CS 241: Foundations of Sequential Programs

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Gordon Cormack's lectures.

1 Introduction & Character Encodings

← January 7, 2013

1.1 Course Structure

The grading scheme is 50% final, 25% midterm, and 25% assignments. There are eleven assignments. Don't worry about any textbook. See the course syllabus for more information.

1.2 Abstraction

Abstraction is the process of removing or hiding irrelevant details. Everything is just a sequence of bits (binary digits). There are two possible values for a bit, and those values can have arbitrary labels such as:

- Up / down.
- Yes / no.
- 1 / 0.
- On / off.
- Pass / fail.

Let's say we have four projector screens, each representing a bit of up/down, depending on if the screen has been pulled down or left up (ignoring states between up and down). These screens are up or down independently. There are sixteen possible combinations:

Screen 1	Screen 2	Screen 3	Screen 4
Up (1)	Down (0)	Up (1)	Down (0)
Down (0)	Down (0)	Down (0)	Up (1)
⋮	⋮	⋮	⋮

Note that there are sixteen combinations because $k = 4$, and there are always 2^k combinations since there are two possible values for each of k screens.

1.3 Endianness

Let's consider the sequence 1010. This sequence of bits has a different interpretation when following different conventions.

- **Unsigned, little-endian**: $(1 \times 2^0) + (0 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) = 1 + 4 = 5$.
- **Unsigned, big-endian**: $(0 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3) = 2 + 8 = 10$.
- **Two's complement, little-endian**: $5 - 16 = -10$.
- **Two's complement, big-endian**: $10 - 16 = -6$.
- **Computer terminal**: LF (line feed).

Note that a two's complement number n will satisfy $-2^{k-1} \leq n < 2^{k-1}$.

1.4 ASCII

ASCII is a set of meanings for 7-bit sequences.

Bits	ASCII Interpretation
0001010	LF (line feed)
1000111	G
1100111	g
0111000	8

In the latter case, 0111000 represents the character '8', not the unsigned big- or little-endian number 8.

ASCII was invented to communicate text. ASCII can represent characters such as A-Z, a-z, 0-9, and control characters like ();!. Since ASCII uses 7 bits, $2^7 = 128$ characters can be represented with ASCII. As a consequence of that, ASCII is basically only for Roman, unaccented characters, although many people have created their own variations of ASCII with different characters.

1.5 Unicode

Unicode was created to represent more characters. Unicode is represented as a 32-bit binary number, although representing it using 20 bits would also be sufficient. The ASCII characters are the first 128, followed by additional symbols.

A 16-bit representation of Unicode is called **UTF-16**. However, there's a problem: we have *many* symbols ($> 1M$) but only $2^16 = 65,536$ possibilities to represent them. Common characters are represented directly, and there is also a 'see attachment' bit for handling the many other symbols that didn't make the cut to be part of the $65,536$. Similarly, there is an 8-bit representation of Unicode called **UTF-8**, with the ASCII characters followed by additional characters and a 'see attachment' bit.

The bits themselves do not have meaning. Their meaning is in your head – everything is up for interpretation.

1.6 A Message for Aliens

← January 9, 2013

In a computer, meaning is in the eye of the beholder. We must agree on a common interpretation – a convention. However, the English language and numbers also have their meaning determined by a set of conventions.

NASA wanted to be able to leave a message for aliens on a plaque on their spacecraft, however it was clear that aliens would not understand our language or even 0s and 1s. NASA wanted their message to be a list of prime numbers. They decided they would use binary to represent the numbers, but since 0s and 1s would be ambiguous to aliens, they used a dash (-) instead of 0, and 1 for 1. It's only a convention, but it's one that NASA determined aliens would have a higher chance of understanding.

1.7 Hexadecimal

Hexadecimal (hex) is base 16. It has sixteen case-insensitive digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f.

Why is hex useful? It makes conversions easy. We group bits into sequences of four:

$$\underbrace{\underbrace{0011}_{3}\underbrace{1010}_{A}}_{3A}$$

Conversions are made especially easy when the sequences of bits are lengthy:

$$\underbrace{\underbrace{10}_{2}\underbrace{1110}_{E}\underbrace{0111}_{7}\underbrace{0011}_{3}\underbrace{1011}_{B}\underbrace{1001}_{9}\underbrace{1000}_{8}\underbrace{0011}_{3}}_{2E73B983}$$

2 Stored Program Computers

Stored program computers are also known as the **Von Neumann architecture**. They group bits into standard-sized sequences.

In modern times, standard-sized sequences of bits are:

- **Bytes**. A byte is 8-bits (256 possible values). Example: 00111010.
- **Words**. A word is only guaranteed to be "more than a byte." Words are often 16-bit ($2^{16} = 65,536$ possible values), 32-bit ($2^{32} \approx 4 \times 10^9$), or 64-bit ($2^{64} \approx 10^{19}$).

2.1 Storage Devices

2.1.1 Registers

There are typically a finite number of fixed-sized sequence of bits, called **registers**. You can put bits in, peek at them, and modify them. A "64-bit CPU" just means it's a CPU that uses 64-bit words.

Calculators typically have 2-3 registers for recalling numbers and maintaining state.

There are a couple of downsides to registers. They're expensive to build, which is why there is a finite number of them. They're also difficult to keep track of.

2.1.2 RAM (Random Access Memory)

RAM is essentially a physical array that has **address lines**, **data lines**, and **control lines**. Data is fed into RAM using electrical lines. Data will remain in RAM until overwritten.

If you want to place a happy face character at address 100, you set the address lines to 100, the data lines to 10001110 (which is the Unicode representation of a happy face), and give the control lines a kick.

RAM could be implemented in several different ways. It could even be created with a **cathode ray tube**. The **core** method is synonymous with RAM, however. It involves a magnetic core, and the data remains magnetized after the magnet is removed. Bits are read by toggling the state (toggling the magnetic poles) and seeing if it was easier to toggle than expected (similar to unlocking an already-unlocked door), and then toggling back after. No one really uses magnetic cores anymore.

Capacitive memory (also known as dynamic RAM or **DRAM**) is still used today. It involves an insulator, and two conductive plates, one more negatively-charged than the other. The electrons will remain in their state even when the poles are removed. There is a problem, however. Insulators are not perfect – electrons will eventually make their way through the insulator. In order to alleviate this, we have to refresh the charge fairly often (every second, for instance).

Switches are typically used only for registers and cache. They produce more heat, but are much faster.

2.2 Control Unit Algorithm

The CPU contains a **control unit**, several **registers**, PC (**program counter**), and IR (**instruction register**), and is connected to RAM with electrical lines.

```
PC <- some fixed value (e.g. 0)
loop
  fetch the word of RAM whose address is in PC, put it in IR
  increment PC
  decode and execute the machine instructon that's in IR
end loop
```

IR would contain an instruction like "add register 1 to register 2, and put the result into register 7."

3 First Steps with MIPS

← January 11, 2013

3.1 Unix

You'll need Unix to use MIPS in this course. Unix was originally created in the 1970s at AT&T Bell Labs. Unix is still popular today, especially for servers. Linux is a Unix dialect, and Mac OS X is also based on Unix.

Unix has three types of files:

- **Binary files**. A sequence of arbitrary bytes.
- **Text files**. A sequence of ASCII characters, with lines terminated by a LF / newline.
- **Tools**. These are programs, which are technically binary files.

3.1.1 Getting Access to a Unix System

If you use Linux or Mac OS X, you're in good shape. However, Windows is not Unix-based, so you'll have to pursue one of these alternative options:

- Install Linux. You can dual-boot it alongside Windows if you'd like, or you could install it inside a virtual machine.
- Install Cygwin. When installing it, choose to install everything.
- Login to the `student.cs` servers remotely. You can use PuTTY for that.

3.1.2 Commands You Should Know

- `ssh username@linux.student.cs.uwaterloo.ca` – logs you into the student.cs systems remotely through SSH.
- `cat unix_text_file.txt` – copies the contents of the file to the current terminal. If a non-text file is given to `cat`, incorrect output will result.
- `xxd -b unix_file.txt` – prints the binary representation of the file to the terminal. The numbers in the left column are the location in the file. If it's a Unix text file, the ASCII representation is presented on the right, with all non-printable characters printed as dots. `xxd` is not aware of newline characters – it arbitrarily splits the file into 16 bytes per line.
- `xxd unix_file.txt` – prints the hex representation of the file to the terminal. Identical to the previous command (`-b`) in every other way.
- `ls -l` – lists all files in the current directory in the long-listing form, which shows the number of bytes in the file, permissions, and more.

3.2 Getting Started with MIPS

The MIPS CPU uses 32-bit words since it's a 32-bit machine, and it's big-endian. You can use `xxd` to inspect MIPS files. MIPS has 32 registers (numbered 0 to 31).

At the end of our MIPS programs, we will copy the contents of register $31 to the program counter (PC) to "return".

3.2.1 Running MIPS Programs

Upon logging in to the `student.cs` servers, run `source ~cs241/setup` in order to add the required executables to your `PATH`. Then, when given a MIPS executable called `eg0.mips`, you can run `java mips.twoints eg0.mips` in order to run the program.

`mips.twoints` is a Java program that requests values for registers $1 and $2 and then runs the given MIPS program. There are other MIPS runner programs, such as `mips.array`, which populate the 31 registers in different ways.

