of correspondence between code and specification (compare the function `expr()` with its BNF counterpart) which massively simplifies both implementation and testing.

Compiler

The language semantics – namely, lexical scope resolution and type checking – will be implemented by the compiler stage ("semantic analyser" takes too long to type). It will also expand all macros, check for semantic errors such as redefining functions, and try to eliminate certain classes of programmer error.



Lexical scoping is a scheme for deciding which instance of a variable a given name refers to – it originated with ALGOL, and is common today in languages such as C and Haskell.

See the following example:

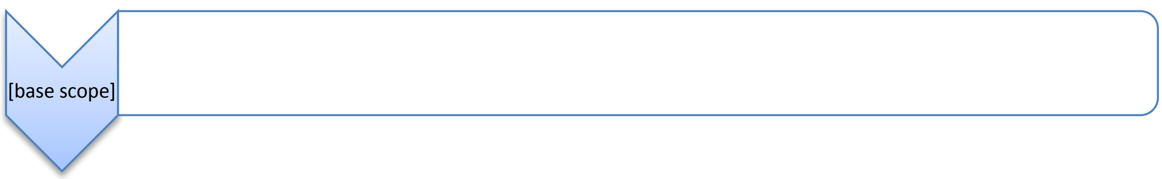
```
function main()
{
    var int a, b, c;
    a = 5;
    if (a)
    {
        var int a;
        a = increment(a);
    }
    output(a, debugout);
}
```

The correct output for this code is 5. The `if` block creates a new variable, which is also called `a`; this variable is incremented, but the original `a` of value 5 is unaltered, as the new variable has lexical priority.

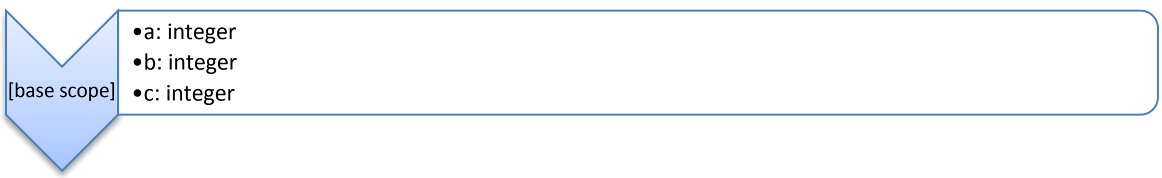
The most suitable data structure for this is a stack of `map`s. Upon entering a function block, a new `map` is pushed – new variable names are placed in this `map`. When leaving a block, the previous block's variables are exposed. This "Last In First Out" (LIFO) behaviour is exactly what stacks were designed for.

When looking up a variable, the first `map` is searched – if the variable is not present then the next level is searched, and the next, until either the variable is found, or the search eventually bottoms out and determines that the variable does not exist. The compiler should report this error to the user.

To illustrate this data structure:

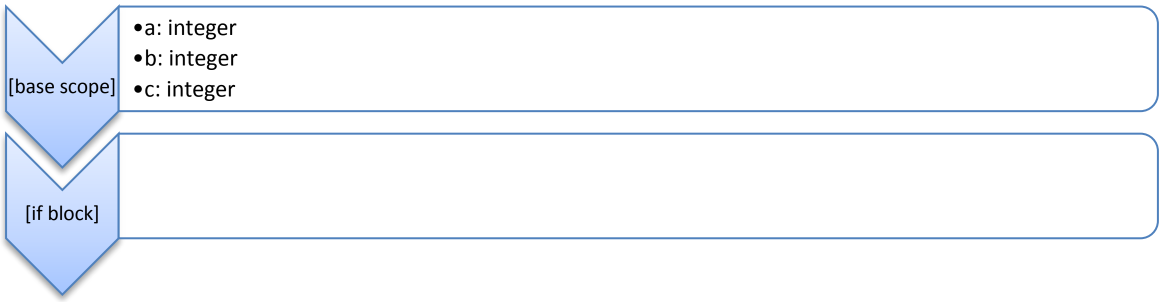


Initially, there is one empty scope. The compiler will reach the declaration of `a`, `b`, `c` and add these to the current scope:

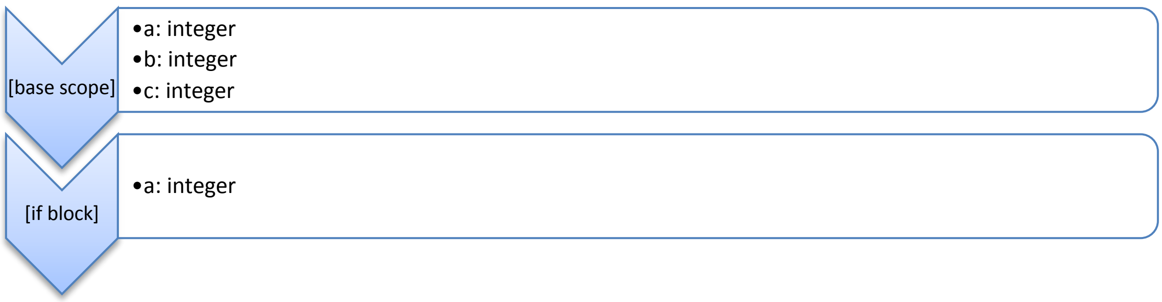


It reaches the assignment statement: it looks up the variable with the name “a”, and writes 5 to its location.

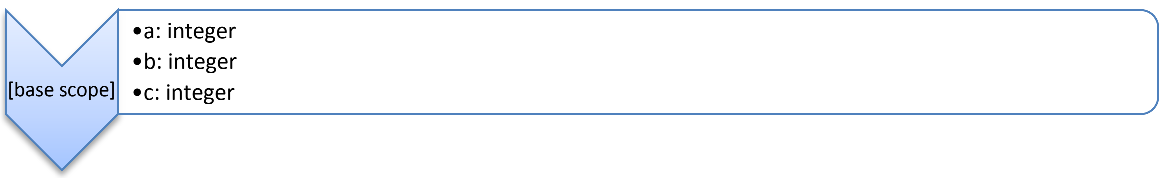
The compiler then enters an `if` block: this is a new scope, so another scope is pushed onto the stack (it grows downwards):



It reaches the new declaration of a, and pushes it onto the stack:



The assignment operation searches for a variable called “a” – our algorithm finds this in the innermost scope, so uses this location. The end of the block is now reached, so the if scope is popped:



The function call to “output” takes the variable a as its argument – the top of the stack now exposes the original a, not the one that was incremented, giving the correct value of five.

The compiler will traverse the tree using a set of recursive functions (most likely to just be overloaded functions for the various types present in the tree). Tree traversal should mostly be of the pre-order variety: it is necessary to make decisions based upon what type of node is being processed before attending to its leaves, as an `if` node will behave fundamentally differently to a `while` node.

Type checking will also be implemented as a recursive traversal of the tree, applying a set of rules to infer the type of an expression from its constituents, and to check that expression’s actual type is compatible with the expected type (e.g. assigning a 16 bit value to an 8 bit variable should raise an error).

The “program object” will be a list of function definitions, which must include a main function (the entry point for the linker). It will use the same tree structure as the original AST but with new type information, and variable names resolved to globally-unique names, as the linker will be ignorant of scope.

Linker

The linker will perform low-level code generation, and allocate memory to variables. A separate component will be responsible for managing the memory, and allocating it in the most efficient way possible that does not cause conflicts when code shares memory (“clobbering”).

As the assembly code is only an intermediate representation, a textual format would be awkward and inefficient (it need not be human-readable). A list of `struct`s with tag fields would be more appropriate.

The linker will also prepend and append a prologue and epilogue to the main function as it’s generated – these pieces of executable code will set up the machine’s memory for language runtime features such as pointers (yes, pointers are a runtime feature…), and the epilogue will halt execution so that the computer does not continue to execute undefined memory. All functions after the main function will be linked into the executable following the prologue of the main function.

The assembly code will allow the linker to be more flexible in how it deals with addresses: for instance, when generating a jump forward to a location, it will request a unique label name, pass that name in lieu of the address, and mark the actual location of the label when it reaches that point in the code (similar to labels in a textual assembly language). It will also support expressions, in a simple recursive manner.

The simplest way to allocate memory is to have an array of Booleans denoting whether each memory location is in use at a given point in the program, and to search through this table for the first free space of sufficient size. This is only practical because of the extremely small size of RAM (16 384 bytes); the approach breaks down for larger sizes.[3] See the following pseudocode (aka BASIC with arrows instead of equals):

```
MemInUse[MemSize]: Boolean Array
FirstSpace = 0: Integer
Vars: Dictionary

Function FindFirstSpace
    For i From 0 To MemSize - 1
        If Not MemInUse[i] Then
            Return i
        End If
    Next i
    Error "No more free space!"
End Function
```

[3] A typical implementation of `malloc()` provided by an operating system will often use a tree structure of progressively-halving memory blocks at the large scale (for O(log n) memory complexity) and may then coarsen the leaves (drop back to linear) at a smaller scale for performance reasons.

```
Function GetSpace(Size)
    Pos ← FirstSpace
    While Pos < MemSize
        Start ← Pos
        EnoughSpace ← True
        While Pos < Start + Size And EnoughSpace
            If MemInUse[Pos] Then EnoughSpace ← False
            Pos ← Pos + 1
        End While
        If EnoughSpace Then
            For i From Start To Start + Size
                MemInUse[i] ← True
            Next i
            If Start Is FirstSpace Then FindFirstSpace
            Return Start
        End If
    End While
    Error "No more space!"
End Function

Function AddVar(Name, Size)
    Vars.Add(Name, {Position: GetSpace(Size), Size: Size})
    Return Vars[Name].Position
End Function

Function Remove(Name)
    If Not Vars.Contains(Name) Then Error "Tried to free unknown variable"
    Var ← Vars[Name]
    For i From Var.Position To Var.Position + Var.Size - 1
        MemInUse[i] ← False
    Next i
    If Var.Position < FirstSpace Then FindFirstSpace
    Vars.Remove(Name)
End Function
```

The brunt of the work is in the `GetSpace` function; this is essentially a variant on linear search, with two pointers instead of one. It searches for the first contiguous block of memory of size `Size`, and it passes over each location in the array no more than once. `FindFirstSpace` makes this job slightly simpler, by giving `GetSpace` a sensible starting point. The rest of the code is simple bookkeeping.

The linker should be able to compile to different memory spaces: growing downwards from the end of RAM for programs that are resident in ROM, and growing upwards relative to the end of the program for programs that are to be loaded into RAM by the operating system. The machine supports direct addressing only, so the relative addresses will be resolved by the linker and calculated by the assembler.

The linker should also have an option to strip out unused functions to reduce executable size: the mark and sweep algorithm, used in garbage collectors, is well suited to this task. Essentially:

1) Build a directed graph of which functions depend on which other functions. This graph may have cycles. (Aka a dependency list for each function, the compiler phase should generate this as it goes through.)
2) Starting at the root node (the main function), explore the directed dependencies in a recursive manner. For each node explored:
   a) If the node has already been marked as explored, bottom out.
   b) Otherwise mark it as explored (used).
   c) Repeat the procedure recursively for each node (function) depended on by this one.

3) Once the tree is explored, iterate through the list of nodes (functions) and erase any that are not marked.

Assembler

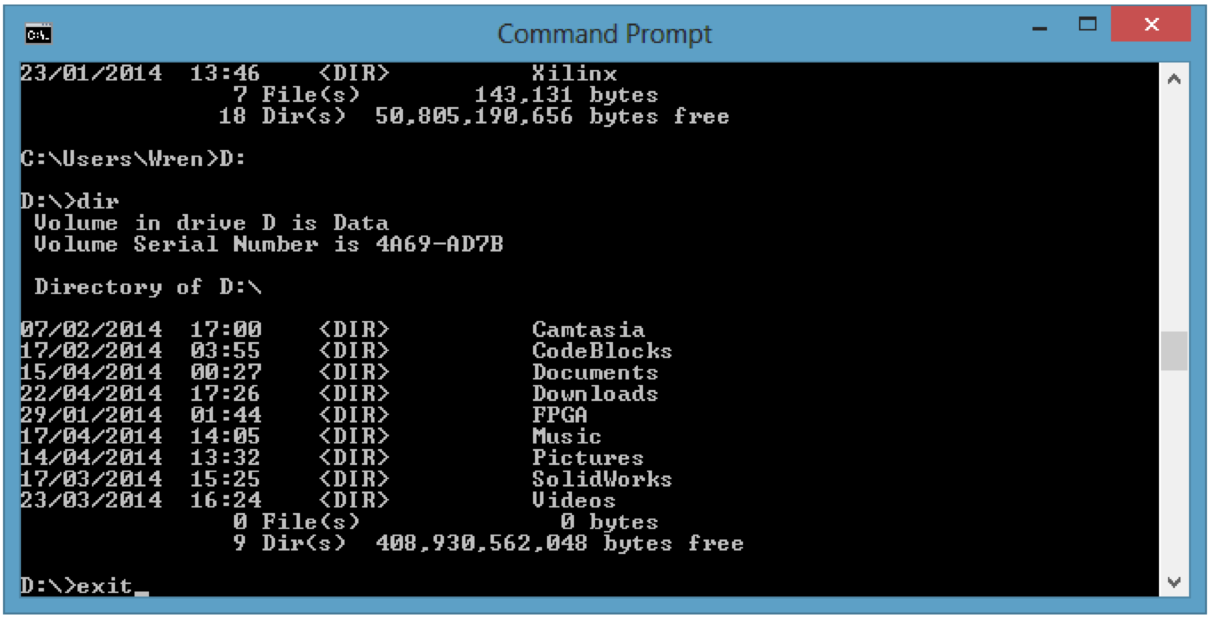
The assembler will be the simplest part of the compiler: the linker will already have generated code down to the level of individual instructions, and the job of the assembler will be to produce the final executable binary.

The linker will provide an assembly listing for the entire program as well as the symbol table describing which value should be substituted for each symbol used in the assembly code. The assembler's task is then simply to scan through the assembly and perform any necessary substitutions, as well as evaluating any expressions (e.g. when the linker has output an address relative to an unknown location).

It is also the assembler's task to build the final ROM image: this includes padding the machine code to the correct size and inserting the constant lookup tables at the end of ROM.

Interface

The compiler itself will be a standalone executable with a simple command-line interface. Details such as the input filename, output location, options such as stripping unused functions from the executable and various output formats will be specified with command-line arguments, and the executable's main function will parse these arguments and handle the sequencing of the compiler stages.



An example of a command-line interface. At the bottom, the user is entering a command.

Input and output of files will be done directly to the file system, and stdout will be used to report error or success (so the IDE can read results easily with `popen()` or similar). The compiler will output nothing on success[4] and return zero, and will output an informative preformatted error message on error and return non-zero.

An IDE will provide a more user-friendly graphical interface to the compiler, although this is a separate program – the command line interface is sufficient for programming.

Commands

A command-line invocation of the program will consist of the following:

- The name of the compiler executable
- The input file name and the output file path
- Any option flags to be used for this program

The parts of the command will be delimited by spaces. Each option will consist of a dash (-) followed by a single lower-case letter. The options will be as follows:

- -s: strip
    - Any functions defined in the code but not used will be stripped from the final executable.
    - This will reduce program size, but will not be suitable if another program links against this one (i.e. if the -e option is given).
- -r: ram
    - This is for use when the program being compiled is being loaded into RAM.
    - Program instructions will be addressed relative to 0x8000 (the start of RAM) rather than 0x0000 (the start of ROM), and variables will be allocated from the end of the program forward rather from the end of RAM backwards.
- -e: export
    - This will create and addition ".def" file which exports any functions and global variables defined/declared in the program.
    - This is used when functions need to be used/linked against by other programs, e.g. operating system syscalls.

For example:

For example:

```
spoon -s helloworld.spn helloworld.bin
```

A command like this may be used to compile a small example program: the strip option would remove all of the unused library code and make the executable smaller.

If we were compiling an operating system or bootstrap program which we want to later link against, we would use the export option:

```
spoon -e os.spn osimage.bin
```

[4] "Silence is golden"- UNIX and similar

This command would create a second file, osimage.bin.def, containing export declarations for all of the functions and global variables, as well as actually compiling the source file.

Later, when the user has written a program to be loaded into RAM by the operating system, the user would run a command similar to the following:

```
spoon -r hexedit.spn hexedit.bin
```

The simple command-driven interface makes it easy to automate the compilation of large numbers of files with tools such as GNU make, or to create an front-end GUI that interfaces with the compiler program at the back end.

Reports

The user will need some form of feedback from the compiler as to what happened – whether bad or good – and to provide some sort of information about faults in the source file that allows the programmer to track them down and fix them.

I mocked up these reports by typing white monospace text into a black textbox. This proved to be rather authentic.

Success

Upon success, the compiler should return zero, to inform the calling program (the shell, an IDE) that no errors have occurred.

The only output from a successful compilation will be a brief message stating the size of the file:

```
Executable Size: 1368 (0x530) bytes.
```

Failure

Upon success, the compiler will return a non-zero value (0xffff). The calling program will then know immediately that there has been a problem, without having to in some way parse or interpret the output on stdout.

There is a range of errors that may occur. If we take a chronological view of the points in the compilation process at which they may arise, the first would be the user entering an incorrect command. At this point a “usage message” will be displayed:

```
Usage: spoon [-ers] (inputfile) (outputfile)
```

This will tell the user the correct format for a command.

It may also occur that a source file can not be opened, because the supplied file path is wrong or the file does not exist. In this case, the message will be simple:

```
Error: could not open file: somefile.spn
```

All errors will start with the word “Error:”, just to make things *really obvious*.

The tokenizer will report errors with the character-level syntax: if a character is received where it is not expected, a generic message will be printed, similar to the following:

```
Error: unexpected character near "@", on line 1
```

And more specific messages for things such as comments and strings:

```
Error: expected " to close string near EOF
```

Syntax errors will always be in the form of something not being where it is expected, or being where it is not expected. This means we can use a very generic form of error, “Error in (file): unexpected token near (token text) on line (line number): expected (expected type), got (received type).”

For example, the user may see the following:

```
Error in wrong.spn: unexpected token near "123" on line 3:
expected semicolon, got number
```

The compiler can have a list of “friendly” token type strings, to convert from its own internal representation to something understandable by the user.

The compiler will report when a function call has the wrong number of arguments:

```
Error: too many arguments to function f on line 123
```

And when there is a type mismatch:

```
Error: Type mismatch in assignment of variable p: was expecting
pointer but got int (line 42)
```

The only other class of error is that which will occur at link-time. This stage should be fairly bullet proof, but there is a hard limit upon program size. Upon reaching it, the user will see the following:

```
Error: program too big!
```

Language Specification

The language is called Spoon. (As the successor to Fork, this was the only logical choice, besides “Knife”. Knife would be banned in schools anyway, unless I rounded the ends and made it out of plastic.)

The syntax of the language is defined by the following BNF/regular expressions:

```
<program> ::= {<constdef>|<funcdef>|<macrodef>|<vardecl>}

<constdef> ::= `const` <type> <name> `=` <expression> `;`

<funcdef> ::= `function` [<type>] <name> `(` <type> <name> {`,`
<type> <name>} `)` <block>

<type> ::= `int`|`pointer`|`void`

<macrodef> ::= `macro` <name> `(` <name> {`,` <name>} `)` <block>

<block> ::=  <statement>
         |   `{` {<vardecl>} {<statement>} `}`

<vardecl> ::= <type> <varname> {`,` <varname>} `;`

<varname> ::= <name> [`[` <number> `]`] [`=` <expression>]

<statement> ::=  <name> `(` <expression> {`,` <expression>} `)` `;`
             |   <name> `=` <expression> `;`
             |   `if` `(` <expression> `)` <block> [`else` <block>]
             |   `while` `(` <expression> `)` <block>
             |   <name> `:`
             |   `goto` <name> `;`
             |   (`break` | `continue` | `return`) `;`



<expression> ::= <value> [(`&&` | `||`) <expression>]

<value> ::= `!`<value>
         |  `(`<expression>`)`
         |  <name>
         |  <name> `(`<expression> {`,` <expression>}`)`
         |  <name> `[`<number>`]`
         |  <string>
         |  `{`<number> {`,` <number>} `}`
         |  <number>

<name> ::= [a-zA-Z_][a-zA-Z0-9_]*

<number> ::= (0x[0-9a-fA-F]+)|([0-9]+)
          |  `'`<char>`'`

<string> ::= \"[^\"]*\"
```

Where non-terminals are in `<angle-brackets>`, terminals are in `` `backquotes` ``, `[x]` represents optional `x` and `{x}` represents optional multiple `x` (a Kleene closure).

Identifiers and operators are all case sensitive.

A program is a set of declarations: of functions, of macros, of variables and of constants. The only stipulation is that there must exist a function called `main`, which the linker will use as the program entry point. `main` takes no arguments and returns `void`.

Types

There are 3 main types: `void`, `int` and `pointer`. You can also use the name `char` instead of `int`, or `int16` instead of `pointer`, but they are logically and internally the same.

`void` is a pseudo-type, and represents the result of a function that does not return anything. You can define a variable of type void, but trying to assign anything but the result of a void function to it will generate a type error. (Why you would want to do this is beyond me, but it is logically consistent). It has a size of 0 bytes.

`int` has a size of one byte (8 bits). It is the basic unit of arithmetic, and can also represent a single ASCII character. The nickname `char` is a completely equivalent type, but is a little more natural when writing text processing code (a la C).

`pointer` is 2 bytes long, or 16 bits. It refers to a single 8-bit word of machine memory, or can be used to represent a large integer, such as the size of a file in bytes (hence the pseudonym `int16`).

The only compound type is the array. If you need to represent a complex data structure (like C's `struct`), use an array.

Variable Declarations and Scope

Variable declarations can take place anywhere inside a block, but scoping is at a block-level – this has the effect that all variable declarations are processed upon entering a block.

Multiple variables of the same type can be declared at once, and each variable can be initialised with any valid expression:

```
var int x = 5, y = f(x);
```

The initialisations are guaranteed to be performed left-to-right, so f will receive an argument of 5 in the above example.

While the declarations are processed as though they were at the top of a block, the initializations are compiled in the position they are written, so there is no loss of sequencing.

Variables are lexically scoped, as per Algol, C, Haskell and Common Lisp. For example:

```
function main()
{
    var int a, b, c;
    a = 5;
    if (true)
    {
        var int a;
        a = increment(a);
    }
    output(a, debugout);
}
```

Will result in an output of 5, because the new `a` defined inside the `if` block "shadows" the old one, so the original `a` is unchanged.

Arrays

The language allows you to define arrays:

```
var int arr[5];
```

An area of size n * sizeof(type) bytes is allocated by the linker and guaranteed not to be touched by other variables for the duration of the current block, e.g. an array of 5 pointers would be 10 bytes. You can assign into arrays if you have a constant offset:

```
arr[3] = f(x);
```

The following, however, is invalid if x is a variable:

```
arr[x] = f(x);
```

As addition of pointers is a library function rather than a language builtin. (It is not possible to generate the above code statically). This can be written as the following:

```
write(addPointer(arr, x), f(x));
```

Using addPointer() from the stdmath library.

Array initialisers are not supported:

```
var int numbers[5] = {1, 2, 3, 4, 5};
```

Because this list syntax is actually a special case of a string literal, so is of type `pointer` (consider that the table must be stored in ROM).

The following:

```
var pointer numbers = {1, 2, 3, 4, 5};
```

And

```
var int numbers[5];
memcpy(numbers, {1, 2, 3, 4, 5}, 5);
```

Are both perfectly valid (using memcpy from the stdmem library).

Strings

Strings are constant arrays in ROM. When evaluated in expressions, they result in pointers. Attempting to write to these pointers is valid code, but will not result in any change (it's called read only memory for a good reason).

These:

```
var pointer str = "Hello!";
```

And

```
var pointer str = {'H', 'e', 'l', 'l', 'o', 33, 0};
```

Are the same (ASCII, null-terminated).

String literals are immutable (as a consequence of the laws of physics as well as language semantics) – in order to perform mutating string operations you must first copy the string into a memory buffer using `strcpy()` or similar.

```
var char buffer[64];
var pointer p = buffer;
p = strcpy(p, "I'm going to ");
p = strcpy(p, "concatenate some strings!");
println(buffer);
```

Functions

Functions are defined with the following syntax:

```
function int f(int x)
{
   f = increment(shiftleft(x));
}
```

Note that “BASIC-style” returns are used (assign to function name). The `return` statement does exist, but it takes no arguments and simply exits the function.

The return type is optional –if omitted, the function returns void.

Arguments are passed by value, so the following:

```
function int sumto(n)
{
     sumto = 0;
     while (n)
     {
        sumto = add(sumto, n);
        n = decrement(n);
     }
}
```

Does not modify the original expression for n.

Recursion is not supported, as there is no machine stack – however, any recursive algorithm can be easily converted into an iterative one through the use of a stack, and stacks are easy to implement with pointers. Indirection to the rescue.

Memory allocated in one function is guaranteed to not be touched by another function (unless other functions do pointer arithmetic and don’t use bounds checking). This means all variables are the equivalent of `static` variables in C. See the following:

```
function int howManyTimesHaveIBeenCalled(int initialise)
{
     if (initialize)
          howManyTimesHaveIBeenCalled = 0;
     else
          howManyTimesHaveIBeenCalled =
increment(HowManyTimesHaveIBeenCalled);
}
```

(This `static`ness includes the function variable). After having been called initially with an argument of true, the above function will return 1, then 2, then 3… etc.

Operators and Operator Precedence

There is no operator precedence. The only operators which are syntactically separate, as opposed to builtins using the function call syntax, are `&&` and `||`, which are really control flow operators; they operate based on short-circuit behaviour, as in C, Python, Lua and other languages:

- `&&`: evaluate the first argument. If it is true, evaluate the second argument and return it. If the first argument is false, return false immediately.
- `||`: evaluate the first argument. If it is true, return true immediately. If it is false, evaluate the second argument and return it.

These are useful if you want to make a decision based on a function of a pointer, but are not sure whether the pointer is valid(non-null):

```
if (pobject && some_function_of(pobject))
    do_something_to(pobject);
```

The function of the object will only be evaluated if the pointer is non-null.

These operators have equal precedence, and are left-associative. Precedence can be changed by the use of brackets to group expressions (see the logic operator test program).

Control Flow

Besides && and ||, the available control flow keywords are `if`, `else`, `while`, `break`, `continue`, `return` and `goto`. These are almost entirely "borrowed" from C.

`if` and `else` work as expected. The `else` clause is optional. If the result of the `if` expression is non-zero, then the `if` branch is taken. If false, the `else` branch is taken, or the `if` branch is simply skipped if the `else` clause is omitted. The only quirk is that, if the expression results in a multi-byte value (i.e. a pointer), the decision is made based on the first (most significant) byte only. Use the `first()` and `second()` builtins if this is not the behaviour you're after.

As a consequence of the fact that a block can be either a statement or multiple statements between braces, we can "chain up" `if`s and `else`s to make an `else if`:

```
if (a)
    func_a();
else if (b)
    func_b();
else
    func_c();
```

There is in fact no `else if` keyword. If the above were written out fully, it would look like this:

```
if (a)
{
    func_a();
}
else
{
    if (b)
    {
        func_b();
    }
    else
    {
        func_c();
    }
}
```

`while` loops run the loop body for as long as the result of the expression is non-zero (with the same caveat as the `if` expression). If the expression is zero upon the first evaluation, the loop body is not entered at all. The `continue` statement jumps back up to the test expression (e.g. when we want to move on the the next item in a list) and the `break` statement exits the current loop.

The `return` statement exits a function (jumps to the epilogue) but, as noted in the function specification, it does not take an argument, as value returns are achieved by assigning to the function variable.

`goto` jumps to an address or label:

```
loop:
    x = increment(x);
    goto loop;
```

`goto`s are not very useful on their own, and in fact it is generally best to avoid them (see: Djikstra, Goto Considered Harmful). However, in conjunction with macros and the existing control flow operators, it is possible to implement entirely new control flow operators from within the language, as per `TAGBODY`s and `GO`s in Lisp.

Labels follow lexical scope, in the same way as variables.

Macros

Macros allow frequently-used blocks of code to be defined, and then used in-line.

For example:

```
macro output(src, dest)
{
    var int temp;
    nfc(temp, val(0xff));
    nfc(temp, src);
    nfc(dest, temp);
}
```

This macro from stddefs is used to output a value to an output port: this is a frequent operation, and code becomes much more descriptive as a result of the macro's use.

Each "call" to a macro is expanded inline. This means they can take up more space than functions because of repetition, but the performance is usually higher. Furthermore, each time a macro argument is referred to within the macro body, this is also an inline substitution: this may have unintended side effects, as functions passed as macro arguments can be called multiple times.

Builtin Functions

To make the language useful, a number of functions are built-in. These are usually sufficiently small to be generated in-place (and will be, for reasons of speed), and so do not use the regular calling convention.

Name (Signature)	Description
`int and(int, int)`	Returns the bitwise and of two integers. E.g. and(3, 5) = 1.
`int andnot(int, int)`	Returns the bitwise and of the first integer and the complement of the second. This takes only three machine instructions (the

	same as a byte copy).
`int decrement(int)`	Returns the value of the argument – 1. Implemented as a table lookup.
`int first(pointer)`	Returns the first (big endian => more significant) byte of a pointer.
`void nfc(ref, ref)`	Compiles down to one single NFC machine instruction. This one is a special case – arguments are evaluated as follows: • Functions are called and the return address ends up in the NFC instruction. • Variables' addresses end up in the NFC instruction. • Literal numbers are used directly as addresses. In other words, everything is as standard unless you pass a number or constant in directly, e.g. nfc(debugout,debugin), in which case it will use that constant as an address. To perform NFCs with a value and not an address, use the val(int) operator.
`void nfc4(ref, ref, ref, ref)`	Same as nfc(ref, ref) but arguments for all 4 fields are supplied, as opposed to the compiler stitching in the next instruction automatically. Used in combination with macros and labels, you can create your own control structures.
`int not(int)`	Returns the bitwise complement of the argument, e.g. not(0) = 0xff.
`int or(int, int)`	Returns the bitwise or of its arguments.
`pointer pair(int, int)`	Returns the pointer value that results from concatenating its two arguments. The first argument is the more significant, i.e. first in memory (big endian).
`int read(pointer)`	Returns the result of reading from the passed pointer (dereferences the pointer). If a literal number is passed in as the address this will compile down to *0* instructions. (This is still semantically significant, e.g. compare the meaning of "x = debugin;" and "x = read(debugin);": these are both single instructions but one assigns 0xc001 to x whilst the other reads a value from the debug input port).
`int second(pointer)`	Returns the second (less significant) byte of a pointer.
`int shiftleft(int)`	Shifts its argument to the left by one bit, e.g. shiftleft(10) = 20. Implemented as a table lookup.
`int shiftright(int)`	Shifts its argument to the right by one bit, e.g. shiftright(10) = 5. Implemented as a table lookup.
`pointer val(int)`	Takes an 8 bit integer as its input and returns an address that corresponds to that integer, e.g. "nfc(debugout, val(0xff));" will clear the debug output port.
`void write(pointer, int)`	Writes a byte to a given location (lval pointer dereferencing).
`int xor(int, int)`	Returns the logical xor of the two arguments.

Other functions such as addition, copying chunks of memory, finding the length of strings etc. will not be built into the compiler, but instead be part of the standard library. A complete listing of the standard library can be found in Appendix 2.

Headers and Libraries

Rather than writing everything from scratch every time you create a new program, it is convenient to be able to reuse common code and functionality. These collections of code are known as libraries.

The header/library model used by spoon is very simple: libraries are included with a `#include` statement, as such:

```
#include "stddefs"
#include "forkos/forkos.bin.def"
```

Each include statement is processed at parse-time, and the included file is loaded and parsed, and its definitions added to the parser's output. Headers can contain either function definitions or merely declarations, so long as each function is defined exactly once in total in the main file and all included files.

Header guards are not used or necessary; the compiler will ignore an include statement for a file that has already been included.

The compiler searches for the included file first in the directory of the main file, and then in the compiler executable's directory. If it is found in neither directory then the user is informed of the problem and the compiler exits.

Implementation

My plan for implementation was to start at the front end of the compiler – i.e., the tokenizer – and progressively work backwards.

This meant that I could use each completed stage to provide input for the next one – for instance, in order to generate a token stream to input into the parser, one could simply push some sample source code through the tokenizer.

I avoided the use of any libraries apart from C++'s standard template library (STL), to make the code more portable (compiles and runs unaltered on my phone and my Raspberry Pi) and to make things easier when swapping between machines (don't have to build hundreds of dependencies on various platforms).

Code such as flex and yacc is available to take away the heavy lifting of the lexer and parser, but these are heavy-weight tools and there's no net benefit[5] when using these for a language specifically designed to be easy to parse.

Safe Practices

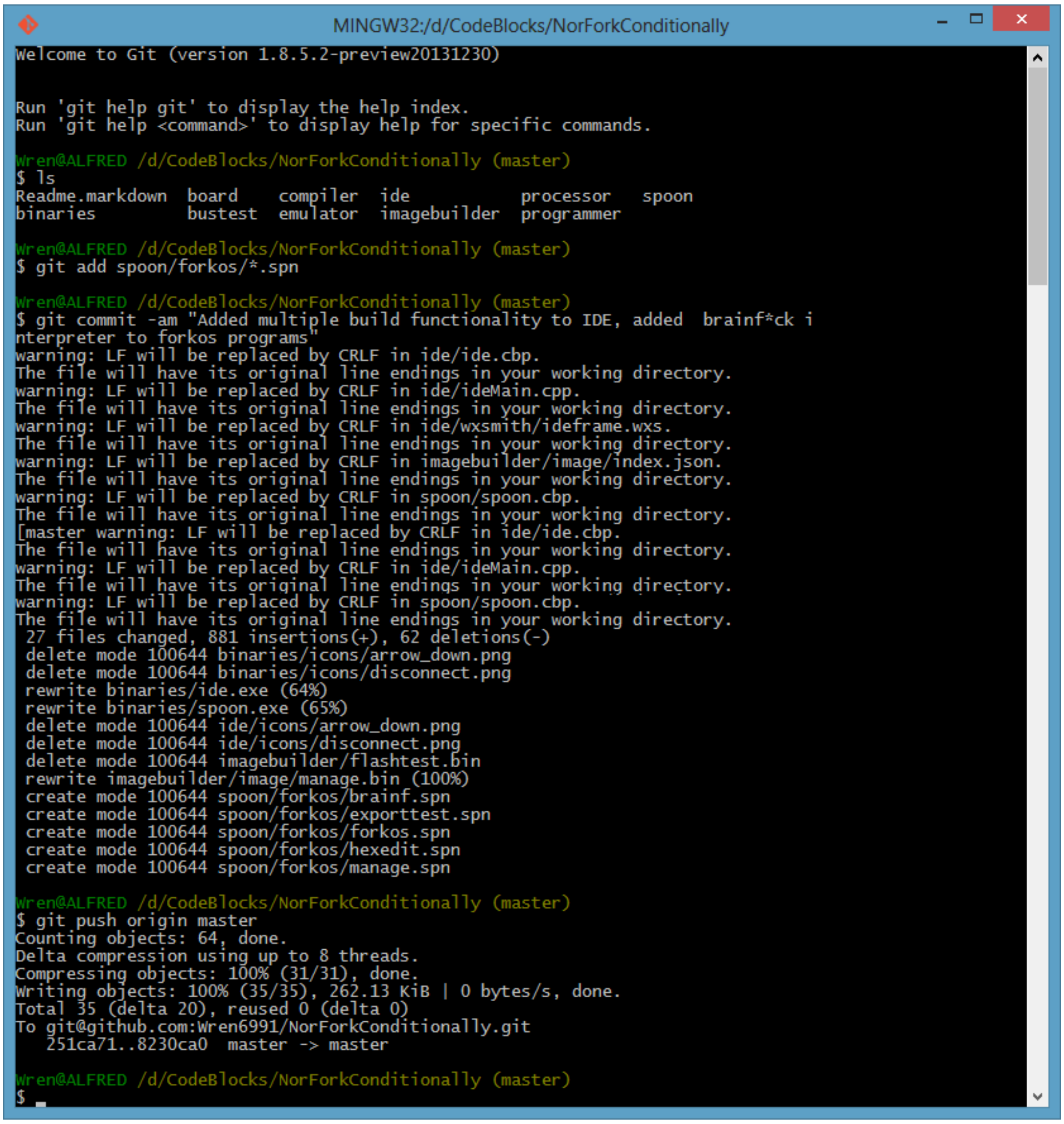
During development, I kept backups of my code. The classic way to do this is to periodically zip everything up and copy that zip to another place, hopefully offsite, where it will be safe in the case of thermonuclear war or your cat walking on your keyboard.

We live in modern times, however, and the tool of choice today is a distributed version control system, or DVCS. As a matter of preference, I used git. This gives me an audit trail of every single change that has been made to the code – I can look at the code as it was at any point in time, see development comments, and see deltas between prior and successive "commits".

This level of control over your own source code is invaluable – upon discovering a bug, it is possible to examine the entire history of the code to find the changes that may have introduced it.

[5] The "Don't use it" criterion: complexity of tool ≥ complexity of problem

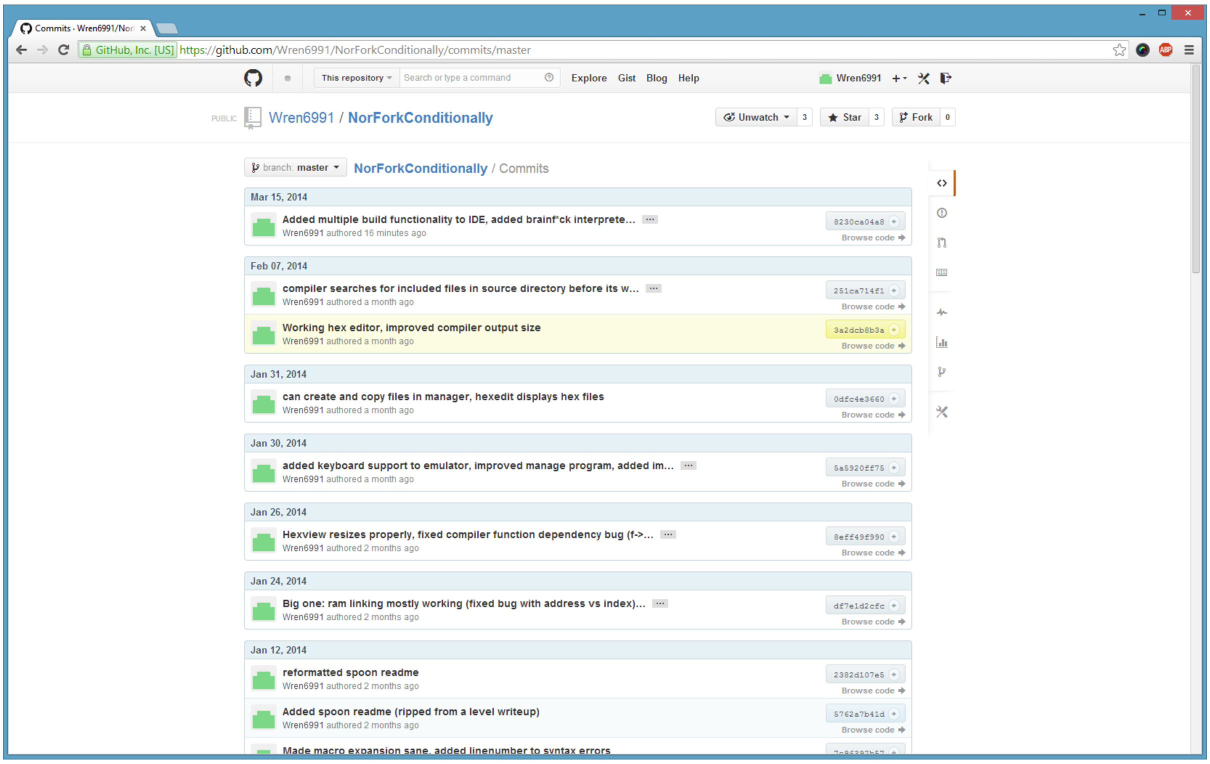
Here's an example git session – I add some files, commit changes to the repository, and push these changes (the "master") to a remote backup (the "origin").



```
MINGW32:/d/CodeBlocks/NorForkConditionally
Welcome to Git (version 1.8.5.2-preview20131230)


Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

Wren@ALFRED /d/CodeBlocks/NorForkConditionally (master)
$ ls
Readme.markdown  board    compiler  ide           processor   spoon
binaries         bustest  emulator  imagebuilder  programmer

Wren@ALFRED /d/CodeBlocks/NorForkConditionally (master)
$ git add spoon/forkos/*.spn

Wren@ALFRED /d/CodeBlocks/NorForkConditionally (master)
$ git commit -am "Added multiple build functionality to IDE, added  brainf*ck i
nterpreter to forkos programs"
warning: LF will be replaced by CRLF in ide/ide.cbp.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in ide/ideMain.cpp.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in ide/wxsmith/ideframe.wxs.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in imagebuilder/image/index.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in spoon/spoon.cbp.
The file will have its original line endings in your working directory.
[master warning: LF will be replaced by CRLF in ide/ide.cbp.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in ide/ideMain.cpp.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in spoon/spoon.cbp.
The file will have its original line endings in your working directory.
 27 files changed, 881 insertions(+), 62 deletions(-)
 delete mode 100644 binaries/icons/arrow_down.png
 delete mode 100644 binaries/icons/disconnect.png
 rewrite binaries/ide.exe (64%)
 rewrite binaries/spoon.exe (65%)
 delete mode 100644 ide/icons/arrow_down.png
 delete mode 100644 ide/icons/disconnect.png
 delete mode 100644 imagebuilder/flashtest.bin
 rewrite imagebuilder/image/manage.bin (100%)
 create mode 100644 spoon/forkos/brainf.spn
 create mode 100644 spoon/forkos/exporttest.spn
 create mode 100644 spoon/forkos/forkos.spn
 create mode 100644 spoon/forkos/hexedit.spn
 create mode 100644 spoon/forkos/manage.spn

Wren@ALFRED /d/CodeBlocks/NorForkConditionally (master)
$ git push origin master
Counting objects: 64, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (31/31), done.
Writing objects: 100% (35/35), 262.13 KiB | 0 bytes/s, done.
Total 35 (delta 20), reused 0 (delta 0)
To git@github.com:Wren6991/NorForkConditionally.git
   251ca71..8230ca0  master -> master

Wren@ALFRED /d/CodeBlocks/NorForkConditionally (master)
$
```

My reason behind choosing git is the free service they provide: github. This provides offsite hosting of git repositories: as well as my own personal backups, github have their own backup schemes to ensure the safety of anything they store, so the code is safe against even the most determined cat on a keyboard, if not worldwide thermonuclear war. Most of their datacentres are in somewhere in North America. The entire transfer process is cryptographically secure (it takes place over SSH, using some form of RSA for the encryption).

Using github's web interface I can quickly browse through my source history for changes:



Git also means I can coordinate development across the multiple machines I use: when I sit down at a different desk I simply have to run `git checkout` and I can continue development. For large teams this is vital; for me, it's just convenient.

Tokenizer

The classic lexical analyser (tokenizer, lexer, scanner can be used interchangably) is based around a finite state machine (FSM). It scans through a character stream in one pass; a state variable keeps track of the type of token that is currently being read, which allows the FSM to make decisions based upon its input history as well as the current input.

Formally, an FSM consists of:

- ∑: the input alphabet
  - In our case this is all ASCII characters (the ASCIIbet), or whichever character encoding our host operating system happens to use.
- S: the set of states
  - This will be represented with an enumerated type (effectively a fancy integer, with some rules such that it will usually only be assigned certain well defined values).
- S_0: the initial state
  - This will be the start state: the start state's transition function will contain the switching logic which makes a decision as to what type of token is currently being read based on the first character.
- δ: the transition function
  - This defines the output and new state based on the previous state and the input character.
  - E.g. $\delta: S_{number} \times \Sigma_{letter} \rightarrow S_{start} \times \omega_{number}$

    - We will implement this as a `switch`/`case` statement in a loop, with control logic for each state that makes decisions based on the current character.
- ω: the output function
    - As our output is a function of both state and input, this FSM is a Mealy machine.
    - Each state case statement will have logic that will decide what and whether to output based on the current character.

With a consistent mental model of how my code should function, I set out typing. I began by defining the header file:

```
#ifndef TOKENIZER_H_INCLUDED
#define TOKENIZER_H_INCLUDED

#include <iostream>
#include <vector>
#include <map>

#include "error.h"

typedef enum {
	t_eof = 0,	//end of file
	t_and,
	........ snippity snip
} token_type_enum;

struct token
{
	token_type_enum type;
	std::string value;
	int linenumber;
	token();
	token(token_type_enum type, std::string value, int linenumber);
	token(const token &other, int linenumber);	// for different line
number instances of const token
};

std::vector <token> tokenize(std::string);

#endif // TOKENIZER_H_INCLUDED
```

(Full listing page 94)

As is typical with C and C++ headers, the header is enclosed within a “header guard” – this is a set of preprocessor commands that causes the file to be effectively ignored if the (hopefully) unique symbol has already been defined in the current translation unit.

The header defines the different types of token, declares a `struct` that represents each individual token, and declares the `tokenize()` function so that other code can use it by including the header.

We can now start defining the source/implementation file. I began by defining the tokenizer’s possible states:

```
enum state_enum
{
	s_start = 0,
```

```
    s_number,
    s_number_hex,
    s_name,
    s_string,
    s_slashaccepted,
    s_linecomment,
    s_streamcomment,
    s_staraccepted,
    s_whitespace,
    s_charliteral,
    s_expectingapostrophe,
    s_logicoperator
};
```

I added some stub functions to make character checks a little more descriptive:

```
inline bool is_digit(char c)
{
    return c >= '0' && c <= '9';
}

inline bool is_hex_digit(char c)
{
    return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'f') ||
    (c >= 'A' && c <= 'F');
}

inline bool allowed_in_name(char c)
{
    // uppercase, lowercase, digit or underscore:
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||
           (c >= '0' && c <= '9') || c == '_' || c == '+' ||
           c == '-' || c == '*';
}

inline bool is_whitespace(char c)
{
    return c == ' ' || c == '\t' || c == '\n' || c == '\r';
}
```

The function body then essentially consists of a loop that reads characters, and contains a switch statement which implements the switching logic:

```
std::vector <token> tokenize(std::string str)
{
    std::map<std::string, token_type_enum> keywords;
    keywords["break"] = t_break;
    keywords["char"] = t_type;
    ........

    std::map<char, token> symbols;
    symbols[':'] = token(t_colon , ":");
    ........

    std::vector <token> tokens;
    int linenumber = 1;
```

```
    int index = -1;
    const char *buffer = str.c_str();
    char c;
    int startindex = index;
    state_enum state = s_start;
    do
    {
        c = buffer[++index];
        if (c == '\n')
            linenumber++;
        switch(state)
        {
            case s_start:
                startindex = index;
                if (is_digit(c))
                    state = s_number;
                else if (symbols.find(c) != symbols.end())
                    tokens.push_back(token(symbols.find(c)->second,
linenumber));
                else if (allowed_in_name(c))
                    state = s_name;
                else if (is_whitespace(c))
                    state = s_whitespace;
                ........
                break;
            case s_string:
                ........
                break;
        }
    } while (c);
    return tokens;
}
```

This implementation of an FSM is absolutely classic – the other pattern you will see is to use `gotos`[6] and labels, and have the state be implicit in the machine program counter rather than have a centralised dispatch state.

I tried to avoid mixing different models in my code; while in rapid implementation I had some lookahead logic (`if (buffer[index + ...]...)`) in some of my states, which does not fit at all well with the state machine model. The code's structure became more consistent when I replaced the lookaheads with extra states, which has the added benefit that all of the buffer access occurs in one location (abstracted the FSM from the buffer).

Parser

The compiler's structure is very much a linear flow of data – each stage has a different intermediate representation for both its input and its output. Before implementing the parser, I defined its output representation (syntaxtree.h, page 100) – I won't reproduce it here, but it essentially consists of

[6] As an aside: this will actually run faster, because it makes better use of the CPU's branch predictor (creating a correlation betweens FSM state and CPU program counter lets the predictor make useful decisions) but I decided on maintainability over a few percent efficiency. Also, GCC tends to monkey around with the jumps and negate the benefits. (See computed gotos, distributed dispatch in VMs and emulators).

`structs`[7] which contain pointers to other `structs` and a few STL containers/types for data storage (classic tree structure). This is the closest thing there is to a physical embodiment of the BNF.

It is an abstract syntax tree (AST) in that it implements the underlying tree structure of the syntax, but doesn't care about trivialities such as where the brackets are or whether the commas are in the right place – we can say it is *abstracted* from the user-facing syntax, and this makes the job easier later. The structure also includes a few extra fields which the parser doesn't use (the compiler does), but these are just ignored at this point.

The parser is based around the recursive descent algorithm, as described in the parser design section starting page 20. In short, the idea is to take the recursive definition of the BNF and mirror it with a set of recursive functions.

The parser object contains the token list to be parsed, some state (current position, current token etc.) and the declarations of the functions that perform the actual recursive descent.

```
#ifndef _PARSER_H_INCLUDED_
#define _PARSER_H_INCLUDED_

#include "syntaxtree.h"
#include "tokenizer.h"

#include <map>
#include <set>
#include <vector>

class parser
{
    std::string filename;
    std::vector<token> tokens;
    std::map <std::string, type_enum> typestrings;
    std::set <std::string> includedfiles;
    int index;
    token t;
    token lastt;
    void gettoken();
    bool accept(token_type_enum type);
    void expect(token_type_enum type);
    public:
    parser(std::vector<token> _tokens, std::string _filename = "file");

    // these functions are more or less 1-1 with syntaxtree.h
    program* getprogram();
    void do_preprocessor(program *prog);
    void throw_unexpected(std::string value, int linenumber = 0,
token_type_enum expected = t_eof, token_type_enum got = t_eof);
    definition *getdefinition();
        constdef* getconstdef();
        funcdef* getfuncdef();
    ........
    block* getblock();
    statement* getstatement();
    ........
};
```

[7] Some people shun `structs` in C++ and will choose instead to write `class { public:` every single time, or have a mystical belief that `structs` shouldn't have constructors. This is silly.

```
#endif // _PARSER_H_INCLUDED_
```

With this top level code in place, implementation was more or less a case of fill in the gaps, testing as I went. I started with the token stream accessors, so that I could test drive the rest of the code immediately.

```
// read the next token, put it in t; make t a blank token if no more
tokens.
void parser::gettoken()
{
    lastt = t;
    index++;
    if ((unsigned)index < tokens.size())
    {
        t = tokens[index];
    }
    else
    {
        t = token();
    }
}

// optionally gobble up a token, and return whether or not we have gobbled.
bool parser::accept(token_type_enum type)
{
    if (t.type == type)
    {
        gettoken();
        return true;
    }
    else
    {
        return false;
    }
}

// gobble up a token, or raise an error if it doesn't match
void parser::expect(token_type_enum type)
{
    if (!accept(type))
    {
        throw_unexpected(t.value, t.linenumber, type, t.type);
    }
}
```

The comments should make it fairly clear what these do; `gettoken()` is only used internally by `accept()` and `expect()`, and its job is to advance the “read head” in a safe, well-defined way. A blank token has the type `t_eof` (zero); this type of token will not be the result of any characters typed by the user, so it behaves as a useful sentinel value at the end of the token stream. This also makes the errors nicer without adding special cases to the code: e.g. “Error: expected semicolon near <EOF>”.

The recursive descent model worked out very well – here is a typical function:

```
if_stat* parser::getif()
```

```
{
    resourcep <if_stat> ifs;
    expect(t_lparen);
    ifs.obj->expr = getexpression();
    expect(t_rparen);
    ifs.obj->ifblock = getblock();
    if (accept(t_else))
        ifs.obj->elseblock = getblock();
    return ifs.release();
}
```

The above function implements the following BNF:

```
<statement> ::= `if` `(` <expression> `)` <block> [`else` <block>]
```

Here we can see a few patterns that appear frequently in the parser code:

- High degree of correspondence between BNF specification and the code
  - This makes implementation much simpler, and allows useful sanity checks while debugging.
- `resourcep` template class
  - This is a tiny RAII[8] wrapper that deletes a pointer resource when the stack is unwound, unless the resource is released (source: page 106).
  - This stops memory leaks when errors are encountered and the tree is not returned (and hence left unreachable).
  - This was not used in later code as the compiler exits on the first error, meaning there was no advantage to this approach over simply relying on the operating system to release all memory on exit.
- Use of `accept()` and `expect()`
  - These are the only two ways in which the recursive descent functions will advance the token stream.
  - `expect()` behaves similarly to `accept()` except that it is used when a syntax element *must* be there; it encapsulates code which throws a nice preformatted syntax error, including line number and filename, meaning this code is kept separate from the actual parsing.
- Some of these functions are very small
  - Each node on the syntax tree is also small; the size of the function corresponds with this.
  - It’s a common feature of recursive code that complex and expressive structures “fall out” of simple code, seemingly by magic.

Not all of these were necessarily planned from the outset: I went through an iterative process of expanding, debugging, refactoring and debugging my code, and it’s natural for common functionality to rise out and bubble up to the top during this process. The rule of thumb is that one function does one job. *(Writing debugging twice was intentional)*.

[8] Resource Acquisition Is Initialization: a design pattern in unmanaged languages where the destructor of a wrapper object on the stack is used to deallocate a heap object when the stack is unwound. I actually reinvented the wheel here: I could have used something like `std::shared_ptr`.

Partway through implementation, I realised the need to read back the tree in a textual form in order to check the parser's results. The code I wrote to do this is in printtree.* (page 162). I found checking my work as I progressed to be absolutely essential: writing your entire program in a single stint and expecting everything to work at the end is beyond wishful thinking.

Having filled in all the gaps, I now had a parser which implemented the language's syntax and threw nicely formatted and informative error strings when I fed it some malformed code.

Compiler

With a working tokenizer and parser, I had a convenient way of generating abstract syntax trees. However, a syntactically valid statement may be utter nonsense when interpreted in context; this stage of the compiler traverses the syntax tree and makes sure that it is semantically valid. The two prior stages provided a working input pipeline, meaning that as a programmer I could work with source code as my data source rather than trying to manipulate the tree objects directly.

As almost always, I started by bashing out the header file:

```
class compiler
{
    scope *globalscope;
    scope *currentscope;
    std::map<std::string, symbol> globalsymboltable;
    std::set<std::string> defined_funcs;
    std::map<std::string, func_signature> functions;
    std::map<std::string, expression*> expression_subs;
    funcdef *currentfuncdef;
    void pushscope();
    void popscope();
    public:
    compiler();
    object* compile(program*);

    void compile(macrodef*);
    void compile(funcdef*);
    ...... (and the rest)
    void compile(expression*&);
    void gettype(expression*);
    bool match_types(type_t expected, type_t &received);
    void addvar(std::string name, type_t type, int ptr, int
linenumber, bool isConstant = false, int constvalue = 0);
};
```

This compiler class contains the state and functionality responsible with directly handling the AST. It maintains the scopes and the symbol tables, providing functions to manipulate them in a safe and convenient way.

The `void compile(*)` functions traverse the tree and perform whatever manipulation is necessary; the full source on page 121 includes annotations on each function, briefly stating its individual purpose.

The header also declares some supporting classes:

```
struct func_signature
```

```
{
    type_t return_type;
    std::vector<type_t> arg_types;
    bool is_macro;
    bool args_must_match;
    funcdef *def;
    bool operator==(func_signature &rhs) const;
    bool operator!=(func_signature &rhs) const;
    func_signature();
};

struct symbol
{
    std::string name;
    type_t type;
    bool is_constant;
    int value;
    symbol(){is_constant = false;}
    std::string tostring();
};

class scope
{
  private:
    std::map <std::string, symbol> variables;
  public:
    scope(scope *_parent = 0);
    scope *parent;
    void insert(std::string name, symbol var);
    symbol& get(std::string name);
    bool exists(std::string name);
    bool inthisscope(std::string name);
};
```

These objects represent the signature of a function (its return type and argument types, and operators to check that signatures match), a symbol (variable name) object which represents a variable name in context, and a scope object: this stack-of-maps structure is described in detail on page 22.

The compiler performs scope resolution: each time it encounters a variable declaration, this symbol is assigned a globally unique identifier (GUID) by concatenating its name with some unique data – the pointer to this syntax node in machine memory. When it encounters a variable in situ, the compiler checks with the appropriate scope shadowing to decide which variable it refers to, and replaces the ambiguous local name with the globally unique identifier.

Type checking is implemented by this function, matchtypes():

```
// check that received matches accepted, and refine the generic "number"
type.
// if there is no possible match, return false.
bool compiler::match_types(type_t expected, type_t &received)
{
    if (expected == received)
    {
        return true;
```

```
    }
    else if ((expected == type_int || expected == type_pointer) && received
== type_number)
    {
        received = expected;    // replace generic number with int/pointer.
        return true;
    }
    else if (expected == type_pointer && received.type == type_array)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

This applies a set of simple rules to check that the types match, and to make the general "number" type more specific (the linker will need to know this later).

A post-order tree traversal is used to climb through the expression trees and decide bottom-up the type of the expression, and whether this type is valid in the situation.

Linker

Syntax trees are now fully processed, expanded and verified. All local and scoped variables have been resolved to global symbols, meaning the linker can assume all variable names are globally unique. The code is both syntactically and semantically correct, although at this point no trace of the syntax is left. It's now the linker's job to finish things off and create the final binary code.

One of the operations the linker performs is to strip out any unused functions, to reduce output program size, if requested. It does this with a mark and sweep algorithm. In pseudocode:

```
Procedure Mark(RootFunc)
    RootFunc.Used ← True
    For Each Func In RootFunc.DependsOn:
        If Not Func.Used Then
            Mark(Func)
        End If
    Next Func
End Procedure

Procedure Sweep
    For Each Func in Funcs
        If Not Func.Used Then
            Erase Func
        End If
    Next Func
End Procedure
```

The general sequence of events here is:

- Start at the main function. Mark this function as used.
- For each dependee of the main function not currently marked as used, repeat the process.

Eventually, the entire tree will have been recursively explored, and all used functions will be marked as such. The second part of the process (the “sweep” phase) is to sweep through the list of all functions and erase any that were not marked as being in use.

This takes place as following in the code:

```
std::set<std::string> linker::analysedependencies(std::string rootfunc)
{
    funcdef *rootdef = (funcdef*)defined_funcs[rootfunc];
    rootdef->is_used = true;
    std::set<std::string> &dependencies = rootdef->dependson;
    for (std::set<std::string>::iterator i = dependencies.begin(); i !=
dependencies.end(); i++)
    {
        funcdef *def = (funcdef*)defined_funcs[*i];
        if (!def || def->type != dt_funcdef)
            continue;
        if (!def->is_used)
        {
            // analyse each dependency's sub-dependencies recursively, and
add them to this function's
            // dependencies, so we know all dependencies of a function (not
just sub dependencies) (this also marks used functions)
            std::set<std::string> nextleveldeps = analysedependencies(def-
>name);
            for (std::set<std::string>::iterator iter =
nextleveldeps.begin(); iter != nextleveldeps.end(); iter++)
                dependencies.insert(*iter);
        }
    }
    return dependencies;
}

void linker::removeunusedfunctions()
{
    std::map<std::string, definition*>::iterator fiter;
    bool noincrement = false;
    for (fiter = defined_funcs.begin(); fiter != defined_funcs.end();
fiter++)
    {
        if (noincrement)
        {
            fiter--;
            noincrement = false;
        }
        funcdef *def = (funcdef*)fiter->second;
        if (!def || def->type != dt_funcdef)
            continue;   // ignore the builtin functions.
        if (!def->is_used)
        {
            defined_funcs.erase(fiter++);
            if (fiter == defined_funcs.begin())
                noincrement = true;
            else
                fiter--;
        }
```

```
        else
            std::cout << "function " << fiter->first << " is used\n";
    }
}
```

(The first function implements the mark phase, the second one the sweep.)

In general though, the linker's job is one of final code generation and variable allocation.

The memory allocation code made for a fairly well-contained subunit, so I factored this out into a separate class:

```
class vardict;

struct variable
{
    friend class vardict;
    type_t type;
    variable *next;     // for stack operations we can build a linked list,
in case we get the same symbol twice. (Shadowing)
    linkval address;
    variable() {next = 0;}
private:
    int offset;     // we don't want to use this by mistake...
};

class vardict
{
    std::map<std::string, variable*> vars;
    std::vector<bool> memory_in_use;
    std::vector<bool> has_been_used;
    int first_available_space;
    int getspace(int size);
    void find_first_available_space(int searchstart = 0);
    std::vector<std::vector<std::string> > tempscopes;       // in an
if/while test we may use multiple temp locations, and we don't want them to
clobber each other, so we keep track of temps and clean up after test
finished.
public:
    bool start_from_top;
    linkval addvar(std::string name, type_t type);
    void registervar(std::string name, type_t type, linkval address);    //
for when we want to push an existing address as a var, and the memory is
already allocated. (it's removed in the same way)
    void remove(std::string name);
    variable* getvar(std::string name);
    bool exists(std::string name);
    void push_function_scope();     // so functions can't clobber each
other's memory, we mark all memory used by other functions as currently in
use.
    void push_temp_scope();
    void pop_temp_scope();
    void remove_on_pop(std::string name);
    vardict();
};
```

This class is intended to keep track of memory and which variables are currently in use, and to allocate and remove variables at the linker's request. (Source listing page 131).

The linker also performs the task of code generation; this is grouped into a series of functions, `emit*()`:

```
    void write8(linkval);
    void write16(linkval);
    void padto8bytes();

// Code generation routines:
    void emit_nfc2(linkval x, linkval y);
    void emit_branchifzero(linkval testloc, linkval dest, bool
amend_previous = false, bool invert = false);
    void emit_branchifnonzero(linkval testloc, linkval dest, bool
amend_previous = false);
    void emit_branchalways(linkval dest, bool always_emit = false);
    void emit_copy(linkval src, linkval dest);
    void emit_copy_inverted(linkval src, linkval dest);
    void emit_writeconst(uint8_t val, linkval dest);
    void emit_copy_multiple(linkval src, linkval dest, int nbytes);
    void emit_writeconst_multiple(int value, linkval dest, int nbytes);
    void emit_writelabel(std::string label, linkval dest);
```

`write8()` outputs a single byte to the output stream, and updates the `index` and `address` variables. The two are distinct: `index` refers to the current position in the output stream, whilst `address` refers to the current instruction's position in the memory space (the two may be different, e.g. while compiling to RAM). `write16()` writes a 16 bit value (2 bytes) – it is convenient shorthand. The implementation of `write16` is worth mentioning:

```
void linker::write16(linkval val)
{
    write8(val.gethighbyte());
    write8(val.getlowbyte());
}
```

Rather than numbers, the "linkval" class is used. This may be a simple wrapper round a number – in which case simple shift operations are performed – or it may be a symbol, or an expression. It is possible to perform shifts on and write to the output stream values which are *not yet known.*

The idea of the `linkval` is pivotal to the linker's structure: one option would have been to take a multi-pass approach, performing all of the code generation, then going back and figuring out where all the labels are, and then going and computing and filling in all of the jump targets. A linkval can take the shape of a symbol, or an "IOU" for an actual value: this means

Testing

In order to make sure that the program works as expected, and actually meets its specification, extensive testing is necessary – this is perhaps more true for a compiler than any other software, as it is utterly useless without absolute one hundred percent reliability.

It's instructive to perform both implementation-based tests – at the low, modular level, making sure that each piece of code behaves as specified – and functionality-based tests, ensuring that the system as a whole behaves as it should and produces the correct output for a range of possible tasks. Testing done with knowledge of the internals is known as white box testing, and the converse is black box testing.

In a traditional waterfall model, testing is done after implementation. In real life, this doesn't always work out. Waterfall is for engineering, not programming. Testing is an iterative process performed throughout implementation, which informs the process; the formal testing here is just the icing on the cake.

Implementation Testing

Compilation is a very well defined operation: for a certain input, a certain output is expected.

I tested each subsystem (tokenizer, parser, compiler, linker, assembler) in isolation; I defined a test plan, with a set of inputs and expected outputs, and then ran these tests on each section to ensure that the implementation was functionally correct and bug-free.

Tokenizer

During testing, I outfitted the tokenizer with a simple "test harness" – stub routines for input and output and for outputting the token stream in a human-readable way.

The tokenizer consists of only one function (`tokenize()`) which contains the lexing state machine, so the simplest way to proceed is to provide a range of stimuli and assess the response.

Correct Test Cases

These input data are well-formed: the tokenizer should not output any errors, and should output the expected token stream.

Input	Expected Output		Comment
`x _x1`	`name:` `name:`	`"x"` `"_x1"`	Ensure that names are correctly lexed, including those beginning with underscores and containing numbers.
`123 0xff`	`number:` `number:`	`"123"` `"255"`	Numbers should be correctly parsed, and hexadecimal literals should be parsed into decimals.

`comment1`
 	
`hello //comment2`
 	
`comment3`
 	
`/*comment`
`4~~~~~~~*/`
	`name:`
	`name:`
	`name:`
		`"comment1"`
		`"hello"`
		`"comment3"`
			Line and stream comments should be ignored. The line following a line comment should be correctly lexed. The part of a line preceding the line comment should be correctly lexed.

`char int int16`
`pointer void`
	`type:`
	`type:`
	`type:`
	`type:`
	`type:`
		`"char"`
		`"int"`
		`"int16"`
		`"pointer"`
		`"void"`
			These should be correctly recognised as type keywords.

`break const`
`continue else`
`function goto if`
	`"break":`
	`"const":`
	`"continue":`
		`"break"`
		`"const"`
		`"continue"`
			All these keywords should be correctly identified and assigned the relevant type

<table>
<tr><td>macro return var
while</td><td>"else":
"function":
"goto":
"if":
"macro":
"return":
"var":
"while":</td><td>"else"
"function"
"goto"
"if"
"macro"
"return"
"var"
"while"</td><td>descriptor.</td></tr>
<tr><td>"Hello world!"</td><td>string:
world!"</td><td>"Hello</td><td>Strings should be correctly parsed, with no missing or extra characters at the beginning or end.</td></tr>
<tr><td>'a'</td><td>number:</td><td>"97"</td><td>Char literals should be parsed as numbers.</td></tr>
<tr><td>:,=#!(){}[];</td><td>colon:
comma:
"=":
"#":
"!":
"(":
")":
"{":
"}":
"[":
"]":
semicolon:</td><td>":"
","
"="
"#"
"!"
"("
")"
"{"
"}"
"["
"]"
";"</td><td>Symbols should be correctly recognised and lexed.</td></tr>
<tr><td>&& ||</td><td>"&&":
"||":</td><td>"&&"
"||"</td><td>Logic operators should be correctly lexed into their individual tokens.</td></tr>
</table>

Boundary Cases

This data set consists of data that are on the very verge of being incorrect, or are simply unusual.

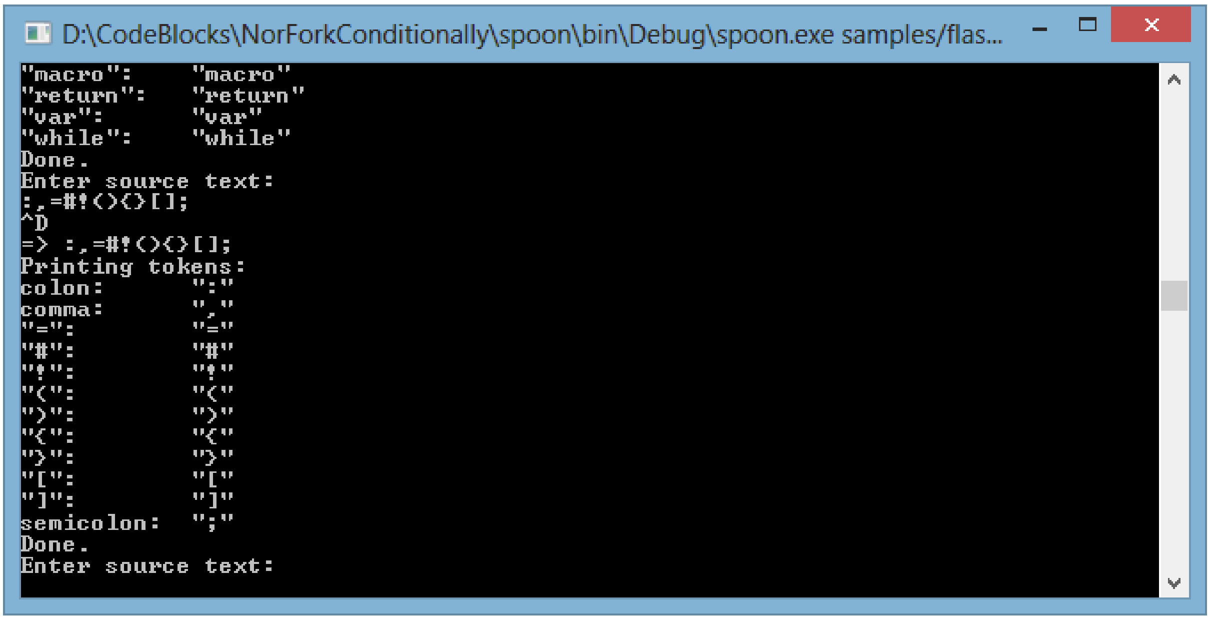
Input	Expected Output		Comment
(empty string)			An empty string should be valid input for the tokenizer (surprisingly this is the minimum length).
"Hello world!"	string: world!"	"Hello	Embedded linefeeds and carriage returns inside strings are perfectly valid (there are no escape sequences).
1230xff	number: name:	"1230" "xff"	Names may not start with numbers: these are two separate lexemes.
'''	number: (assuming ASCII)	"39"	Any character can appear inside a char literal, including another single quote.
/* /* /* /* Comment! */			Stream comment symbols are not required to be balanced (GCC, clang etc. would output a warning if this happened)

Erroneous Test Cases

These input sequences are malformed: the tokenizer should output an error.

Input	Expected Output	Comment
`/* comment !`	`Error: unclosed stream comment near EOF`	An unclosed comment should raise an error.
`Comment */`	`Error: unexpected character near "*"`	An asterisk is not at the start of any valid token.
`"Hello world! Whoops forgot to close this string`	`Error: expected " to close string near EOF`	Strings should always be closed with a matching " character.
`'a`	`Error: expected ' to close char literal near EOF`	Char literals should always be closed.
`''`	`Error: expected ' to close char literal near EOF`	The second quote is treated as being “inside” the literal, so there is no closing quote here.
`'ab'`	`Error: expected ' to close char literal near "b"`	Char literals should contain one and only one character.
`|\`	`Error: malformed logic operator near "\"`	Slipped off the shift key. (Double pipe is valid but single pipe is never valid.)
`&7`	`Error: malformed logic operator near "7"`	Whoops! (An ampersand should always be followed by another one, as per C’s short circuit operators.)

For testing purposes, I also whipped up a quick interactive prompt – here is an example session:



Parser

Once the tokenizer is tested, you have a known working tool for generating token streams. This is good!

It is now easy to produce test data for the parser from source code. However, in order to assess the parser's output, we need some code to traverse the tree and output it in a textual format – this is itself a nontrivial operation, so the code has been included for your viewing enjoyment (printtree.*, page 162).

This is an example of input and output for tokenizer->parser->printtree.cpp: *(Input:)*

```
const pointer debugout = 0xc000;
const pointer debugin = 0xc001;
const int true = 0xff;
const int false = 0x00;

function int decrement(int x);
function int read(pointer p);

function sleep(int time)
{
    while (time)
        time = decrement(time);
}

function main()
{
    while (true)
    {
        var int period;
        period = read(debugin);
        sleep(period);
        nfc(debugout, val(0xff));
        sleep(period);
        nfc(debugout, val(0x00));
    }
}
```

(Output:)

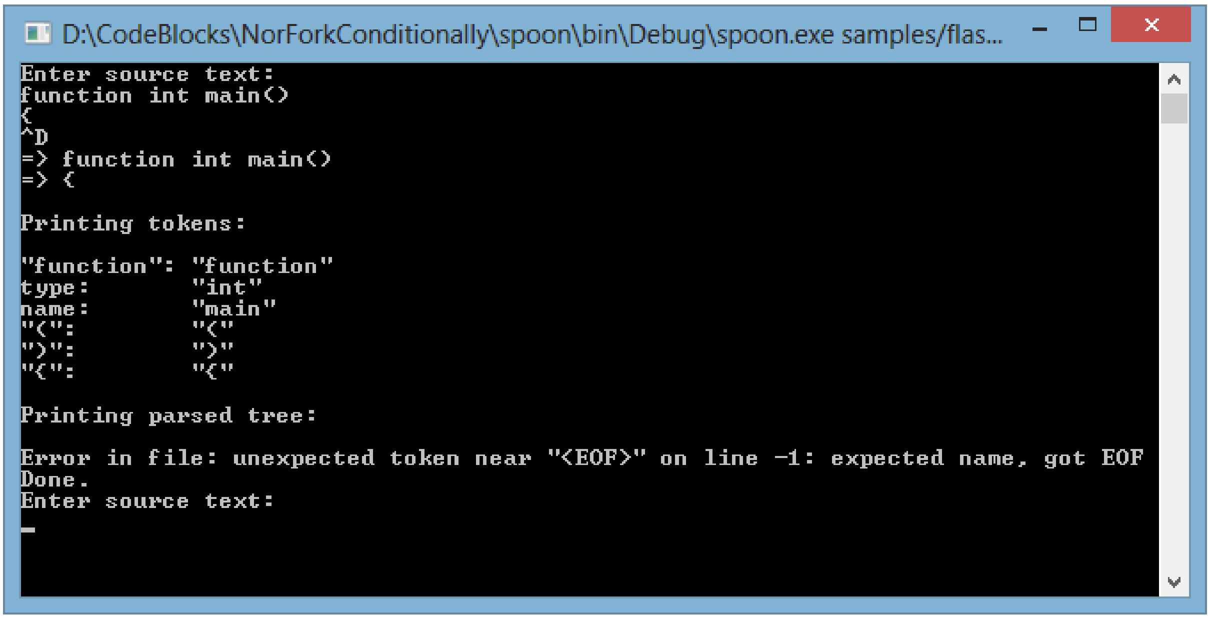
```
Printing parsed tree:

constant pointer debugout: 49152
constant pointer debugin: 49153
constant int true: 255
constant int false: 0

Definition of function decrement: (int) => int:
 Depends on:
  (declaration only)

Definition of function read: (pointer) => int:
 Depends on:
  (declaration only)
```

```
Definition of function sleep: (int) => void:
 Depends on:
{
  While time Do
  {
    Setting time to decrement(time)
  }
}