3.2.2 Creating MIPS Programs

Start with `vi thing.asm` (or use your favorite editor). Inside this file, you'll create an **assembly language file**, which is a textual representation of the binary file you want to create. Each line in this file should be in the form `.word 0xabcdef12` (that is, each line should start with `.word 0x` – the `0x` is a convention that indicates that hex follows). You can add comments onto the end of lines, starting with a semi-colon (Scheme style).

Next, you'll need to convert your assembly language file into a binary file. You can do that by running `java cs241.wordasm < thing.asm > thing.bin`. You can then inspect `thing.bin` with `xxd` in hex, or in binary if you're masochistic.

A few important things you should know for developing MIPS programs:

- $0 is a register that will always contain 0. It's special like that.
- $30 points to memory that could be used as a stack.
- $31 will be copied to the program counter at the end of execution in order to "return".
- You can specify register values using base 10 values or as hex values (if prefixed by `0x`).
- It takes 5-bits to specify a register, since $2^5 = 32$.
- It's convention to call S and T (as indicated in various documentation) **source registers**, and D is the **destination register**.
- MIPS uses two's complement numbers by default, unless specified otherwise.
- Loops and conditionals are accomplished by adding or subtracting from the program counter.

There is a MIPS reference sheet available on the course website that you'll find to be quite useful. It contains the binary representations for all MIPS instructions. Convert the binary into hex and put them into an assembly language file.

3.2.3 A Few Important MIPS Instructions

1. **Load Immediate & Skip** (`lis`): loads word from the program counter. Loads the next word of memory into the D register. You specify a `lis` instruction followed by an arbitrary word next. You need to also skip the appropriate number of bytes by incrementing the program counter.
2. **Set Less Than [Unsigned]** (`slt`): compares S to T. If S $<$ T, 1 is put into the D register, otherwise 0 is put into the D register.
3. **Jump Register** (`jr`): copies S to the program counter.
4. **Jump and Link Register** (`jalr`): assigns the program counter to register 31, then jumps to it.
5. **Branch on Equal** (`beq`): if S is equal to T, it adds the specified number to the program counter (times 4). There is also **Branch on Unequal** (`bne`) which does the opposite.

3.2.4 MIPS Program Workflow

← January 14, 2013

The MIPS CPU understands **binary machine language programs**, however we cannot write them directly. Instead, we write **assembly language programs** in text files. By convention, we name these text files with the extension `.asm`. Assembly language contains instructions like `.word 0x00221820`. We feed the assembly language program into `cs241.wordasm`, which is an **assembler**. An assembler translates assembly language into binary machine code.

Assembly language can also look like this: `add $3, $1, $2`. Assembly language in this form has to be fed into a different assembler (`cs241.binasm`) that understands that flavor of assembly syntax.

There is a MIPS reference manual available on the course website. It might be useful in situations such as:

- When you want to be an assembler yourself. You'll need to lookup the mapping between assembly instructions like `add $3, $1, $2` and their binary equivalents.
- When you need to know what's valid assembly code that an assembler will accept.
- When you want to write your own assembler you'll need a specification of which instructions to handle.

3.2.5 The Format of MIPS Assembly Language

MIPS assembly code is placed into a Unix text file with this general format:

```
labels instruction comment
```

Labels are any identifier followed by a colon. For example, `fred:`, `wilma:`, and `x123:` are some examples of valid labels.

Instructions are in the form `add $3, $1, $2`. Consult the MIPS reference sheet for the syntax of each MIPS instruction.

Comments are placed at the end of lines and must be prefixed by a semicolon. Lines with only comments (still prefixed with a semicolon) are acceptable as well. For example: `; hello world`.

It's important to note that there is a **one-to-one correspondence** between instructions in assembly and instructions in machine code. The same MIPS instructions will always produce the same machine code.

3.2.6 More MIPS Instructions

Here's a more comprehensive overview of the instructions available to you in the CS 241 dialect of MIPS. Note that for all of these instructions, $0 \leq d, s, t \leq 31$, since there are 32 registers in MIPS numbered from 0 to 31.

- `.word`. This isn't really a MIPS instruction in and of itself. Words can be in several different forms. For example:

  - `.word 0x12345678` (hex)
  - `.word 123` (decimal)
  - `.word -1` (negative decimals whose representation will eventually be represented in two's complement)

- `add $d, $s, $t`. Adds $s to $t and stores the result in $d.
- `sub $d, $s, $t`. Subtracts $t from $s and stores the result in $d ($d = $s - $t).
- `mult $s, $t`. Multiplies $s and $t and stores the result in the `HI` and `LO` registers. Uses two's complement.
- `multu $s, $t`. Provides the same functionality as `mult`, but uses unsigned numbers.
- `div $s, $t`. Divides $s by $t. The remainder is stored in `HI` and the quotient is stored in `LO`.
- `divu $s, $t`. Provides the same functionality as `div`, but uses unsigned numbers.
- `mflo $d`. Copies the contents of the `LO` register to $d.
- `mfhi $d`. Copies the contents of the `HI` register to $d.
- `lis $d` (load immediate and skip). Copies the word from the program counter (PC), adds 4 to PC in order to skip the word you just loaded.
- `lw $t, i($s)` (load word, $-32,768 \leq i \leq 32,767$). For example: `lw $3, 100($5)` will get the contents of $5, add 100, treat the result as an address, fetch a word from RAM at that address, and put the result into $3.
- `sw $t, i($s)` (store word, $-32,768 \leq i \leq 32,767$). This works in a similar way to `lw`, except it stores the contents of $t at RAM at this address.
- `slt $d, $s, $t` (set less than). Sets $d to 1 if $s < $t, or to 0 otherwise.
- `sltu $d, $s, $t` (set less than unsigned). Sets $d to 1 if $s < $t, or to 0 otherwise. Interprets the numbers as unsigned numbers.
- `beq $s, $t, i` (branch if equal, $-32,768 \leq i \leq 32,767$). Adds $4i$ to the program counter if $s is equal to $t. Note that 4 is still added (in addition to adding the $4i$ for this specific command) as you move to the next instruction, as with all instructions.
- `bne $s, $t, i` (branch if not equal, $-32,768 \leq i \leq 32,767$). Works the same way as `beq`, except it branches if $s is not equal to $t.
- `jr $s` (jump register). Copies $s to the program counter.
- `jalr $s` (jump and link register). Copies $s to the program counter and copies the previous value of the program counter to $31.

3.2.7 Example Program: Sum from 1 to N

We want a program that sums the numbers from 1 to n, where n is the contents of \$1, and we want the result to be placed in \$3. Aside: it's only a convention that we reserve \$1 and \$2 as registers for input parameters and \$3 as the register for the result – the MIPS system itself does not treat these registers in a special way.

```
; $1 is N.
; $3 is the sum.
; $2 is temporary.

add $3, $0, $0  ; zero accumulator

; beginning of loop
add $3, $3, $1  ; add $1 to $3
lis $2          ; decrement $2
.word -1
add $1, $1, $2
bne $1, $0, -5  ; n = 0? If not, branch to beginning of loop

jr $31          ; return
```

If we enter 10 for \$1 (to get the sum of the numbers from 1 to 10), we should get 55. But the actual result is `0x00000037`. Note that $37_{16} = 55_{10}$, so the program works as expected. The end result is \$1 being `0x00000000` (0_{10}), \$2 being `0xffffffff` (-1_{10}), and \$3 being `0x00000037` (55_{10}).

3.2.8 Housekeeping Notes

- `cs241.binasm` will be available on Thursday after the assignment 1 deadline has passed. You can use this for future assignments as necessary.
- You don't need to memorize the binary representation of MIPS commands for exams, or the ASCII representation of characters. You'll be provided with the MIPS reference sheet and an ASCII conversion chart for the exams.

3.2.9 Labels

← January 16, 2013

Part of the assembler's job is to count instructions and keep track of their locations (`0x00000004`, `0x00000008`, `0x0000000c`, etc.). The assembler can also simplify the programmer's job at with **labels**.

Labels are identifiers in the form `foo:` (a string followed by a colon). A label `foo:` is equated to the **location** of the ine on which it is defined.

Some instructions like `beq` and `bne` rely on relative locations of lines. Counting these yourself is tedious, and can be troublesome in some situations. The locations you specify, both in places where they're specified relatively and in places where they're specified absolutely (`jr`), may become invalid if you add or remove any lines to your codebase.

Labels can be used in place of integer constants. If you have an instruction like `bne $1, $2, -5`, you can replace it with `bne $1, $2, foo`. The assembler will compute:

$$\frac{\text{location(label) - location(next instruction)}}{4}$$

The third argument of `bne` is always a number. It can an integer literal, or it can be a label which will be converted to an integer by the assembler. MIPS itself has no knowledge of labels – only the assembler does.

4 Accessing RAM in MIPS

4.1 RAM vs. Registers

There are some key differences between RAM and registers:

- There is lots of RAM available, but there are a finite number of registers available (usually not very many).
- You can compute addresses with RAM, but registers have fixed names that cannot be computed (i.e. you can compute memory address `0x00000008 = 0x00000004 + 0x0000004`, but you can't compute $2.
- You can create large, rich data structures in RAM. Registers provide small, fixed, fast storage mechanisms.

4.2 Storing in RAM

```
lis $5
.word 100000
sw $1, 0($5)
lw $3, 0($5)
jr $31
```

The example above uses memory address 100000. But how do we know that we have that much RAM? How do we know it's not already being used by someone else? This is clearly a bad practice.

We really shouldn't just use an arbitrary memory address without any type of safety checking. So, we'll reserve some memory ourselves. We can add a word after the last `jr` instruction, which means memory will be allocated for the word instruction, however it'll never be executed.

MIPS requires that we actually specify a word. The contents of it don't matter, so we'll just use `.word 28234`, which is entirely arbitrary. We can then replace 100000 in the above example with 20. For now, we can assume that our MIPS program will always run in memory starting at memory address 0, so memory addresses and locations in our code can be treated as being the same.

But wait! Hard-coding 20 is a bad idea, in case the program changes, and it's tedious to calculate the proper location (20). We should use a label instead.

4.2.1 Stack

$30 is the conventional register to place the **stack pointer** in (sometimes abbreviated as $sp). The stack pointer points to the first address of RAM that's reserved for use by other people. Here's an example of storing and fetching something in the stack:

```
sw $1, -4($30)
lw $3, -4($30)
jr $31
```

All memory with an address less than the value of $30 could be used by your program. You can use this method to create 100,000+ storage locations, and that wouldn't have been possible with registers without having 100,000 registers, and without hard-coding $1, $2, ... $100000.

The stack pointer isn't magical. It doesn't change on its own, but you can change it yourself if you'd like. Just make sure to change the stack pointer back to its original state before you return (before `jr $31`).

Here's another example of a program which sums the numbers from 1 to n without modifying anything except $3. Actually, it's okay to modify $1 and $2, so long as they are returned to their original state before returning.

```
sw $1, -4($30)   ; save on stack
sw $2, -8($30)   ; save on stack

lis $2
.word 8
sub $30, $30, $2 ; push two words

add $3, $0, $0

; beginning of loop
foo: add $3, $3, $1
  lis $2
  .word -1
  add $1, $1, $2
  bne $1, $0, foo

lis $2
.word 8
add $30, $30, $2 ; restore stack pointer

lw $1, -4($30)   ; restore from stack
lw $2, -8($30)

jr $31
```

`mips.array` is a MIPS runner that passes an array A of size N into your MIPS program. The address of A will be in $1, and the size of A (which is N) will be in $2.

To access array elements, you would execute instructions such as these:

```
lw $3, 0($1)
sw $4, 4($1)
```

Note that each array index increases by 4.

You can also compute the array index. In C/C++, you might have an expression A[i]. A is in \$1 and i is in \$3. How can we fetch A[i] into x (let's say, into \$7)?

1. Multiply i by 4.
2. Add to A.
3. Fetch RAM at the resulting address.

```
add $3, $3, $3
add $3, $3, $3 ; these two lines give i * 4

add $3, $3, $1 ; A + i * 4
lw $7, 0($3)
```

Note that the two first lines each double the value in \$3, so the two lines together effectively multiplied i by 4.

Here's an example program to sum the integers in an array A of length N. \$1 contains the address of A, \$2 contains N, and \$3 will contain the output (the sum). \$4 is used temporarily.

```
add $3, $0, $0

loop:
  lw $5, 0($1)     ; fetch A[i]
  add $3, $3, $5   ; add A[i] to sum
  lis $4           ; load -1 into $4
  .word -1
  add $2, $2, $4   ; decrement $2
  lis $4
  .word 4
  add $1, $1, $4
  bne $2, $0, loop ; loop if not done.

jr $31
```

5 Procedures in MIPS

← January 18, 2013

Recall the `sum.asm` program from earlier, which sums the numbers from 1 to N (\$1), and puts the result in \$3:

```
sum:              ; only needed for next example, sum10.asm
  add $3, $0, $0 ; $3 is the accumulator A.

loop:
  add $3, $3, $1 ; A = A + N
  lis $2
  .word -1
  add $1, $1, $2 ; A = A + (-1)
  bne $1, $0, loop

jr $31
```

Now, let's create a program `sum10.asm` which sums the numbers from 1 to 10, and puts the result in $3. We'll add the `sum:` label to the top of our `sum.asm` file, as indicated, so we have a way to jump to the `sum.asm` line (which is part of how procedures are called in MIPS).

```
; PROLOGUE
sw $31, -4($30)  ; push word onto stack
lis $2
.word 4
sub $30, $30, $2

; PROCEDURE CALL
lis $1
.word 10
lis $4
.word sum        ; address of sum procedure is in $4
jalr $4          ; puts old PC value into $31, jumps to $4

; EPILOGUE
lis $2
.word 4
add $30, $30, $2 ; restore stack pointer
lw $31, -4($30)  ; restore $31
jr $31
```

Note that if you ever get into an infinite loop while executing a MIPS program, you can push CTRL-C to forcefully end the process immediately.

We use `jalr` instead of `jr` so the `sum` routine knows how to get back. `jalr` is the only instruction that can access the contents of the `PC`.

How do we actually run this program? We `cat` together the two programs! It really is that simple. You execute `cat sum10.asm sum.asm | java cs241.binasm > foo.mips` to get a MIPS program in binary.

5.1 Recursion

Recursion is nothing special. You need to save any local variables (which are stored in registers), including given parameters and the return address, onto the stack so we can change

them back when we're done. We don't want subroutines (recursive calls) to mess with those values, so subroutines must preserve their own values. "It's always good hygiene to save your registers."

Let's build `gcd.asm`, where $1 = a, $2 = b, and $3 will hold the result. We will use the following algorithm:

$$gcd(a,b) = \begin{cases} b & a = 0 \\ gcd(b\%a, a) & a \neq 0 \end{cases}$$

Here's `gcd.asm`:

```
gcd:
  sw $31, -4($30)  ; save return address
  sw $1, -8($30)   ; and parameters
  sw $2, -12($30)
  lis $4
  .word 12
  sub $30, $30, $4

  add $3, $2, $0   ; tentatively, result = b
  beq $1, $0, done ; quit if a = 0
  div $2, $1       ; stores quotient in LO, remainder in HI
  add $2, $1, $0   ; copy a to $2
  mfhi $1          ; $1 <- b % a
  lis $4
  .word gcd
  jalr $4

done:
  lis $4
  .word 12
  add $30, $30, $4
  lw $31, -4($30)
  lw $31, -8($30)
  lw $2, -12($30)
  jr $31
```

An **invariant** means if something is true as a pre-condition then it is always true as a post-condition.

Notice in the `gcd.asm` example, you aren't actually erasing the stack contents. If you're storing secret data, you should overwrite it with zeroes, or (ideally) garbage data. For assignment 2, at least, we can just leave our garbage lying around.

5.2 Input and Output

`getchar` and `putchar` simulate RAM, however they actually send the data to/from the user's keyboard/monitor. `getchar` is located at memory address `0xffff0004` and `putchar`

is at address `0xffff000c`. If you store or load a byte at either of these addresses, you will send or retrieve the byte to/from standard input (STDIN) or standard output (STDOUT).

We will create an example program, `cat.asm`, to copy input to output:

```
lis $1
.word 0xffff0004   ; address of setchar()
lis $3
.word -1           ; EOF signal

loop:
  lw $2, 0($1)     ; $2 = getchar()
  beq $2, $3, quit ; if $2 == EOF, then quit
  sw $2, 8($1)     ; putchar() since getchar() and putchar() are 8 apart
  beq $0, $0, loop

quit: jr $31
```

6 Building an Assembler

← January 21, 2013

An assembler is just a program that reads input and produces output. The input and output of an assembler just happen to be programs.

You need to be more familiar with the MIPS assembly language in order to write an assembler, compared to just writing MIPS assembly code. You need to know *exactly* what is a valid MIPS assembly program and what isn't in order to write a proper assembler.

Ensure you implement and test everything on the MIPS reference sheet. You need to test the range for all numbers and reject all programs that contain numbers that are not within the valid ranges, for instance.

Sometimes when we look at a MIPS program there is no meaning because the MIPS assembly code is not well-formed (it's invalid). In that case, we need to have the assembler output an error report. For us, our assembler will identify well-formed MIPS programs in the 'all or nothing' sense – that is, if the program is valid we will produce valid binary machine code, otherwise we'll indicate that there's a problem.

Don't imagine all the ways a program can be wrong. Write your assembler to identify correct MIPS assembly code as per the specification, and if the program does not follow those finite number of rules, then it is invalid and should be rejected.

6.1 Assemblers In This Course

It'd be nice if our assembler's error reports could produce helpful error messages. However, that involves mind reading (pretty much) and is beyond the scope of this course.

For assignments in this course, you can use Scheme, C++, or Java to write your assembler. Scheme is recommended, especially since you'll be dealing with data structures of lengths

that aren't pre-determined, which is easier to handle in Scheme than in C++ or Java. The tools provided for Java are supported for tests, so you can use them, but we won't actively be covering them.

For assignments 3 and 4, you will be provided a scanner for use in your assembler.