Definition of function main: (void) => void:
 Depends on:
{
  While true Do
  {
    Declared int period
    Setting period to read(debugin)
    Call to function sleep
      Argument: period
    Call to function nfc
      Argument: debugout
      Argument: val(255)
    Call to function sleep
      Argument: period
    Call to function nfc
      Argument: debugout
      Argument: val(0)
  }
}
Done.
```

As you can see, the output is more or less the same as the input – this is what we're hoping for. The syntax should survive the round trip: if it doesn't, something somewhere has gone wrong.

Do *not* be fooled into thinking that the transformation is purely textual in nature; all internal operations are based around a recursive tree structure, the textual output is simply more easily understandable by a squishy human operator.

Here's an example of an interactive prompt session testing some syntax (^D is the terminator):



Correct Test Cases

Input	Expected Output	Comment
`var int x;`	`Declared int x`	

`macro m(a, b) { }`	`Definition of macro m:`
	`(a, b):`
	`{`
	`}`
		Macro definitions should be correctly parsed, with argument names.
		(The parsing of the block is not specific to `macro` so will be tested later.)

`function f();`	`Definition of function f: (void) => void:`
	` Depends on:`
	`  (declaration only)`
		Function declarations should be properly parsed.
		Note how the parser should take missing types to be implicit `void` (optional type keywords are part of the syntax, not the semantics).

`function f() {}`	`Definition of function f: (void) => void:`
	` Depends on:`
	`{`
	`}`
		Ignore the empty “Depends on” list: this is set by the compiler and then used by the linker’s mark and sweep phase to remove unused functions if the user requests it.

`function int f(int x, pointer y) {}`	`Definition of function f: (int, pointer) => int:`
	` Depends on:`
	`{`
	`}`	

`function int main()`	`Definition of function`	`main()` should not

{
var int x = 5;
x = f(x);
g();
}
	main: (void) => int:
	Depends on:
	{
	Declared int x
	Setting x to 5
	Setting x to f(x)
	Call to function g
	}
		return int, but the parser doesn’t care about this. Ensure that variables can be declared inside blocks, and that initializers work (although an initializer outside of a block is a syntax error unless const). Ensure function calls are parsed both as expressions and statements.

function main()
{
f();
var int x, y = 5;
}
	Definition of function main: (void) => void:
	Depends on:
	{
	Declared int x
	Declared int y
	Call to function f
	Setting y to 5
	}
		Multiple declarations should be parsed into a series of individual declarations for the compiler to process. Declarations should take effect at the top of the block irrespective of their position, but assignments should appear in correct sequence.

var char buffer[32];	Declared int[32] buffer	Test the array declaration syntax.

function main()
{
var pointer str = "Hello world!";
}
	Definition of function main: (void) => void:
	Depends on:
	{
	Declared pointer str
	Setting str to "Hello world!"
	}
		Make sure that string constants work (str could actually be of any type, the parser doesn’t understand these things).

function main()
{
var pointer hexdigits = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f’};
}
	Definition of function main: (void) => void:
	Depends on:
	{
	Declared pointer hexdigits
	Setting hexdigits to "0123456789abcdef"
	}
		Test the alternative form of string constants (value lists).

function main()
{
if (x)
f();
else
g();
}
	Definition of function main: (void) => void:
	Depends on:
	{
	If x Then
	{
	Call to function f
	}
	Else
	{
	Call to function g
		Test if statements and else clauses with single statements.

<table>
<tr><td></td><td><pre>
  }
}</pre></td><td></td></tr>
<tr><td><pre>function main()
{
    if (x || y)
    {
        var int a;
        f(a);
    }
}</pre></td><td><pre>Definition of function
main: (void) => void:
 Depends on:
{
  If (x || y) Then
  {
    Declared int a
    Call to function f
      Argument: a
  }
}</pre></td><td>Should also be able to use blocks instead of single statements.
Should be able to declare variables inside the nested block.
(Short circuit operators included).
The parser doesn’t care that <code>x</code> and <code>y</code> are undeclared.</td></tr>
<tr><td><pre>function main()
{
    while (a)
        b();
    while (x)
    {
        break;
        continue;
    }
    return;
}</pre></td><td><pre>Definition of function
main: (void) => void:
 Depends on:
{
  While a Do
  {
    Call to function b
  }
  While x Do
  {
    Break
    Continue
  }
  Return
}</pre></td><td>While statements, for both single statements and blocks.
Breaks and continues.</td></tr>
<tr><td><pre>function main()
{
    loopstart:
    if (x && !y)
        goto loopstart;
}</pre></td><td><pre>Definition of function
main: (void) => void:
 Depends on:
{
  Label: loopstart
  If (x && !(y)) Then
  {
    Goto loopstart
  }
}</pre></td><td>Test <code>!</code> operator nested inside of an expression.
Test labels and <code>goto</code>s.</td></tr>
<tr><td><pre>function f()
{
    x = a || b || c ||
d;
}</pre></td><td><pre>Definition of function
f: (void) => void:
 Depends on:
{
  Setting x to (((a ||
b) || c) || d
}</pre></td><td>Test left-associativity of short circuit operators.
(Note carefully the grouping of brackets.)</td></tr>
<tr><td><pre>function main()
{
    x = a && b || !(c
&& d) || e;
}</pre></td><td><pre>Definition of function
main: (void) => void:
 Depends on:
{
  Setting x to (((a &&
b) || !((c && d))) ||
e)
}</pre></td><td>Test use of brackets to alter precedence.</td></tr>
</table>

Boundary Cases

Input	Expected Output	Comment
`(empty string)`		The parser should have no problems with empty strings: it should simply do nothing.

`function main()`
`{`
`    var pointer p;`
`    var int x = p;`
`}`
	`Definition of function main: (void) => void:`
	`Depends on:`
	`{`
	`  Declared pointer p`
	`  Declared int x`
	`  Setting x to p`
	`}`
		The parser should not check for type errors; the code to the left is semantically nonsense but syntactically valid.

Erroneous Cases

Input	Expected Output	Comment
`int x;`	`Error: expected keyword near "int" on line 1`	The correct syntax is "`var int x;`".

`macro m(a, b)`
`    do_stuff(a, b);`
`}`
	`Error in (file): unexpected token near "}" on line 1: expected "{", got "}"`	Macro definitions should be correctly parsed, with argument names.
		(The parsing of the block is not specific to `macro` so will be tested later.)

`function f;`	`Error in (file): unexpected token near ";" on line 1: expected "(", got semicolon`	Forward declarations of functions must still include the argument signature.
`function int f(x, y) {}`	`Error in (file): unexpected token near "x" on line 1: expected ")", got name`	Each argument name must be prefixed by a type – otherwise the argument list should end.
`var int x = 5;`	`Error: initialization of globals not supported. (do it in main())`	Variables should not be initialized outside of a block, unless const (this is defined by syntax).
`var char[32] buffer;`	`Error in (file): unexpected token near "[" on line 1: expected name, got "["`	The name should come before the array qualifier.

`function main()`
`{`
`    if x`
`        f();`
`    else`
`        g();`
`}`
	`Error in (file): unexpected token near "x" on line 3: expected "(", got name.`	If statements must have brackets around the conditions.

`function main()` `{`	`Error in (file): unexpected token near`	`&&` and `||` can only appear after a valid

<table>
<tr><td><pre>   x = && x;
}</pre></td><td><pre>"&&" on line 3:
expected number, got
"&&"</pre></td><td>expression.</td></tr>
<tr><td><pre>function main()
{
    x = x && ;
}</pre></td><td><pre>Error in (file):
unexpected token near
";" on line 3: expected
number, got semicolon</pre></td><td><code>&&</code> and <code>||</code> must be followed by a valid expression.</td></tr>
</table>

Compiler

At this point in the test plan, we have verified that the first two stages can take source code as input and produce a valid abstract syntax tree (AST) for further processing. This means we can use source code as our input rather than raw binary ASTs.

These test input sequences (source code samples) are intended to test the language's semantics: type checking, scope and variable name resolution, checking function arguments and macro expansion.

Correct Test Cases

<table>
<tr><th>Input</th><th>Expected Output</th><th>Comment</th></tr>
<tr><td><pre>
function main()
{
    var int a, b;
    a = b;
}</pre></td><td><pre>No errors</pre></td><td>Assignments of variables of the same type should parse correctly.</td></tr>
<tr><td><pre>function int f(int x)
{
    f = x;
}
function main()
{
    var int a;
    a = f(a);
}</pre></td><td><pre>No errors</pre></td><td>Result of function can be assigned to a variable of the correct type.</td></tr>
<tr><td><pre>function int f(int x)
{
    f = x;
}

function main()
{
    var int a;
    f(a);
}</pre></td><td><pre>No errors</pre></td><td>No need to do anything with the result of a function, even if it is non-void.</td></tr>
<tr><td><pre>function main()
{
    var int a, b;
    if (a || b)
        a = b;
    if (a && b)
        a = b;
    while (!a)
        a = b;
}</pre></td><td><pre>No errors</pre></td><td>Should be able to use any two integers for || and &&, and any single integer for !.
Integers should be valid conditions for if and while loops.</td></tr>
</table>

Boundary Cases

Input	Expected Output	Comment

function f()
{}
 	
function main()
{
    var void x;
    x = f();
}
	No errors	void is a valid type. Assigning void to void is a valid no-op.

	Definition of function main: (void) => void:
	 Depends on:
	{
	  Declared pointer p
	  Declared int x
	  Setting x to p
	}
		The parser should not check for type errors; the code to the left is semantically nonsense but syntactically valid.

Erroneous Cases

Input	Expected Output	Comment

function main()
{
   var int x;
   var pointer p = x;
}
	Error: Type mismatch in assignment of variable p: was expecting pointer but got int	Only variables of matching types can be assigned to one another.

function main()
{
    var int x;
    var int x;
}
	Error: duplicate declaration of variable x on line	Variables can not be redeclared within the same scope.

function int f(int x)
{
    f = x;
}
function main()
{
    var int a;
    var pointer p;
    a = f(p);
}
	Error: Type mismatch in argument to function f: was expecting int but got pointer(line 9)	Arguments to functions must be of matching types.

function int f(int x)
{
    f = x;
}
function main()
{
    var int a;
    var pointer p;
    p = f(a);
}
	Error: Type mismatch in assignment of variable p: was expecting pointer but got int(line 9)	Can not assign a function result to a variable of non-matching type.

function int f(int x) {	Error: not enough arguments to function	Must supply the correct

`    f = x;`
`}`
`function main()`
`{`
`    var int a;`
`    a = f();`
`}`
	`f`	number of arguments for a function.

`function int f(int x)` **`{`** **`    f = x;`** **`}`** **`function main()`** **`{`** **`    var int a;`** **`    a = f(a, a);`** **`}`**	`Error: too many arguments to function f`	Functions must be supplied with the correct number of arguments.

Assembler

As one departure from order, we test the assembler before the linker. The linker is vastly more complex, and it is much easier to test the linker if the actual resulting machine code can be tested (the assembler is effectively a second stage of the linker in the case of this compiler).

As the representation is binary (see page 135), there is no syntax as such – any input sequence will produce a “valid” output, and it is up to the preceding stages to make the assembler do something useful. The only erroneous cases that the assembler checks for are missing/incorrect linker symbols, and null pointers for expression arguments; these simple sanity checks prevent a lot of headaches and crashes.

Linker

We can create valid program objects: local variable names are resolved to globally unique symbols, all types are guaranteed to be compatible, functions are defined once and only once, and each statement is in a semantically valid location and its arguments have been checked, resolved and expanded.

This means we can use source code to represent program objects, and provide guaranteed valid test data for the linker; if the compiler stage passes all of its tests then the program object must be valid.

We need to check code generation, static variable allocation, function scoping (avoiding memory clobbering between functions), label generation, function linking and the generation of

Correct Test Cases

Input	Expected Output	Comment

`function main()`
`{}`
(empty program)
	Test the prologue: load the output in the emulator and run it. The memory starting at 0xbfc0 in RAM should now read:
	`bfc0: xxxx 7d00 bfc8 bfc8`
	`bfc8: xxxx bff0 bfe0 bfe0`
		This prologue writes code into RAM which will later be modified and used by the program for the purpose of reading and writing pointers.

	bfd0: bff0 7d00 bfd8 bfd8 bfd8: bff0 xxxx bfe0 bfe0 bfe0: bff1 7d00 xxxx xxxx	
function main() {} (empty program)	Program should halt at a fixed address	Ensures that the halt instruction inserted at the end of the program works correctly.
function main() {} (empty program)	Starting at 0x7c00 in ROM, the constant tables should be correctly inserted as such: • For 0x7c00->0x7cff, the value should be the low order address byte + 1. • For 0x7d00->0x7dff, the value should be the low order address byte – 1. • For 0x7e00->0x7eff, the value should be the low order address byte times 2. • For 0x7f00->0x7fff, the value should be the integer part of the low order address byte divided by 2.	These four lookup tables are necessary for all of the arithmetical operations the computer may carry out.

function main()
{
    var int x, y;
}
 	
//(run compiler in debug mode)
	adding var x@9565f8 0x0
	adding var y@9565f8 0x1
		Each variable should be assigned its own location in memory.

function main()
{
    var int x;
    {
        var int x;
    }
}
 	
//(run compiler in debug mode)
	adding var x@3f6950 0x0
	adding var x@3f6810 0x1
		Variables with the same names in different scopes should have the same name internally, and any clashes should be avoided by assigning separate locations.

function main()
{
    {
        var int x;
    }
    {
        var int x;
    }
}
 	
//(run compiler in debug mode)
	adding var x@d965c8 0x0
	adding var x@d96588 0x0
		Variables that can not possibly clash (being in distinct blocks within the same function) should be assigned to the same location to save space.

function main()
{
    var pointer x, y, z;
}
 	
//(run compiler in debug mode)
	adding var x@3f6768 0x0
	adding var y@3f6768 0x2
	adding var z@3f6768 0x4
		Variables of sizes larger than one byte should be spaced appropriately in memory.

function main()
{
    var char str[32];
    var int x;
}
 	
//(run compiler in debug mode)
	adding var str@906648 0x0
	adding var x@9066e8 0x20
		Arrays should have the appropriate storage space allocated (0x20 = 32).

Boundary Cases

Input	Expected Output	Comment

function main()
{
    var int x, y;
    x = y;
    x = y;
    x = y;
    ...... (1303 times)
}
	(Extremely large program)
	No errors; output size of 31736 bytes.
		This program is right at the size limit of what can be fit into the 31KiB of available ROM space before the constant tables.pointers.

Erroneous Cases

Input	Expected Output	Comment

function main()
{
    var int x, y;
    x = y;
    x = y;
    x = y;
    ...... (2000 times)
}
	Error: program is too big!	The compiler should inform the user of the problem rather than crashing or forging blindly ahead and chopping off part of the program.

(empty program)	Error: no definition of function main.	The main function is the program's entry point: the previous stages do not care about this, but the program can not be linked if it is missing.

function int f(int x);
 	
function main()
{
    var int a;
    a = f(a);
}
	Linker Error: unknown builtin function f	The function f has been forward declared, so the semantic stage should have no problems.
		The linker should notice that f has no definition, and should complain.

Functional Testing

However thorough the test plan, nothing beats actually *using* your software for an extended project and watching for the flaws that inevitably crop up.

Operating System

The rationale behind my functional testing was as such:

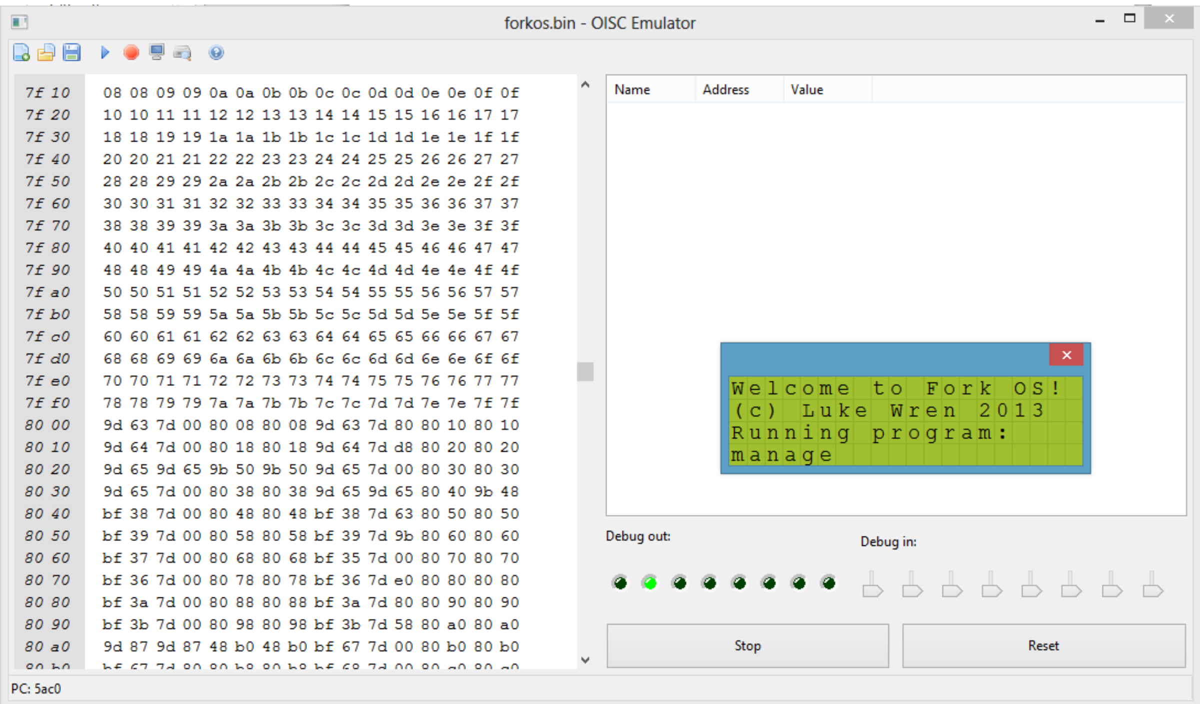
"If you can write an operating system in it, it's a pretty reliable language."

With this principle in mind, I set out and wrote a simple operating system with the following features:

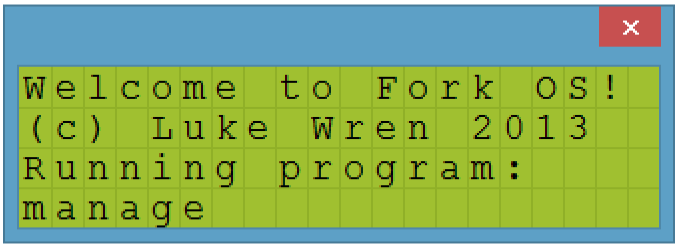
- LCD driver to control a small character liquid crystal display
- Keyboard driver for input
- Mass storage driver and file system
- Ability to load and run programs from mass storage
- Suite of included programs:
    - File manager
    - Hex editor
    - Some simple games
    - An interpreter for another simple language, written in my compiled language

The entire listing of this operating system and the mass storage disk image can be found in Appendix 3.

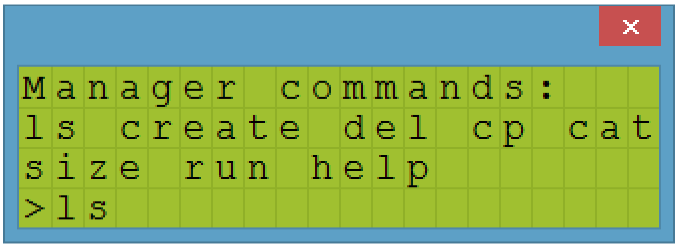
To give an idea of the scope of this piece of software, I opened it up in my emulator and took a few screenshots:

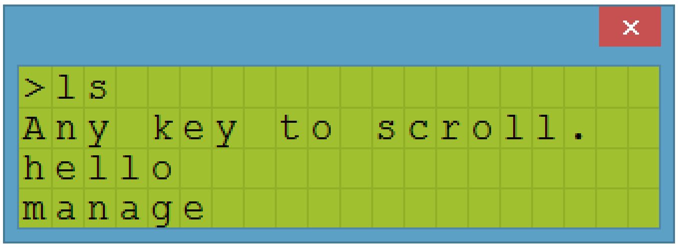


Upon boot, and after running the runtime setup for the language, it prints a welcome message to the user and starts loading the shell program (the driver for the LCD was also written in Spoon: it controls a real physical LCD controller, the Hitachi HD44780, which this emulator simulates).

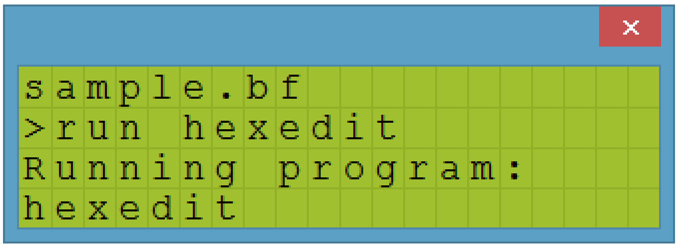


Using the flash driver and library it is essentially one line of code to load a program: the binary is loaded verbatim from flash storage into memory and then jumped to. The user is given a list of commands and prompted for input – here I enter an `ls` command:

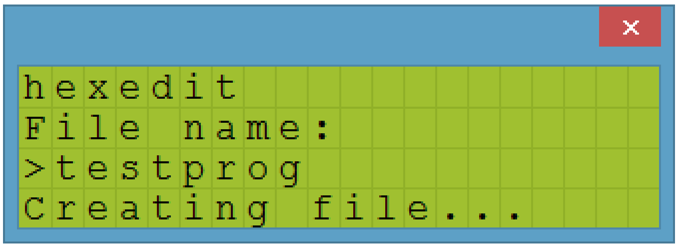




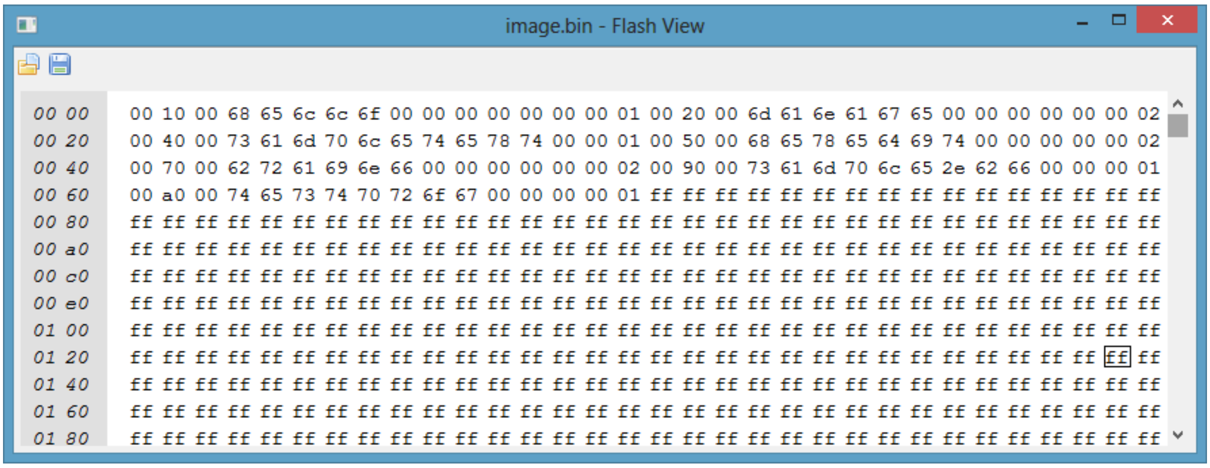
As with Unix (etc) this lists files. The file system is flat (directory-less) so this lists all the files on disk.



I scroll down the list of files and see the name that I'm looking for – the hex editor. I issue the run command. The shell parses this, looks through the file system to find the location of this file on disk, loads it into memory and off we go!

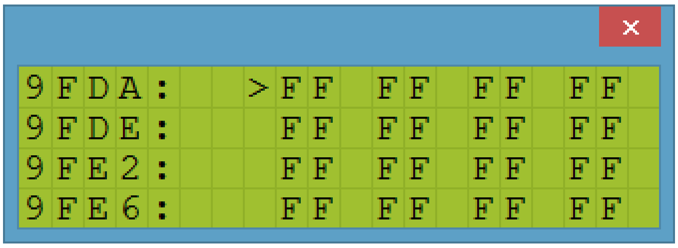


Notice how the scrolling screen buffer is persistent between different programs. The hex editor asks for a filename – this file doesn't yet exist, so it creates it.

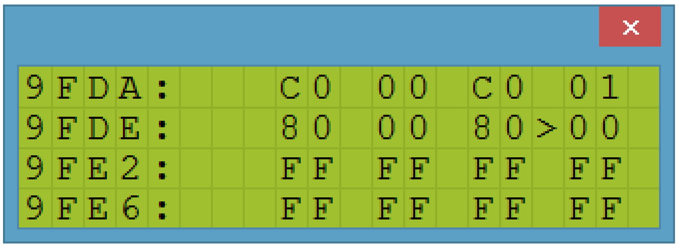


On the fourth line of this hex display you can see the entry for the new file in the file allocation table (FAT). The `createfile` function uses a one dimensional version of AABB intersections to find the first space on disk into which the file will fit without overlapping any existing files, erases the target sectors, and creates the new entry in the FAT.

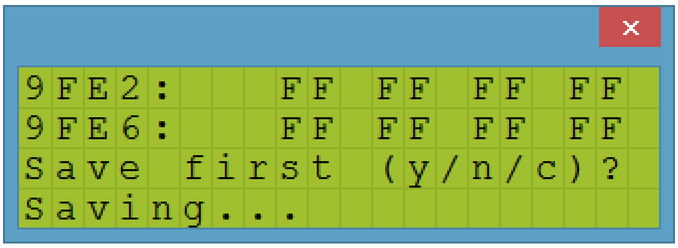
I'm presented with a hex-editor interface:



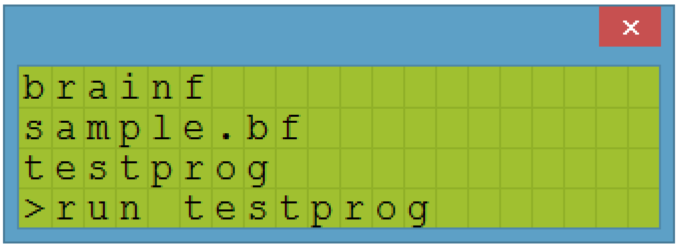
Using the arrow keys to navigate, I enter a very simple program in machine code (inverts the value on the debug input port and writes to debug input, then loops).



I press x to exit, and then y to confirm that I would like to save the file.

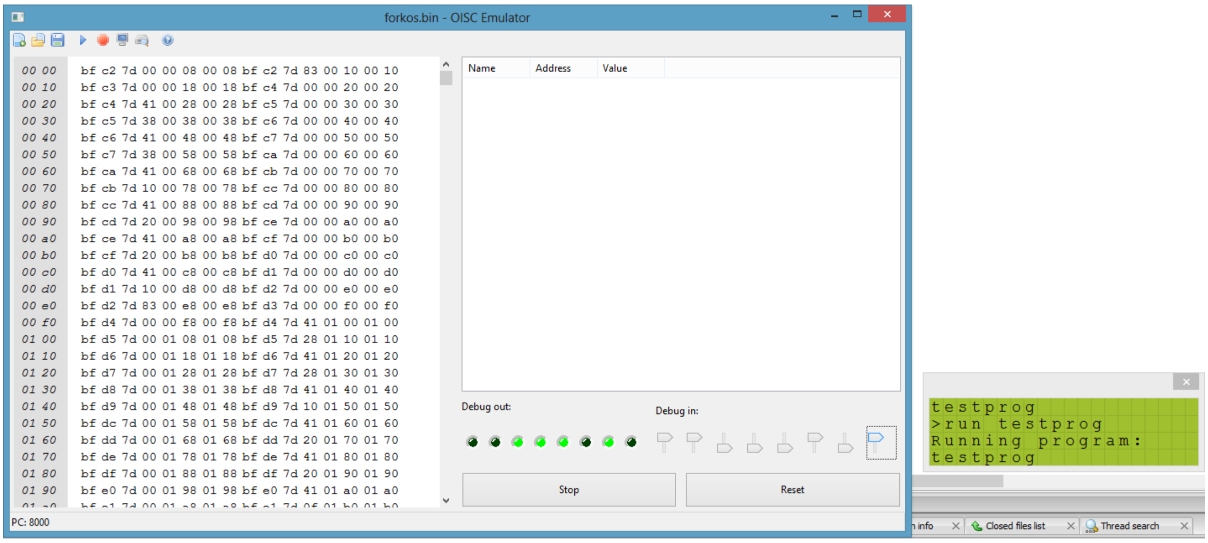


The computer restarts. I list files to make sure that the file has been created, and then enter a command to run the new file as a program.



(Yes, that's a brainf*ck interpreter! I can run brainf*ck programs other people have written provided they don't require more than 4KiB of memory.)

The program runs, and I enter the value 0xc5 on the debug input switches:



0x3a: Success!

To recap:

- Language runtime support is set up on first boot (this takes about 0.25ms) – without this, not even *pointers* work.
- In order to print the welcome message, the screen driver must be working.
- In order to load the shell program, both the flash driver and the filesystem must be working.
- A keyboard driver (such as it is) is needed for interacting with the shell, and most programs also use it.
- Most of the arithmetic functions provided by the standard libraries are also used in just the basic operating system.

All of these things compile and work.

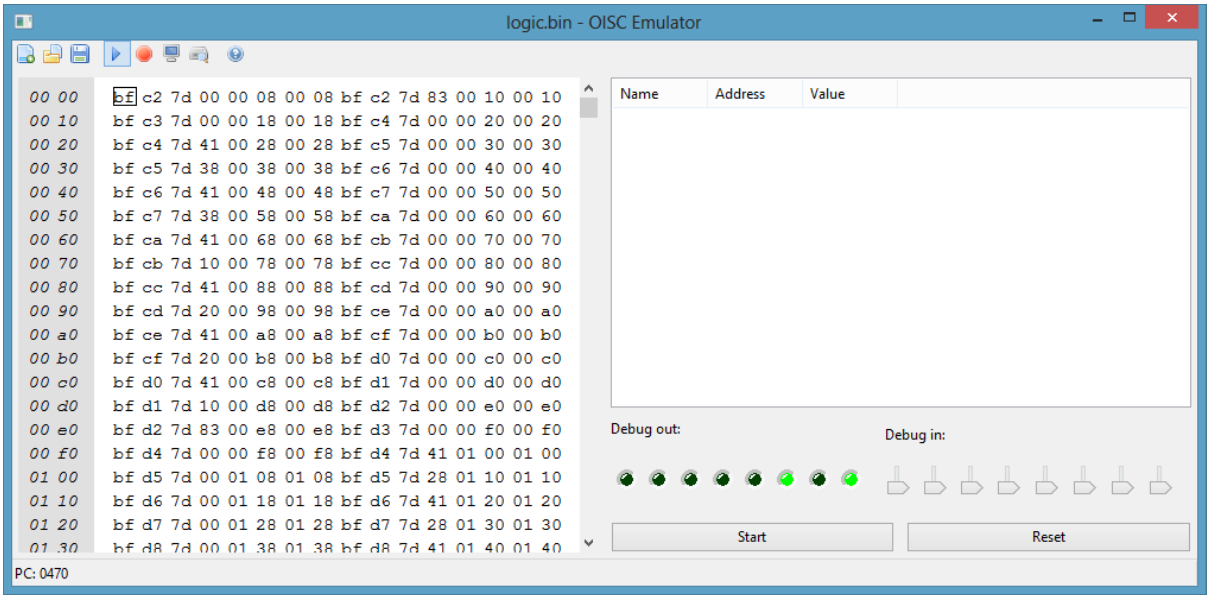
With half a megabyte of flash storage I can continue to write programs, and develop the machine into something I could plausibly do my homework on. While **not a formal test**, this leaves little room for doubt that the language is robust.

Test Programs

I also wrote a set of small test programs, and defined input and expected output for each:

Program	Input Sequence	Expected Output Sequence
Beep	Begin with all debug switches set to zero. Plug a piezo transducer into the debug output port. Power the computer on. Set first the least significant bit and then the higher-order bits of the debug output switches to 1, pausing after each switch.	A high-pitched tone should be heard upon powering on the computer. The tone should descend with each switch that is flipped to the "1" position.
Fibonacci	Power up the computer. Set the debug switches to 0, then 1, 2, 3, 4, 5, pausing in between each.	The debug output should go through the sequence 0, 1, 1, 2, 3, 5, 8 as the buttons are manipulated through the sequence. (In binary: 0, 1, 1, 10, 11, 101, 1000).
Hello World	Power up the computer	The display should show the string "Hello world!"
String comparison test	Power on the computer	If the test passes, the value 2 will be displayed on debug output. 1 indicates a failure.
Logic operator test	Load the program in the emulator and step through, one instruction at a time.	The output port should go through the sequence 1, 3, 5, 7. The lack of one of these numbers, or the appearance of an even number, indicates a test failure.

Here logicoperators.spn is shown mid-test:



The complete listing of these test programs is in Appendix 4 (page 187).

Maintenance

A compiler is a complex beast, and is often maintained by a small group of people who have intimate knowledge of its workings.

It is still possible however that a programmer who is unfamiliar with the codebase will need to make changes: whether they be corrections and bugfixes, improvements, or adapting the compiler to target a new architecture or front-end language.

Most of the technical details of the compiler's structure are described in detail in the design section (page 19), which will be referred back to. The structure of the *code*, however, is documented here, for the most part on a file-by-file basis.

Overview

The program is a cross-compiler for a high-level language ("Spoon"), targeting a custom single-instruction architecture, and intended to run on any system with a C++ compiler. The file-handling portions of the code will need to be adapted if use operating systems other than Unix-derivatives (e.g. Linux, Mac OS) and Windows is intended.

The code consists of 11 C++ source files and 12 header files. It has no external dependencies, other than STL (the standard template library). Four header files in the target language (Spoon) are also included, and are necessary for the compilation of most programs.

The compiler's interface is command-line based, and so requires some sort of shell program for operator interaction.

Summary of Header Files

Here the headers are listed in alphabetical order, along with a brief description and list of declarations:

Name	Description	Declarations
compiler.h	Defines the compiler class, and associated classes concerned with semantic analysis, such as scopes and function signatures.	Classes: `func_signature`, `symbol`, `scope` and `compiler`
error.h	Defines an error class that contains a string and is thrown by functions within the compiler (could have used `std::exception` whoops).	Classes: `error`

linker.h	Declares the `linker` class – responsible for the final code generation and resolving all the variable links generated by the compiler to addresses, as well as linking in exported functions from previously compiled files (see page 25).
	Defines constants associated with the machine's memory map, how programs are laid out in memory, and
		Classes: linker
		Constants:
		`const int ROM_SIZE`
		`const int INCREMENT_START`
		`const int DECREMENT_START`
		`const int LEFTSHIFT_START`
		`const int RIGHTSHIFT_START`
		`const int RAM_SIZE`
		`const int RAM_TOP`
		`const int HEAP_TOP`
		`const int HEAP_BOTTOM`
		`const int HEAP_SIZE`

	the layout of the language's runtims support (all the `VECTOR`s and `INSTRUCTION`s).	`const int POINTER_READ_INSTRUCTION`
		`const int POINTER_READ_PVECTOR`
		`const int JUMP_INSTRUCTION JUMP_PVECTOR`
		`const int POINTER_READ_RESULT`
		`const int POINTER_WRITE_VALUE`
		`const int POINTER_WRITE_CLEAR_INSTRUCTION`
		`const int POINTER_WRITE_COPY_INSTRUCTION`

linkval.h	Declares the `linkval` class; this is the equivalent of the assembly languages, allowing the linker to emit symbols instead of absolute numbers and then retroactively substitute. See page 27.	Classes: `linkval` Enum: `lv_type`
object.h	Declares the `object` class – this is the internal representation used to transfer data between the compiler (semantic analysis) stage and the linker. Essentially a wrapper around the AST with some extra information.	Classes: `object`
parser.h	Declares the parser functions associated with the recursive descent parsing method (see page 20 for a worked example of this program structure).	Classes: `parser`

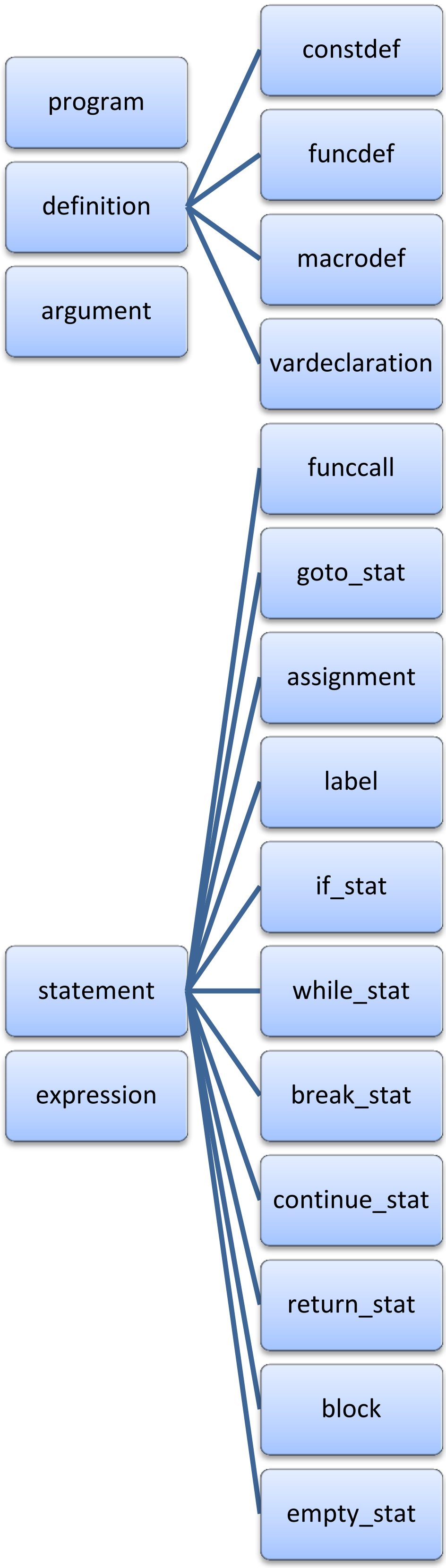
printtree.h	Declares prototypes for functions that will print out the AST in a textual form.
	The functions are (sometimes mutually) recursive, mimicking the AST's tree structure.
		Functions:
		`void printtree(program *prog, int indentation = 0);`
		`void printtree(definition *def, int indentation);`
		`void printtree(funcdef *def, int indentation);`
		`void printtree(macrodef *def, int indentation);`
		`void printtree(constdef *def, int indentation);`
		`void printtree(block *blk, int indentation);`
		`void printtree(vardeclaration *decl, int indentation);`
		`void printtree(statement *stat, int indentation);`
		`void printtree(expression *expr);`

resourcep.h	Declares (and defines inline) a very simple RAII wrapper class.	Classes: `struct resourcep <typename T>`
syntaxtree.h	Declares the classes that form the abstract syntax tree (AST).	Classes: `program`, `definition`, `constdef`, `argument`, `funcdef`, `macrodef`, `vardeclaration`, `statement`,

		funccall, goto_stat, assignment, label, if_stat, while_stat, break_stat, continue_stat, return_stat, block, empty_stat, expression Functions: *::getcopy() Enums: def_type, stat_type, exp_type
tokenizer.h	Declares prototypes for the tokenizer, which turns input text into a list of tokens (lexemes).	Classes: token Functions: tokenize Enums: token_type_enum
type.h	Declares the "type" type. (The "t" in type_t also stands for type, so when you use this type you're actually declaring a variable of type type_type!)	Classes: type_t Enums: type_enum
vardict.h	Declares the prototypes for the vardict class, which manages variables for the linker and allocates memory for them (see page 25 for details of the allocation process).	Classes: variable vardict

Class Hierarchy for syntaxtree.h

For the vast majority of the program, the class hierarchy is never more than one deep. The only exception is in the syntaxtree header, where inheritance is used to factor out repeated code from similar syntax nodes. Following is a class hierarchy for the file with the base classes on the left:



Summary of Source and Classes

The remaining classes are summarised below – each data member and member function is briefly described.

File	Class	Field/Function	Description
compiler.cpp	(file scope)	`std::string makeguid(std::string name, int ptr)`	Makes a globally unique identifier (GUID) by concatenating the symbol's name with an "@" (does not occur in symbols) and some unique information, usually the pointer to the AST object.
		`throw_type_error(std::string context, type_t expected, type_t got, int linenumber)`	Throws a preformatted type error informing the user of a type mismatch.
	`func_signature`	`type_t return_type`	The type of the variable returned by this function
		`bool is_macro`	Is this the signature of a macro?
		`bool args_must_match`	Do the types of the arguments matter or just the count? (Used by special builtins such as NFC).
		`funcdef *def`	Pointer to the actual function definition.
		`bool operator==()`	Equality operator for testing that signatures match.
		`bool operator!=()`	The inverse of ==. (Just calls that and returns the logical negation.)
		`(constructor)`	Sets up sensible defaults: `args_must_match = true; is_macro = false; def = NULL;`
	`symbol`	`std::string name`	The name of the symbol (should be globally unique).
		`type_t type`	The type of the variable to which this symbol refers.
		`bool is_constant`	Does this symbol refer to a constant?
		`int value`	If so, what is the value of that constant?
		`std::string tostring()`	Turn this type into a formatted string (used for error reporting).
	`scope`	`std::map <std::string, symbol> variables`	Contains the list of variables contained within this scope. (For a detailed description of the scope data structure, see page 22.)
		`(constructor)`	Sets the `parent` variable (default NULL if none given).
		`scope *parent`	Pointer to the next-highest scope; `NULL` if this is the global scope.
		`void insert(std::string`	Inserts a symbol directly into this scope's `std::map`.

		name, symbol var)	
		symbol& get(std::string name)	Gets an existing symbol from this scope, or the scopes above it if this scope doesn't recognise it.
		bool exists(std::string name)	Checks whether a symbol exists in this scope or a higher one.
		bool inthisscope(std::string name)	Checks whether a symbol exists in this scope specifically.
	compiler	scope *globalscope; scope *currentscope;	Pointers to the current and global scopes.
		void pushscope(); void popscope();	Push a new scope or pop the current one and delete it. These operate by manipulating the pointers declared above, and using each scope's parent pointer to form linked lists.
		std::map<std::string, symbol> globalsymboltable	Every single symbol that has been defined during the course of compilation.
		std::set<std::string> defined_funcs	The list of all function names that have been defined (as opposed to merely declared). This stage doesn't actually use this information; it's only interested in the signatures, so it passes the list to the next stage (the linker).
		std::map<std::string, func_signature> functions	The list of all functions, whether defined or merely declared.
		std::map<std::string, expression*> expression_subs;	List of expressions to be subbed in as macro arguments; when compiling the macro body, any variable names whose symbols indicate they are the macro's arguments get looked up agains this table.
		object* compile(program*)	Top-level compile function: calls lower-level functions to progress the definitions and then stuff everything into the program object.
		void compile(macrodef*); void compile(funcdef*);	Compile the arguments and the function body, and perform bookkeeping.
		void compile(statement*&, std::string exitlabel = "",	Delegates to the lower-level compilation functions based on the type enumerator.

		`std::string toplabel = "", std::string returnlabel = "")`	The optional arguments are names for the start and end of the current block and for the end of the current function, which are used by the `break`, `continue` and `return` statements.
		`void compile(block*, std::string exitlabel = "", std::string toplabel = "", std::string returnlabel = "");`	Compiles all the statements within the `block`, after compiling each `vardeclaration`. Has the same optional arguments as above.
		`void compile(vardeclaration *dec)`	Adds each declared variable to the current scope and the global symbol table. Assigns to each variable its GUID.

		`void compile(funccall*&)`
		`void compile(goto_stat*)`
		`void compile(label*)`
		`void compile(if_stat*, std::string exitlabel, std::string toplabel, std::string returnlabel)`
		`void compile(while_stat*, std::string returnlabel)`
		`void compile(assignment*)`
			Processes each of the corresponding statement types.
			The `if` statement always has the exit, top and return labels passed in, because it contains a new block.
			The `while` statement only takes the return label, because the loop defines its own top and exit.

		`void compile(expression*&)`	Takes a reference to a pointer so that it can replace the expression with something new (e.g. expression subsitutions for macro arguments). If the expression is a symbol which is a constant, the constant is substituted in. Otherwise the existence of the symbol is confirmed. If the expression is a function call, its arguments are evaluated. Calls `gettype()` to find the expression's type.
		`void gettype(expression*)`	Finds the expression's type, possibly by use of recursion.

		`bool match_types(type_t expected, type_t &received);`	Checks whether two types are compatible. If the RHS is a generic type such as `t_number`, it is updated to be more specific (so that its storage size etc are known), hence the reference argument type.
		`void addvar (std::string name, type_t type, int ptr, int linenumber, bool isConstant = false, int constvalue = 0)`	Adds a variable to the current scope and the global symbol table, and does all the appropriate bookkeeping, such as generating the variable's GUID.
linker.cpp	`linker`	`vardict vars`	The dictionary of all vars currently in use. Also allocates memory.
		`std::map<std::string, definition*> defined_funcs`	The list of all defined function names.
		`std::vector<definition*> definitions`	All the things that have been defined: at this stage, this means functions and global variables.
		`int index`	Our current location in the output buffer (not necessarily in the program's memory space!)
		`int address`	Our current location in the program's memory space (do not mix up with `index`!)
		`bool compile_to_ram`	Is this program being compiled into a RAM memory space? (changes whether runtime support prologue is added, the base of `address` etc.)
		`std::vector <linkval> buffer;`	The output buffer.
		`std::map<std::string, linkval> valtable`	Contains addresses that will later be substituted: e.g. maps a label's name to the address the label eventually ended up at.
		`std::vector<std::pair<std::string, std::string> > stringvalues;`	Contains each string literal that has been saved, as well as the name of the label it was saved at (the linker links string literals onto the end of the programs, the labels are what the string pointers get resolved to).
		`std::stringstream defstring`	The stream that the linker writes function and global variable export commands to (this is written to a file if the –e

		command is used).
	`void savelabel(std::string, linkval)`	Once a label's location is known, this helper function is used to write the relevant information (the `_HI` and `_LO` bytes of the address) into the `valtable`.
	`void write8(linkval)`	Writes an 8-bit value to the output buffer. (Note that the argument is not a `char` or `uint8_t`; we do not actually have to know what the value is at the time of writing.)
	`void write16(linkval)`	Treats the linkval as a 16-bit value – it ends up calling `write8()` twice. The utility of this function comes when writing e.g. a label: it knows to append `_HI` and `_LO` to the symbol name for the two writes, and to do the appropriate things for expressions. For literals it just bitshifts.
	`void padto8bytes()`	Inserts sufficient zeroes to align the output head to an 8-byte instruction boundary.
	`void emit_nfc2(linkval x, linkval y)`	Emits an NFC (nor and fork conditionally) instruction that NORs the bytes at x and y. The skip fields are automatically filled in with `address` + instruction size. All operations compile down to an NFC.
	`void emit_branchifzero(linkval testloc, linkval dest, bool amend_previous = false, bool invert = false);`	Emits a branch to `dest` if `testloc` is equal to zero; in doing so it also copies from `testloc` to an unused location. The `amend_previous` flag permits the function to amend the skip fields of the last written instruction, to perform the branch "for free". The `invert` flag inverts the branch (see below).
	`void emit_branchifnonzero(linkval testloc, linkval dest, bool amend_previous = false);`	A shim for `emit_branchifzero()`, calling the above function with `invert` set to true. This inverts the behaviour of the branch, making it a branch if non-zero.

	`void emit_branchalways(linkval dest, bool always_emit = false)`	Emits a nonconditional branch (a jump) to dest. If `always_emit` is false it may do this "for free" by retroactively modifying the last instruction, otherwise it will perform a NOR on an unused location (effectively a NOP). `always_emit` is used when the jump instruction may itself be the target of a jump, such as at the top of an else block.
	`void emit_copy(linkval src, linkval dest)`	Copies a byte from `source` to `dest`. If `source` == `dest` then no code is emitted (as a copy to self is destructive; the destination is always cleared before the copy).
	`void emit_copy_inverted(linkval src, linkval dest)`	Copies a byte and inverts it in transit. This takes one less instruction than a regular copy (2 vs 3). There are cases (e.g. pointer reads) where the required value is known to exist inverted at the `src`: by using this rather than inverting and then copying, the instruction count is halved.
	`void emit_writeconst(uint8_t val, linkval dest)`	Writes a constant value to a location by looking up against the constant tables at the end of ROM. This is effectively an inverted copy from the address of `~val` (2 instructions), unless `val` is 0 in which case it is a single clear instruction.
	`emit_copy_multiple(linkval src, linkval dest, int nbytes)`	Copies several bytes.
	`void emit_writeconst_multiple(int value, linkval dest, int nbytes);`	Writes a multi-byte constant in an efficient manner. See how this is used to set up the pointer runtime code.
	`void emit_writelabel(std::string label, linkval dest)`	Writes the address of a label (which may not yet be known) to dest in the same efficient manner used by `writeconst_multiple`. How does it do this? Linkval magic.
	`bool`	A predicate that does exactly

	`last_instruction_points_to_this_one();`	what it says. Used by `emit_branchifzero`'s `amend_previous` behaviour to check if the previous instruction is not already performing a branch or jump.
	`void link(funcdef*)`	Performs bookkeeping (notes down the function's start vector), links the function body into the program and exports a function declaration.
	`void exportfuncdef(funcdef*)`	Exports function information in the following order: call address, return value location, return vector, argument locations
	`exportvardeclaration(vardeclaration *vardec)`	Exports a declaration containing the variable's name, type and location.
	`void link(block*)`	Process the variable declarations (tell the vardict what they are) and link all the statements. Clean up the new variables from the vardict once the block has been linked in.
	`void link(statement*)`	Delegates to the lower-level functions based on the type enum.
	`void link(funccall*)`	Decide whether the function is a builtin or it's user defined, and invoke the appropriate routine.
	`linkval linkfunctioncall(std::vector<expression*>&, funcdef*)`	Takes a list of expressions (the arguments) and a pointer to the definition of a user-defined function. Does the following: • First, evaluate all of the arguments which are not the first argument and are calls to functions which depend on this one, and copy the results to temporary variables (to avoid collisions: these calls may ultimately call this function, and would clobber our attempts to fill in all the arguments if it was called mid-way through assembling the argument list.) • Go through the rest of the arguments. If it's a literal, emit a writeconst. If it's

		already been evaluated, copy the temp to the argument location. Otherwise, evaluate and copy. • Note down the current address, and write the value of the address after this into the function's return vector. • Jump to the function body. Note that this does not support recursion; there is no machine stack.
	linkval linkbuiltinfunction(std::vector<expression*> &args, std::string name, bool givenpreferred, linkval preferred)	Generate one of the compiler's built in functions (and(), xor(), read() etc.). If givenpreferred is true, the linker tries to make sure the result ends up in the preferred location (this makes things faster as the caller's copy then becomes a no-op). This function always returns where the result actually ends up, so the caller knows whether it needs to perform a copy.
	void link(goto_stat*)	This compiles to a jump. always_emit is false.
	void link(if_stat*)	Evaluates the test expression, adds the branch, links in the blocks, adds labels and adds in any appropriate jumps. If the test evaluates to zero, we jump past the if block: whether this goes to the else block or simple past the if depends on whether the else block is present.
	void link(while_stat*)	A branch at the top to exit when the expression evaluates to zero, and a jump at the bottom to return to the top. Does bookkeeping such as noting the positions of the break and continue labels.
	void link(assignment*)	Evaluates and copies the expression to the destination. If the expression is a constant, emit_writeconst_multiple is used.
	linkval	Evaluates an expression and

	`evaluate(expression*, bool givenpreferred = false, linkval preferred = 0)`	returns a linkval pointing to the result. `givenpreferred` has the same effect as with `linkfunctioncall()`. If the expression is a literal, an address in a constant table is returned. Names are looked up; if the name is a label, we just cast the name to a linkval (or look up the next layer of indirection in `valtable` if there is already an entry for it). If it is an array, then a temp for the pointer to the array is created and written to, or an `emit_writelabel` is issued for the preferred address if one is given (labels are used because the array may be linked onto the end of the program, the location of which is not known until code generation is complete). Function calls are handled with the appropriate function: user-defined functions are linked with `linkfunctioncall()`, and builtins are linked with `linkbuiltinfunction()`, with the `givenpreferred` information passed on to improve code generation. Strings are handled with labels, in the exact same way as arrays. The short-circuit operators (`&&`, `||`, `!`) are handled by chaining off the arguments to further calls of `evaluate()`, and inserting a branch based on the result of the first evaluation as to what is written to the return. The return location is either a temporary value or the preferred address if one is supplied.
	`uint16_t evaluate(linkval)`	Resolves a linkval down to its final integer value, after all the code generation is complete. Literals are returned, symbols are looked up, and expressions are handled appropriately based

		on the evaluation of their arguments (recursively evaluated).
	`std::vector<char> std::vector<char> assemble()`	The money shot: the generated code is passed byte-by-byte through `evaluate(linkval)` to turn it into an array of actual bytes, which the target computer can run. Also inserts the runtime-setup prologue at the start of the program, puts a halt instruction on the end and links the constant table into the last kiB after padding, giving the final 32kiB ROM image.
	`void allocatefunctionstorage()`	A bookkeeping function called early on in the linking process: for each function (apart from `main`), it allocates variables for the return value, the return vector and all of the arguments.
	`std::set<std::string> analysedependencies(std::string rootfunc);`	Implements the "mark" phase of the mark-and-sweep algorithm: starting at the root node (main), each used function is marked as being used. This allows unused functions to be removed later. It also amends each function's dependency list to include all of its dependencies, not just the immediate ones. This allows the function call code to detect circular dependencies between functions and perform some shuffling with temporary variables to avoid clashes (problems caused by a lack of stack!).
	`void removeunusedfunctions()`	The "sweep" phase of mark and sweep: deletes all unused functions from the program to reduce size if the "-s" option (strip) is given.
	`void add_object(object* obj);`	Adds all the definitions from the given object into the linker's own data structures, and performs some basic sanity checks.
	`std::vector<char> link()`	The top-level function: coordinates and calls all the

			other functions responsible for management, code generation and assembly, analysis of dependencies and sticking all of the strings onto the end.
		`std::string getdefstring()`	Returns the declarations string contained by the defstring stringstream, which contains all of the function and global variable exports.
		`void setcompiletoram(bool ram)`	Sets up flags and variables (e.g. the initial value of `address`) that depend on whether the program is to be loaded into RAM or ROM.
vardict.cpp	`vardict`		

Algorithms

A number of algorithms are used throughout the compiler. Many of them are well-known standard algorithms: these are mentioned by name only, as they should be easy to look up. The others are described at a broad level of detail, which should hopefully be sufficient to make the source code easily understandable.

Where pseudocode is available elsewhere in the document, it is cross-referenced by page number.

Operation	Algorithm
Tokenization	Finite state machine (see page 41) based on a loop and a state variable.
Parsing	Recursive descent parsing (see page 20), based on BNF on page 19.
Type Checking	Rules-based approach to finding compatible types. Post-order tree traversal for assessing validity of expressions.
Scope Resolution	A “stack of maps” data structure is used, with each scope having a pointer to its parent (enclosing scope) and a list of its variables. When looking up a variable name in context, the immediate scope is checked: if this scope does not have the variable, it “chains off” recursively to the enclosing scope. If the global scope does not have the variable, it returns false. (See page 22 for more details.)
Memory Allocation	A variant on linear search: maintain two pointers. Keep one in place and advance the second until it has covered a sufficiently large free space, at which point the first pointer is returned, as it is at the start of this space. Mark the space as occupied. If the search pointer hits an occupied space, advance it by one more space, and catch the start pointer up to it. Go back to the top of the loop. For pseudocode see page 25.
Stripping Unused Functions	Classic mark and sweep algorithm. One function starts at the main function and performs a pre-order traversal of the dependency tree built during semantic analysis (a list of dependee functions for each function; names act as pointers). A second function then “sweeps” through the list of functions and removes any that were not “marked” as used during the first phase, if the -s option is passed through the command line.

	The sweep function (`analysefunctiondependencies()`) also amends each dependency list to include all (not necessarily immediate!) dependent functions, so that the linker can recognise circular dependencies and avoid certain types of memory clashes when writing function arguments.
Assembly Expression Evaluation	Standard post-order tree traversal.

The compiler also makes use of the following STL classes (mainly ADTs), many of which have their own associated algorithms:

- std::vector
- std::map
- std::set
- std::string
- std::stringstream
- std::pair

The tree traversal operation is described here with pseudocode:

```
Struct Node:
    Pointer Data
    Pointer Left
    Pointer Right
End Struct

Sub Process(Node N)
    ` code to perform operation on node defined here
End Sub

Sub PreOrderTraverse(Node N)
    Process N
    PreorderTraverse N.Left
    PreorderTraverse N.Right
End Sub

Sub InOrderTraverse(Node N)
    InOrderTraverse N.Left
    Process N
    InOrderTraverse N.Right
End Sub

Sub PostOrderTraverse(Node N)
    PostOrderTraverse N.Left
    PostOrderTraverse N.Right
    Process N
End Sub
```

The different orders are used in different situations throughout the code: the specific order used will have been mentioned above.

Also reproduced here is the pseudocode for the memory allocation operation:

```
MemInUse[MemSize]: Boolean Array
FirstSpace = 0: Integer
Vars: Dictionary

Function FindFirstSpace
```

```
    For i From 0 To MemSize – 1
        If Not MemInUse[i] Then
            Return i
        End If
    Next i
    Error "No more free space!"
End Function

Function GetSpace(Size)
    Pos ← FirstSpace
    While Pos < MemSize
        Start ← Pos
        EnoughSpace ← True
        While Pos < Start + Size And EnoughSpace
            If MemInUse[Pos] Then EnoughSpace ← False
            Pos ← Pos + 1
        End While
        If EnoughSpace Then
            For i From Start To Start + Size
                MemInUse[i] ← True
            Next i
            If Start Is FirstSpace Then FindFirstSpace
            Return Start
        End If
    End While
    Error "No more space!"
End Function

Function AddVar(Name, Size)
    Vars.Add(Name, {Position: GetSpace(Size), Size: Size})
    Return Vars[Name].Position
End Function

Function Remove(Name)
    If Not Vars.Contains(Name) Then Error "Tried to free unknown variable"
    Var ← Vars[Name]
    For i From Var.Position To Var.Position + Var.Size – 1
        MemInUse[i] ← False
    Next i
    If Var.Position < FirstSpace Then FindFirstSpace
    Vars.Remove(Name)
End Function
```

Appraisal

Objective Assessment

Before even going to the client, we can appraise the software by evaluating it against our own requirements.

Key:

Met Requirement	Failed Optional Requirement	Failed Key Requirement

Feature	Achieved?
Must support variables: • Variables of different types must exist. • Compiler must automatically allocate memory for each variable. • Programmer must be able to assign to and from variables within the code.	Variables of type pointer and int/char can be declared. Memory is allocated automatically.
Conditional statements: • `if` and `while` as an absolute minimum. • Should function as in C.	If and while statements are available, and function with the exact same semantics as C.
Parameter passing: • Necessitates functions and function calls. • Function calling convention should be clearly documented.	Parameters are passed by value. Pointers can be passed as parameters. The calling convention is described in the technical documentation.
Arrays • Makes pointers twice as useful. Allows storing of (large amount of | indexed) data.	Arrays can be declared and used with the exact same syntax as C. Can take pointer to array and perform pointer arithmetic.
Traditional C-like syntax • Code should only differ from C syntax in the keywords and the semantics.	Code is both objectively and subjectively C-like.
Pointers • Must work: programmer can read to/write from indirectly addressed locations. • Mechanics of pointers should be clearly exposed in machine code.	Pointers can be written to/read from, and pointer arithmetic can be performed. The mechanism behind pointers is documented.
Recursion • Recursive data structures (requires use of pointers) should be implementable by programmer. • Recursive code/function calls should be usable.	Function recursion is not supported (it would be too inefficient, due to the lack of machine stack).
Machine code output should have a limited number of instructions (less than 10) and a simple (fixed) format.	Machine code has ONE instruction. Fixed format.

Syntax should be documentable (documented?) with BNF	Syntax is documented with BNF.
Have a physical computer platform (custom homemade CPU) specifically designed to run the compiler's output.	Custom CPU made with 30 CMOS chips, looks *kickin rad*.
Support for multiple platforms: Linux etc. (Essentially don't do anything to make it platform-dependent).	Compiler is platform-independent: has been compiled on my Raspberry Pi (Linux) and my phone (Android) to test this.

Feature	Achieved?
Report syntax errors to user – code must be well formed. • State the type of error (what is wrong/must be corrected) • State the line and file in which the error occurs	These reports are given as stated and described on page 29.
Report missing/undeclared variables and type mismatches. • The names of undeclared but used variable names should be reported. • Type mismatches should be stated, along with the expected type, the received type, the line number and the file in which the error occurred.	Semantic errors are also reported.
Report if a file is missing/cannot be found. • State the name of the missing file.	Missing file errors are reported.

At a glance, this looks very promising: All key requirements were met, and most of the optional requirements. The only sticking point is the lack of function recursion: this was tested, but was too inefficient (in terms of code size more than speed) due to the lack of machine stack/SP.[9]

Client Feedback

Having determined that the software met my internal requirements, I packed everything up and sent it to the client for evaluation. The King's School IT department received full program binaries for the compiler, emulator and IDE, a set of sample programs and the Spoon Programming Manual. The tutorial course from the manual was followed by a class of students over the course of a small number of lessons with, on the whole, a large degree of the success.

Mr M Greenhalgh sent me the following letter (email):

> The Spoon IDE has a simple but effective interface and all the students found it easy to enter code, edit it and save it to their home drives. Having *tabs* was great because they could have several programs open at once and compare or copy parts they needed to. The icons were fine and having a large window to edit the code on was ideal. The build options were interesting and led to a lot of discussion about code and compiler optimisation possibilities. The Log Window had plenty of useful messages about the success or failure of a build. Some students commented that their syntax (typing errors) would have been easier to find if line numbers were present for the errors. Having an option

[9] This problem was later solved by writing a bytecode stack VM in Spoon for another compiled language, which allowed much more dense and expressive code to be written.

to printout the code directly would have been helpful too. The colour coding of language statements and ability to indent meant the code was easier to read, structure and edit.

The emulator was great and whilst it took students a while to understand the parts of the screen fully, they were soon able to load and run their programs, see the various memory contents and follow what their program was doing. Having the 'LCD screen' was great and they could see the program output and check it.

From my point of view, the compiler worked well, was quick and robust – there were no errors encountered. The fact it supports a C like language, with variables, arrays, functions and the typical language constructs was ideal from the teaching viewpoint. As we teach a lot of Visual Basic, it was very useful for students to be able to write in a more C like language and appreciate the differences. It prompted much discussion on how much clearer and better structured the programs would potentially be. There was some talk about the language itself and some students felt that support for a simple For..Next type loop would be better as would support for recursion. The data types supported by the compiler seemed limited but for teaching purposes this did not matter. In the longer term, a long int, real and simple string data types would definitely extend the usefulness of the compiler to write some different types of program. The compiler's support for pointers generated a lot of interest as the concept is sometimes hard for students to grasp!
The user manual is extremely good. It is very easy to follow, informative and clear. The screenshots help a great deal to show exactly how to write a program, compile it and run it. The introduction is really helpful as it covers the basic architecture and the memory arrangement. This really makes it clear on where the compiled code is stored and what the other parts of memory are used for. The step-by-step instructions with example code is easy to follow so I was able to get students to work through the examples themselves, before trying some tasks by writing and compiling their own short programs. Inclusion of the BNF and explanations of more advanced ideas, such as variable scope, is very concise and helpful.

In essence, this has been a very successful project. The original objectives are all met as far as I can see and it will be a valuable teaching tool within the dept, and will hopefully stimulate some students to explore the related areas more – such as compilers, programming in C, the Von Neumann memory model and so on.

Our internal objectives have been met, and the client is satisfied. At this point we can consider the project complete and successful, albeit with scope for expansion in the future.

Future Improvements

In his email and in a later conversation, the client and I identified the following possible future improvements:

- A simple for…next type loop in addition to the existing while loop.
- Support for recursion.
- More data types, such as long integer, real (floating point) and string.
- Support for user-defined types, as with C's structs.
- Give line numbers for all forms of error, not just syntax errors.

For the most part, these are extensions rather than amendments.

The language extensions require both syntactic as well as semantic change. This means I (or another future programmer) would have to begin at the compiler's front end: adding the relevant token definitions to the tokenizer, and including the new syntax in the syntax tree declarations. For instance, adding C-style structs would necessitate the addition of a new `struct` keyword and the a syntax node describing the arrangement of the keyword, opening and closing braces and a list of variable declarations. This could be made up of smaller existing program nodes, e.g. the existing `block` node could be reused, and the semantic constraint added that the block may only contain variable declarations.

This would also require some back-end semantic changes. The thorough documentation would make it relatively simple for another programmer to make the relevant changes. Large parts of the code would remain completely unchanged, thanks to the modular design.

The initial objectives have been met, but with future development the program could become more flexible, powerful and feature-rich.

Appendix 1: Code Listing

tokenizer.h

```
#ifdef _WIN32
    #include <direct.h>
    #define getcwd _getcwd
    #define FILE_SEP_CHAR "\\"
#else
    #include <unistd.h>
    #define FILE_SEP_CHAR "/"
#endif
#include <fstream>
#include <iomanip>
#include <iostream>
#include "tokenizer.h"
#include "parser.h"
#include "printtree.h"
#include "compiler.h"
#include "linker.h"

#ifndef TOKENIZER_H_INCLUDED
#define TOKENIZER_H_INCLUDED

#include <iostream>
#include <iostream>
#include <vector>
#include <map>

#include "error.h"

typedef enum {
    t_eof = 0,  //end of file
    t_and,
    t_break,
    t_colon,
    t_comma,
```

```
    t_const,
    t_continue,
    t_else,
    t_export,
    t_equals,
    t_function,
    t_goto,
    t_hash,
    t_if,
    t_lbrace,
    t_lparen,
    t_lsquareb,
    t_macro,
    t_name,
    t_not,
    t_number,
    t_or,
    t_rbrace,
    t_return,
    t_rparen,
    t_rsquareb,
    t_semicolon,
    t_string,
    t_type,
    t_var,
    t_while
} token_type_enum;

class token
{
    public:
    token_type_enum type;
    std::string value;
    int linenumber;
    token();
    token(token_type_enum, std::string, int);
    token(const token&, int);   // for different line number instances of token
};

std::vector <token> tokenize(std::string);

#endif // TOKENIZER_H_INCLUDED
```

tokenizer.cpp

```
#include <iostream>
#include <map>
#include <sstream>
#include <vector>

#include "tokenizer.h"

std::string friendly_tokentype_names[] = {
    "EOF",
    "\"&&\"",
    "\"break\"",
    "colon",
    "comma",
    "\"const\"",
    "\"continue\"",
```

```
    "\"else\"",
    "\"export\"",
    "\"=\"",
    "\"function\"",
    "\"goto\"",
    "\"#\"",
    "\"if\"",
    "\"{\"",
    "\"(\"",
    "\"[\"",
    "\"macro\"",
    "name",
    "\"!\"",
    "number",
    "\"||\"",
    "\"}\"",
    "\"return\"",
    "\")\"",
    "\"]\"",
    "semicolon",
    "string",
    "type",
    "\"var\"",
    "\"while\""
};

enum state_enum
{
    s_start = 0,
    s_number,
    s_number_hex,
    s_name,
    s_string,
    s_slashaccepted,
    s_linecomment,
    s_streamcomment,
    s_staraccepted,
    s_whitespace,
    s_charliteral,
    s_expectingapostrophe,
    s_logicoperator
};



extern std::string token_type_names[];


token::token()
{
    type = t_eof;
    value = "<EOF>";
    linenumber = -1;
}

token::token(token_type_enum type_, std::string value_, int linenumber_ = -1)
{
    type = type_;
    value = value_;
    linenumber = linenumber_;
}
```

```
token::token(const token &other, int linenumber_)
{
	type = other.type;
	value = other.value;
	linenumber = linenumber_;
}


inline bool is_digit(char c)
{
	return c >= '0' && c <= '9';
}

inline bool is_hex_digit(char c)
{
	return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'f') || (c >= 'A' && c <=
'F');
}

inline bool allowed_in_name(char c)
{
	// uppercase, lowercase, digit or underscore:
	return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <=
'9') || c == '_' ||
		c == '+' || c == '-' || c == '*';
}

inline bool is_whitespace(char c)
{
	return c == ' ' || c == '\t' || c == '\n' || c == '\r';
}

std::vector <token> tokenize(std::string str)
{
	std::map<std::string, token_type_enum> keywords;
	keywords["break"] = t_break;
	keywords["char"] = t_type;
	keywords["const"] = t_const;
	keywords["continue"] = t_continue;
	keywords["else"] = t_else;
	keywords["export"] = t_export;
	keywords["goto" ] = t_goto;
	keywords["function"] = t_function;
	keywords["if" ] = t_if;
	keywords["int"] = t_type;
	keywords["int16"] = t_type;
	keywords["macro"] = t_macro;
	keywords["pointer"] = t_type;
	keywords["return"] = t_return;
	keywords["var"] = t_var;
	keywords["void"] = t_type;
	keywords["while"] = t_while;

	std::map<char, token> symbols;
	symbols[':'] = token(t_colon , ":");
	symbols[','] = token(t_comma , ",");
	symbols['='] = token(t_equals, "=");
	symbols['#'] = token(t_hash  , "#");
	symbols['!'] = token(t_not   , "!");
	symbols['('] = token(t_lparen, "(");
```

```
    symbols[')'] = token(t_rparen, ")");
    symbols['{'] = token(t_lbrace, "{");
    symbols['}'] = token(t_rbrace, "}");
    symbols['['] = token(t_lsquareb, "[");
    symbols[']'] = token(t_rsquareb, "]");
    symbols[';'] = token(t_semicolon, ";");

    std::vector <token> tokens;
    int linenumber = 1;

    int index = -1;
    const char *buffer = str.c_str();
    char c;
    int startindex = index;
    state_enum state = s_start;
    do  // while(c);  - breaks at end of loop instead of start, so we can
process ids etc. that rest up against the end of the string.
    {
        c = buffer[++index];
        if (c == '\n')
            linenumber++;
        switch(state)
        {
            case s_start:
                startindex = index;
                if (is_digit(c))
                    state = s_number;
                else if (symbols.find(c) != symbols.end())  //if we find a
matching symbol, push a matching token onto the list.
                    tokens.push_back(token(symbols.find(c)->second,
linenumber));
                else if (allowed_in_name(c))
                    state = s_name;
                else if (is_whitespace(c))   // swallow all the whitespace
                    state = s_whitespace;
                else if (c == '/')
                    state = s_slashaccepted;
                else if (c == '"')
                    state = s_string;
                else if (c == '\'')
                    state = s_charliteral;
                else if (c == '|' || c == '&')
                    state = s_logicoperator;
                else if (c)
                {
                    std::stringstream ss;
                    ss << "Error: unexpected character near \"" << c << "\", on
line " << linenumber;
                    throw(error(ss.str()));
                }

                break;
            case s_string:
                if (c == '"')
                {
                    tokens.push_back(token(t_string, str.substr(startindex + 1,
index - startindex - 1), linenumber));
                    state = s_start;
                }
                break;
            case s_slashaccepted:
```

```
                if (c == '/')
                    state = s_linecomment;
                else if (c == '*')
                    state = s_streamcomment;
                else
                    tokens.push_back(token(symbols['/'], linenumber));
                break;
            case s_linecomment:
                if (c == '\n' || c == '\r')
                    state = s_start;
                break;
            case s_streamcomment:
                if (c == '*')
                    state = s_staraccepted;
                break;
            case s_staraccepted:
                if (c == '/')
                    state = s_start;
                else
                    state = s_streamcomment;
                break;
            case s_number:
                if (!is_digit(c))
                {
                    std::string val = str.substr(startindex, index -
startindex);
                    if (c == 'x' && val == "0")
                    {
                        state = s_number_hex;
                        startindex += 2;
                    }
                    else
                    {
                        tokens.push_back(token(t_number, val, linenumber));
                        state = s_start;
                        index--;
                    }
                }
                break;
            case s_number_hex:
                if (!is_hex_digit(c))
                {
                    int value;
                    std::stringstream ss;
                    ss << std::hex << str.substr(startindex, index -
startindex);
                    ss >> value;
                    std::stringstream ss2;
                    ss2 << value;
                    tokens.push_back(token(t_number, ss2.str(), linenumber));
                    state = s_start;
                    index--;
                }
                break;
            case s_name:
                if (!allowed_in_name(c))
                {
                    token t(t_name, str.substr(startindex, index - startindex),
linenumber);
                    if (keywords.find(t.value) != keywords.end())
                        t.type = keywords.find(t.value)->second;
```

```
                    tokens.push_back(t);
                    state = s_start;
                    index--;
                }
                break;
            case s_whitespace:
                if (!is_whitespace(c))
                {
                    index--;    // we've now encountered a character that isn't
whitespace; unget it so the next loop can pick it up.
                    state = s_start;
                }
                break;
            case s_charliteral:
                state = s_expectingapostrophe;
                break;
            case s_expectingapostrophe:
            {
                if (c != '\'')
                {
                    std::stringstream ss;
                    std::cout << "Error: expected ' to close char literal near
\"" << c << "\" on line " << linenumber;
                    throw(error(ss.str()));
                }
                std::stringstream ss;
                ss << (int)(str[index - 1]);
                tokens.push_back(token(t_number, ss.str(), linenumber));
                state = s_start;
                break;
            }
            case s_logicoperator:
                if (c == '&' && buffer[index - 1] == '&')
                    tokens.push_back(token(t_and, "&&", linenumber));
                else if (c == '|' && buffer[index - 1] == '|')
                    tokens.push_back(token(t_or, "||", linenumber));
                else
                {
                    std::stringstream ss;
                    ss << "Error: malformed logic operator near \"" <<
str.substr(startindex, 2) << "\" on line " << linenumber;
                    throw(error(ss.str()));
                }
                state = s_start;
                break;
        }
    } while (c);
    if (state == s_string)
        throw(error("Error: expected \" to close string near EOF"));
    else if (state == s_charliteral || state == s_expectingapostrophe)
        throw(error("Error: expected ' to close char literal near EOF"));
    return tokens;
}
```

syntaxtree.h

```
#ifndef SYNTAXTREE_H_INCLUDED
#define SYNTAXTREE_H_INCLUDED

#include <set>
```

```
#include <string>
#include <vector>

#include "type.h"

typedef enum {
    dt_constdef = 0,
    dt_funcdef,
    dt_macrodef,
    dt_vardec       // dec vs. def I know...
} def_type;

typedef enum {
    stat_block,
    stat_call,
    stat_empty,
    stat_goto,
    stat_assignment,
    stat_label,
    stat_if,
    stat_while,
    stat_break,
    stat_continue,
    stat_return
} stat_type;

typedef enum {
    exp_name,
    exp_number,
    exp_funccall,
    exp_string,
    exp_and,
    exp_not,
    exp_or
} exp_type;


struct definition;
struct block;
struct expression;


struct program
{
    std::vector <definition*> defs;
};

struct definition
{
    def_type type;
    int linenumber;
};

struct constdef: public definition
{
    type_t valtype;
    std::string name;
    int value;                    // ought really to be an expression
    constdef() {type = dt_constdef;}
};
```