6.2 The Assembly Process

1. Read in your text file containing MIPS assembly code. [Input]
2. Scan each line, breaking it into components. [Analysis]
3. Parse components, checking well-formedness. [Analysis]
4. Other error checking. [Analysis]
5. Construct equivalent binary MIPS code. [Synthesis]
6. Output binary code. [Output]

6.2.1 Output

How do we actually output binary code? In C, the only *safe* way is to use `putchar`, which outputs one byte. Here's how to output a 32-bit big-endian word in C:

```
putchar(...)
putchar(...)
putchar(...)
putchar(...)
```

You can't use a built-in C function that outputs a whole word at once, because your computer architecture (Intel machines) will probably force that word to be written in little-endian, which won't work with MIPS.

In Scheme, you can use `(write-byte ...)` which works in a similar way.

6.2.2 Scanners

Scanners are way more annoying than they seem at first glance. We'll be given a scanner for assignments 3 and 4, called `asm.cc`, `asm.ss`, or `asm.java`.

The scanner takes a line of text (a string) and gives you a list of all components in the string. For example, the line `foo: bar: add $1, $2, foo` will give you:

- label `foo`
- label `bar`
- instruction `add`
- register `$1`
- comma

- register `$2`
- comma
- identifier `foo`

You should have code to check the validity of `add`'s parameters – you shouldn't have code to check for indentifiers in place of registers, etc. You must also ensure that there aren't too many or two few arguments, but to do so you should check that you have exactly the correct number of arguments.

You can use `cs241.binasm` combined with `xxd` as a reference implementation, because it's a valid MIPS assembler.

The assembler builds a **symbol table** that keeps track of mappings between identifiers and their locations. The symbol table is later used during the synthesis process to check the validity of label usage, since labels can be used before they're defined. Not all error checking occurs in the analysis steps, since this check during the synthesis process is considered error checking.

If `foo` did not exist in the symbol table, then you should error out. Whether `foo` was defined before it was used is irrelevant because it's perfectly valid to use labels before they're defined.

When an error is encountered, we must write to an error file. It our case, we'll write our output (the binary file) to standard output (STDOUT) and any error messages to standard error (STDERR). Ensure the assembler does not crash when it runs into an invalid program.

If there's an error, it doesn't matter what you've already written to standard output – it'll be ignored. You could have written half a program or nothing, but it doesn't matter if an error message has been written to standard error.

The scanner will check hex numbers to ensure their general format is correct. That is, it will let you know that `0xg` is not a valid hex number. However, it may or may not check the ranges of numbers.

One common hiccup is not to throw an error when you are given `.word 10, $1`, which is not valid. `.word` commands can only take one number, in decimal, hex, or a label identifier.

7 Outline of an Assembler

← January 23, 2013

You'll be writing a two-pass assembler.

1. **Pass 1** – analysis (build intermediate representation, construct symbol table).
2. **Pass 2** – synthesis (construct equivalent MIPS binary machine code and output it).

7.1 Pass 1 – Analysis

Pass 1 should generally follow this pseudocode:

location_counter = 0;
for *every line of input in source file* **do**
    read the line;
    scan line into a sequence of tokens;
    for *each LABEL at the start of the sequence* **do**
        if *LABEL is already in the symbol table* **then**
            output ERROR to standard error;
            quit;
        end
        add(label, location_counter) to symbol table
    end
    if *next token is an OPCODE* **then**
        if *remaining tokens are NOT exactly what is required by the OPCODE* **then**
            output ERROR to standard error;
            quit;
        end
        output a representation of the line to the intermediate representation (which can be text, a token sequence, etc.);
        location_counter += 4;
    end
end

7.1.1 Efficient Symbol Tables

Scheme:

```
(define st (make-hash))
(hash-set! st ‘‘foo’’ 42)
(hash-ref st ‘‘foo’’ #f) ; returns #f if key not found

(hash-ref st ‘‘foo’’ #f) => 42
(hash-ref st ‘‘bar’’ #f) => #f
```

C++:

```
using namespace std;
#include <map>
#include <string>

map<string, int> st;
st[‘‘foo’’] = 42;

// Incorrect way of accessing elements:
x = st[‘‘foo’’]; // x gets 42
y = st[‘‘bar’’]; // y gets 0, (bar, 0) gets added to st.
```

```
// Correct way of accessing elements:
if (st.find(``biff'') != st.end()) { ... not found ... }
```

Why do we use `map`s? It's more efficient because it converts strings into numbers in order to make lookups more performant.

7.1.2 The Supplied Scanners

The supplied scanners will find a sequence of tokens on the given line for you. For each token, you'll get a tuple (kind, lexeme, value), where the lexene is the literal text of the token. For example, take the line `foo: beq $3, $6, -2\verb`. The provided scanners will return a list with this data:

- (LABEL, "for:", ?)
- (OPCODE, "beq", ?)
- (REGISTER, "$3", 3)
- (COMMA, ",", ?)
- (REGISTER, "$6", 6)
- (COMMA, ",", ?)
- (INT, "-2", -2)

7.2 Pass 2 – Synthesis

Pass 2 should generally follow this pseudocode:

location_counter = 0;
for *each OPCODE in the intermediate representation* **do**
    construct corresponding MIPS binary instruction (inside of an int variable);
    output the instruction;
    location_counter += 4;
end

7.2.1 Creating MIPS Binary Instructions

We can form a MIPS binary instruction using a template and appropriate values for any variables in the command (usually denoted s, t, and d). The template is the integer that represents the binary associated with a particular instruction, with all of the variables (s, t, and d) being zeroes.

Bitwise operations compare integers bit by bit and perform the requested operation at that level. For example, given the numbers a and b that can represented in binary as 000111000 and 11011000, respectively, $a|b$ (a bit-wise OR) will result in 11011100 and `a & b` (a bit-wise AND) will result in 00011000.

Suppose we want to zero out all but the last byte of b. We'd do `b & 255`, which is the same as `b & 0xff`.

In Scheme, `(bitwise-and a b)` and `(bitwise-ior a b)` are the provided utilities for performing bit-wise operations. In C++, you use `a & b` and `a | b`.

Shifting bits can also be useful. You can use `a << n` in C++ to perform a left-shift by n bits (adding n zeroes to the right, discarding from the left). C++ also supports `a >> 5`, which adds n zeroes to the left and discards from the right. Scheme supports left and right shifts using `(arithmetic-shift a n)`, where positive n shifts right and negative n shifts left.

To fill our template with the appropriate s, t, and d values, you would perform an operation like this:

```
template | (s << 21) | (t << 16) | (0xffff & i)
```

Note that this adjusts s to the proper position, which in this case is 21 bits from the end. The `0xffff` which is ANDed with i is 16-bits, as required by i.

You can output bytes using shifts and several output calls. Here's some sample code for outputting a byte in C++:

```
void outbyte(int b) {
   putchar(b >> 24);
   putchar(b >> 16);
   putchar(b >> 8);
   putchar(b >> 4);
}
```

Here's similar code for Scheme:

```
(define (outbyte b)
   (write-byte (bitwise-and (arithmetic-shift b -24) 255))
   (write-byte (bitwise-and (arithmetic-shift b -16) 255))
   (write-byte (bitwise-and (arithmetic-shift b -8) 255))
   (write-byte (bitwise-and b 255)))
```

7.3 Assembler Requirements

← January 25, 2013

Your assembler is a Scheme/C/C++/Java program. You could also submit it in Scala, if you like.

In the end, your assembler should be able to be run such that it takes a `.asm` file as standard input, and produces MIPS binary code as standard output. That is, you should be able to run it like so:

- Scheme: `racket asm.ss < myprog.asm 1> myprog.mips 2>myprog.err`
- C++:

```
g++ asm.cc
valgrind --log-file=foo ./a.out < myprog.asm 1> myprog.mips 2> myprog.err
grep ‘‘ERROR SUMMARY’’ foo
```

For C++, leak checking is turned off. You still shouldn't leak memory all over the place, though.

`grep ‘‘ERROR’’ myprog.err` should not match anything if your program is valid, or should match otherwise (if there's an error).

To check the accuracy of your assembler, you can compare it with `cs241.binasm`:

```
java cs241.binasm < myprog.asm > myprog.ref.mips
diff myprog.mips myprog.ref.mips
```

This should give no output. Remember: there is a one-to-one mapping of valid MIPS assembly code to valid MIPS binary machine code.

You're going to have to make a large test suite to test your assembler. For valid code, you only need a handful of test cases. However, for error cases, you'll need a separate test case for each error since our assembler will quit as soon as it encounters the first error. Write a test script to run all of these cases, because it'll become tedious otherwise. You can share test cases you make with others, however you are not allowed to share information about how Marmoset test cases Marmoset uses to test your code.

We're going to take a look at the specification of the CS 241 dialect of MIPS. You'll likely want to print this out and mark every statement/condition twice: once after you implement it and once after you test it. [Note: you should look at the specification itself rather than relying on the notes here.]

7.3.1 Locations and Labels

Locations start at $0, 4, 8, \ldots$. The size of a program is the number of instructions multiplied by four. Locations are typically written in hex.

Each line may or may not contain labels, an instruction, and a comment. All three are optional. You can have none, one of these, two of these, or all three of these.

Lines without an instruction are known as **NULL lines**. Each non-NULL line has output into machine code.

Labels must occur at the beginning of the line. You may have zero of more labels on a line. Multiple labels are permitted on a single line. A label looks like this: `foo: ...`

Any given label can only be given a single definition in a MIPS program. You cannot have a label defining multiple lines or locations.

The **location** of a line is simply $4 \times$ (number of non-NULL lines that precede it). For example:

```
; hello                       0
foo:                          0
  add $1, $8, $3              0
bar: bif:  ; hello again      4
  lw $7, 0($2)                4
jr $31                        8
.word 32                      12
```

Note that all lines have a location (including NULL lines), however some lines may have the same location due to non-NULL lines. You are guaranteed that every line that contains an instruction (that is, every line that will be converted to machine code) will have a distinct location associated with it.

You should have a location counter variable which is incremented by 4 after you translate each instruction.

A **label** is simply a mapping between an identifier and a location.

7.3.2 Comments

Comments are ignored entirely. The provided scanners will throw away these comments for you.

7.3.3 Instructions

An **instruction** is an opcode (name of command) followed by zero or more operands.

After a label, you must either hit the end of the line (either it was a blank line, or a line with comments that were already stripped away) or an opcode.

For assignment 3, you can assume `.word` is the only opcode in existence.

If you do encounter a valid opcode, ensure it has the proper operands. The existence, types, and ranges of all operands must be verified for acceptability. Also, you must ensure there are no extra operands.

There are several types of operands:

- Registers.
- Unsigned numbers.
- Signed numbers.
- Hex numbers (`0x...`; the case of a to f does not matter).
- Labels (the *use* of labels, not a label definition).

7.3.4 Operand Format

Each instruction has specific requirements for its operands. Some instructions have the same requirements for their operands as other instructions, so they can be grouped together to remove duplication. In the second pass, you simply use a different template to fill different instructions from within the same grouping.

- `add, sub, slt, slt` – `add`, register, comma, register, comma, register, nothing else.
- `mult, div, multu, divu` – `mult`, register, comma, register, nothing else.
- `mfhi, mflo, lis` – `mfhi`, register, nothing else.
- `lw, sw` – lw, register, comma, i, bracket(, register, bracket), nothing else, where i can be an unsigned, negative, or hex number within the 16-bit range. Test cases needed: in/valid with unsigned, in/valid with hex, etc. Note: there is no rule stating that you must load a multiple of four.
- `beq, bne` – `beq`, register, comma, register, i, nothing else. i in this case can be an unsigned, negative, or hex number in the 16-bit range, or it could be the use of a label. In the case of a label, you calculate the numerical value by calculating: $\frac{\text{location(label)} - \text{location counter} - 4}{4}$.
- `jr, jalr` – `jr`, register, nothing else.
- `.word` – `.word`, i, nothing else, where i can be an unsigned, negative, or hex number in the 16-bit range, or it could be the use of a label.

8 Loaders, Relocation, and Linkers

← January 28, 2013

8.1 Loaders

For the use of this section, we have `ft.asm`:

```
ASSEMBLY            RAM         LOCATION        BINARY
lis $3              0           0               00001814
.word ft            4           4               00000010
lw $3, 0($3)        8           8               8c630000
jr $31              c           c               03e00008
ft: .word 42        10          10              0000002a
```

After assembling `ft.asm`, you'll have a MIPS binary machine program (say, `ft.mips`). But how does the program get into the CPU? The program is sent to the IO registers of RAM, which sends it to a loader, which stores it in RAM and executes it for you.

The **loader** is itself a MIPS program. It has exist in the MIPS CPU (or in RAM accessible from the MIPS CPU). A loader is a program in RAM that:

- Figures out the length (n words) of a binary MIPS program.
- Finds n words of available RAM at some address α.
- Reads the file into RAM starting at α.

- Puts α (the address) into some register (let's say `$5`).
- Executes the program (`jalr $5`).
- Does something else afterwards (`mips.twoints` prints the values of all the registers and ends, for instance).

This raises another question: how does the loader itself get into RAM? In the old days, there were hardware switches on RAM that allowed you to physically punch words into RAM. Today, we have manufacturing processes that do this for us. Regardless, we still want our loader to be *really* small (in terms of the number of words).

Instead of hard-coding the loader into RAM, a mini-loader is written (with less code than a full loader would require) to load the loader. This process is known by various names, including:

- Initial Program Loader (IPI) in IBM mainframe land.
- Bootstrapping (bootstrap) in mini/micro.

The mini-loader is permanently stored in a portion of RAM that is read-only. The mini-loader is actually manufactured into RAM.

Exercise: write the smallest file `mini.mips` such that you can write an arbitrary program and run it like this: `java mips.twoints mini.mips < program.mips`.

8.2 Relocation

Up until this point, we have assumed that $\alpha = 0$. That is, the location in the program has been equal to its address in RAM. Suppose instead that $\alpha = 8$. In fact, we can test this with `mips.twoints` like so: `java mips.twoints ft.mips 8` (the last parameter is the value for α, which must be a multiple of 4).

In the case where $\alpha = 8$, the `ft.mips` program would set `$3` to `0x8c63000`, which isn't correct. We need to offset our label values when $\alpha \neq 0$.

We use "sticky notes" to indicate where α needs to be added. We encode these sticky notes using a **MERL file** (MIPS executable relocatable linkable file). A MERL file is a binary file with three components:

1. **Header**. A MERL header is three words. The first word is a cookie, which is an arbitrary identifier that indicates that this is a MERL file. The cookie for MERL files is `0x10000002`. Note that `0x10000002` is the hex representation of `beq $0, $0, 2`, which means our program will execute normally if the MERL file is itself run directly where $\alpha = 0$. The second word is the size of the header, MIPS code, and sticky notes (indicates end/length of file). The third word is the size of the header and MIPS code (indicates the location of the first sticky note).
2. **MIPS code**. This is regular binary MIPS code, starting at location `0c`.
3. **Sticky notes**. A sticky note is two words: a sticky-note-type indicator (in this case it's 1 to indicate the sticky note means to add α), and the location in MERL to add α to.

Here's an example MERL file:

```
ASSEMBLY            RAM         LOCATION        BINARY
.word 0x10000002    8           0               10000002
.word 32            c           4               00000020
.word 40            10          8               00000028
lis $3              14          c               00001814
.word ft            18          10              00000010
lw $3, 0($3)        1c          14              8c630000
jr $31              20          18              03e00008
ft: .word 42        24          1c              0000002a
.word 1             28          20              00000001
.word 0x10          2c          24              00000010
```

← January 30, 2013

For more information about MERL, you should consult the MERL specification.

The easy way to encode a MERL file is to put a label at the end of the code, and another label at the end of the whole file (after your sticky notes).

The general process for creating a MERL file was to take your MIPS assembly, add the MERL header and sticky notes into an augmented asm program, then run it through `cs241.binasm` to get a `.merl` file. However, you can use `java cs241.linkasm` instead to do all of this for you.

8.3 Linkers

If you want to use `proc.asm` in multiple programs, you don't want to have to re-assemble it over and over again. With the procedure process that involved `cat` (as previously discussed), we had to worry about identifier conflicts when defining labels. Linkers solve both of these problems.

A **linker** allows for the separate assembly of procedures. You can assemble `main.asm` and `proc.asm` separately with `cs241.linkasm` to generate two MERL files (`main.merl` and `proc.merl`), then you can run both files through a linker and get `both.merl`. You needn't worry about identifier conflicts for re-assembling.

If you feed `main.asm` into `cs241.binasm` (or `cs241.linkasm`, or any other assembler), you'll get an error stating that you're using an undefined identifier (label). `cs241.linkasm` provides a mechanism for specifying that certain identifiers will be defined later.

```
main.asm:
  .import proc
  lis $1
  .word proc
  jr $31
```

```
proc.asm:
   .export proc
   proc: jr $31
```

This is an example of the `.import` and `.export` features of `cs241.linkasm`. `.import proc` indicates to the assembler to that the `proc` identifier must be defined later. It creates a sticky note that says to place the value of `proc` into certain places of your program where the identifier is used. In `proc.asm`, we specify `.export proc`, which creates a note that

`proc` is available to other programs at location `0xc` (for instance).

`.import` and `.export` instructions create sticky notes with a specific format code (`0x11` for `.import` and `0x05` for `.export`), a location to be encoded (`.import`) or value (`.export`), and name (a 32-bit word containing the number of words that encode the name of the identifier, followed by those n words).

By convention, references to currently-undefined identifiers get zero-filled (`0x00000000`) prior to being defined.

```
beq $0, $0, 2 ; skip over header ; 0x00000000  0x10000002
.word endmodule                  ; 0x00000004  0x0000003c
.word endcode                    ; 0x00000008  0x0000002c
   lis $3                        ; 0x0000000c  0x00001814
   .word 0xabc                   ; 0x00000010  0x00000abc
   lis $1                        ; 0x00000014  0x00000814

reloc1:
   .word A                       ; 0x00000018  0x00000024
   jr $1                         ; 0x0000001c  0x00200008
   B:
   jr $31                        ; 0x00000020  0x03e00008
   A:
   beq $0, $0, B                 ; 0x00000024  0x1000fffe
reloc2:
   .word B                       ; 0x00000028  0x00000020

endcode:
.word 1      ; relocate          ; 0x0000002c  0x00000001
.word reloc1 ; location          ; 0x00000030  0x00000018
.word 1      ; relocate          ; 0x00000034  0x00000001
.word reloc2 ; location          ; 0x00000038  0x00000028
```

When a linker links two MERL files together, the resulting MERL file will contain a new header, code from file 1, code from file 2 (relocated to be after code 1), and sticky notes (combined and relocated). Code 1 does not need to be relocated because it was and remains at location `0xc`. Code 2 needs to be relocated to `0xc + (size of code 1)`.