```
struct argument
{
    type_t type;
    std::string name;
    argument(type_t type_ = type_none, std::string name_ = "")
    {
        type = type_;
        name = name_;
    }
};

struct funcdef: public definition
{
    type_t return_type;
    std::string name;
    std::vector <argument> args;
    block *body;
    bool defined;        // vs. merely declared
    bool exported;
    std::vector <int> exportvectors;
    // Only used by compiler and linker:
    std::set<std::string> dependson;
    bool is_used;    // For mark and sweep of functions by linker
    funcdef() {type = dt_funcdef; defined = false; is_used = false; exported =
false;}
};

struct macrodef: public definition
{
    std::string name;
    std::vector<std::string> args;
    block *body;
    macrodef() {type = dt_macrodef;}
};

struct vardeclaration: public definition
{
    struct varpair
    {
        std::string name;
        type_t type;
        bool exported;
        int exportvector;
        varpair() {exported = false;}
    };
    std::vector <varpair> vars;
    vardeclaration *getcopy()
    {
        vardeclaration *vardec = new vardeclaration;
        for (unsigned int i = 0; i < vars.size(); i++)
            vardec->vars.push_back(vars[i]);
        return vardec;
    }
    vardeclaration() {type = dt_vardec;}
};

struct statement
{
    stat_type type;
    int linenumber;
    virtual statement* getcopy() = 0;
```

```
    virtual ~statement() {};
};

struct funccall: public statement
{
    std::string name;
    std::vector <expression*> args;
    virtual statement* getcopy();
    funccall() {type = stat_call;}
    virtual ~funccall() {}
};

struct goto_stat: public statement
{
    expression *target;
    virtual statement* getcopy();
    goto_stat() {type = stat_goto;}
    virtual ~goto_stat() {}
};

struct assignment: public statement
{
    std::string name;
    bool indexed;
    int index;
    expression *expr;
    virtual statement* getcopy();
    assignment() {type = stat_assignment; indexed = false;}
    virtual ~assignment() {}
};

struct label: public statement
{
    std::string name;
    virtual statement* getcopy();
    label() {type = stat_label;}
    virtual ~label() {}
};

struct if_stat: public statement
{
    expression *expr;
    statement *ifblock;
    statement *elseblock;
    virtual statement* getcopy();
    if_stat() {type = stat_if; elseblock = 0;}
    virtual ~if_stat() {}
};

struct while_stat: public statement
{
    expression *expr;
    statement *blk;
    virtual statement* getcopy();
    while_stat() {type = stat_while;}
    virtual ~while_stat() {}
};

struct break_stat: public statement
{
    break_stat() {type = stat_break;}
```

```
    virtual statement* getcopy() { return new break_stat; }
    virtual ~break_stat() {}
};

struct continue_stat: public statement
{
    continue_stat() {type = stat_continue;}
    virtual statement* getcopy() { return new continue_stat; }
    virtual ~continue_stat() {}
};

struct return_stat: public statement
{
    return_stat() {type = stat_return;}
    virtual statement* getcopy() { return new return_stat; }
    virtual ~return_stat() {}
};

struct block: public statement
{
    std::vector <vardeclaration*> declarations;
    std::vector <statement*> statements;
    block() {type = stat_block;}
    virtual block* getcopy();
    virtual ~block() {}
};

struct empty_stat: public statement
{
    empty_stat() {type = stat_empty;}
    virtual statement* getcopy() {return new empty_stat;}
    virtual ~empty_stat() {}
};

struct expression
{
    exp_type type;
    std::string name;
    int number;
    int linenumber;
    bool indexed;
    type_t val_type;          // <- Not touched by the parser: the compiler sets
it when it reads types, and the linker reads it later.
    std::vector<expression*> args;
    expression* getcopy();
    expression() {indexed = false;}
    expression(std::string _name) {type = exp_name; name = _name; indexed =
false;}
};

inline statement* funccall::getcopy()
{
    funccall *f = new funccall;
    f->name = name;
    f->linenumber = linenumber;
    for (unsigned int i = 0; i < args.size(); ++i)
        f->args.push_back(args[i]->getcopy());
    return f;
}

inline statement* goto_stat::getcopy()
```

```
{
    goto_stat *g = new goto_stat;
    g->linenumber = linenumber;
    g->target = target->getcopy();
    return g;
}

inline statement* assignment::getcopy()
{
    assignment *assg = new assignment;
    assg->name = name;
    assg->linenumber = linenumber;
    assg->indexed = indexed;
    assg->index = index;
    assg->expr = expr->getcopy();
    return assg;
}

inline statement* label::getcopy()
{
    label *lbl = new label;
    lbl->name = name;
    lbl->linenumber = linenumber;
    return lbl;
}

inline statement* if_stat::getcopy()
{
    if_stat *ifs = new if_stat;
    ifs->linenumber = linenumber;
    ifs->expr = expr->getcopy();
    ifs->ifblock = ifblock->getcopy();
    ifs->elseblock = elseblock->getcopy();
    return ifs;
}

inline statement* while_stat::getcopy()
{
    while_stat *whiles = new while_stat;
    whiles->linenumber = linenumber;
    whiles->expr = expr->getcopy();
    whiles->blk = blk->getcopy();
    return whiles;
}

inline block* block::getcopy()
{
    block *blk = new block;
    blk->linenumber = linenumber;
    for (unsigned int i = 0; i < declarations.size(); ++i)
        blk->declarations.push_back(declarations[i]->getcopy());
    for (unsigned int i = 0; i < statements.size(); ++i)
        blk->statements.push_back(statements[i]->getcopy());
    return blk;
}

inline expression* expression::getcopy()
{
    expression *expr = new expression;
    expr->type = type;
    expr->name = name;
```

```
    expr->linenumber = linenumber;
    expr->number = number;
    expr->val_type = val_type;
    for (unsigned int i = 0; i < args.size(); ++i)
        expr->args.push_back(args[i]->getcopy());
    return expr;
}


#endif // SYNTAXTREE_H_INCLUDED
```

resourcep.h

```
#ifndef RESOURCEP_H_INCLUDED
#define RESOURCEP_H_INCLUDED

// tiny RAII wrapper class.
// Instantiates a new T when it's instantiated.
// If it gets popped off the stack, the T gets destroyed.
// Call release() when it's safe to take control of the pointer yourself.

template <typename T>
struct resourcep
{
    T *obj;

    resourcep(T *_obj = 0)
    {
        if (_obj)
            obj = _obj;
        else
            obj = new T();
    }
    ~resourcep()
    {
        if (obj)
            delete obj;
    }
    T* release()
    {
        T *handle = obj;
        obj = 0;
        return handle;
    }
};


#endif // RESOURCEP_H_INCLUDED
```

parser.h

```
#ifndef _PARSER_H_INCLUDED_
#define _PARSER_H_INCLUDED_

#include "syntaxtree.h"
#include "tokenizer.h"

#include <map>
#include <set>
```

```
#include <vector>

class parser
{
    std::string filename;
    std::string filedirectory;
    std::vector<token> tokens;
    std::map <std::string, type_enum> typestrings;
    std::set <std::string> includedfiles;
    int index;
    token t;
    token lastt;
    void gettoken();
    bool accept(token_type_enum type);
    void expect(token_type_enum type);
    public:
    parser(std::vector<token> tokens_, std::string _filename = "file",
std::string _filedirectory = "");

    program* getprogram();
    void do_preprocessor(program *prog);
    void throw_unexpected(std::string value, int linenumber = 0, token_type_enum
expected = t_eof, token_type_enum got = t_eof);
    definition *getdefinition();
        constdef* getconstdef();
        funcdef* getfuncdef(bool exported = false);
        macrodef* getmacrodef();
    block* getblock();
    vardeclaration* getvardeclaration(std::vector<statement*> *statlist = 0,
bool exported = false);    // we tell the function where the statements are, so
it's allowed to push assignments. (See implementation for more notes)
    vardeclaration::varpair getvarname_and_type(type_enum basetype,
std::vector<statement*> *statlist);
    statement* getstatement();
        assignment* getassignment();
        goto_stat* getgoto();
        funccall* getfunccall();
        label* getlabel();
        if_stat* getif();
        while_stat* getwhile();
        break_stat* getbreak();
        continue_stat* getcontinue();
        return_stat* getreturn();
    expression* getexpression();
    expression* getsinglevalue();
};

#endif // _PARSER_H_INCLUDED_
```

parser.cpp

```
#include "parser.h"
#include "resourcep.h"

#include <fstream>
#include <sstream>

extern std::string friendly_tokentype_names[];

// Throw a preformatted error that we've received an unexpected token.
void parser::throw_unexpected(std::string value, int linenumber, token_type_enum
expected, token_type_enum got)
```

```
{
	std::stringstream ss;
	ss << "Error in " << filename << ": unexpected token near \"" << value << 
"\"";
	if (linenumber)
		ss << " on line " << linenumber;
	if (expected)
		ss << ": expected " << friendly_tokentype_names[expected] << ", got " << 
friendly_tokentype_names[got];
	throw(error(ss.str()));
}

int intfromstring(std::string str)
{
	std::stringstream ss;
	int result;
	ss << str;
	ss >> result;
	return result;
}


// Set up the type dicts and the state vars
parser::parser(std::vector<token> tokens_, std::string _filename, std::string 
_filedirectory)
{
	filename = _filename;
	filedirectory = _filedirectory;
	typestrings["char"] = type_int;
	typestrings["int"] = type_int;
	typestrings["int16"] = type_pointer;
	typestrings["pointer"] = type_pointer;
	typestrings["void"] = type_none;
	tokens = tokens_;
	index = 0;
	if (tokens.size() > 0)
		t = tokens[0];
}

// read the next token, put it in t; make t a blank token if no more tokens.
void parser::gettoken()
{
	lastt = t;
	index++;
	if ((unsigned)index < tokens.size())
	{
		t = tokens[index];
	}
	else
	{
		t = token();
	}
}

// optionally gobble up a token, and return whether or not we have gobbled.
bool parser::accept(token_type_enum type)
{
	if (t.type == type)
	{
		gettoken();
		return true;
```

```
    }
    else
    {
        return false;
    }
}

// gobble up a token, or raise an error if it doesn't match
void parser::expect(token_type_enum type)
{
    if (!accept(type))
    {
        throw_unexpected(t.value, t.linenumber, type, t.type);
    }
}

// top level function: delegates to the function, macro and const
// definition functions.
program* parser::getprogram()
{
    resourcep <program> prog;
    while (t.type != t_eof)
    {
        if (accept(t_function))
        {
            prog.obj->defs.push_back(getfuncdef());
        }
        else if (accept(t_macro))
        {
            prog.obj->defs.push_back(getmacrodef());
        }
        else if (accept(t_const))
        {
            prog.obj->defs.push_back(getconstdef());
        }
        else if (accept(t_var))
        {
            prog.obj->defs.push_back(getvardeclaration());
        }
        else if (accept(t_hash))
        {
            expect(t_name);
            do_preprocessor(prog.obj);
        }
        else if (accept(t_export))
        {
            if (accept(t_var))
                prog.obj->defs.push_back(getvardeclaration(NULL, true));
            else if (accept(t_function))
                prog.obj->defs.push_back(getfuncdef(true));
            else
                expect(t_var);
        }
        else
        {
            throw_unexpected(t.value, t.linenumber, t_function, t.type);
        }
    }
    return prog.release();
}
```

```
void parser::do_preprocessor(program *prog)
{
    if (lastt.value == "include")
    {
        expect(t_string);
        // Files do not get included twice:
        if (includedfiles.find(lastt.value) != includedfiles.end())
            return;
        includedfiles.insert(lastt.value);
        std::string includefiledirectory = filedirectory;
        std::fstream *includefile = new std::fstream(filedirectory +
lastt.value, std::ios::in | std::ios::binary);
        if (!includefile->is_open())
        {
            delete includefile;
            includefile = new std::fstream(lastt.value, std::ios::in |
std::ios::binary);
            includefiledirectory = "";
            if (!includefile->is_open())
                throw(error(std::string("Error: could not open include file ") +
lastt.value));
        }
        includefile->seekg(0, std::ios::end);
        int sourcelength = includefile->tellg();
        includefile->seekg(0, std::ios::beg);
        std::vector<char> source(sourcelength);
        includefile->read(&source[0], sourcelength);
        source.push_back(0);
        includefile->close();
        delete includefile;
        std::vector<token> includetokens = tokenize(&source[0]);
        parser p(includetokens, lastt.value, includefiledirectory);
        p.includedfiles = includedfiles;
        program *includedefs = p.getprogram();
        includedfiles = p.includedfiles;
        for (unsigned int i = 0; i < includedefs->defs.size(); i++)
            prog->defs.push_back(includedefs->defs[i]);
    }
    else
    {
        throw(error("Error: unrecognised preprocessor directive: \"" +
lastt.value + "\""));
    }
}

// sort out all the syntax stuff, return the type, name and
// value of the constant.
constdef* parser::getconstdef()
{
    resourcep <constdef> def;
    def.obj->linenumber = lastt.linenumber;
    expect(t_type);
    def.obj->valtype = typestrings[lastt.value];
    expect(t_name);
    def.obj->name = lastt.value;
    expect(t_equals);
    expect(t_number);
    std::stringstream ss;
    ss << lastt.value;
    ss >> def.obj->value;  // convert string to int
    expect(t_semicolon);
```

```
    return def.release();
}

// read in the list of args and parse the macro body
macrodef* parser::getmacrodef()
{
    resourcep <macrodef> def;
    def.obj->linenumber = lastt.linenumber;
    expect(t_name);
    def.obj->name = lastt.value;
    expect(t_lparen);
    if (accept(t_name))
    {
        def.obj->args.push_back(lastt.value);
        while (accept(t_comma))
        {
            expect(t_name);
            def.obj->args.push_back(lastt.value);
        }
    }
    expect(t_rparen);
    def.obj->body = getblock();
    return def.release();
}


funcdef* parser::getfuncdef(bool exported)
{
    resourcep <funcdef> def;
    def.obj->linenumber = lastt.linenumber;
    if (accept(t_type))
    {
        def.obj->return_type = typestrings[lastt.value];
    }
    else
    {
        def.obj->return_type = type_none;
    }
    expect(t_name);
    def.obj->name = lastt.value;
    expect(t_lparen);
    if (accept(t_type))
    {
        type_t type = typestrings[lastt.value];
        expect(t_name);
        def.obj->args.push_back(argument(type, lastt.value));
        while (accept(t_comma))
        {
            expect(t_type);
            type_t type = typestrings[lastt.value];
            expect(t_name);
            def.obj->args.push_back(argument(type, lastt.value));
        }
    }
    expect(t_rparen);
    if (exported)
    {
        expect(t_colon);
        def.obj->exported = true;
        def.obj->defined = true;
        for (unsigned int i = 0; i < def.obj->args.size() + 3; i++)
```

```
        {
            // One for location, one for return value, one for return vector,
one for each argument:
            expect(t_number);
            def.obj->exportvectors.push_back(intfromstring(lastt.value));
            if (!accept(t_comma))
            {
                expect(t_semicolon);
                break;
            }
        }
        if (lastt.type != t_semicolon)
            throw(error("Error: too many export vectors for function " +
def.obj->name));
        else if (def.obj->exportvectors.size() != def.obj->args.size() + 3)
            throw(error("Error: not enough export vectors for function " +
def.obj->name));
    }
    else if (accept(t_semicolon))
    {
        def.obj->defined = false;
    }
    else
    {
        def.obj->defined = true;        // there is a function definition body,
so it's not just a declaration.
        def.obj->body = getblock();
    }
    return def.release();
}

// block is either a single statement, or a bunch of statements
// and definitions between two braces.
block* parser::getblock()
{
    resourcep <block> blk;
    expect(t_lbrace);
    blk.obj->linenumber = lastt.linenumber;
    while(t.type != t_rbrace)
    {
        if (accept(t_var))
        {
            blk.obj->declarations.push_back(getvardeclaration(&blk.obj-
>statements));
        }
        else
        {
            blk.obj->statements.push_back(getstatement());
        }
    }
    expect(t_rbrace);
    return blk.release();
}

// Returns the type and a list of names
// We tell the function where the statements are, so it's allowed to push
assignments.
// (This avoids sequencing problems with vardecs compiled at start of block,
otherwise assignments
// would all take place at start of block when vardecs are processed.)
vardeclaration* parser::getvardeclaration(std::vector<statement*> *statlist,
```

```
bool exported)
{
    resourcep <vardeclaration> vardec;
    vardec.obj->linenumber = lastt.linenumber;
    expect(t_type);
    type_enum basetype = typestrings[lastt.value];
    expect(t_name);
    vardec.obj->vars.push_back(getvarname_and_type(basetype, statlist));  //
handles all of the array length arguments and stuff for us too :)
    while (accept(t_comma))
    {
        expect(t_name);
        vardec.obj->vars.push_back(getvarname_and_type(basetype, statlist));
    }
    if (exported)
    {
        expect(t_colon);
        expect(t_number);
        vardec.obj->vars[0].exported = true;
        vardec.obj->vars[0].exportvector = intfromstring(lastt.value);
    }
    expect(t_semicolon);
    return vardec.release();
}

vardeclaration::varpair parser::getvarname_and_type(type_enum basetype,
std::vector<statement*> *statlist)
{
    vardeclaration::varpair var;
    var.name = lastt.value;
    if (accept(t_lsquareb))
    {
        var.type.type = type_array;
        var.type.second = basetype;
        expect(t_number);
        std::stringstream ss;
        ss << lastt.value;
        ss >> var.type.count;
        expect(t_rsquareb);
    }
    else
    {
        var.type.type = basetype;
    }
    if (accept(t_equals))
    {
        if (statlist)
        {
            assignment *assg = new assignment;
            assg->name = var.name;
            assg->expr = getexpression();
            statlist->push_back(assg);
        }
        else
        {
            throw(error("Error: initialization of globals not supported. (do it
in main())"));
        }
    }
    return var;
}
```

```
statement* parser::getstatement()
{
    int linenumber = t.linenumber;
    statement *stat;
    if (t.type == t_lbrace)
    {
        stat = getblock();
    }
    else if (accept(t_goto))
    {
        stat = getgoto();
    }
    else if (accept(t_if))
    {
        stat = getif();
    }
    else if (accept(t_while))
    {
        stat = getwhile();
    }
    else if (accept(t_break))
    {
        resourcep <break_stat> breaks;
        expect(t_semicolon);
        stat = breaks.release();
    }
    else if (accept(t_continue))
    {
        resourcep <continue_stat> continues;
        expect(t_semicolon);
        stat = continues.release();
    }
    else if (accept(t_return))
    {
        resourcep <return_stat> returns;
        expect(t_semicolon);
        stat = returns.release();
    }
    else if (accept(t_semicolon))
    {
        stat = new empty_stat;
    }
    else
    {
        expect(t_name);
        if (t.type == t_lparen)
        {
            stat = getfunccall();
        }
        else if (t.type == t_colon)
        {
            stat = getlabel();
        }
        else
        {
            stat = getassignment();
        }
    }
    stat->linenumber = linenumber;
```

```
    return stat;
}

assignment* parser::getassignment()
{
    resourcep <assignment> assg;
    assg.obj->name = lastt.value;
    if (accept(t_lsquareb))
    {
        expect(t_number);
        assg.obj->indexed = true;
        assg.obj->index = intfromstring(lastt.value);
        expect(t_rsquareb);
    }
    expect(t_equals);
    assg.obj->expr = getexpression();
    expect(t_semicolon);
    return assg.release();
}

goto_stat* parser::getgoto()
{
    resourcep <goto_stat> sgoto;
    sgoto.obj->target = getexpression();
    expect(t_semicolon);
    return sgoto.release();
}

funccall* parser::getfunccall()
{
    resourcep <funccall> fcall;
    fcall.obj->name = lastt.value;
    expect(t_lparen);
    if (t.type != t_rparen)
    {
        fcall.obj->args.push_back(getexpression());
        while (accept(t_comma))
        {
            fcall.obj->args.push_back(getexpression());
        }
    }
    expect(t_rparen);
    expect(t_semicolon);
    return fcall.release();
}

label* parser::getlabel()
{
    resourcep <label> lbl;
    lbl.obj->name = lastt.value;
    expect(t_colon);
    return lbl.release();
}

if_stat* parser::getif()
{
    resourcep <if_stat> ifs;
    expect(t_lparen);
    ifs.obj->expr = getexpression();
    expect(t_rparen);
    ifs.obj->ifblock = getstatement();
```

```
    if (accept(t_else))
        ifs.obj->elseblock = getstatement();
    return ifs.release();
}

while_stat* parser::getwhile()
{
    resourcep <while_stat> whiles;
    expect(t_lparen);
    whiles.obj->expr = getexpression();
    expect(t_rparen);
    whiles.obj->blk = getstatement();
    return whiles.release();
}

expression* parser::getexpression()
{
    resourcep <expression> expr = getsinglevalue();

    while (accept(t_and) || accept(t_or))
    {
        expression *temp = expr.obj;
        expr.obj = new expression;
        expr.obj->type = (lastt.type == t_and) ? exp_and : exp_or;
        expr.obj->args.push_back(temp);
        expr.obj->args.push_back(getsinglevalue());
        expr.obj->linenumber = temp->linenumber;
    }
    return expr.release();
}

expression* parser::getsinglevalue()
{
   resourcep <expression> expr;
   expr.obj->linenumber = lastt.linenumber;

    if (accept(t_not))
    {
        expr.obj->type = exp_not;
        expr.obj->args.push_back(getsinglevalue());
    }
    else if (accept(t_lparen))
    {
        resourcep<expression> innerexpr(getexpression());
        expect(t_rparen);
        return innerexpr.release();
    }
    else if (accept(t_name))
    {
        expr.obj->name = lastt.value;
        if (accept(t_lparen))
        {
            expr.obj->type = exp_funccall;
            if (t.type != t_rparen)
                expr.obj->args.push_back(getexpression());
            while (accept(t_comma))
                expr.obj->args.push_back(getexpression());
            expect(t_rparen);
        }
        else
        {
```

```
            expr.obj->type = exp_name;
            if (accept(t_lsquareb))
            {
                expect(t_number);
                expr.obj->indexed = true;
                expr.obj->number = intfromstring(lastt.value);
                expect(t_rsquareb);
            }
        }
    }
    else if (accept(t_string))
    {
        expr.obj->type = exp_string;
        expr.obj->name = lastt.value;
    }
    else if (accept(t_lbrace))
    {
        expr.obj->type = exp_string;
        std::vector<char> values;
        do
        {
            expect(t_number);
            values.push_back(intfromstring(lastt.value));
        }
        while (accept(t_comma));
        expect(t_rbrace);
        expr.obj->name = std::string(&(values[0]), values.size());
//std::string is not actually null-terminated, so the array can contain zeroes.
    }
    else
    {
        expr.obj->type = exp_number;
        expect(t_number);
        expr.obj->number = intfromstring(lastt.value);
    }
    return expr.release();
}
```

type.h

```
#ifndef TYPE_H_INCLUDED
#define TYPE_H_INCLUDED

#include <string>

typedef enum {
    type_none = 0,
    type_int,
    type_pointer,
    type_label,
    type_array,
    type_number,        // generic number type, for compiler use only.
    type_expression,
    n_types
} type_enum;

struct type_t
{
    type_enum type;
```

```
    type_enum second;       // secondary type: e.g. array of *int*.
    int count;
    type_t(type_enum _type = type_none, type_enum _second = type_none, int
_count = 0) {type = _type; second = _second; count = _count;}
    bool operator== (const type_t &rhs) const;
    int getsize() const;
    int getstoragesize() const; // NB the difference between these two! for an
array, getsize() is the pointer size, and storagesize() is the underlying array
size.
    std::string getname() const;
};

#endif // TYPE_H_INCLUDED
```

type.cpp

```
#include "type.h"

#include <sstream>

const std::string friendly_type_names[] = {
    "void",
    "int",
    "pointer",
    "label",
    "array",
    "number",
    "expression"
};

const int type_sizes[] = {
    0,
    1,
    2,
    0,
    2,
    1,
    0
};

bool type_t::operator==(const type_t &rhs) const
{
    if (type == rhs.type)
        return true;
    else if (type == type_array && rhs.type == type_array)
        return second == rhs.second && (count == 0 || count == rhs.count);
    else
        return false;
}

// tells us how much space the thing takes up in memory - used primarily by
vardict when allocating memory.
int type_t::getstoragesize() const
{
    if (type == type_array)
        return count * type_sizes[second];
    else
        return type_sizes[type];
}
```

```
// tells us the size of the type, as interacted with by the programmer: e.g.
arrays have size of two, as the programmer uses them as const pointers.
int type_t::getsize() const
{
	return type_sizes[type];
}

std::string type_t::getname() const
{
	if (type == type_array)
	{
		std::stringstream ss;
		ss << ((second >= 0 && second < n_types) ? friendly_type_names[second] :
std::string("UNDEFINEDTYPE")) << "[";
		if (count)
			ss << count;
		ss << "]";
		return ss.str();
	}
	else
		return (type >= 0 && type < n_types) ? friendly_type_names[type] :
std::string("UNDEFINEDTYPE");
}
```

compiler.h

```
#ifndef COMPILER_H_INCLUDED
#define COMPILER_H_INCLUDED

#include "syntaxtree.h"
#include "object.h"

#include <map>
#include <vector>
#include <set>

struct func_signature
{
	type_t return_type;
	std::vector<type_t> arg_types;
	bool is_macro;
	bool args_must_match;		// to be used with macros, and special builtins
like NFC.
	funcdef *def;
	bool operator==(func_signature &rhs) const {return rhs.return_type ==
return_type && rhs.arg_types == arg_types;}
	bool operator!=(func_signature &rhs) const {return !operator==(rhs);}
	func_signature() {args_must_match = true; is_macro = false; def = 0;}
};

struct symbol
{
	std::string name;
	type_t type;
	bool is_constant;
	int value;
	symbol(){is_constant = false;}
	std::string tostring();
};
```

```
class scope
{
  private:
    std::map <std::string, symbol> variables;
  public:
    scope(scope *_parent = 0);
    scope *parent;
    void insert(std::string name, symbol var);
    symbol& get(std::string name);
    bool exists(std::string name);
    bool inthisscope(std::string name);
};


// The scopes map a local name to a global symbol (name@ptr)
// The global symbol table gets exported as part of the object, also makes it
easy to look up types in type checking.
class compiler
{
    scope *globalscope;
    scope *currentscope;
    std::map<std::string, symbol> globalsymboltable;
    std::set<std::string> defined_funcs;
    std::map<std::string, func_signature> functions;
    std::map<std::string, expression*> expression_subs;
    funcdef *currentfuncdef;
    void pushscope();
    void popscope();
    public:
    compiler();
    object* compile(program*);
    void compile(macrodef*);
    void compile(funcdef*);
    void compile(statement*&, std::string exitlabel = "", std::string toplabel =
"", std::string returnlabel = ""); // optionally supply labels for
break/continue
    void compile(block*, std::string exitlabel = "", std::string toplabel = "",
std::string returnlabel = "");      // (block is a subclass of statement...)
    void compile(vardeclaration *dec);      // returns list of assignments
specified by variable initializers.
    void compile(funccall*&);
    void compile(goto_stat*);
    void compile(label*);
    void compile(if_stat*, std::string exitlabel, std::string toplabel,
std::string returnlabel);
    void compile(while_stat*, std::string returnlabel);
    void compile(assignment*);
    void compile(expression*&);
    void gettype(expression*);
    bool match_types(type_t expected, type_t &received);
    void addvar(std::string name, type_t type, int ptr, int linenumber, bool
isConstant = false, int constvalue = 0);
};


#endif //COMPILER_H_INCLUDED
```

compiler.cpp

```
#include "compiler.h"
#include "error.h"

#include <sstream>
#include <iomanip>

#include <iostream>


#include "printtree.h"

// take a local symbol and a globally unique pointer value,
// concatenate into a guid string
std::string makeguid(std::string name, int ptr)
{
    std::stringstream ss;
    ss << name << "@" << std::hex << ptr;
    return ss.str();
}

void throw_type_error(std::string context, type_t expected, type_t got, int
linenumber)
{
    throw_error("Error: Type mismatch in " << context << ": was expecting " <<
expected.getname() <<
                 " but got " << got.getname() << "(line " << linenumber << ")");
}

// Scope definitions: //
scope::scope(scope *_parent)
{
    parent = _parent;
}

void scope::insert(std::string name, symbol var)
{
    //std::cout << "Inserting variable " << name << " into this scope -> " <<
var.tostring() << "\n";
    variables[name] = var;
}

// If this scope has a copy then return that, else refer upwards.
symbol& scope::get(std::string name)
{
    if (variables.find(name) != variables.end())
        return variables[name];
    else if (parent)
        return parent->get(name);
    else
        throw_error("Error: undefined name _!");
}

// If the current scope has a match then return true, else refer upwards
bool scope::exists(std::string name)
{
    if (variables.find(name) != variables.end())
        return true;
    else if (parent)
        return parent->exists(name);
```

```
    else
        return false;
}

bool scope::inthisscope(std::string name)
{
    return variables.find(name) != variables.end();
}

// Compiler definitions //

// set up scopes and signatures:
compiler::compiler()
{
    globalscope = new scope();
    currentscope = globalscope;
    func_signature sig;
    sig.args_must_match = false;
    sig.arg_types.push_back(type_number);
    sig.arg_types.push_back(type_number);
    functions["nfc"] = sig;
    sig.arg_types.push_back(type_number);
    sig.arg_types.push_back(type_number);
    functions["nfc4"] = sig;
    func_signature val_sig;
    val_sig.return_type = type_pointer;
    val_sig.arg_types.push_back(type_int);
    functions["val"] = val_sig;
    currentfuncdef = 0;
}

void compiler::pushscope()
{
    currentscope = new scope(currentscope);
}

void compiler::popscope()
{
    scope *oldscope = currentscope;
    currentscope = currentscope->parent;
    delete oldscope;
}

// add a new variable to the current scope: it is saved in the current scope
// with its local name, and in the global symbol table with its global name,
// so we can do type checking and stuff in the second pass.
void compiler::addvar(std::string name, type_t type, int ptr, int linenumber,
bool isConstant, int constvalue)
{
    if (currentscope->inthisscope(name))
        throw(error("Error: duplicate declaration of variable " + name + " on
line "));
    symbol var;
    var.name = makeguid(name, ptr);
    var.type = type;
    var.is_constant = isConstant;
    if (isConstant)
        var.value = constvalue;
    globalsymboltable[var.name] = var;
    currentscope->insert(name, var);
}
```

```
// Run through all the definitions that make up the program - delegate to the
proper
// functions depending on the definition type.
object* compiler::compile(program *prog)
{
    for (unsigned int i = 0; i < prog->defs.size(); i++)
    {
        definition *def = prog->defs[i];
        if (def->type == dt_constdef)
        {
            constdef *cdef = (constdef*)def;
            addvar(cdef->name, cdef->valtype, (long)cdef, cdef->linenumber,
true, cdef->value);
        }
        else if (def->type == dt_macrodef)
        {
            compile((macrodef*)def);
        }
        else if (def->type == dt_funcdef)
        {
            funcdef *previousfuncdef = currentfuncdef;
            currentfuncdef = (funcdef*)def;
            compile(currentfuncdef);
            currentfuncdef = previousfuncdef;
        }
        else if (def->type == dt_vardec)
        {
            compile((vardeclaration*)def);
        }
    }
#ifdef EBUG // gcc -DEBUG :o)
    std::cout << "\nGlobal symbol table:\n";
    std::map<std::string, symbol>::iterator iter = globalsymboltable.begin();
    for(; iter != globalsymboltable.end(); iter++)
    {
        std::cout << iter->first << "\n";
    }
#endif
    object *obj = new object;
    obj->defined_funcs = defined_funcs;
    obj->tree = prog;
    std::vector<definition*>::iterator idef = obj->tree->defs.begin();
    while (idef != obj->tree->defs.end())
    {
        definition *def = *idef;
        if (def->type == dt_constdef || (def->type == dt_funcdef &&
!((funcdef*)def)->defined) || def->type == dt_macrodef)
        {
            idef = obj->tree->defs.erase(idef);     // strip out everything
apart from function definitions. (macros have already been subbed by this point)
        }
        else
        {
            idef++;
        }
    }
    return obj;
}

void compiler::compile(macrodef *mdef)
```

```
{
    //std::string guid = makeguid(mdef->name, (long)mdef);
    pushscope();
    func_signature sig;
    sig.args_must_match = false;
    sig.is_macro = true;
    sig.return_type = type_none;
    sig.def = (funcdef*)mdef;
    for (unsigned int i = 0; i < mdef->args.size(); i++)
    {
        addvar(mdef->args[i], type_expression, (long)&(mdef->args[i]), mdef-
>linenumber);
        //std::cout << "Renaming " << mdef->args[i] << " to " << currentscope-
>get(mdef->args[i]).name << "\n";
        symbol sym = currentscope->get(mdef->args[i]);
        mdef->args[i] = sym.name;
        globalsymboltable[mdef->args[i]] = sym;
        sig.arg_types.push_back(type_none);
    }
    if (defined_funcs.find(mdef->name) != defined_funcs.end())
        throw_error("Error: conflicting definitions of macro \"" << mdef->name
<< "\"");
    popscope();
    functions[mdef->name] = sig;
    defined_funcs.insert(mdef->name);
}


// check for signature conflicts, check for definition conficts,
// and then compile the function body if there is one.
void compiler::compile(funcdef *fdef)
{
    func_signature sig;
    sig.return_type = fdef->return_type;
    pushscope();
    for (unsigned int i = 0; i < fdef->args.size(); i++)
    {
        argument *arg = &(fdef->args[i]);
        addvar(arg->name, arg->type, (long)arg, fdef->linenumber);
        arg->name = currentscope->get(arg->name).name;
        sig.arg_types.push_back(arg->type);
    }
    if (functions.find(fdef->name) != functions.end() && sig != functions[fdef-
>name])
    {
        throw_error("Error: conflicting type declarations for function \"" <<
fdef->name << "\"");
    }
    else
    {
        functions[fdef->name] = sig;
    }

    if (defined_funcs.find(fdef->name) == defined_funcs.end()) // i.e. function
is currently undefined
    {
        if (fdef->defined)
        {
            symbol retsym;
            retsym.type = fdef->return_type;
            retsym.name = fdef->name + ":__returnval";
```

```
            currentscope->insert(fdef->name, retsym);
            globalsymboltable[retsym.name] = retsym;
            if (!fdef->exported)
                compile(fdef->body, "", "", makeguid("__return", (long)fdef));
            defined_funcs.insert(fdef->name);
        }
    }
    else if (fdef->defined)  // function exists (this is else clause of above)
and we are processing a definition as opposed to merely a declaration
    {
        throw_error("Error: conflicting definitions of function \"" << fdef-
>name << "\"");
    }
    popscope();
}

// Push all the variable declarations into a new scope.
// Run through all the statements and delegate their compilation.
void compiler::compile(block *blk, std::string exitlabel, std::string toplabel,
std::string returnlabel)
{
    pushscope();
    // Process variable declarations:
    for (unsigned int i = 0; i < blk->declarations.size(); i++)
    {
        compile(blk->declarations[i]);
    }
    // scan through for labels: (because they break the forward-view scoping
rule)
    for (unsigned int i = 0; i < blk->statements.size(); i++)
    {
        // NOTE: we put a label in the declarations, but a pointer variable in
the current scope (for type checking purposes). Compiler thinks pointer, linker
knows label.
        if (blk->statements[i]->type == stat_label)
        {
            addvar(((label*)blk->statements[i])->name, type_pointer, (long)(blk-
>statements[i]), blk->linenumber);
            vardeclaration *dec = new vardeclaration;
            vardeclaration::varpair var;
            var.type = type_label;
            var.name = currentscope->get(((label*)blk->statements[i])-
>name).name;
            dec->vars.push_back(var);
            blk->declarations.push_back(dec);
        }
    }

    // compile the remaining statements:
    for (unsigned int i = 0; i < blk->statements.size(); i++)
    {
       compile(blk->statements[i], exitlabel, toplabel, returnlabel);      //
reference to pointer to statement (so we can reassign it)
    }
    popscope();
}

void compiler::compile(statement *&stat, std::string exitlabel, std::string
toplabel, std::string returnlabel)
{
    switch (stat->type)
```

```
    {
    case stat_block:
        compile((block*)stat, exitlabel, toplabel, returnlabel);
        break;
    case stat_call:
        compile((funccall*&)stat);
        break;
    case stat_empty:
        break;
    case stat_goto:
        compile((goto_stat*)stat);
        break;
    case stat_label:
        ((label*)stat)->name = currentscope->get(((label*)stat)->name).name;
// replace local label with globally unique one
        break;
    case stat_if:
        compile((if_stat*)stat, exitlabel, toplabel, returnlabel);
        break;
    case stat_while:
        compile((while_stat*)stat, returnlabel);
        break;
    case stat_assignment:
        compile((assignment*)stat);
        break;
    case stat_break:
        if (exitlabel == "")
            throw(error("Error: no loop to break from"));
        delete stat;
        stat = new goto_stat;
        ((goto_stat*)stat)->target = new expression(exitlabel);
        break;
    case stat_continue:
        if (toplabel == "")
            throw(error("Error: no loop to continue"));
        delete stat;
        stat = new goto_stat;
        ((goto_stat*)stat)->target = new expression(toplabel);
        break;
    case stat_return:
        if (returnlabel == "")
            throw(error("Error: no function to break from"));
        delete stat;
        stat = new goto_stat;
        ((goto_stat*)stat)->target = new expression(returnlabel);
        break;
    default:
        throw(error("Error: unrecognised statement type"));
    }
}

void compiler::compile(vardeclaration *dec)
{
    for (unsigned int j = 0; j < dec->vars.size(); j++)
    {
        vardeclaration::varpair &var = dec->vars[j];
        addvar(var.name, var.type, (long)dec, dec->linenumber);
        var.name = currentscope->get(var.name).name;    // replace the
declaration with the global name of the var; makes linking easier.
    }
}
```