8.4 What a Linker Does

- Creates a combined header.
- Concatenates all code segments.
  - Relocates code_i by $\sum_{j<i} length(\text{code}_j)$.
  - Relocates the locations in the sticky notes by the same amount.
- REL (relocation entry) entries are copied directly to the new MERL, once they're relocated.

- Relocated ESD (external symbol definition) entries are copies.
- For each ESR (external symbol reference):
  - If there is an ESD with the same name...
    - Store the value of the ESD at the location specified in the ESR.
    - Add REL entry with ESR location to the combined stickies.
  - Otherwise, if there is no ESD...
    - Add ESR to combined stickies.

  When the ESRs are all gone, you can run your program.

8.5 Assemblers, Linkers, and Loaders on Linux

← February 1, 2013

In this course, we've been using tools designed for a MIPS CPU. We'll now take a brief look at how assemblers, linkers, and loaders work on Linux machines instead of on MIPS CPUs.

The **ELF format** is the Linux equivalent to MERL for MIPS. ELF means executable linkable format. ELF is a *really* complicated format. You can learn more about the ELF format by running `man elf` on a Linux machine.

The assembler on Linux is called `as`. Linux uses a different assembly language than MIPS because your CPU is probably an Intel or AMD CPU, not a MIPS CPU, so the assembly language differs.

The linker is called `ld`.

The loader is implicit. Fun fact: `a.out` was a format like ELF (but simpler), and it literally meant "the **out**put from the **a**ssembler."

8.6 Libraries

You may have a bunch of procedures that you want to use in many `asm` files. You can package these in a **library** so the linker can search the sticky notse for the ESD (external symbol definition) for the procedures it needs. Think of it as a match.com for MERLs.

How do you create an archive? It could be just a folder. But on Linux, you use `ar` (which is an abbreviation for "archive") which produces an archive similar to a zip file.

The standard libraries on Linux systems are stored at `/usr/lib`.

So far, we've been discussing **static linking**, where all external symbol references are resolved by combining with external symbol definitions to create REL entries. There are several shortcomings to static linking, however.

- If many programs use the same procedures/libraries, you get many copies of the procedures which wastes file space and RAM.
- If you change a library procedure, you have to find and relink every program that uses it.

An alternative to static linking is **dynamic linking**. Dynamic linking defers the resolution of ESRs until execution time. The library is searched when the ESR-referenced routine is called. Dynamic linking is implemented by statically linking a **stub** for each external symbol reference. The stub is effectively a pointer to the (external) dynamic library, which can be updated without changing references to the stub (i.e. without relinking).

Importantly, if multiple programs are running at once that use the same dynamically-linked library, the procedure will still only exist once in RAM. There won't be a separate instance of the library in RAM for each running program that needs it.

On Windows, a dynamic (link) library is called a **DLL**. If you're on Windows and have two programs that need the same DLL, but different versions of that DLL, you're in for a bad time. Windows also requires restarting in order to update DLLs, which is a fundamental design flaw. Linux, on the other hand, preserves old versions of dynamic link libraries even after newer versions are installed.

9 Formal Languages

A mathematical definition of what is and is not in a language. Formal language is useful for:

- Automation (if possible).
- Recognizers (human recognizers, too).
- Parsers (prove that X is valid per a mathematical definition).
- Translators (translate to an equivalent representation in another language).

← February 4, 2013

We need mathematical notions of what's in and not in a language.

Definition. A **formal language** L is a set of strings on a finite alphabet Σ.

Definition. An **alphabet** Σ is a finite set of symbols.

For example, $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ or $\Sigma = \{\text{dog}, \text{cat}, \text{monkey}\}$. Note that an alphabet can contain any arbitrary symbols – it does not need to consist of just single Roman characters.

Definition. A **string** is a sequence whose elements are symbols. The **empty string** is a sequence with zero elements, typically denoted ϵ or λ.

In this course, we'll denote the empty string as ϵ.

Language L is a set of strings. $L_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \ldots\}$, $L_2 = \{0, 1, 00, 11, 000, 111, 0000, 1111, \ldots\}$, and $L_3 = \{\text{dogcat}, \text{catdog}, \text{dogdogmonkey}\}$ are three examples of languages. It's important to note that unlike an alphabet, a language can be infinite in size.

There is a certain imprecision in the definitions of L_1 and L_2. "..." is not mathematically precise. For example, does L_1 contain counting numbers with no extra (padding) zeroes ... except zero? It gets complicated. There are also some even more complicated languages:

- $L_1 = \{$set of coded messages sent by NASA on probes$\}$. We could ask NASA, but it still gets complicated. What if a probe had a coded message on-board but the probe blew up on the launchpad? We need to be more precise in our definition.
- $L_2 = \{$set of coded messages that aliens will understand$\}$. This involves mind-reading.
- $L_3 = \{$set of all MIPS binary programs that halt for all inputs$\}$. You can't write a program to determine membership in this language.

Some languages are clearly easier to specify (precisely) than others.

Noam Chomsky came up with a hierarchy that expressed four different types of formal languages.

- Regular languages (easiest, nice mathematical properties, but limited).
- Context-free languages (more general, tools are available to work with these but they're harder to use).
- Context-sensitive languages (even more broad, harder to work with).
- Unrestricted languages (you can specify what you want to be in the language but you cannot build a computer to specify it).

9.1 Regular Languages

There are two equivalent definitions for regular languages: a generative definition (a method for building languages) and an analytic definition (how you can build a recognizer for the language).

Definition. A language L on alphabet Σ is regular if *any* of the following are true:

- L is finite.
- $L = L_1 \cup L_2$ (that is, L is the union of regular languages L_1 and L_2).
- $L = L_1 L_2$ (that is, L is the concatenation of regular languages L_1 and L_2). More precisely, $L_1 L_2 = \{xy | x \in L_1, y \in L_2\}$.
- $L = L_1^\star$ of a regular language L_1. This is known as repetition or a Kleene closure. That is, $L = \{\epsilon\} \cup LL_1$. Note that L contains ϵ, L_1, and every element of L being concatenated with L_1 continuously. Alternatively, this could be expressed as $L^\star = \{\epsilon\} \cup L_1 \cup L_1 L_1 \cup L_1 L_1 L_1 \cup \ldots$.

Definition. A language L is regular if it can be recognized by a computer with finite memory. If you need a stack or another unbounded data structure, then it is not regular.

For the analytic definition, we use an abstraction known as **finite state machines** or finite automaton to represent *any* machine with finite memory.

9.1.1 Deterministic Finite Automaton (DFA)

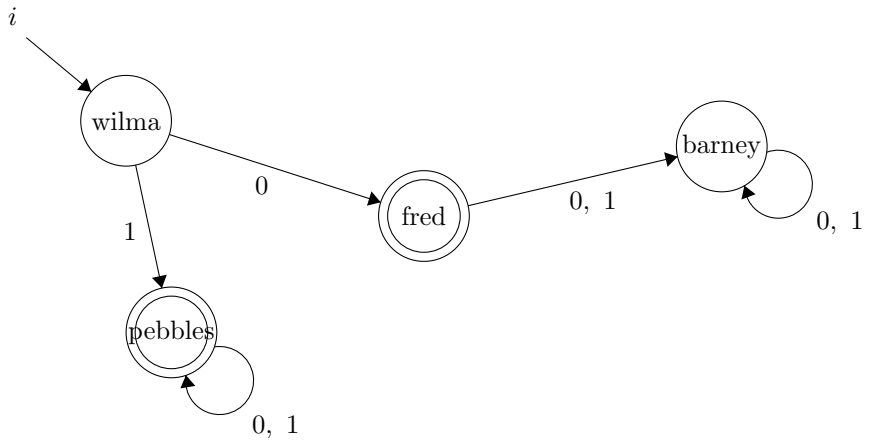
You have an alphabet Σ, a finite set of states S, an initial state $i \in S$, a set of final states $f \subseteq S$, and a transition function $T : S \times \Sigma \to S$.

For example: $\Sigma = \{0, 1\}$, $S = \{\text{wilma}, \text{fred}, \text{pebbles}\}$, $i = \text{wilma}$, $f = \{fred, pebbles\}$, and T is defined by the table:

$S \times \Sigma$	S
wilma, 0	fred
wilma, 1	pebbles
pebbles, 0	pebbles
pebbles, 1	pebbles

This is a partial function T. It's tedious to construct a table like this, and it doesn't easily convey the information to us in a visual way. Instead, we could use a bubble diagram.

A bubble diagram is more visual way to illustrate a finite state machine / finite automaton. You draw a circle for each state, an arrow for the initial state, and arrows between the circles to represent the transition function. You indicate the final states in some way, such as a double circle.



You follow the arrows through the states, as necessary, and if you end up on a final state then the element is in the language.

We prefer total functions to partial functions. We add a new state, which we'll call "barney", that acts as a black hole. A finite automaton only needs one black hole, where all states loop back to itself. We can always make T total by directing transitions that are not otherwise specified to the black hole. The black hole state is *not* a final state.

We can name our states whatever we'd like on our bubble diagram. Intelligent names should be chosen, like "start", "zero", "nonzero", and "fail", for instance.

Fun fact: regular languages have been proven to be closed under intersection and set difference.

← February 6, 2013

We sometimes denote a DFA as DFA $=< \Sigma, S, i, f, T >$.

For simplicity's sake, we'll setup some notation we'll be using to discuss DFAs. This is by convention only.

- a, b, c, d (and other letters at the beginning of the standard alphabet) represent symbols in $\Sigma = \{a, b, c, d\}$.
- Particular strings are a concatenation of symbols from Σ. Some examples are $abca$, $bbca$, and ϵ (the empty string).
- x, y, z (and other letters near the end of the standard alphabet) are variables that represent strings. For example, let $x = abc$, then $x = x_0x_1x_2 \dots x_{n-1}$ where $n = |x|$.

The **DFA algorithm** is pretty simple. Given an input x (which is implicitly a DFA $= < \Sigma, S, i, f, T >$, where T is a total function), the output of the algorithm will be:

$$\begin{cases} \text{“accept”} & x \in L \\ \text{“reject”} & x \notin L \end{cases}$$

The DFA algorithm is as follows:

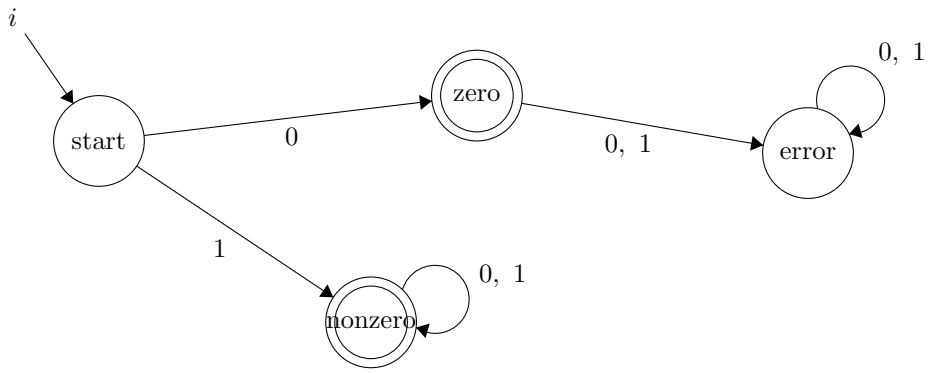
state = i;
for $a = x_0, x_1, x_2, \dots, x_{n-1}$ **do**
  state = T[state, a];
end
if *state* $\in f$ **then** accept;
else reject;

The bonus question on A5 involves implementing this algorithm. You may want to use an array or map. It's the easiest bonus question all term.

Example 9.1. We're given the binary integers, $\Sigma = \{0, 1\}, S = \{\text{start}, \text{zero}, \text{nonzero}, \text{error}\}, i = \text{start}, f = \{\text{zero}, \text{nonzero}\}$, and function T defined by:

$S \times \Sigma$	S
start, 0	zero
start, 1	nonzero
zero, 0	error
zero, 1	error
nonzero, 0	nonzero
nonzero, 1	nonzero
error, 0	error
error, 1	error

This table for T isn't very intuitive. Let's look at the bubble diagram representing this DFA for a clearer picture.



Let's look at x = 10. x = 10 will execute the following:

state = start;
state = T[start, 1]; // state = nonzero
state = T[nonzero, 0]; // state = nonzero
(end loop)
nonzero ∈ f? Yes, accept.

Next, let's look at x = 01. x = 01 will execute the following:

state = start;
state = T[start, 0]; // state = zero
state = T[zero, 1]; // state = error
(end loop)
error ∈ f? No, reject.

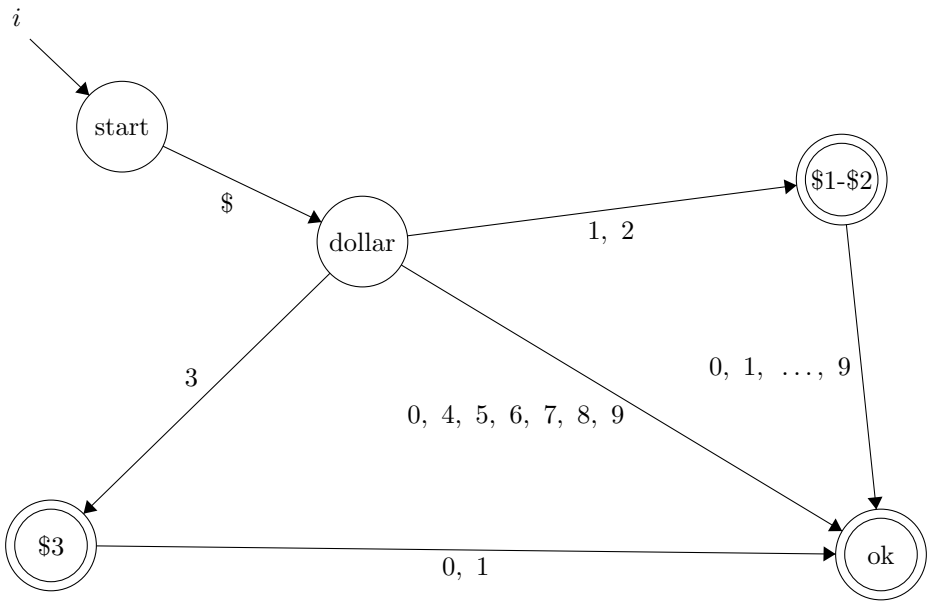
Example 9.2. Let's construct a finite automaton that represents the MIPS assembly notation for registers.

Given $\Sigma = \{\$, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, L = MIPS assembly notation for registers (i.e. $\{\$0, \$1, \$2, \ldots, \$31\}$).

It's useful to name your DFA states with what you've learned so far based on the conditions that led to that state, because that's the only way we know what we've determined in the past. These names don't matter, but it'll make the finite automaton much easier for you to understand.

We're going to assume there is an error state that all undefined transitions lead to. We'll make this assumption in general in this course. You need to draw the error state only when you need a *total* function T (such as when you intend to run the DFA algorithm), or when you're explicitly asked to draw it.

Let's look at the bubble diagram for this finite automaton that represents the notation of MIPS registers.

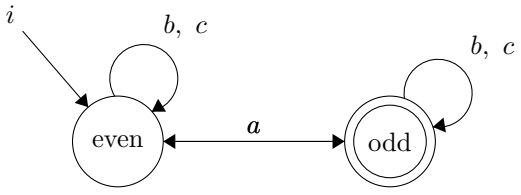


Every MIPS register must start with the dollar sign ($). Keep in mind that only $\$0, \ldots, \$31$ are valid MIPS registers. So, if the next number is a 1 or a 2, we can have any second digit (from 0-9) to form a valid MIPS register. If the number is 3, we can only follow that up by a 0 or 1 ($30 and $31). We also need to handle all other single digit registers ($0, $4, $5, $6, $7, $8, $9).

We could've made this differently by having multiple "ok" states, but those aren't necessary so all valid registers point to the same "ok" state in this diagram. We can point all complete paths to a single complete state because what happens from the complete state onwards does not differ. In this case, all complete states have no further path (except to an implied black hole), and all complete states are final states, so having one complete state (denoted "ok" on the bubble diagram) is acceptable.

In general, you can combine two states if everything that follows from that point is the same among all of the states you're combining.

Example 9.3. Let $\Sigma = \{a, b, c\}, L =$ any string with an odd number of a's. For example: $a, abaa, babbc, \ldots$.

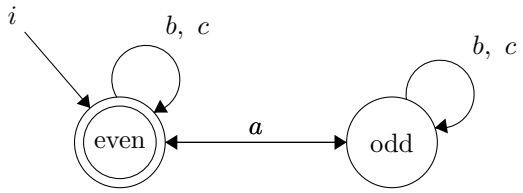


(Note that there are two arrows between those nodes: one for each direction.)