```
// compile ALL the arguments!
void compiler::compile(funccall *&fcall)
{
	//std::cout << "Call to function " << fcall->name << "\n";
	if (functions.find(fcall->name) == functions.end())
	{
		throw_error("Error: implicit declaration of function \"" << fcall->name
<< "\"");
	}
	if (currentfuncdef)
	{
		currentfuncdef->dependson.insert(fcall->name);
	}
	func_signature &sig = functions[fcall->name];
	if (fcall->args.size() < sig.arg_types.size())
		throw(error("Error: not enough arguments to function " + fcall->name));
	else if (fcall->args.size() > sig.arg_types.size())
		throw(error("Error: too many arguments to function " + fcall->name));
	if (!sig.is_macro) // is function
	{
		for (unsigned int argnum = 0; argnum < fcall->args.size(); argnum++)
		{
			compile(fcall->args[argnum]);
			if (sig.args_must_match && !match_types(sig.arg_types[argnum],
fcall->args[argnum]->val_type))
				throw_type_error(std::string("argument ") + "" + " to function "
+ fcall->name, sig.arg_types[argnum], fcall->args[argnum]->val_type, fcall-
>linenumber);
		}
	}
	else
	{
		pushscope();
		block *macrobody = ((macrodef*)sig.def)->body->getcopy();
		for (unsigned int argnum = 0; argnum < fcall->args.size(); argnum++)
		{
			std::string argname = ((macrodef*)sig.def)->args[argnum];
			expression_subs[argname] = fcall->args[argnum];
			currentscope->insert(argname.substr(0, argname.find('@')),
globalsymboltable[argname]);
		}
		compile(macrobody);
		fcall = (funccall*)macrobody; // It's still actually a block, and the
type tag will tell the linker this, we just need to cast the pointer for the
assignment.
		popscope();
	}
}

// just compile the target expression...
void compiler::compile(goto_stat *sgoto)
{
	compile(sgoto->target);
}

/*void compiler::compile(label *lbl)
{
	//... do we really need this?
}*/
```

```
// we only need to compile the expression and if bodies:
// the actual code generation and labelling happens at link time.
void compiler::compile(if_stat *ifs, std::string exitlabel, std::string
toplabel, std::string returnlabel)
{
	compile(ifs->expr);
	compile(ifs->ifblock, exitlabel, toplabel, returnlabel);
	if (ifs->elseblock)
		compile(ifs->elseblock, exitlabel, toplabel, returnlabel);
}

void compiler::compile(while_stat *whiles, std::string returnlabel)
{
	compile(whiles->expr);
	compile(whiles->blk, makeguid("__exit", (long)whiles->blk),
makeguid("__top", (long)whiles->blk), returnlabel);
}

void compiler::compile(assignment *assg)
{
	if (!currentscope->exists(assg->name))
	{
		throw(error("Error: undeclared variable " + assg->name));
	}
	assg->name = currentscope->get(assg->name).name;
	compile(assg->expr);
	type_t assg_type = globalsymboltable[assg->name].type;
	if (assg->indexed)
	{
		if (assg_type.type != type_array)
			throw(error("Error: attempt to index non-array type."));
		assg_type.type = assg_type.second;	// match expression against
content type, not type_array.
	}
	// Check for type mismatch:
	type_t secondtype = assg->expr->val_type.second;
	if (!match_types(assg_type, assg->expr->val_type) &&
		!(assg->expr->indexed && match_types(assg_type, secondtype)))
		throw_type_error("assignment of variable " + assg->name.substr(0, assg-
>name.find("@")), assg_type, assg->expr->val_type, assg->linenumber);
}

// To compile an expression:
// - if it's a name, check that it exists in the current scope
//	- if so, replace the local name with the global one.
//	- if it refers to a constant, swap the constant value in for the name.
// - otherwise, leave it
// - get its type once we've compiled it
void compiler::compile(expression *&expr)
{
	bool typeAlreadyDetermined = false;
	if (expr->type == exp_name)
	{
		if (!currentscope->exists(expr->name))
		{
			throw_error("Error: undeclared name \"" << expr->name << "\" in
expression on line " << expr->linenumber);
		}
		//std::cout << "Expression containing " << expr->name;
		expr->name = currentscope->get(expr->name).name;		// replace local
name with globally unique name;
```

```
        //std::cout <<  "\n    " << globalsymboltable[expr->name].tostring() << 
"\n";
        if (globalsymboltable[expr->name].is_constant)          // if it's a 
constant, fetch the value from the global symbol table and replace.
        {
            expr->type = exp_number;
            expr->number = globalsymboltable[expr->name].value;
            expr->val_type = globalsymboltable[expr->name].type;
            typeAlreadyDetermined = true;
        }
        else if (globalsymboltable[expr->name].type == type_expression)
        {
            expr = expression_subs[expr->name]->getcopy();
            //std::cout << "Subbing expression: (";
            //printtree(expr);
            //std::cout << ")\n";
            compile(expr);
        }
    }
    else if (expr->type == exp_funccall)
    {
        if (functions.find(expr->name) == functions.end())
            throw(error("Error: no such function: " + expr->name));
        func_signature &sig = functions[expr->name];
        if (currentfuncdef)
        {
            currentfuncdef->dependson.insert(expr->name);
        }
        if (expr->args.size() < sig.arg_types.size())
            throw(error("Error: not enough arguments to function " + expr-
>name));
        else if (expr->args.size() > sig.arg_types.size())
            throw(error("Error: too many arguments to function " + expr->name));
        for (unsigned int i = 0; i < expr->args.size(); i++)
        {
            compile(expr->args[i]);
            if (sig.args_must_match && !match_types(sig.arg_types[i], expr-
>args[i]->val_type))
                throw_type_error(std::string("argument ") + "" + " to function " 
+ expr->name, sig.arg_types[i], expr->args[i]->val_type, expr->linenumber);
        }
    }
    else if (expr->type == exp_and || expr->type == exp_or || expr->type == 
exp_not)
    {
        for (unsigned i = 0; i < expr->args.size(); i++)
        {
            compile(expr->args[i]);
            if (!match_types(type_int, expr->args[i]->val_type))
                throw_error("Error: logical operators apply only to booleans 
(line " << expr->linenumber << ")");
        }
    }

    else if (expr->type != exp_number && expr->type != exp_string)
    {
        throw_error("Error: attempted to compile unknown expression type on line 
" << expr->linenumber);
    }
    // tell the compiler to make note of the type: (if we haven't already gotten 
it from the symbol table)
```

```
    if (!typeAlreadyDetermined)
        gettype(expr);
}

// Find the type of the expression (possibly by looking at subexpressions), and
// then note it in the type field.
// As this is called at the end of the end of expression compilation, it is
// guaranteed that subexpressions will already be compiled, so their types
// will be known.
void compiler::gettype(expression *expr)
{
    switch (expr->type)
    {
    case exp_name:
        expr->val_type = globalsymboltable[expr->name].type;
        if (expr->val_type.type == type_array && expr->indexed)
            expr->val_type.type = expr->val_type.second;
        break;
    case exp_number:
        expr->val_type = type_number;
        break;
    case exp_funccall:
        expr->val_type = functions[expr->name].return_type;
        break;
    case exp_string:
        expr->val_type = type_t(type_array, type_int, expr->name.size() + 1);
        break;
    case exp_and:
    case exp_not:
    case exp_or:
        expr->val_type = type_int;
    }
}


// check that received matches accepted, and refine the generic "number" type.
// if there is no possible match, return false.
bool compiler::match_types(type_t expected, type_t &received)
{
    if (expected == received)
    {
       return true;
    }
    else if ((expected == type_int || expected == type_pointer) && received ==
type_number)
    {
        received = expected;   // replace generic number with int/pointer.
        return true;
    }
    else if (expected == type_pointer && received.type == type_array)
    {
        return true;
    }
    else
    {
        return false;
    }
}

std::string symbol::tostring()
{
```

```
    std::stringstream ss;
    ss << "(Name: " << name << ", type: " << type.getname();
    if (is_constant)
        ss << " (" << value << ")";
    ss << (is_constant ? ", constant)" : ", not constant)");
    return ss.str();
}
```

vardict.h

```
#ifndef _VARDICT_H_
#define _VARDICT_H_

#include <map>
#include <string>
#include <vector>

#include "object.h"
#include "linkval.h"

class vardict;

struct variable
{
    friend class vardict;
    type_t type;
    variable *next;      // for stack operations we can build a linked list, in case we get the same symbol twice. (Shadowing)
    linkval address;
    variable() {next = 0;}
private:
    int offset;     // we don't want to use this by mistake...
};

class vardict
{
    std::map<std::string, variable*> vars;
    std::vector<bool> memory_in_use;
    std::vector<bool> has_been_used;
    int first_available_space;
    int getspace(int size);
    void find_first_available_space(int searchstart = 0);
    std::vector<std::vector<std::string> > tempscopes;       // in an if/while test we may use multiple temp locations, and we don't want them to clobber each other, so we keep track of temps and clean up after test finished.
public:
    bool start_from_top;
    linkval addvar(std::string name, type_t type);
    void registervar(std::string name, type_t type, linkval address);    // for when we want to push an existing address as a var, and the memory is already allocated. (it's removed in the same way)
    void remove(std::string name);
    variable* getvar(std::string name);
    bool exists(std::string name);
    void push_function_scope();      // so functions can't clobber each other's memory, we mark all memory used by other functions as currently in use.
    void push_temp_scope();
    void pop_temp_scope();
    void remove_on_pop(std::string name);
```

```
    vardict();
};

#endif // _VARDICT_H_
```

vardict.cpp

```
#include "vardict.h"

#include <iostream>

#include "error.h"
#include "linker.h"

void vardict::find_first_available_space(int searchstart)
{
    for (int i = 0; i < HEAP_SIZE; i++)
    {
        if (!memory_in_use[i])
        {
            first_available_space = i;
            return;
        }
    }
    // If it fell through:
    throw(error("Error: no more free space in heap!"));
}

// Search through memory for a block large enough
// If we find it, block it out and return the start.
int vardict::getspace(int size)
{
    int pos = first_available_space;
    while (pos < HEAP_SIZE)
    {
        int start = pos;
        bool enoughSpace = true;
        for (; pos < start + size && enoughSpace; pos++)
        {
            if (memory_in_use[pos])
            {
                enoughSpace = false;
                pos++;          // start checking again at the next unchecked
location
            }
        }
        if (enoughSpace)
        {
            for (int i = start; i < start + size; i++)
            {
                memory_in_use[i] = true;
                has_been_used[i] = true;
            }
            if (start == first_available_space)
                find_first_available_space(first_available_space);        // if
we've covered the first known free space, find a new one.
            return start;
        }
    }
    throw(error("Error: no more free space in heap!"));
```

```
}

// Note: returns an offset from the heap start, you'll have to add
// the heap bottom to this value to get a machine address.
linkval vardict::addvar(std::string name, type_t type)
{
	variable *var = new variable;
	var->type = type;
	var->offset = getspace(type.getstoragesize());
	if (vars.find(name) != vars.end())
		var->next = vars[name];  // push the stack down one...
	vars[name] = var;
	if (start_from_top)
		var->address = HEAP_TOP - var->offset - type.getstoragesize() + 1;
	else
		var->address = linkval("__program_end") + var->offset;
#ifdef EBUG
	std::cout << "adding var " << name << " 0x" << std::hex << var->offset <<
"\n";
#endif
	return var->address;
}

void vardict::registervar(std::string name, type_t type, linkval address)
{
	variable *var = new variable;
	var->type = type;
	var->offset = -1;
	if (vars.find(name) != vars.end())
		var->next = vars[name];
	vars[name] = var;
	var->address = address;
}

void vardict::remove(std::string name)
{
	// sanity check:
	std::map<std::string, variable*>::iterator iter = vars.find(name);
	if (iter == vars.end())
		throw_error("Error: tried to free non-existent variable \"" << name <<
"\"! (Link-time)");
	variable *var = iter->second;
	// release the memory:
	if (var->offset >= 0)   // registervar() doesn't allocate memory, so it sets
offset to -1.
	{
		for (int i = var->offset; i < var->offset + var->type.getstoragesize();
i++)
			memory_in_use[i] = 0;
		if (var->offset < first_available_space)
			first_available_space = var->offset;
	}
	// update the dictionary: pop or remove
	if (var->next)
	{
		iter->second = var->next;
		delete var;
	}
	else
	{
		delete var;
```

```
        vars.erase(iter);
    }
}

variable* vardict::getvar(std::string name)
{
    std::map<std::string, variable*>::iterator iter = vars.find(name);
    if (iter == vars.end())
        return 0;
    else
        return iter->second;
}

bool vardict::exists(std::string name)
{
    return vars.find(name) != vars.end();
}

// When we link a new function body, we mark all memory that is used by other
functions as currently in use.
// This means memory is still packed as efficiently as possible within
functions, but functions
// can't clobber memory used by others when they declare vars.
void vardict::push_function_scope()
{
    memory_in_use = has_been_used;
}

void vardict::push_temp_scope()
{
    tempscopes.push_back(std::vector<std::string>());
}

void vardict::pop_temp_scope()
{
    if (tempscopes.size() < 1)
        throw(error("Linker error: no tempscope to pop!"));
    std::vector<std::vector<std::string> >::iterator iter = tempscopes.end() -
1;
    std::vector<std::string> &names = *iter;
    for (unsigned int i = 0; i < names.size(); i++)
    {
#ifdef EBUG
        std::cout << "popping " << names[i] << "\n";
#endif
        remove(names[i]);
    }
    tempscopes.erase(iter);
}

void vardict::remove_on_pop(std::string name)
{
    if (tempscopes.size() < 1)
        remove(name);               // if we're not in a temp scope, assume we
can just chuck the variable away.
    else
        tempscopes[tempscopes.size() - 1].push_back(name);
}

vardict::vardict()
{
```

```
    first_available_space = 0;
    start_from_top = true;
    for (int i = 0; i < HEAP_SIZE; i++)
    {
        memory_in_use.push_back(0);
        has_been_used.push_back(0);
    }
}
```

linkval.h

```
#ifndef LINKVAL_H_INCLUDED
#define LINKVAL_H_INCLUDED

#include <stdint.h>
#include <string>

// linkvals are our "Assembly language" - they let us pass symbols and
expressions for machine code
// instead of just the literal addresses, e.g. with labels where we don't know
the location til we reach it.
// They get evaluated in the final "assemble" step.

typedef enum
{
    lv_literal = 0,
    lv_symbol,
    lv_expression
} lv_type;

struct linkval
{
    enum op_type {op_add, op_sub, op_gethigh, op_getlow};
    lv_type type;
    uint16_t literal;
    std::string sym;
    linkval *argA;
    linkval *argB;
    op_type operation;
    linkval() {argA = 0; argB = 0;}
    linkval(uint16_t lit):linkval() {type = lv_literal; literal = lit;}
    linkval(std::string s):linkval() {type = lv_symbol; sym = s;}
    linkval operator+(linkval rhs) const;
    linkval operator-(linkval rhs) const;
    bool operator==(linkval rhs) const;
    bool operator!=(linkval rhs) const;
    linkval gethighbyte() const;
    linkval getlowbyte() const;
    std::string tostring() const;
};

#endif // LINKVAL_H_INCLUDED
```

linkval.cpp

```
#include "linkval.h"
#include <iostream>
#include <sstream>
```

```
// If they're both literals, just add their values
// Otherwise, return an expression with pointers to the two arguments.
linkval linkval::operator+(linkval rhs) const
{
    linkval result(0);
    switch (type)
    {
    case lv_literal:
        if (rhs.type == lv_literal)
        {
            result.type = lv_literal;
            result.literal = literal + rhs.literal;
            break;
        }   // fall through if not:
    case lv_symbol:
    case lv_expression:
        result.type = lv_expression;
        result.operation = op_add;
        result.argA = new linkval(0);
        *result.argA = *this;
        result.argB = new linkval(0);
        *result.argB = rhs;
        break;
    }
    return result;
}

// If they're both literals, just subtract their values
// Otherwise, return an expression with pointers to the two arguments.
linkval linkval::operator-(linkval rhs) const
{
    linkval result(0);
    switch (type)
    {
    case lv_literal:
        if (rhs.type == lv_literal)
        {
            result.type = lv_literal;
            result.literal = literal - rhs.literal;
            break;
        }   // fall through if not:
    case lv_symbol:
    case lv_expression:
        result.type = lv_expression;
        result.operation = op_sub;
        result.argA = new linkval(0);
        *result.argA = *this;
        result.argB = new linkval(0);
        *result.argB = rhs;
        break;
    }
    return result;
}

// if they're different types, they're not equal.
// otherwise, check whether their values are equal.
// symbols don't get looked up, just directly compared.
bool linkval::operator==(linkval rhs) const
{
    if (type != rhs.type)   // this is a conservative equality: those of
different types may be equal, but we assume not.
```

```
        return false;
    switch (type)
    {
    case lv_literal:
        return literal == rhs.literal;
    case lv_symbol:
        return sym == rhs.sym;
    case lv_expression:
        return operation == rhs.operation &&
        (argA && rhs.argA ? *argA == *rhs.argA : argA == rhs.argA) &&
        (argB && rhs.argB ? *argB == *rhs.argB : argB == rhs.argB);     //
compare by value if both non-null, else compare by reference.
    default:
        return false;
    }
}

bool linkval::operator!=(linkval rhs) const
{
    return !(operator==(rhs));
}

// Shift it, append a "_HI", or return an expression.
linkval linkval::gethighbyte() const
{
    if (type == lv_literal)
        return literal >> 8;
    else if (type == lv_symbol)
        return sym + "_HI";
    else
    {
        linkval lv(0);
        lv.type = lv_expression;
        lv.operation = op_gethigh;
        lv.argA = new linkval(0);
        *lv.argA = *this;
        return lv;
    }
}

// Shift it, append a "_LO", or return an expression.
linkval linkval::getlowbyte() const
{
    if (type == lv_literal)
        return literal & 0xff;
    else if (type == lv_symbol)
        return sym + "_LO";
    else
    {
        linkval lv(0);
        lv.type = lv_expression;
        lv.operation = op_getlow;
        lv.argA = new linkval(0);
        *lv.argA = *this;
        return lv;
    }
}

std::string linkval::tostring() const
{
    std::stringstream ss;
```

```
    if (type == lv_literal)
        ss << "literal: " << std::hex << literal;
    else if (type == lv_symbol)
        ss << "symbol: " << sym;
    else
        ss << "expression: (" << ")";
    return ss.str();
}
```

linker.h

```
#ifndef LINKER_H_INCLUDED
#define LINKER_H_INCLUDED

#include "object.h"
#include "linkval.h"
#include "vardict.h"

#include <map>
#include <sstream>
#include <stdint.h>
#include <vector>

const int ROM_SIZE = 32 * 1024;
const int INCREMENT_START = ROM_SIZE - 1024;
const int DECREMENT_START = INCREMENT_START + 256;
const int LEFTSHIFT_START = DECREMENT_START + 256;
const int RIGHTSHIFT_START = LEFTSHIFT_START + 256;
const int RAM_SIZE = 16 * 1024;
const int RAM_TOP = 0xbfff;
const int HEAP_TOP = 0xbfbf;
const int HEAP_BOTTOM = 0x8000;
const int HEAP_SIZE = HEAP_TOP - HEAP_BOTTOM + 1;

const int POINTER_READ_INSTRUCTION = RAM_TOP - 0x2f;
const int POINTER_READ_PVECTOR = POINTER_READ_INSTRUCTION + 0x0a;       // B
field of the next instruction - the one we read from.
const int JUMP_INSTRUCTION = RAM_TOP - 0x1f;
const int JUMP_PVECTOR = JUMP_INSTRUCTION + 0x6;
const int POINTER_READ_RESULT = RAM_TOP - 0x0f;
const int POINTER_WRITE_VALUE = POINTER_READ_RESULT;
const int POINTER_WRITE_CLEAR_INSTRUCTION = POINTER_READ_INSTRUCTION - 0x10;
const int POINTER_WRITE_COPY_INSTRUCTION = POINTER_WRITE_CLEAR_INSTRUCTION +
0x8;

class linker
{
    vardict vars;
    std::map<std::string, definition*> defined_funcs;
    std::vector<definition*> definitions;
    int index;
    int current_address;
    bool compile_to_ram;
    std::vector<linkval> buffer;
    std::map<std::string, linkval> valtable;        // contains values for
substitution
    std::vector<std::pair<std::string, std::string> > stringvalues;    // label,
contents
    std::stringstream defstring;    // contains global variable and function
export data
```

```
    void savelabel(std::string, linkval);
    void write8(linkval);
    void write16(linkval);
    void padto8bytes();
    linkval evaluate_or_return_literal(expression*);
// Code generation routines:
    void emit_nfc2(linkval x, linkval y);
    void emit_branchifzero(linkval testloc, linkval dest, bool amend_previous =
false, bool invert = false);
    void emit_branchifnonzero(linkval testloc, linkval dest, bool amend_previous
= false);            // shim for e_biz with invert = true
    void emit_branchalways(linkval dest, bool always_emit = false);     // can
pass true to disable the follow behaviour, if the jump is jumped to (e.g. end of
while)
    void emit_copy(linkval src, linkval dest);
    void emit_copy_inverted(linkval src, linkval dest);
    void emit_writeconst(uint8_t val, linkval dest);
    void emit_copy_multiple(linkval src, linkval dest, int nbytes);
    void emit_writeconst_multiple(int value, linkval dest, int nbytes);
    void emit_writelabel(std::string label, linkval dest);
    bool last_instruction_points_to_this_one();
// Tree traversal:
    void link(funcdef*);
    void exportfuncdef(funcdef*);
    void exportvardeclaration(vardeclaration *vardec);
    void link(block*);
    void link(statement*);
    void link(funccall*);
    linkval linkfunctioncall(std::vector<expression*>&, funcdef*);
    linkval linkbuiltinfunction(std::vector<expression*>&, std::string, bool
givenpreferred = false, linkval preferred = 0);
    void link(goto_stat*);
    void link(if_stat*);
    void link(while_stat*);
    void link(assignment*);
    linkval evaluate(expression*, bool givenpreferred = false, linkval preferred
= 0);
    uint16_t evaluate(linkval);
    std::vector<char> assemble();
    void allocatefunctionstorage();
    std::set<std::string> analysedependencies(std::string rootfunc);
    void removeunusedfunctions();
public:
    linker();
    void add_object(object* obj);
    std::vector<char> link();
    std::string getdefstring();
    void setcompiletoram(bool ram);
    bool strip_unused_functions;
};

#endif // LINKER_H_INCLUDED
```

linker.cpp

```
#include "error.h"
#include "linker.h"
```

```
// REMOVE:
#include "printtree.h"

#include <iostream>
#include <set>
#include <sstream>

//////////////////////// General defs ////////////////////////////

std::string makeguid(std::string name, int ptr);

std::string getlabel()
{
    static int lcount = 0;
    std::stringstream ss;
    ss << "__L" << std::hex << lcount++;
    return ss.str();
}

uint16_t getconstaddress(uint8_t val)
{
    return DECREMENT_START + ((val + 1) % 256);
}


//////////////////////// Linker defs ////////////////////////////


linker::linker()
{
    // Mark all of the builtin funcs as defined but with NULL definitions
    defined_funcs["nfc"] = 0;
    defined_funcs["nfc4"] = 0;
    defined_funcs["val"] = 0;
    defined_funcs["read"] = 0;
    defined_funcs["write"] = 0;
    defined_funcs["increment"] = 0;
    defined_funcs["decrement"] = 0;
    defined_funcs["shiftleft"] = 0;
    defined_funcs["shiftright"] = 0;

    index = 0;
    current_address = 0;
    compile_to_ram = false;
    strip_unused_functions = false;
    buffer.reserve(ROM_SIZE);
}

// Make note of a label's location and store it in the substitution table
void linker::savelabel(std::string name, linkval current_address)
{
    valtable[name] = current_address;
    valtable[name + "_HI"] = current_address.gethighbyte();
    valtable[name + "_LO"] = current_address.getlowbyte();
}

// NB: index refers to the location _about_ to be written. (I.e. next location)
void linker::write8(linkval val)
{
    if (index >= ROM_SIZE)
        throw(error("Error: program too big!"));
```

```
    buffer.push_back(val);
    index++;
    current_address++;
}

void linker::write16(linkval val)
{
    write8(val.gethighbyte());
    write8(val.getlowbyte());
}

void linker::padto8bytes()
{
    while ((buffer.size() % 8) != 0)
        write8(0);
}

void linker::emit_nfc2(linkval x, linkval y)
{
    padto8bytes();
    uint16_t next = current_address + 8;
    write16(x);
    write16(y);
    write16(next);
    write16(next);
}

// clear temp; set temp to not(x);  invert temp and branch on result.
void linker::emit_branchifzero(linkval testloc, linkval dest, bool
amend_previous, bool invert)
{
    //amend_previous = false; /// /// /// ///
    bool emit_full = !amend_previous;
    if (amend_previous)
    {
        if (testloc.gethighbyte() != buffer[index - 8] || testloc.getlowbyte()
!= buffer[index - 7])
            emit_full = true;
        if (!invert)
        {
            buffer[index - 2] = dest.gethighbyte(); // branch on zero
            buffer[index - 1] = dest.getlowbyte();
        }
        else
        {
            buffer[index - 4] = dest.gethighbyte(); // branch on non zero
            buffer[index - 3] = dest.getlowbyte();
        }
    }
    if (emit_full)
    {
        padto8bytes();
        linkval temploc = vars.addvar("__brztemp", type_int);
        emit_nfc2(temploc, getconstaddress(0xff));
        emit_nfc2(temploc, testloc);
        uint16_t next = current_address + 8;
        write16(temploc);
        write16(temploc);
        if (!invert)
        {
            write16(next);
```

```
            write16(dest);  // dest if 0
        }
        else
        {
            write16(dest);  // dest if non zero
            write16(next);
        }
        vars.remove("__brztemp");
    }
}

void linker::emit_branchifnonzero(linkval testloc, linkval dest, bool
amend_previous)
{
    // simple shim for emit_branchifzero
    emit_branchifzero(testloc, dest, amend_previous, true);
}

void linker::emit_branchalways(linkval dest, bool always_emit)
{
    padto8bytes();
    // Check if the last instruction points at this one:
    if (!always_emit && last_instruction_points_to_this_one())
    {
        // If so we can just retroactively amend it to the new jump target.
        buffer[index - 4] = dest.gethighbyte();
        buffer[index - 3] = dest.getlowbyte();
        buffer[index - 2] = dest.gethighbyte();
        buffer[index - 1] = dest.getlowbyte();
    }
    else
    {
        // Otherwise do a pointless copy (a NOP) and jump from this instruction.
        linkval temploc = vars.addvar("__bratemp", type_int);
        write16(temploc);
        write16(temploc);
        write16(dest);
        write16(dest);
        vars.remove("__bratemp");
    }

}

void linker::emit_copy(linkval src, linkval dest)
{
    // Copy to self is destructive! (As destination is cleared before copy... x
= x; would otherwise set x to 0)
    // Note however that linkval inequality will compare false in some cases
such as labels vs. literals, so don't trust it.
    if (src == dest)
        return;
    // Note that this also protects against __brztemp being in the same place as
a temporary function return loc.
    emit_nfc2(dest, getconstaddress(0xff));
    emit_nfc2(dest, src);
    emit_nfc2(dest, dest);

}

void linker::emit_copy_inverted(linkval src, linkval dest)
{
```

```
    if (src == dest)
    {
        emit_nfc2(dest, dest);  // just invert, no copy (so it still does what
we expect)
    }
    else
    {
        emit_nfc2(dest, getconstaddress(0xff));
        emit_nfc2(dest, src);
    }
}

void linker::emit_writeconst(uint8_t val, linkval dest)
{
    emit_nfc2(dest, getconstaddress(0xff));
    if (val)    // no need to do this if 0, as already cleared once.
        emit_nfc2(dest, getconstaddress(~val));
}

void linker::emit_copy_multiple(linkval src, linkval dest, int nbytes)
{
    for (int i = 0; i < nbytes; i++)
    {
        emit_copy(src + i, dest + i);
    }
}

void linker::emit_writeconst_multiple(int value, linkval dest, int nbytes)
{
    for (int i = 0; i < nbytes; i++)
    {
        emit_writeconst((value >> (nbytes - i - 1) * 8) & 0xff, dest + i);
    }
}

void linker::emit_writelabel(std::string label, linkval dest)
{
    emit_nfc2(dest, getconstaddress(0xff));
    emit_nfc2(dest, linkval(DECREMENT_START) + (linkval(256) - linkval(label +
"_HI")).getlowbyte());          // 256 - x   == ~x + 1
    emit_nfc2(dest + 1, getconstaddress(0xff));
    emit_nfc2(dest + 1, linkval(DECREMENT_START) + (linkval(256) - linkval(label
+ "_LO")).getlowbyte());        // getlowbyte is equivalent to & 0xff; makes 0 -
> 0 instead of 256.
}

bool linker::last_instruction_points_to_this_one()
{
    // true if both skip fields match, and each pair equals the current index.
    return
        buffer[index - 4] == buffer[index  - 2] && buffer[index - 3] ==
buffer[index - 1] &&
        buffer[index - 4].type == lv_literal && buffer[index - 4].literal ==
index >> 8 &&
        buffer[index - 3].type == lv_literal && buffer[index - 3].literal ==
(index & 0xff);
}

void linker::add_object(object *obj)
{
    // check for collisions:
```

```
    for (std::set<std::string>::iterator iter = obj->defined_funcs.begin(); iter
!= obj->defined_funcs.end(); iter++)
    {
        if (defined_funcs.find(*iter) != defined_funcs.end())
        {
            std::stringstream ss;
            ss << "Error: multiple definitions of function \"" << *iter << "\"";
            throw(error(ss.str()));
        }
    }
    // add all the definitions to linker's symbol table:
    for (unsigned int i = 0; i < obj->tree->defs.size(); i++)
    {
        definition *def = obj->tree->defs[i];
        if ((def->type != dt_funcdef || !((funcdef*)def)->defined) && def->type
!= dt_vardec)
        {
            throw(error("Error: linker only accepts defined functions as symbols
(internal error upstream)"));
        }
        if (def->type == dt_funcdef)
        {
            funcdef *fdef = (funcdef*)def;
            defined_funcs[fdef->name] = fdef;
        }
        definitions.push_back(def);
    }
}

// keep a dict of static vars.
// start at main, pass through the statements. Link in all the hardcoded funcs.
// increment the current machine index as you go.
std::vector<char> linker::link()
{
    if (defined_funcs.find("main") == defined_funcs.end())
    {
        throw(error("Error: no definition of function main."));
    }
    valtable.clear();

    // Allocate static storage for function arguments/return vectors, and global
variables:
    allocatefunctionstorage();
    for (unsigned int i = 0; i < definitions.size(); i++)
    {
        if (definitions[i]->type == dt_vardec)
        {
            vardeclaration *dec = (vardeclaration*)definitions[i];
            for (unsigned int i = 0; i < dec->vars.size(); i++)
            {
                vardeclaration::varpair &var = dec->vars[i];
                if (var.exported)
                    vars.registervar(var.name, var.type, var.exportvector);
                else
                    vars.addvar(var.name, var.type);
                if (var.type.type == type_array)
                    savelabel(var.name, vars.getvar(var.name)->address);
                exportvardeclaration(dec);
            }
        }
    }
```

```
    // set up pointer read/write instructions:
    // bfc0:    qqqq 'ff  bfc8 bfc8     ; clear target location q (q field
written by caller)
    // bfc8:    qqqq bff0 bfe0 bfe0     ; q = ~ bff0 (bff0 previously set to ~x;
q must be set here again.) Skip to jump instruction.
    // bfd0:    bff0 'ff  bfd8 bfd8     ; clear bff0
    // bfd8:    bff0 pppp bfe0 bfe0     ; set bff0 to ~p (p field written by
caller)
    // bfe0:    bff1 'ff  xxxx rrrr     ; jump to rrrr
    // To write:
    // we write ~val to bff0, jump to RAM.
    // in RAM we clear dest, write ~(~val) to dest, and then jump back to ROM
using the same return instruction as for reads.
    // To read:
    // we write to bfda (rrrr) to set the pointer read location
    // when we jump to bfd0, it clears bff0 and then reads ~*ptr into it.
    // bff1 is cleared - as the result is always 0, the machine jumps to the
return (rrrr) which was written beforehand.
    if (!compile_to_ram)
    {
        emit_writeconst_multiple(0x7d00, 0xbfc2, 2);
        emit_writeconst_multiple(0xbfc8bfc8, 0xbfc4, 4);
        emit_writeconst_multiple(0xbff0, 0xbfca, 2);
        emit_writeconst_multiple(0xbfe0bfe0, 0xbfcc, 4);
        emit_writeconst_multiple(0xbff07d00, 0xbfd0, 4);
        emit_writeconst_multiple(0xbfd8bfd8, 0xbfd4, 4);
        emit_writeconst_multiple(0xbff0, 0xbfd8, 2);
        emit_writeconst_multiple(0xbfe0bfe0, 0xbfdc, 4);
        emit_writeconst_multiple(0xbff17d00, 0xbfe0, 4);
    }


    // Link in the main function body
#ifdef EBUG
    std::cout << "Linking main\n";
#endif
    analysedependencies("main");
    vars.addvar(makeguid("__return", (long)defined_funcs["main"]), type_label);
    link(((funcdef*)defined_funcs["main"])->body);
    savelabel(makeguid("__return", (long)defined_funcs["main"]),
current_address);

    // generate halt instruction:
    emit_branchalways(current_address, true);

    // Link in the rest of the functions afterwards.
    if (strip_unused_functions)
        removeunusedfunctions();

    for(std::map<std::string, definition*>::iterator iter =
defined_funcs.begin(); iter != defined_funcs.end(); iter++)
    {
        if (!iter->second)  // skip it if it's hardcoded! (largely 'cause we
don't want null dereferencing.)
            continue;
        if (iter->second && iter->second->type == dt_funcdef && ((funcdef*)iter-
>second)->name != "main")
            link((funcdef*)iter->second);
    }
```

```
	// Finally, push all the strings onto the end.
	for (unsigned int stringnum = 0; stringnum < stringvalues.size(); 
stringnum++)
	{
		savelabel(stringvalues[stringnum].first, current_address);
		std::string &str = stringvalues[stringnum].second;
		for (unsigned int i = 0; i <= str.size(); i++)		// <= instead of < 
because we want to include the terminating zero.
			write8(str[i]);
	}
	savelabel("__program_end", current_address);
#ifdef EBUG
	std::cout << "__program_end: " << evaluate(linkval("__program_end")) << 
"\n";
#endif // EBUG
//#ifdef EBUG
	std::cout << "Executable size: " << std::dec << index << std::hex << " (0x" 
<< index << ") bytes.\n";
//#endif // EBUG

	return assemble();
}

void linker::link(funcdef* fdef)
{

#ifdef EBUG
	std::cout << "Linking function: " << fdef->name << ", @" << std::hex << 
current_address << "\n";
#endif
	vars.push_function_scope();
	if (!fdef->exported)
	{
		// Save the start label, link the body and save the end label.
		std::string returnstat_target = makeguid("__return", (long)fdef);
		vars.addvar(returnstat_target, type_label);
		savelabel(fdef->name + ":__startvector", current_address);
		link(fdef->body);
		savelabel(returnstat_target, current_address);
		emit_copy_multiple(vars.getvar(fdef->name + ":__returnvector")->address, 
				JUMP_PVECTOR, type_t(type_pointer).getsize());
		emit_branchalways(JUMP_INSTRUCTION);
	}
	else
	{
		// If this is an exported function existing else where, we just need to 
make public its location.
		savelabel(fdef->name + ":__startvector", fdef->exportvectors[0]);
	}
	exportfuncdef(fdef);
}

void linker::exportfuncdef(funcdef *fdef)
{
	// No export if hardcoded:
	if (defined_funcs[fdef->name] == 0)
		return;
	defstring << "export function " << fdef->return_type.getname() << " " << 
fdef->name << "(";
	if (fdef->args.size() > 0)
	{
```

```
        std::string fullname = fdef->args[0].name;
        defstring << fdef->args[0].type.getname() << " " << fullname.substr(0, 
fullname.find("@"));
        for (unsigned int i = 1; i < fdef->args.size(); i++)
        {
            std::string fullname = fdef->args[i].name;
            defstring << ", " << fdef->args[i].type.getname() << " " << 
fullname.substr(0, fullname.find("@"));
        }
    }
    // Export function information in the following order: call address, return 
value location, return vector, argument locations
    defstring << "): 0x";
    defstring << std::hex << ((valtable[fdef->name + 
":__startvector_HI"].literal << 8) + valtable[fdef->name + 
":__startvector_LO"].literal);
    defstring << ", 0x" << vars.getvar(fdef->name + ":__returnval")-
>address.literal;
    defstring << ", 0x" << vars.getvar(fdef->name + ":__returnvector")-
>address.literal;
    for (unsigned int i = 0; i < fdef->args.size(); i++)
        defstring << ", 0x" << vars.getvar(fdef->args[i].name)->address.literal;
    defstring << ";\r\n";
}

void linker::exportvardeclaration(vardeclaration *vardec)
{
    for (unsigned int i = 0; i < vardec->vars.size(); i++)
    {
        vardeclaration::varpair &var = vardec->vars[i];
        defstring << "export var ";
        std::string type_str = var.type.getname();
        std::string subscript = "";
        if (type_str.find('[') != std::string::npos)
        {
            int bracketpos = type_str.find('[');
            subscript = type_str.substr(bracketpos);
            type_str = type_str.substr(0, bracketpos);
        }
        std::string name = var.name;
        name = name.substr(0, name.find('@'));
        defstring << type_str << " " << name << subscript <<  ": ";
        defstring << "0x" << std::hex << vars.getvar(var.name)->address.literal 
<< ";\r\n";
    }
}


void linker::link(block *blk)
{
    for (std::vector<vardeclaration*>::iterator iter = blk-
>declarations.begin(); iter != blk->declarations.end(); iter++)
    {
        for (unsigned int i = 0; i < (*iter)->vars.size(); i++)
        {
            vardeclaration::varpair &var = (*iter)->vars[i];
            if (var.exported)
                vars.registervar(var.name, var.type, var.exportvector);
            else
                vars.addvar(var.name, var.type);
            if (var.type.type == type_array)
```

```
                savelabel(var.name, vars.getvar(var.name)->address);
            }
        }
        for (std::vector<statement*>::iterator iter = blk->statements.begin(); iter
!= blk->statements.end(); iter++)
        {
            link(*iter);
        }
        // TODO: make this a simple scope pop operation.
        for (std::vector<vardeclaration*>::iterator iter = blk-
>declarations.begin(); iter != blk->declarations.end(); iter++)
        {
            for (unsigned int i = 0; i < (*iter)->vars.size(); i++)
            {
                vars.remove((*iter)->vars[i].name);
            }
        }
}

void linker::link(statement *stat)
{
    switch(stat->type)
    {
    case stat_block:
        link((block*)stat);
        break;
    case stat_empty:
        break;
    case stat_call:
        link((funccall*)stat);
        break;
    case stat_goto:
        link((goto_stat*)stat);
        break;
    case stat_label:
        savelabel(((label*)stat)->name, current_address);
        break;
    case stat_if:
        link((if_stat*)stat);
        break;
    case stat_while:
        link((while_stat*)stat);
        break;
    case stat_assignment:
        link((assignment*)stat);
        break;
    default:
        {
            std::stringstream ss;
            ss << "Error: linking unrecognized statement type: " << stat->type;
            throw(error(ss.str()));
        }
        break;
    }
}

// For the special case of NFC, where inputs are treated as hard addresses so we
want the actual
// literal value instead of getconstaddress().
linkval linker::evaluate_or_return_literal(expression *expr)
{
```

```
    if (expr->type == exp_number)
        return expr->number;
    else
        return evaluate(expr);
}

void linker::link(funccall *call)
{
    if (defined_funcs.find(call->name) == defined_funcs.end())
    {
        std::stringstream ss;
        ss << "Error: no definition for function \"" << call->name << "\"";
        throw(error(ss.str()));
    }
    definition *def = defined_funcs[call->name];
    if (!def)   // it's a hardcoded function...
    {
        linkbuiltinfunction(call->args, call->name);
    }
    else
    {
        linkfunctioncall(call->args, (funcdef*)def);
    }
}

// passes in arguments, sets up return vector and jumps.
// returns: the location of the function returnval register.
linkval linker::linkfunctioncall(std::vector<expression*> &args, funcdef *fdef)
{
    // First: scan the arguments for immediate evaluations of functions that
ultimately depend on this one (start from 1 not 0 because evaluated before write
so no overwrite for 0)
    std::map<int, std::pair<linkval, int>> temps;
    for (unsigned int i = 1; i < args.size(); i++)
    {
        if (args[i]->type != exp_funccall)
            continue;
        funcdef *dependedfunc = (funcdef*)defined_funcs[args[i]->name];
        if (!dependedfunc)  // i.e. if builtin
            continue;
        if (fdef->dependson.find(fdef->name) != fdef->dependson.end());
        {
            type_t return_type = dependedfunc->return_type;
            linkval temp = vars.addvar("__fdep_temp", return_type);
            emit_copy_multiple(evaluate(args[i], true, temp), temp,
return_type.getsize());
            temps[i] = std::pair<linkval, int>(temp, return_type.getsize());
        }
    }
    // For each argument: if it is a constant then emit a constant copy, else
evaluate the expression and copy the result to the argloc.
    for (unsigned int i = 0; i < args.size(); i++)
    {
        if (args[i]->type == exp_number)
            emit_writeconst_multiple(args[i]->number, vars.getvar(fdef-
>args[i].name)->address, fdef->args[i].type.getsize());
        else if (temps.find(i) != temps.end())
        {
            emit_copy_multiple(temps[i].first, vars.getvar(fdef->args[i].name)-
>address, temps[i].second);
        }
```

```
        else
        {
            linkval argloc = vars.getvar(fdef->args[i].name)->address;
            // Pass argloc as the preferred location, store the value in
arg_val_loc: if argloc is successfully used then these linkvals are the same and
emit_copy_multiple() is a no-op.
            linkval arg_val_loc = evaluate(args[i], true, argloc);
            emit_copy_multiple(arg_val_loc, argloc, fdef-
>args[i].type.getsize());
        }
    }
    // Create a label name which refers to the first instruction after the call:
pass this to the call-writing code. Once the call code is written we then save
the actual location that label refers to.
    // (Avoids trying to guess the number of instructions: unique label acts as
an "IOU" for the location)
    std::string returnlabel = getlabel();
    // We write the location of the label to the function's return vector (even
if we don't know it yet - this gets sorted out at the final assembly), then jump
into the function.
    emit_writelabel(returnlabel, vars.getvar(fdef->name + ":__returnvector")-
>address);
    emit_branchalways(linkval(fdef->name + ":__startvector"));
    savelabel(returnlabel, current_address);
    return vars.getvar(fdef->name + ":__returnval")->address;
}

linkval linker::linkbuiltinfunction(std::vector<expression*> &args, std::string
name, bool givenpreferred, linkval preferred)
{
        if (name == "nfc")
        {
            vars.push_temp_scope();
            emit_nfc2(evaluate_or_return_literal(args[0]),
evaluate_or_return_literal(args[1]));
            vars.pop_temp_scope();
        }
        else if (name == "nfc4")
        {
            padto8bytes();
            vars.push_temp_scope();
            write16(evaluate_or_return_literal(args[0]));
            write16(evaluate_or_return_literal(args[1]));
            write16(evaluate_or_return_literal(args[2]));
            write16(evaluate_or_return_literal(args[3]));
            vars.pop_temp_scope();
        }
        else if (name == "val")
        {
            return getconstaddress(evaluate_or_return_literal(args[0]).literal);
        }
        else if (name == "first")
        {
            if (args[0]->type == exp_number)
            {
                return getconstaddress(args[0]->number >> 8);
            }
            else
            {
                vars.push_temp_scope(); // The scope is pushed partly to clean
up variables as soon as possible, but largely so that the evaluations take place
```

```
in a valid scope to avoid internal clobbering.
                linkval resultloc = evaluate(args[0]);
                vars.pop_temp_scope();
                if (givenpreferred)
                {
                    emit_copy(resultloc, preferred);
                    return preferred;
                }
                else
                {
                    linkval temploc = vars.addvar("__firsttemp", type_int);
                    emit_copy(resultloc, temploc);
                    vars.remove_on_pop("__firsttemp");
                    return temploc;
                }

            }
        }
        else if (name == "second")
        {
            if (args[0]->type == exp_number)
            {
                return getconstaddress(args[0]->number & 0xff);
            }
            else
            {
                vars.push_temp_scope();
                linkval resultloc = evaluate(args[0]) + 1;
                vars.pop_temp_scope();
                if (givenpreferred)
                {
                    emit_copy(resultloc, preferred);
                    return preferred;
                }
                else
                {
                    linkval temploc = vars.addvar("__secondtemp", type_int);
                    emit_copy(resultloc, temploc);
                    vars.remove_on_pop("__secondtemp");
                    return temploc;
                }
            }
        }
        else if (name == "pair")
        {
            linkval temploc = vars.addvar("__pairtemp", type_pointer);
            vars.push_temp_scope();
            if (givenpreferred)
            {
                emit_copy(evaluate(args[0], true, temploc), temploc);
// We still use temploc for the first byte, in case the expression depends upon
the pointer at preferredloc.
                emit_copy(evaluate(args[1], true, preferred + 1), preferred +
1);    // The second byte goes straight to where it needs to go, as there are no
further evaluations that may depend on it.
                emit_copy(temploc, preferred);
            }
            else
            {
                emit_copy(evaluate(args[0], true, temploc), temploc);
                emit_copy(evaluate(args[1], true, temploc + 1), temploc + 1);
```

```
            }

            vars.pop_temp_scope();
            vars.remove_on_pop("__pairtemp");
            return givenpreferred ? preferred : temploc;
        }
        else if (name == "write")
        {
            std::string returnlabel = getlabel();
            vars.push_temp_scope();
            emit_copy_inverted(evaluate(args[0], true, POINTER_WRITE_VALUE),
POINTER_WRITE_VALUE);
            linkval pointerpos = evaluate(args[1], true,
POINTER_WRITE_CLEAR_INSTRUCTION);
            vars.pop_temp_scope();
            emit_copy_multiple(pointerpos, POINTER_WRITE_CLEAR_INSTRUCTION,
type_t(type_pointer).getsize());
            emit_copy_multiple(pointerpos, POINTER_WRITE_COPY_INSTRUCTION,
type_t(type_pointer).getsize());
            emit_writelabel(returnlabel, JUMP_PVECTOR);
            emit_branchalways(POINTER_WRITE_CLEAR_INSTRUCTION);
            savelabel(returnlabel, current_address);
// URGH this is so inefficient :(
        }
        else if (name == "read" || name == "increment" || name == "decrement" ||
                 name == "shiftleft" || name == "shiftright")
        {
            std::string returnlabel = getlabel();
            if (name == "read")
            {
                if (args[0]->type == exp_number)
                    return args[0]->number; // this should ONLY get reached with
a call such as read(debugin). In this case, no need to worry about timing or
clobbering!
                vars.push_temp_scope();
                emit_copy_multiple(evaluate(args[0], true,
POINTER_READ_PVECTOR), POINTER_READ_PVECTOR, type_t(type_pointer).getsize());
                vars.pop_temp_scope();
            }
            else
            {
                if (name == "increment")
                    emit_writeconst(INCREMENT_START >> 8, POINTER_READ_PVECTOR);
                else if (name == "decrement")
                    emit_writeconst(DECREMENT_START >> 8, POINTER_READ_PVECTOR);
                else if (name == "shiftleft")
                    emit_writeconst(LEFTSHIFT_START >> 8, POINTER_READ_PVECTOR);
                else //if (name == "shiftright")
                    emit_writeconst(RIGHTSHIFT_START >> 8,
POINTER_READ_PVECTOR);
                if (args[0]->type == exp_number)
                    emit_writeconst(args[0]->number, POINTER_READ_PVECTOR + 1);
                else
                    emit_copy(evaluate(args[0], true, POINTER_READ_PVECTOR + 1),
POINTER_READ_PVECTOR + 1);
            }
            emit_writelabel(returnlabel, JUMP_PVECTOR);
            emit_branchalways(POINTER_READ_INSTRUCTION);
            savelabel(returnlabel, current_address);
            if (givenpreferred)
            {
```

```
                emit_copy_inverted(POINTER_READ_RESULT, preferred);
                return preferred;
            }
            else
            {
                emit_nfc2(POINTER_READ_RESULT, POINTER_READ_RESULT);    // NB
multiple functions are returning here!
                return POINTER_READ_RESULT;                             //
Careful of collisions.
            }
        }
        else if (name == "andnot")
        {
            linkval returnloc = givenpreferred ? preferred :
vars.addvar("__andnottemp", type_int);
            vars.push_temp_scope(); // For the arguments
            emit_copy_inverted(evaluate(args[0], true, returnloc), returnloc);
            emit_nfc2(returnloc, evaluate(args[1]));
            vars.pop_temp_scope();
            if (!givenpreferred)
                vars.remove_on_pop("__andnottemp");
            return returnloc;
        }
        else if (name == "and")     // the problem occurs when the argument
starts off in the finish location: it is destroyed by the invcopy to self
(destructive).
        {
            linkval returnloc = givenpreferred ? preferred :
vars.addvar("__andreturnloc", type_int);
            vars.push_temp_scope(); // For the arguments
            linkval temploc = vars.addvar("__andtemploc", type_int);
            emit_copy_inverted(evaluate(args[0], true, returnloc), returnloc);
            emit_copy_inverted(evaluate(args[1]), temploc);
            emit_nfc2(returnloc, temploc);
            vars.pop_temp_scope();
            if (!givenpreferred)
                vars.remove_on_pop("__andreturnloc");
            vars.remove_on_pop("__andtemploc");
            return returnloc;
        }
        else if (name == "not")
        {
            linkval returnloc = givenpreferred ? preferred :
vars.addvar("__notreturnloc", type_int);
            vars.push_temp_scope(); // For the arguments
            emit_copy_inverted(evaluate(args[0], true, returnloc), returnloc);
            vars.pop_temp_scope();
            if (!givenpreferred)
                vars.remove_on_pop("__notreturnloc");
            return returnloc;
        }
        else if (name == "or")
        {
            linkval returnloc = givenpreferred ? preferred :
vars.addvar("__orreturnloc", type_int);
            vars.push_temp_scope(); // For the arguments
            emit_copy(evaluate(args[0], true, returnloc), returnloc);
            emit_nfc2(returnloc, evaluate(args[1]));
            emit_nfc2(returnloc, returnloc);
            vars.pop_temp_scope();
            if (!givenpreferred)
```

```
                vars.remove_on_pop("__orreturnloc");
            return returnloc;
        }
        else if (name == "xor")
        {
            linkval returnloc = givenpreferred ? preferred : 
vars.addvar("__xorreturnloc", type_int);
            linkval temp1 = vars.addvar("__xortemp1", type_int);
            linkval temp2 = vars.addvar("__xortemp2", type_int);
            vars.push_temp_scope(); // For the arguments
            linkval argA = evaluate(args[0], true, temp1);
            linkval argB = evaluate(args[1], true, temp2);
            emit_copy_inverted(argA, temp1);
            emit_nfc2(temp1, argB);
            emit_copy_inverted(argB, temp2);
            emit_nfc2(temp2, argA);
            emit_copy(temp1, returnloc);   // we can't do away with temp1 
because if the second argument is the same loc as the preferred return, writing 
~the first argument to returnloc would have side effects on the second argument.
            emit_nfc2(returnloc, temp2);
            emit_nfc2(returnloc, returnloc);
            vars.pop_temp_scope();
            if (!givenpreferred)
                vars.remove_on_pop("__xorreturnloc");
            vars.remove_on_pop("__xortemp1");
            vars.remove_on_pop("__xortemp2");
            return returnloc;
        }
        else
        {
            throw(error("Linker Error: unknown builtin function " + name));
        }
        return 0;
}

void linker::link(goto_stat *sgoto)
{
    linkval temploc = vars.addvar("temp", type_int);
    write16(temploc);
    write16(temploc);
    linkval target = evaluate_or_return_literal(sgoto->target);
    write16(target);
    write16(target);
    vars.remove("temp");
}


//  BRNZ L1
//  <if block>
//  BRA L2
//L1:
//  <else block
//L2:
void linker::link(if_stat* ifs)
{
    vars.push_temp_scope();
    std::string elselabel = getlabel();
    std::string endlabel = getlabel();
    unsigned int address_before_evaluate = current_address;
    linkval testloc = linker::evaluate(ifs->expr);
    emit_branchifzero(testloc, linkval(elselabel) );//,current_address !=
```

```
address_before_evaluate);   // the top of an if-statement has no associated
label, so it's reasonable to not emit an instruction for the branch
    link(ifs->ifblock);
    if (ifs->elseblock)
    {
        // finished the if clause: nonconditional jump past else clause.
        emit_branchalways(linkval(endlabel), true);        // the same is not
true for the else jump, so always_emit is true.
        savelabel(elselabel, current_address);
        link(ifs->elseblock);
    }
    else
        savelabel(elselabel, current_address);
    savelabel(endlabel, current_address);
    vars.pop_temp_scope();
}

//L1:
//  <test>
//  BRZ L2
//  <body>
//  BRA L1
//L2:
void linker::link(while_stat *whiles)
{
    vars.push_temp_scope();
    std::string toplabel = makeguid("__top", (long)whiles->blk);
    std::string exitlabel = makeguid("__exit", (long)whiles->blk);
    vars.addvar(toplabel, type_label);
    vars.addvar(exitlabel, type_label);
    savelabel(toplabel, current_address);
    unsigned int address_before_evaluate = current_address;
    linkval result = evaluate(whiles->expr);
    emit_branchifzero(result, exitlabel  );//, current_address !=
address_before_evaluate);   // if we have generated code in the course of
evaluating the condition then we are free to use amend_previous (otherwise
possibly not!)
    link(whiles->blk);
    // jump unconditionally to top:
    emit_branchalways(linkval(toplabel), true);
    savelabel(exitlabel, current_address);
    vars.remove(toplabel);
    vars.remove(exitlabel);
    vars.pop_temp_scope();
}

void linker::link(assignment *assg)
{
    vars.push_temp_scope();
    expression targetexp;
    targetexp.type = exp_name;
    targetexp.name = assg->name;
    if (assg->indexed)
    {
        targetexp.indexed = true;
        targetexp.number = assg->index;
    }
    linkval target = evaluate(&targetexp);
    if (assg->expr->type == exp_number)
    {
        emit_writeconst_multiple(assg->expr->number, target, assg->expr-
```

```
>val_type.getsize());
    }
    else
    {
        type_t type;
        if (assg->expr->indexed)
            type = assg->expr->val_type.second;
        else
            type = assg->expr->val_type;
        // Pass in the target as the preferred location: if resultloc and target
match, no actual copy will be emitted.
        emit_copy_multiple(evaluate(assg->expr, true, target), target,
type.getsize());
    }
    vars.pop_temp_scope();
}


// Can return a literal value or a symbol: this is transparent to using
functions.
// (think of a symbol as an "IOU" for an actual address: e.g. for labels we
don't know the address til we reach them.)
// Always returns an address: if we need code to calculate the value (e.g. a
function call) then this function emits that code
// and then returns the location where the value will be found.
// Caller can pass in a "preferred" destination, e.g. when performing
assignments - if possible, result will end up there.
linkval linker::evaluate(expression *expr, bool givenpreferred, linkval
preferred)
{
    if (expr->type == exp_number)
    {
        if (expr->val_type == type_int)
            return getconstaddress(expr->number);        // NOT the literal
itself. If you want the actual literal (e.g. with NFC) then fetch it directly,
as this is an oddball case. (SEE evaluate_or_return_literal())
        else
        {
            linkval constloc = givenpreferred? preferred :
vars.addvar("__consttemp", expr->val_type);
            emit_writeconst_multiple(expr->number, constloc, expr-
>val_type.getsize());
            if (!givenpreferred)
                vars.remove_on_pop("__consttemp");
            return constloc;
        }
    }
    else if (expr->type == exp_name)
    {
        if (vars.exists(expr->name))
        {
            variable *var = vars.getvar(expr->name);
            if (var->type == type_label)
            {
                // eager evaluation of labels: we get the most recently bound
label identity,
                // not the last one. This is what allows us to change argname
meaning in different calls to a macro.
                if (valtable.find(expr->name) != valtable.end())
                    return valtable[expr->name];
                else
```

```
                    return expr->name;
                }
                else if (var->type.type == type_array)
                {
                    if (expr->indexed)
                        return var->address + expr->number;
                    linkval addressloc = givenpreferred? preferred :
vars.addvar("__arraytemp", type_pointer);
                    emit_writelabel(expr->name, addressloc);
                    if (!givenpreferred)
                        vars.remove_on_pop("__arraytemp");
                    return addressloc;
                }
                else
                    return var->address;
            }
            else
            {
                std::stringstream ss;
                ss << "Error: unknown linker symbol \"" << expr->name << "\"";
                throw(error(ss.str()));
            }
        }
        else if (expr->type == exp_funccall)
        {
            if (!defined_funcs[expr->name])
            {
                return linkbuiltinfunction(expr->args, expr->name, givenpreferred,
preferred);
            }
            else
            {
                if (defined_funcs[expr->name]->type != dt_funcdef)
                    throw(error("Error: only functions can return values (call to "
+ expr->name + ")"));
                return linkfunctioncall(expr->args, (funcdef*)defined_funcs[expr-
>name]);
            }
        }
        else if (expr->type == exp_string)
        {
            std::string stringlocation = getlabel();
            stringvalues.push_back(std::pair<std::string,
std::string>(stringlocation, expr->name));    // the actual string will get
linked in later, tacked onto the end of the program.
            if (givenpreferred)
            {
                emit_writelabel(stringlocation, preferred);
                return preferred;
            }
            else
            {
                linkval temploc = vars.addvar("__stringloctemp", type_pointer);
                emit_writelabel(stringlocation, temploc);
                vars.remove_on_pop("__stringloctemp");
                return temploc;
            }
        }
        else if (expr->type == exp_not)
        {
            linkval return_loc = givenpreferred ? preferred :
```

```
vars.addvar("__lnottemp", type_int);
        std::string skiplabel = getlabel();
        emit_nfc2(return_loc, getconstaddress(0xff));
        unsigned int address_before_evaluate = current_address;
        linkval result = evaluate(expr->args[0]);
        emit_branchifnonzero(result, skiplabel, false && current_address !=
address_before_evaluate); // use option amend_previous to avoid spurious copy of
expression result if there has been code generated.
        emit_nfc2(return_loc, getconstaddress(~1));
        savelabel(skiplabel, current_address);
        if(!givenpreferred)
            vars.remove_on_pop("__lnottemp");
        return return_loc;
    }
    else if (expr->type == exp_and)
    {
        linkval return_loc = givenpreferred ? preferred :
vars.addvar("__landtemp", type_int);
        std::string skiplabel = getlabel();
        unsigned int address_before_evaluate = current_address;
        emit_copy(evaluate(expr->args[0], true, return_loc), return_loc);
        emit_branchifzero(return_loc, skiplabel, false && current_address !=
address_before_evaluate);
        emit_copy(evaluate(expr->args[1], true, return_loc), return_loc);
        savelabel(skiplabel, current_address);
        if (!givenpreferred)
            vars.remove_on_pop("__landtemp");
        return return_loc;
    }
    else if (expr->type == exp_or)
    {
        linkval return_loc = givenpreferred ? preferred :
vars.addvar("__lortemp", type_int);
        std::string skiplabel = getlabel();
        unsigned int address_before_evaluate = current_address;
        emit_copy(evaluate(expr->args[0], true, return_loc), return_loc);
        emit_branchifnonzero(return_loc, skiplabel, false && current_address !=
address_before_evaluate);
        emit_copy(evaluate(expr->args[1], true, return_loc), return_loc);
        savelabel(skiplabel, current_address);
        if (!givenpreferred)
            vars.remove_on_pop("__lortemp");
        return return_loc;
    }
    else
    {
        throw(error("Error: linking unknown expression type"));
    }
    return 0;
}


std::vector<char> linker::assemble()
{
    // assemble literals and symbols into ROM image:
    std::vector<char> image;
    for (std::vector<linkval>::iterator iter = buffer.begin(); iter !=
buffer.end(); iter++)
    {
        image.push_back(evaluate(*iter));
    }
```

```
    if (image.size() > (unsigned)INCREMENT_START)
        throw(error("Error: program is too big!"));
    // pad and put constant tables into last kilobyte.
    if (!compile_to_ram)
    {
        while (image.size() < (unsigned)INCREMENT_START)
            image.push_back(0);
        for (unsigned int i = 0; i <= 255; i++)
            image.push_back((i + 1) & 0xff);
        for (unsigned int i = 0; i <= 255; i++)
            image.push_back((i - 1) & 0xff);
        for (unsigned int i = 0; i <= 255; i++)
            image.push_back((i << 1) & 0xff);
        for (unsigned int i = 0; i <= 255; i++)
            image.push_back((i >> 1) & 0xff);
    }
    return image;
}


// For each function, allocate the following statically:
// - a return value address, of a size matching the return type. The retval is
found here after execution.
// - a return vector address: the caller writes its next address here before
calling, and the callee jumps to this address.
// - an appropriately-sized variable for each argument (pass-by-value).
void linker::allocatefunctionstorage()
{
    for (std::map<std::string, definition*>::iterator iter =
defined_funcs.begin(); iter != defined_funcs.end(); iter++)
    {
#ifdef EBUG
        std::cout << "Allocating for " << iter->first << "\n";
#endif
        if (iter->second && iter->second->type == dt_funcdef)
        {
            funcdef *fdef = (funcdef*)(iter->second);
            if (fdef->name == "main")
                continue;
            if (!fdef->exported)
            {
                vars.addvar(fdef->name + ":__returnval", fdef->return_type);
                vars.addvar(fdef->name + ":__returnvector", type_pointer);
            }
            else
            {
                vars.registervar(fdef->name + ":__returnval", fdef->return_type,
fdef->exportvectors[1]);
                vars.registervar(fdef->name + ":__returnvector", type_pointer,
fdef->exportvectors[2]);
            }
            for (unsigned int argnum = 0; argnum < fdef->args.size(); argnum++)
            {
                if (!fdef->exported)
                    vars.addvar(fdef->args[argnum].name, fdef-
>args[argnum].type);  // name is already resolved to a global symbol by
compiler.
                else
                    vars.registervar(fdef->args[argnum].name, fdef-
>args[argnum].type, fdef->exportvectors[argnum + 3]);
            }
```

```
        }
    }
}

void linker::removeunusedfunctions()
{
    std::map<std::string, definition*>::iterator fiter;
    bool noincrement = false;
    for (fiter = defined_funcs.begin(); fiter != defined_funcs.end(); fiter++)
    {
        if (noincrement)
        {
            fiter--;
            noincrement = false;
        }
        funcdef *def = (funcdef*)fiter->second;
        if (!def || def->type != dt_funcdef)
            continue;   // ignore the builtin functions.
        if (!def->is_used)
        {
#ifdef EBUG
            std::cout << "erasing function " << fiter->first << "\n";
#endif // EBUG
            defined_funcs.erase(fiter++);
            if (fiter == defined_funcs.begin())
                noincrement = true;
            else
                fiter--;
        }
        else
            std::cout << "function " << fiter->first << " is used\n";
    }
}

std::set<std::string> linker::analysedependencies(std::string rootfunc)
{
    #ifdef EBUG
    std::cout << "Finding dependencies for " << rootfunc << "\n";
    #endif
    funcdef *rootdef = (funcdef*)defined_funcs[rootfunc];
    rootdef->is_used = true;
    std::set<std::string> &dependencies = rootdef->dependson;
    #ifdef EBUG
    std::cout << " - Depends on";
    for (std::set<std::string>::iterator i = dependencies.begin(); i !=
dependencies.end(); i++)
        std::cout << " " << *i;
    std::cout << ".\n";
    #endif // EBUG

    for (std::set<std::string>::iterator i = dependencies.begin(); i !=
dependencies.end(); i++)
    {
        funcdef *def = (funcdef*)defined_funcs[*i];
        if (!def || def->type != dt_funcdef)
            continue;
        if (!def->is_used)
        {
            // analyse each dependency's sub-dependencies recursively, and add
them to this function's
            // dependencies, so we know all dependencies of a function (not just
```

```
sub dependencies) (this also marks used functions)
            std::set<std::string> nextleveldeps = analysedependencies(def-
>name);
            for (std::set<std::string>::iterator iter = nextleveldeps.begin();
iter != nextleveldeps.end(); iter++)
                dependencies.insert(*iter);
        }
    }
    return dependencies;
}

std::string linker::getdefstring()
{
    return defstring.str();
}

void linker::setcompiletoram(bool ram)
{
    compile_to_ram = ram;
    if (compile_to_ram)
        current_address = index + HEAP_BOTTOM;
    else
        current_address = index;
    vars.start_from_top = !compile_to_ram;
}

uint16_t linker::evaluate(linkval lv)
{
    switch (lv.type)
    {
    case lv_literal:
        return lv.literal;
    case lv_symbol:
        if (valtable.find(lv.sym) == valtable.end())
            throw(error("Linker error: no such symbol: " + lv.sym));
        else
            return evaluate(valtable[lv.sym]);
    case lv_expression:
        {
            linkval lv1 = *lv.argA;
            uint16_t first_operand = evaluate(lv1);
            uint16_t second_operand = 0;
            if (lv.argB)
                second_operand = evaluate(*lv.argB);
            switch (lv.operation)
            {
            case linkval::op_add:
                return first_operand + second_operand;
            case linkval::op_sub:
                return first_operand - second_operand;
            case linkval::op_gethigh:
                return first_operand >> 8;
            case linkval::op_getlow:
                return first_operand & 0xff;
            }
        }
    }
    return 0;
}
```

printtree.h

```
#ifndef PRINTTREE_H_INCLUDED
#define PRINTTREE_H_INCLUDED

#include "syntaxtree.h"

void printtree(program *prog, int indentation = 0);
void printtree(definition *def, int indentation);
void printtree(funcdef *def, int indentation);
void printtree(macrodef *def, int indentation);
void printtree(constdef *def, int indentation);
void printtree(block *blk, int indentation);
void printtree(vardeclaration *decl, int indentation);
void printtree(statement *stat, int indentation);
void printtree(expression *expr);

#endif // PRINTTREE_H_INCLUDED
```

printtree.cpp

```
#include "printtree.h"

#include <iostream>

void indent(int n)
{
    for (int i = 0; i < n; i++)
    {
        std::cout << "  ";
    }
}

std::string typenames[] =
{
    "void",
    "int",
    "pointer"
};

void printtree(program *prog, int indentation)
{
    for (unsigned int i = 0; i < prog->defs.size(); i++)
    {
        printtree(prog->defs[i], 0);
    }
}

void printtree(definition *def, int indentation)
{
    switch(def->type)
    {
        case dt_constdef:
            printtree((constdef*)def, indentation);
            break;
        case dt_funcdef:
            printtree((funcdef*)def, indentation);
            break;
        case dt_macrodef:
            printtree((macrodef*)def, indentation);
            break;
```

```
        case dt_vardec:
            printtree((vardeclaration*)def, indentation);
            break;
    }
}

void printtree(constdef *def, int indentation)
{
    indent(indentation);
    std::cout << "constant " << def->valtype.getname() << " " << def->name << ":
" << def->value << "\n";
}

void printtree(funcdef *def, int indentation)
{
    indent(indentation);
    std::cout << "\nDefinition of function " << def->name << ": (";
    if (def->args.size() == 0)
        std::cout << "void";
    else
        for (unsigned int i = 0; i < def->args.size(); i++)
        {
            std::cout << def->args[i].type.getname();
            if (i < def->args.size() - 1)
                std::cout << ", ";
        }
    std::cout << ") => " << def->return_type.getname() << ":\n";
    indent(indentation);
    std::cout << " Depends on:\n";
    for (std::set<std::string>::iterator iter = def->dependson.begin(); iter !=
def->dependson.end(); iter++)
    {
        indent(indentation);
        std::cout << "   " << (*iter) << "\n";
    }
    if (def->exported)
    {
        indent(indentation + 1);
        std::cout << "(exported)\n";
    }
    else if (def->defined)
    {
        printtree(def->body, indentation);
    }
    else
    {
        indent(indentation + 1);
        std::cout << "(declaration only)\n";
    }
}

void printtree(macrodef *def, int indentation)
{
    indent(indentation);
    std::cout << "\nDefinition of macro " << def->name << ": (";
    if (def->args.size() == 0)
    {
        std::cout << "<none>";
    }
    else
    {
```

```
        for (unsigned int i = 0; i < def->args.size(); i++)
        {
            std::cout << def->args[i];
            if (i < def->args.size() - 1)
                std::cout << ", ";
        }
    }
    std::cout << "):\n";
    printtree(def->body, indentation);
}

void printtree(block *blk, int indentation)
{
    indent(indentation);
    std::cout << "{\n";
    for (std::vector<vardeclaration*>::iterator iter = blk-
>declarations.begin(); iter != blk->declarations.end(); iter++)
    {
        printtree(*iter, indentation + 1);
    }
    for (std::vector<statement*>::iterator iter = blk->statements.begin(); iter
!= blk->statements.end(); iter++)
    {
        printtree(*iter, indentation + 1);
    }
    indent(indentation);
    std::cout << "}\n";
}

void printtree(vardeclaration *decl, int indentation)
{
    for (unsigned int i = 0; i < decl->vars.size(); i++)
    {
        indent(indentation);
        std::cout << "Declared " << decl->vars[i].type.getname() << " " << decl-
>vars[i].name << "\n";
    }

}

void printtree(statement *stat, int indentation)
{
    indent(indentation);
    if (stat->type == stat_block)
        printtree((block*)stat, indentation);
    if (stat->type == stat_call)
    {
        funccall *fcall = (funccall*)stat;
        std::cout << "Call to function " << fcall->name << "\n";
        for (unsigned int i = 0; i < fcall->args.size(); i++)
        {
            indent(indentation + 1);
            std::cout << "Argument: ";
            printtree(fcall->args[i]);
            std::cout << "\n";
        }
    }
    else if (stat->type == stat_goto)
    {
        goto_stat *sgoto = (goto_stat*)stat;
        std::cout << "Goto ";
```

```
        printtree(sgoto->target);
        std::cout << "\n";
    }
    else if (stat->type == stat_label)
    {
        std::cout << "Label: " << ((label*)stat)->name << "\n";
    }
    else if (stat->type == stat_if)
    {
        std::cout << "If ";
        printtree(((if_stat*)stat)->expr);
        std::cout << " Then\n";
        printtree(((if_stat*)stat)->ifblock, indentation);
        if (((if_stat*)stat)->elseblock)
        {
            indent(indentation);
            std::cout << "Else\n";
            printtree(((if_stat*)stat)->elseblock, indentation);
        }
    }
    else if (stat->type == stat_while)
    {
        std::cout << "While ";
        printtree(((while_stat*)stat)->expr);
        std::cout << " Do\n";
        printtree(((while_stat*)stat)->blk, indentation);
    }
    else if (stat->type == stat_assignment)
    {
        assignment *assg = (assignment*)stat;
        std::cout << "Setting " << assg->name << " to ";
        printtree(assg->expr);
        std::cout << "\n";
    }
    else if (stat->type == stat_break)
    {
        std::cout << "Break\n";
    }
    else if (stat->type == stat_continue)
    {
        std::cout << "Continue\n";
    }
    else if (stat->type == stat_return)
    {
        std::cout << "Return\n";
    }


}

void printtree(expression *expr)
{
    if (expr->type == exp_name)
        std::cout << expr->name;
    else if (expr->type == exp_number)
        std::cout << expr->number;
    else if (expr->type == exp_not)
    {
        std::cout << "!(";
        printtree(expr->args[0]);
        std::cout << ")";
```

```
    }
    else if (expr->type == exp_or)
    {
        std::cout << "(";
        printtree(expr->args[0]);
        std::cout << " || ";
        printtree(expr->args[1]);
        std::cout << ")";
    }
    else if (expr->type == exp_and)
    {
        std::cout << "(";
        printtree(expr->args[0]);
        std::cout << " && ";
        printtree(expr->args[1]);
        std::cout << ")";
    }
    else if (expr->type == exp_funccall)
    {
        std::cout << expr->name << "(";
        for (unsigned i = 0; i < expr->args.size(); i++)
        {
            printtree(expr->args[i]);
            if (i < expr->args.size() - 1)
                std::cout << ", ";
        }
        std::cout << ")";
    }
    else if (expr->type == exp_string)
    {
        std::cout << "\"" << expr->name << "\"";
    }

}
```

main.cpp

```
#ifdef _WIN32
    #include <direct.h>
    #define getcwd _getcwd
    #define FILE_SEP_CHAR "\\"
#else
    #include <unistd.h>
    #define FILE_SEP_CHAR "/"
#endif
#include <fstream>
#include <iomanip>
#include <iostream>
#include "tokenizer.h"
#include "parser.h"
#include "printtree.h"
#include "compiler.h"
#include "linker.h"


void printout(std::vector<char> buffer, bool printasbytes = true)
{
    int nconsecutivezeroes = 0;
    for (unsigned int i = 0; i < buffer.size(); i++)
    {
```

```
        if (i % 8 == 0)
            std::cout << std::hex << std::setw(4) << std::setfill('0') << i <<
":\t";
        std::cout << "0x" <<  std::hex << std::setw(2) << std::setfill('0') <<
(((int)buffer[i]) & 0xff);
        if (i % 8 == 7)
            std::cout << ",\n";
        else if (printasbytes || i % 2 == 1)
            std::cout << ", ";

        if (buffer[i] == 0)
            nconsecutivezeroes++;
        else
            nconsecutivezeroes = 0;
        if (nconsecutivezeroes >= 8)
            break;
    }
}

int imax(int a, int b)
{
    return a > b ? a : b;
}

int main(int argc, char **argv)
{
    std::string usage = "Usage: spoon [-s] (inputfile) (outputfile)\n";
    std::string ifilename, ofilename;
    bool have_ifilename = false, have_ofilename = false;
    bool strip_unused_functions = false;
    bool compile_to_ram = false;
    bool export_symbols = false;
    // Don't know why but it doesn't work on Linux without this code...
    /*for (int i = 1; i < argc; i++)
        argv[i][0] = 1 + (--argv[i][0]); // it's a no-op*/
    try
    {
        for (int i = 1; i < argc; i++)
        {
            if (argv[i][0] == '-')
            {
                switch (argv[i][1])
                {
                case 's':
                    #ifdef EBUG
                    std::cout << "received option strip\n";
                    #endif
                    strip_unused_functions = true;
                    break;
                case 'r':
                    #ifdef EBUG
                    std::cout << "received option compile-to-ram\n";
                    #endif
                    compile_to_ram = true;
                    break;
                case 'e':
                    #ifdef EBUG
                    std::cout << "Received option export\n";
                    #endif // EBUG
                    export_symbols = true;
                    break;
```

```
                default:
                    throw(error(usage));
                }
            }
            else
            {
                if (!have_ifilename)
                {
                    ifilename = argv[i];
                    have_ifilename = true;
                }
                else if (!have_ofilename)
                {
                    ofilename = argv[i];
                    have_ofilename = true;
                }
                else
                {
                    throw(error(usage));
                }
            }
        }
        if (!(have_ifilename && have_ofilename))
            throw(error(usage));
        std::string ifiledirectory;
        if (ifilename.substr(0, 1) == "/" || ifilename.substr(1, 1) == ":")
            ifiledirectory = ifilename.substr(0, imax(ifilename.rfind("/"),
ifilename.rfind("\\"))) + FILE_SEP_CHAR;
        else
        {
            char *buffer = new char[1024];
            getcwd(buffer, 1024);
            ifiledirectory = std::string(buffer) + FILE_SEP_CHAR +
ifilename.substr(0, imax(ifilename.rfind("/"), ifilename.rfind("\\"))) +
FILE_SEP_CHAR;
            delete buffer;
        }
        //std::cout << "Working directory: " << ifiledirectory << "\n";
        std::fstream sourcefile(ifilename, std::ios::in | std::ios::binary);
        if (!sourcefile.is_open())
        {
            throw(error(std::string("Error: could not open file \"") + ifilename
+ "\""));
        }
        sourcefile.seekg(0, std::ios::end);
        int sourcelength = sourcefile.tellg();
        sourcefile.seekg(0, std::ios::beg);
        std::vector<char> source(sourcelength);
        sourcefile.read(&source[0], sourcelength);
        source.push_back(0);
        sourcefile.close();
        std::vector<token> tokens = tokenize(&source[0]);
        parser p(tokens, ifilename, ifiledirectory);
        program *prog = p.getprogram();
        //printtree(prog);
        compiler c;
        object *obj = c.compile(prog);
#ifdef EBUG
        std::cout << "\n\nPost-compile tree:\n\n";
        //printtree(obj->tree);
#endif // EBUG
```

```
        linker l;
        l.strip_unused_functions = strip_unused_functions;
        l.setcompiletoram(compile_to_ram);
        l.add_object(obj);
        std::vector<char> machinecode = l.link();
#ifdef EBUG
        //printout(machinecode);
#endif
        std::fstream outfile(ofilename, std::ios::out | std::ios::binary);
        if (!outfile.is_open())
            throw(error(std::string("Error: could not open file ") +
ofilename));
        for (unsigned int i = 0; i < machinecode.size(); i++)
            outfile.put(machinecode[i]);
        outfile.close();
        if (export_symbols)
        {
            std::fstream deffile(ofilename + ".def", std::ios::out |
std::ios::binary);
            deffile << l.getdefstring();
            deffile.close();
        }
    }
    catch (error e)
    {
        std::cout << e.errstring << "\n";
        return 1;
    }
    return 0;
}
```

Appendix 2: Standard Library

stddefs

```
// stddefs
// Standard Spoon definitions header
// Luke Wren 2013

const pointer debugout = 0xc000;
const pointer debugin = 0xc001;
const pointer screenout = 0xc000;

const int true = 0xff;
const int false = 0x00;

function int first(pointer p);
function int second(pointer p);
function pointer val(int value);
function pointer pair(int a, int b);

function void write(int value, pointer dest);
function int read(pointer src);
function int increment(int x);
function int decrement(int x);
function int shiftleft(int x);
function int shiftright(int x);
```

```
function int andnot(int a, int b);
function int and(int a, int b);
function int not(int x);
function int or(int a, int b);
function int xor(int a, int b);

macro output(src, dest)
{
    var int temp;
    nfc(temp, val(0xff));
    nfc(temp, src);
    nfc(dest, temp);
}

macro nop()
{
    var int dontcare;
    nfc(dontcare, dontcare);
}
```

stdmath

```
// stdmath
// Standard Spoon maths function header
// Luke Wren 2013

#include "stddefs"

// 8 bit

function int add(int a, int b)
{
    var int carry, temp;
    add = xor(a, b);
    carry = shiftleft(and(a, b));
    while (carry)
    {
        temp = add;
        add = xor(add, carry);
        carry = shiftleft(and(temp, carry));
    }
}

function int subtract(int a, int b)
{
    b = increment(not(b));    // two's complement negate
    subtract = add(a, b);
}

function int multiply(int a, int b)
{
    multiply = 0;
    while (b)
    {
        multiply = add(multiply, a);
        b = decrement(b);
    }
}
```

```
function int lessthan(int a, int b);

function int divide(int a, int b)
{
    divide = 0;
    if (!b)
      return;
    while (not(lessthan(a, b)))
    {
        a = subtract(a, b);
        divide = increment(divide);
    }
}

// non-zero if a < b:
// if the top bit is different and only true for b, a must be < b.
function int lessthan(int a, int b)
{
    lessthan = 0;
    var int difference = xor(a, b);
    while (difference)
    {
        if (and(difference, 0x80))
        {
            lessthan = and(b, 0x80);
            return;
        }
        difference = shiftleft(difference);
        a = shiftleft(a);
        b = shiftleft(b);
    }
}

function int equal(int a, int b)
{
    if (xor(a, b))
        equal = 0;
    else
        equal = 1;
}

function int max(int a, int b)
{
    if (lessthan(a, b))
        max = b;
    else
        max = a;
}

function int min(int a, int b)
{
    if (lessthan(a, b))
        min = a;
    else
        min = b;
}

function int shiftrightn(int a, int n)
{
    while (n)
    {
```

```
        a = shiftright(a);
        n = decrement(n);
    }
    shiftrightn = a;
}

function int shiftleftn(int a, int n)
{
    while (n)
    {
        a = shiftleft(a);
        n = decrement(n);
    }
    shiftleftn = a;
}

// 16 bit

function pointer incrementPointer(pointer p)
{
    var int a, b;
    b = increment(second(p));
    if (b)
        a = first(p);
    else
        a = increment(first(p));
    incrementPointer = pair(a, b);
}

function pointer decrementPointer(pointer p)
{
    var int a, b;
    if (second(p))
        a = first(p);
    else
        a = decrement(first(p));
    b = decrement(second(p));
    decrementPointer = pair(a, b);
}

function pointer addPointer(pointer p, int offset)
{
    var int bottom;
    bottom = add(second(p), offset);
    if (lessthan(bottom, second(p)))
        addPointer = pair(increment(first(p)), bottom);
    else
        addPointer = pair(first(p), bottom);
}

function pointer subtractPointer(pointer p, int offset)
{
    var int bottom;
    bottom = subtract(second(p), offset);
    if (lessthan(second(p), offset))
        subtractPointer = pair(decrement(first(p)), bottom);
    else
        subtractPointer = pair(first(p), bottom);
}

function int lessthanPointer(pointer a, pointer b)
```

```
{
    if (lessthan(first(a), first(b)))
        lessthanPointer = 1;
    else if (equal(first(a), first(b)))
        lessthanPointer = lessthan(second(a), second(b));
    else
        lessthanPointer = 0;
}

function int16 add16(int16 a, int16 b)
{
    add16 = addPointer(a, second(b));
    add16 = pair(add(first(add16), first(b)), second(add16));
}

function int16 subtract16(int16 a, int16 b)
{
    subtract16 = subtractPointer(a, second(b));
    subtract16 = pair(subtract(first(subtract16), first(b)),
second(subtract16));
}
```

stdmem

```
// stdmem
// Standard Spoon memory manipulation header
// Luke Wren 2013

#include "stddefs"
#include "stdmath"

function memset(pointer dest, int value, int16 count)
{
    while (first(count) || second(count))
    {
        write(value, dest);
        count = decrementPointer(count);
        dest = incrementPointer(dest);
    }
}

function pointer memcpy(pointer dest, pointer src, int count)
{
    while (count)
    {
        var int x = read(src);
        write(x, dest);
        dest = incrementPointer(dest);
        src = incrementPointer(src);
        count = decrement(count);
    }
    memcpy = dest;
}

// Copies a null-terminated string to dest, and returns the location of the byte
after the last one written.
function pointer strcpy(pointer dest, pointer src)
{
    var char c;
    c = read(src);
```

```
    while (c)
    {
        write(c, dest);
        dest = incrementPointer(dest);
        src = incrementPointer(src);
        c = read(src);
    }
    strcpy = dest;
}

function int16 strlen(pointer src)
{
    strlen = 0;
    var char c = read(src);
    while (c)
    {
        strlen = incrementPointer(strlen);
        src = incrementPointer(src);
        c = read(src);
    }
}

function int strcmp(pointer sa, pointer sb)
{
    strcmp = 0;
    var char ca = read(sa);
    var char cb = read(sb);
    while (ca && cb)
    {
        if (!equal(ca, cb))
        {
            strcmp = 1;
            return;
        }
        sa = incrementPointer(sa);
        sb = incrementPointer(sb);
        ca = read(sa);
        cb = read(sb);
    }
    if (ca || cb)
        strcmp = 1;
}

function int strequal(pointer stra, pointer strb)
{
    strequal = !strcmp(stra, strb);
}

function int instr(pointer str, char tofind)
{
    var char c = read(str);
    instr = 0;
    while (c && xor(c, tofind))
    {
        str = incrementPointer(str);
        instr = increment(instr);
        c = read(str);
    }
}
```

stdscreen

```
// stdscreen
// Standard Spoon screen driver header
// Luke Wren 2013

#include "stddefs"
#include "stdmath"
#include "stdmem"

const int s_enable = 0x80;
const int s_rs = 0x40;

var int screensignal;
var int screenbuffer[80];

function sleep(int time)
{
    while (time)
        time = decrement(time);
}

function pointer int2hexstr(int x)
{
    var pointer hexchars = "0123456789ABCDEF";
    var char buffer[3];
    buffer[0] = read(addPointer(hexchars, shiftrightn(x, 4)));
    buffer[1] = read(addPointer(hexchars, and(x, 0x0f)));
    buffer[2] = 0;
    int2hexstr = buffer;
}

// write data and pulse the clock:
function write4bits(int data)
{
    screensignal = or(and(screensignal, 0xf0), data);
    output(screensignal, debugout);
    output(or(screensignal, s_enable), debugout);
    output(screensignal, debugout);
}

function write8bits(int data)
{
    write4bits(shiftrightn(data, 4));
    write4bits(and(data, 0x0f));
}

function cursorpos(int x, int y)
{
    screensignal = 0x00;
    var int address;
    if (and(y, 0x02))
    {
        if (and(y, 0x01))
            address = 84;
        else
            address = 20;
    }
    else
    {
        if (and(y, 0x01))
```

```
            address = 64;
        else
            address = 0;
    }
    write8bits(or(add(address, x), 0x80));
    screensignal = s_rs;
}

function dumpscreenbuffer()
{
    var int ycount = 4, y = 0;
    var pointer p = screenbuffer;
    while (ycount)
    {
        cursorpos(0, y);
        var int xcount = 20;
        while (xcount)
        {
            write8bits(read(p));
            p = incrementPointer(p);
            xcount = decrement(xcount);
        }
        y = increment(y);
        ycount = decrement(ycount);
    }
}

function emptyscreenbuffer()
{
    memset(screenbuffer, ' ', 80);
}

function scrollscreen()
{
    memcpy(screenbuffer, addPointer(screenbuffer, 20), 60);
    memset(addPointer(screenbuffer, 60), ' ', 20);
}

function printtobuffer(pointer text, int pos)
{
    var pointer dest = addPointer(screenbuffer, pos);
    var int c = read(text);
    while (c)
    {
        write(c, dest);
        text = incrementPointer(text);
        c = read(text);
        dest = incrementPointer(dest);
    }
}

function println(pointer text)
{
    scrollscreen();
    printtobuffer(text, 60);
}

function printlineanddump(pointer text)
{
    printline(text);
    dumpscreenbuffer();
```

```
}

function wakescreen()
{
    screensignal = 0x00;
    write4bits(0x3);
    sleep(255);
    write4bits(0x3);
    sleep(128);
    write4bits(0x3);
    write4bits(0x2);
    write8bits(0x28);
    write8bits(0x0c);
    write8bits(0x01);
    sleep (128);
    screensignal = s_rs;
}
```

Appendix 3: Operating System Listing

forkos.spn (Main Listing)

```
#include "stdscreen"

const pointer flash_sig = 0xc002;
const pointer flash_mid = 0xc003;
const pointer flash_low = 0xc004;
const pointer flash_in = 0xc005;
const pointer flash_out = 0xc006;
const pointer kbd_in = 0xc007;

const int flashsig_oe = 0x40;
const int flashsig_we = 0x80;
const int flashsig_oe_we = 0xc0;

var int flash_signal_byte;

function writeflashsignal(int signal)
{
    flash_signal_byte = or(and(flash_signal_byte, 0x3f), and(signal, 0xc0));
    output(flash_signal_byte, flash_sig);
}

function writeflashsignalpulse(int signal1, int signal2)
{
    flash_signal_byte = or(and(flash_signal_byte, 0x3f), and(signal1, 0xc0));
    var int sig_byte_2 = or(and(flash_signal_byte, 0x3f), and(signal2, 0xc0));
    output(flash_signal_byte, flash_sig);
    output(sig_byte_2, flash_sig);
    output(flash_signal_byte, flash_sig);
}

function writeflashaddress(int top, pointer rest)
{
    flash_signal_byte = or(and(flash_signal_byte, 0xc0), and(top, 0x07));
    output(flash_signal_byte, flash_sig);
    output(first(rest), flash_mid);
```

```
    output(second(rest), flash_low);
}

function int readflashbyte(int top, pointer rest)
{
    writeflashsignal(flashsig_we);
    writeflashaddress(top, rest);
    readflashbyte = read(flash_out);
}

// This doesn't actually perform a write,  unless the correct security sequence
is used.
function int writeflashbyte(int top, pointer rest, int data)
{
    writeflashaddress(top, rest);
    output(data, flash_in);
    writeflashsignalpulse(flashsig_oe_we, flashsig_oe);
}

function programflashbyte(int top, pointer rest, int data)
{
    writeflashbyte(0x00, 0x5555, 0xaa);
    writeflashbyte(0x00, 0x2aaa, 0x55);
    writeflashbyte(0x00, 0x5555, 0xa0);
    writeflashbyte(top, rest, data);
    // a 20 us wait is necessary but during write operations the processing time
will be sufficient
}

function readflashnbytes(int top, pointer rest, pointer dest, int16 count)
{
    var int c1 = increment(first(count)), c2 = second(count);
    while (c1)
    {
        while (c2)
        {
            write(readflashbyte(top, rest), dest);
            rest = incrementPointer(rest);
            if (!(first(rest) || second(rest)))
                top = increment(top);
            dest = incrementPointer(dest);
            c2 = decrement(c2);
        }
        c2 = 0xff;
        c1 = decrement(c1);
    }
}

function programflashnbytes(int top, pointer rest, pointer datap, int16 count)
{
    while (or(first(count), second(count)))
    {
        programflashbyte(top, rest, read(datap));
        rest = incrementPointer(rest);
        if (!(first(rest) || second(rest)))
            top = increment(top);
        datap = incrementPointer(datap);
        count = decrementPointer(count);
    }
}
```

```
function eraseflashsectors(int top, pointer rest, int count)
{
	while (count)
	{
		writeflashbyte(0x00, 0x5555, 0xaa);
		writeflashbyte(0x00, 0x2aaa, 0x55);
		writeflashbyte(0x00, 0x5555, 0x80);
		writeflashbyte(0x00, 0x5555, 0xaa);
		writeflashbyte(0x00, 0x2aaa, 0x55);
		writeflashbyte(top, rest, 0x30);
		var int newfirst = add(first(rest), 0x10);  // 0x10 << 8 bits = 4096
bytes (1 sector)
		if (lessthan(newfirst, first(rest)))
			top = increment(top);
		rest = pair(newfirst, second(rest));
		count = decrement(count);
	}
}

// each file record: 16 bytes:
// 3 bytes: start address
// 12 bytes: name (always null terminated so 11 characters)
// 1 bytes: n sectors (0 if empty record, ff sentinel value if last record)
// the sentinel record is never used as a valid record - it's usually all ffs.
var char currentfilerecord[16];
function pointer findfile(pointer filename)
{
	findfile = 0xffff;
	var pointer recordaddress = 0;
	while (true)
	{
		readflashnbytes(0, recordaddress, currentfilerecord, 16);
		if (equal(currentfilerecord[15], 0xff))
			break;
		if (currentfilerecord[15])
		{
			if (!strcmp(filename, addPointer(currentfilerecord, 3)))
			{
				findfile = recordaddress;
				return;
			}
		}
		recordaddress = addPointer(recordaddress, 16);
	}
}

function erasefilesectors(int sector, int nsectors)
{
	var int newmidbyte = add(currentfilerecord[1], shiftleftn(sector, 4));
	var int topbyte = currentfilerecord[0];
	if (lessthan(newmidbyte, currentfilerecord[1]))
		topbyte = increment(topbyte);
	eraseflashsectors(topbyte, pair(newmidbyte, currentfilerecord[2]),
nsectors);
}

/*function writesectortofile(int sector, pointer datap)
{
	var pointer toptwo = addPointer(pair(currentfilerecord[0],
currentfilerecord[1]), shiftleftn(sector, 4));
	erasefilesectors(sector, 1);
```

```
    programflashnbytes(first(toptwo), pair(second(toptwo),
currentfilerecord[2]), datap, 4096);
}*/

function deletefile(pointer tablepos)
{
    programflashbyte(0, addPointer(tablepos, 15), 0);
}

function pointer findfilespace(int nsectors)
{
    // Algorithm: Assume we can place the file at the start.
    // For each file in the FAT, if it overlaps with our file, move our file to
the end of that file.
    // Repeat this loop until there are no more moves. (Yeah O(n^2) I know.)
    var pointer filestart = 0x0010; // divided by 256 bytes so we can use 16 bit
maths.
    var int filesize = shiftleftn(nsectors, 4);
    var pointer fileend = addPointer(filestart, filesize);
    var int hasmoved;
    while (true)
    {
        hasmoved = false;
        var pointer recordaddress = 0x0000;
        while (true)
        {
            readflashnbytes(0, recordaddress, currentfilerecord, 16);
            if (equal(currentfilerecord[15], 0xff))
                break;
            if (currentfilerecord[15])
            {
                var pointer blockstart = pair(currentfilerecord[0],
currentfilerecord[1]);
                var pointer blockend = addPointer(blockstart,
shiftleftn(currentfilerecord[15], 4));
                if (lessthanPointer(filestart, blockend) &&
lessthanPointer(blockstart, fileend))
                {
                    filestart = blockend;
                    fileend = addPointer(filestart, filesize);
                    hasmoved = true;
                    break;
                }
            }
            recordaddress = addPointer(recordaddress, 16);
        }
        if (!hasmoved)
            break;
    }
    findfilespace = filestart;
}

function createfile(pointer filename, int nsectors)
{
    var pointer recordaddress = findfile(filename);
    if (!equal(first(recordaddress), 0xff))
        deletefile(recordaddress);
    recordaddress = 0;
    while (true)
    {
        readflashnbytes(0, recordaddress, currentfilerecord, 16);
```

```
        if (equal(currentfilerecord[15], 0xff))
            break;
        recordaddress = addPointer(recordaddress, 16);
    }
    var pointer space = findfilespace(nsectors);
    var int topaddress = first(space);
    var pointer rest = pair(second(space), 0x00);
    memset(currentfilerecord, 0, 16);
    currentfilerecord[0] = topaddress;
    currentfilerecord[1] = first(rest);
    currentfilerecord[2] = second(rest);
    memcpy(addPointer(currentfilerecord, 3), filename, 10);
    currentfilerecord[15] = nsectors;
    eraseflashsectors(topaddress, rest, nsectors);
    programflashnbytes(0, recordaddress, currentfilerecord, 16);
}

function copyfile(pointer srcname, pointer destname)
{
    if (equal(first(findfile(srcname)), 0xff))
    {
        printlineanddump("Not found.");
        return;
    }
    var int srctop = currentfilerecord[0];
    var pointer srcrest = pair(currentfilerecord[1], currentfilerecord[2]);
    var int sectorcount = currentfilerecord[15];
    var int16 bytecount = pair(shiftleftn(currentfilerecord[15], 4), 0x00);
    if (!equal(first(findfile(destname)), 0xff))
    {
        printlineanddump("File already exists.");
        return;
    }
    createfile(destname, sectorcount);
    var int desttop = currentfilerecord[0];
    var pointer destrest = pair(currentfilerecord[1], currentfilerecord[2]);
    while (first(bytecount) || second(bytecount))
    {
        programflashbyte(desttop, destrest, readflashbyte(srctop, srcrest));
        srcrest = incrementPointer(srcrest);
        if (!(first(srcrest) || second(srcrest)))
            srctop = increment(srctop);
        destrest = incrementPointer(destrest);
        if (!(first(destrest) || second(destrest)))
            desttop = increment(desttop);
        bytecount = decrementPointer(bytecount);
    }
}

function runprogram(pointer filename)
{
    printlineanddump("Running program:");
    printlineanddump(filename);
    var pointer recordpos = findfile(filename);
    if (equal(first(recordpos), 0xff))
    {
        printlineanddump("Not found.");
    }
    else
    {
        var int16 filesize = pair(shiftleftn(currentfilerecord[15], 4), 0x00);
```

```
        readflashnbytes(currentfilerecord[0], pair(currentfilerecord[1],
currentfilerecord[2]),
                              0x8000, filesize);
        goto 0x8000;
    }
}

function exitprogram()
{
    runprogram("manage");
    goto 0x0000;
}

function char getchar()
{
    getchar = 0;
    while (!getchar)
        getchar = read(kbd_in);
}

function pointer getstring()
{
    var char buffer[20];
    memset(buffer, 0, 20);
    printlineanddump(">");
    var pointer cursorp = buffer;
    var char c = 0;
    var int nwritten = 0;
    while (true)    // until newline
    {
        c = getchar();
        if (equal(c, 10))
        {
            break;
        }
        else if (equal(c, 8))
        {
            if (nwritten)
            {
                nwritten = decrement(nwritten);
                cursorp = decrementPointer(cursorp);
                write(0, cursorp);
            }
        }
        else
        {
            nwritten = increment(nwritten);
            write(c, cursorp);
            cursorp = incrementPointer(cursorp);
        }
        memset(addPointer(screenbuffer, 61), ' ', 19);
        printtobuffer(buffer, 61);
        dumpscreenbuffer();
        while (and(read(debugin), 0x80));
    }
    getstring = buffer;
}

function main()
{
    wakescreen();
```

```
    emptyscreenbuffer();
    printlineanddump("Welcome to Fork OS!");
    printlineanddump("(c) Luke Wren 2013");
    runprogram("manage");
}
```

manage.spn (Shell Program)

```
#include "stddefs"
#include "forkos.spn.bin.def"

function printhelp()
{
    printlineanddump("Manager commands:");
    printlineanddump("ls create del cp cat");
    printlineanddump("size run help");
}

function main()
{
    printhelp();
    while (true)
    {
        var char cmdbuff[32];
        memset(cmdbuff, 0, 32);
        strcpy(cmdbuff, getstring());
        var pointer argstring = addPointer(cmdbuff, instr(cmdbuff, ' '));
        if (equal(read(argstring), ' '))
        {
            write(0, argstring);
            argstring = incrementPointer(argstring);
        }
        if (strequal(cmdbuff, "ls"))
        {
            printlineanddump("Any key to scroll.");
            var pointer recordaddress = 0;
            var int count = 3;
            while (true)
            {
                readflashnbytes(0, recordaddress, currentfilerecord, 16);
                if (equal(currentfilerecord[15], 0xff))
                    break;
                count = decrement(count);
                if (!count)
                {
                    count = 3;
                    getchar();
                }
                if (currentfilerecord[15])
                {
                    printlineanddump(addPointer(currentfilerecord, 3));
                }
                recordaddress = addPointer(recordaddress, 16);
            }
        }
        else if (strequal(cmdbuff, "create"))
        {
            if (!second(strlen(argstring)))
                printlineanddump("Usg: create [filenm]");
            else
```

```
            {
                createfile(argstring, 1);
                printlineanddump("Done.");
            }
        }
        else if (strequal(cmdbuff, "del"))
        {
            var pointer tablepos = findfile(argstring);
            if (equal(first(tablepos), 0xff))
                printlineanddump("No such file.");
            else
                deletefile(tablepos);
            printlineanddump("Done.");
        }
        else if (strequal(cmdbuff, "cp"))
        {
            var pointer targetname = addPointer(argstring, instr(argstring, '
'));
            if (lessthan(second(strlen(targetname)), 2))
            {
                printlineanddump("Usg: cp [src] [dest]");
            }
            else
            {
                write(0, targetname);
                targetname = incrementPointer(targetname);
                copyfile(argstring, targetname);
            }
        }
        else if (strequal(cmdbuff, "cat"))
        {
            var pointer tablepos = findfile(argstring);
            if (equal(first(tablepos), 0xff))
            {
                printlineanddump("Not found.");
                continue;
            }
            else
            {
                printlineanddump("Press q to quit, or");
                printlineanddump("any key to scroll.");
                while (!equal(getchar(), 'q'))
                {
                    readflashnbytes(currentfilerecord[0],
pair(currentfilerecord[1], currentfilerecord[2]), screenbuffer, 80);
                    dumpscreenbuffer();
                    var pointer p = addPointer(pair(currentfilerecord[1],
currentfilerecord[2]), 80);
                    if (currentfilerecord[1] && !first(p))
                        currentfilerecord[0] = increment(currentfilerecord[0]);
                    currentfilerecord[1] = first(p);
                    currentfilerecord[2] = second(p);
                }
            }
        }
        else if (strequal(cmdbuff, "run"))
        {
            runprogram(argstring);
        }
        else if (strequal(cmdbuff, "size"))
        {
```

```
            findfile(argstring);
            println(int2hexstr(currentfilerecord[15]));
            printtobuffer(" sector(s).", 63);
            dumpscreenbuffer();
        }
        else if (strequal(cmdbuff, "help"))
        {
            printhelp();
        }
        else
        {
            printlineanddump("Type 'help' for cmds.");
        }
    }
}
```

hexedit.spn

```
#include "stddefs"
#include "forkos.spn.bin.def"

var int buffer[4096];

function main()
{
    printlineanddump("File name:");
    var pointer filename = getstring();
    var pointer tablepos = findfile(filename);
    if (equal(first(tablepos), 0xff))
    {
        printlineanddump("Creating file...");
        createfile(filename, 1);
    }
    readflashnbytes(currentfilerecord[0], pair(currentfilerecord[1],
currentfilerecord[2]), buffer, 4096);
    var pointer cursorpos = 0;
    var pointer scrollpos = 0x0000;
    while (true)
    {
        emptyscreenbuffer();
        var pointer index = add16(buffer, scrollpos);
        var pointer printpos = screenbuffer;
        var int linecount = 4;
        while (linecount)
        {
            printpos = strcpy(printpos, int2hexstr(first(index)));
            printpos = strcpy(printpos, int2hexstr(second(index)));
            printpos = strcpy(printpos, ":   ");
            var int charcount = 4;
            while (charcount)
            {
                printpos = strcpy(printpos, int2hexstr(read(index)));
                index = incrementPointer(index);
                printpos = strcpy(printpos, " ");
                charcount = decrement(charcount);
            }
            linecount = decrement(linecount);
        }
        var int cursorscreenpos = second(subtract16(cursorpos, scrollpos));
```

```
        printtobuffer(">", read(addPointer({7, 10, 13, 16, 27, 30, 33, 36, 47,
50, 53, 56, 67, 70, 73, 76}, cursorscreenpos)));
        dumpscreenbuffer();
        var char c = getchar();
        if (!((lessthan(c, '0') || lessthan('9', c)) && (lessthan(c, 'a') ||
lessthan('f', c))))
        {
            var pointer byteloc = add16(buffer, cursorpos);
            var int data = read(byteloc);
            data = shiftleftn(data, 4);
            if (lessthan('9', c))
                data = or(data, add(subtract(c, 'a'), 10));
            else
                data = or(data, subtract(c, '0'));
            write(data, byteloc);
        }
        else if (equal(c, 'x'))
        {
            printlineanddump("Save first (y/n/c)?");
            var char c = getchar();
            if (equal(c, 'y'))
            {
                printlineanddump("Saving...");
                erasefilesectors(0, 1);
                programflashnbytes(currentfilerecord[0],
pair(currentfilerecord[1], currentfilerecord[2]), buffer, 4096);
                printlineanddump("Done.");
                goto 0x0000;
            }
            else if (equal(c, 'n'))
            {
                exitprogram();
            }
        }
        else
        {
            if (equal(c, 'j'))
            {
                if (first(cursorpos) || second(cursorpos))
                    cursorpos = decrementPointer(cursorpos);
            }
            else if (equal(c, 'l') || equal(c, ' '))
            {
                cursorpos = incrementPointer(cursorpos);
            }
            else if (equal(c, 'i'))
            {
                if (lessthanPointer(3, cursorpos))
                    cursorpos = add16(cursorpos, 0xfffc);
            }
            else if (equal(c, 'k'))
            {
                cursorpos = addPointer(cursorpos, 4);
                if (lessthanPointer(0xfff, cursorpos))
                    cursorpos = 0xfff;
            }
            if (lessthanPointer(addPointer(scrollpos, 15), cursorpos))
                scrollpos = addPointer(scrollpos, 4);
            if (lessthanPointer(cursorpos, scrollpos))
                scrollpos = add16(cursorpos, 0xfffc);   // -4 in two's
complement
```

```
        }
    }
}
```

Appendix 4: Compiler Test Programs

beep.spn

```
const pointer debugout = 0xc000;
const pointer debugin = 0xc001;
const int true = 0xff;
const int false = 0x00;

function int decrement(int x);
function int read(pointer p);

function sleep(int time)
{
    while (time)
        time = decrement(time);
}

function main()
{
    while (true)
    {
        var int period;
        period = read(debugin);
        sleep(period);
        nfc(debugout, val(0xff));
        sleep(period);
        nfc(debugout, val(0x00));
    }
}
```

fibonacci.spn

```
#include "stddefs"

function int add(int a, int b)
{
    var int carry, temp;
    add = xor(a, b);
    carry = shiftleft(and(a, b));
    while (carry)
    {
        temp = add;
        add = xor(add, carry);
        carry = shiftleft(and(temp, carry));
    }
}

function main()
{
    while (true)
    {
        var int a = 0, b = 1, c, count;
```

```
        count = read(debugin);
        while (count)
        {
            c = add(a, b);
            a = b;
            b = c;
            count = decrement(count);
        }
        output(a, debugout);
    }
}
```

logicoperators.spn

```
#include "stddefs"

function main()
{
    var int a = 1, b = 0;
    if (a || b)
        output(val(1), debugout); // true
    if (a && b)
        output(val(2), debugout); // false
    if (a && !b)
        output(val(3), debugout); // true
    if (!a || b)
        output(val(4), debugout); // false
    if (!b || a)
        output(val(5), debugout); // true
    if (!(b || a))
        output(val(6), debugout); // false
    if ((b && b) || a)
        output(val(7), debugout); // true
    if (b && b || a)
        output(val(8), debugout); // false
}
```

strcmp.spn

```
#include "stdmem"

function main()
{
    if (strcmp("a", "a"))
        output(val(1), debugout);
    else
        output(val(2), debugout);
}
```

helloworld.spn

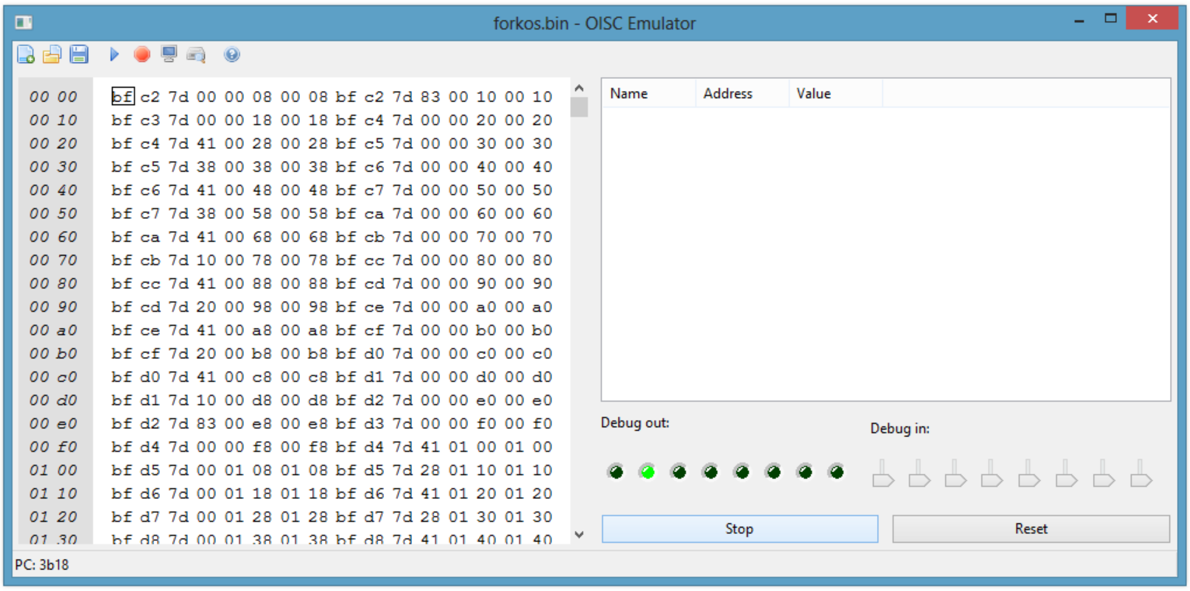
```
#include "stdscreen"

function main()
{
    wakescreen();
    emptyscreenbuffer();
    println("Hello world!");
    dumpscreenbuffer();
}
```

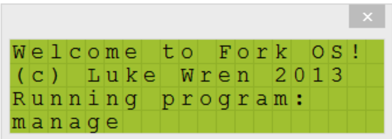
Appendix 5: Supporting Software

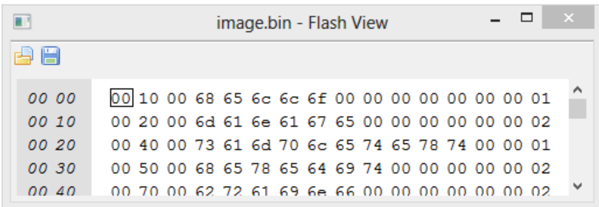
In order to create a useful toolchain, I've developed some additional software to ease development for this architecture.

Emulator

Although I improved the speed of the programmer by around a factor of 50 by using the EEPROM's page write mode, rebuilding the programmer binary and configuring the computer hardware for programming after each code change slowed down the programming feedback loop, making development a little cumbersome. In addition, during the early stages of hardware development, I was not even certain whether the hardware was reliable! This made assessing the reliability of my compiled code very difficult.





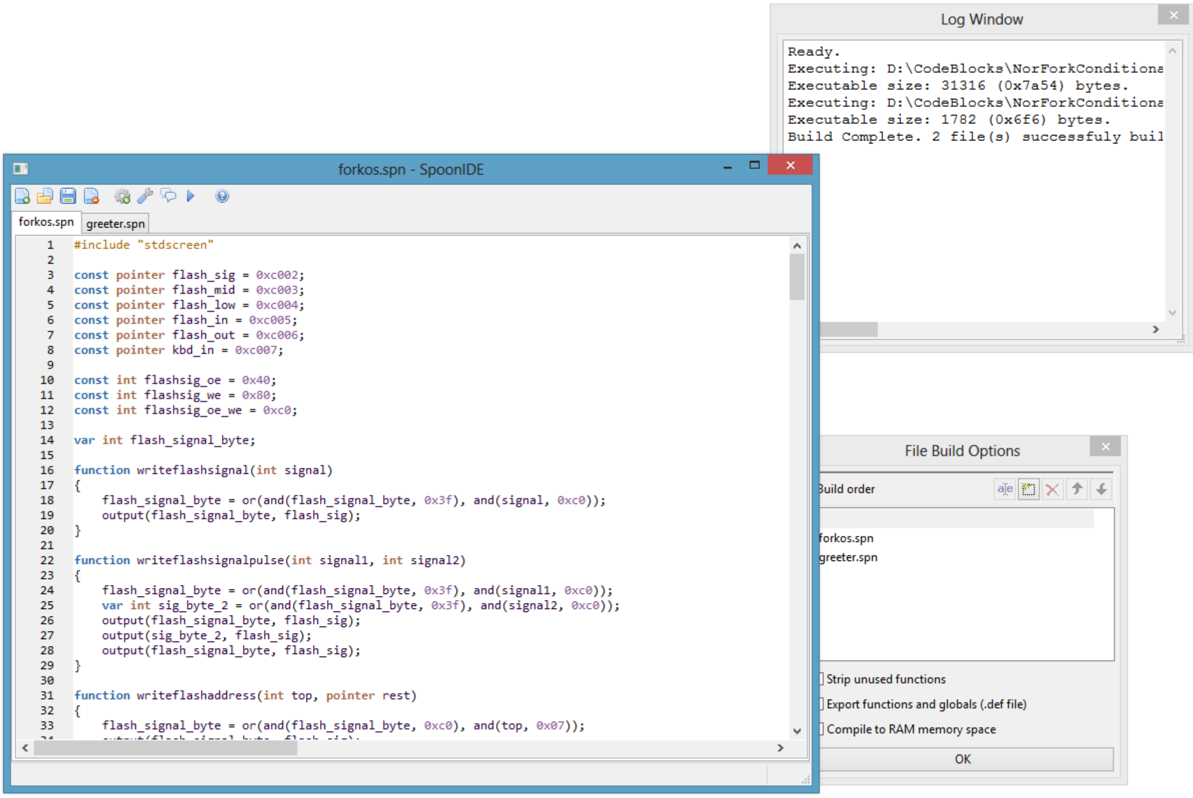


I used C++ and wxWidgets to write an instruction-level emulator for the entire computer system. The interface mirrors the input switch bank and debug LEDs on the computer's board. The code is fairly extensible – I added an LCD window and another window that emulates the flash device and allows you to view and edit its contents in real time. The LCD uses the same industry standard Hitachi HD44780 style interface as the real one, and the flash module mimics the real SST39FS040 on the board, meaning the exact same code will run unmodified on hardware and the emulator (this is important for finding compiler bugs).

The emulator integrates nicely into the IDE – you can simply click "run" and your code will automatically be compiled and run in front of you.

Integrated Development Environment

An integrated development environment (IDE) is a fancy text editor, with a back end that pulls all of the various development tools together into something vaguely cohesive.



Features:

- Pretty syntax highlighting (for enhanced productivity)
- Tabbed interface for quickly swapping between open files
- High level of integration with tools:
    - Compiler errors appear on-screen in front of you
    - Automatically compiles and opens the file in the emulator when you click run