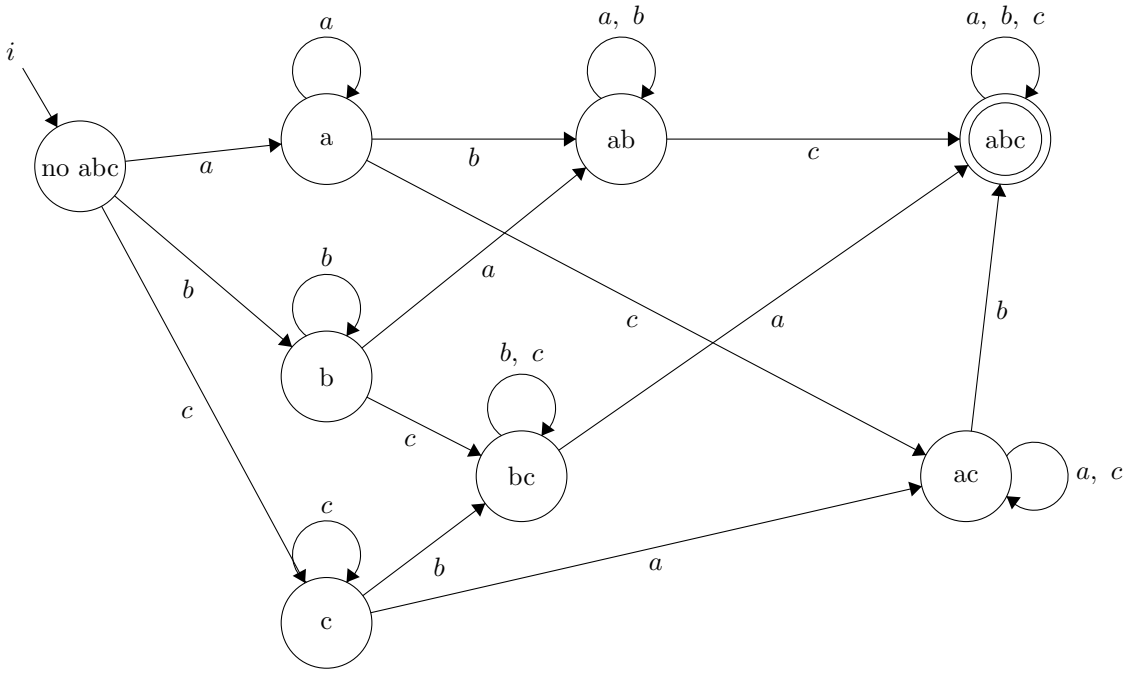
We can keep track of one binary bit of information with a state. In this case, that bit is "do I have an even number of a's?"

Suppose instead we want L = any string with an even number of a's. That's the complement of L from before.

To find the complement of a DFA, you make all non-finish states into finish states, and vice versa. That means the error (black hole) state will become a finish state as well.



Example 9.4. Let $\Sigma = \{a, b, c\}$ and L = all strings with at least one a, one b, and one c. For example: $abaaaaaca \in L$, but $bcbcbccc \notin L$.



T in this case is a total function. It has eight states because we need to keep track of three binary facts, which requires three bits of information ($2^3 = 8$).

Similarly, if we had a DFA to determine if every letter in the English alphabet is included in a particular string, we would need 2^{26} states because we would need to store 26 bits of information. At some point, we say a number is large enough that *in practice* we can treat it as an infinite number, despite it being finite. We wouldn't solve this problem with a DFA because 2^{26} is a number large enough that we wouldn't want to create a state for each of them.

9.1.2 Searching and Scanning with DFAs

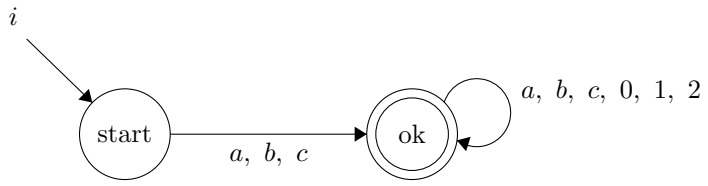
← February 11, 2013

- **Recognition** answers the question "is $x \in L$?" This is what we've used DFAs for so far.
- **Search** answers the question "does x contain $y \in L$?". More formally, Does $\exists w, y, z.x = wyz$ and $y \in L$?
- **Find** answers the question "where in x is $y \in L$" (where $x = x_0x_1 \dots x_{|x|-1}$)? This question could have multiple answers, depending on what you're asked for, such as:
  - y begins at x_i.
  - y ends at x_j.
  - y begins at x_i and ends at x_j. This can be expressed as the pair (i, j).

  Find is a more general result than search.
- **Scan** partitions x into $x = y_0y_1y_2y_3 \dots y_n$ and $\forall i y_i \in L$.

Example 9.5. Let $\Sigma = \{a, b, c, 0, 1, 2, !\}$ be an alphabet. We're interested in L_{id} – that is, the language of identifiers.

We can determine a recognizer for L_{id}:



Searching for L_{id} in x, where $x =!abc!cba!$ yields the vacuous result of "yes!". There is an identifier in x, somewhere.

Some possible answers for find include:

- $i = 1$ is the start of $y \in L$.
- $i = 5$ is the start of $y \in L$.
- $i = 3$ is the end of $y \in L$.
- $i = 7$ is the end of $y \in L$.
- $(i, j) = (1, 3)$ is $y \in L$.
- $(i, j) = (5, 7)$ is $y \in L$.

Those are not the only answers for find, however. "bc" is a valid identifier, and it's contained within x. So, we have a number of additional solutions:

- $i = 2$ is the start of $y \in L$.
- $i = 3$ is the start of $y \in L$.

- $i = 1$ is the end of $y \in L$.
- $i = 2$ is the end of $y \in L$.
- etc.

There are $O(|x|)$ possible solutions that indicate where $y \in L$ either begins or ends. There are also many additional pair solutions for the same reason:

- $(i, j) = (1, 1)$ is $y \in L$.
- $(i, j) = (1, 2)$ is $y \in L$.
- $(i, j) = (1, 3)$ is $y \in L$.
- $(i, j) = (5, 5)$ is $y \in L$.
- $(i, j) = (5, 6)$ is $y \in L$.
- $(i, j) = (6, 7)$ is $y \in L$.
- etc.

There are $O(|x^2|)$ possible (i, j) pair solutions.

We would like to specify a unique solution. How could we do this? First, you have to choose whether you want i, j, or $\overline{(i, j)}$ as the format of your solution. If you want just i or j, then simply pick the first solution. But if you want the pair (i, j), "first" is ill-defined. For example: does $(10, 20)$ come before $(5, 25)$? There's no correct answer, it's all about definitions.

The most common choice for a unique solution is to use the leftmost longest match.

Definition. The **leftmost longest match** is found by finding (i, j) such that $x_i x_{i+1} \dots x_j \in L$, i is minimized, and given i, j is maximized.

A subproblem of the leftmost longest match algorithm is the longest prefix problem.

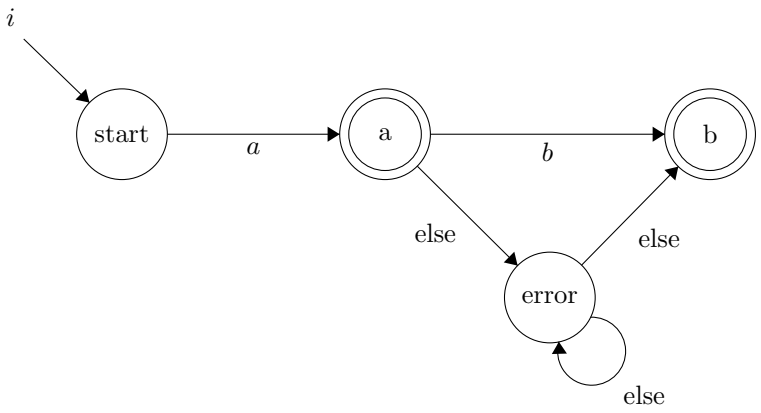
Definition. The **longest prefix problem** is when given x, find the longest prefix (of x), $y \in L$.

For example, if $x = abc123!abc$, the longest prefix is $abc123$. This can be identified by j, where x_j is the end of the prefix of y.

Example 9.6. Suppose $L = \{a, aaa\}$. Let's find the longest match.

If we have $x = aa \dots$, after we examine the first a, we have no idea if that's the longest match or not. Only after examining more of the string can we be sure that it is. In aab, a is the longest match. In aaa, aaa is the longest match and so a is not the longest match.

Example 9.7. Suppose $L = \{a\} \cup \{$ any string beginning with a and ending in $b\}$. We can determine a recognizer for this:



If we have $x = acc012cca12\ldots$ and we're examining a at the beginning, we're unsure if that's the longest prefix $y \in L$ yet. We have to continue looking to be sure.

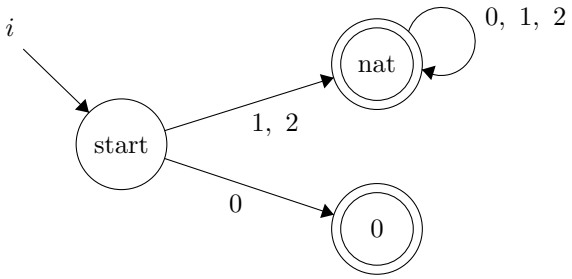
Here's the pseudocode for determining the longest prefix using a DFA, given an input $x = x_0x_1\ldots x_{|x|-1}$.

```
state = start;
j = -1;
for i = 0 to |x| - 1 do
    if state ∈ final then j = i;
    ;
    state = T[state, x_i];
    if state is blackhole then reject;
    ; // You could always hope... this is optional though.
end
if j ≥ 0 then accept, y = x_0 ... x_{j-1} ∈ L;
else reject;
```

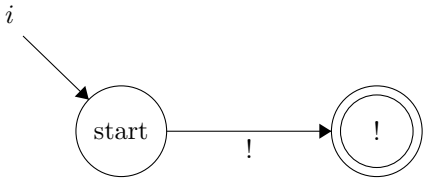
Next, let's take a deeper look at scanning. With scanning, we want to partition x using the longest prefix. This is also called leftmost longest, or more colloquially, the **maximal munch** scanning algorithm.

```
start with x;
while x ≠ ϵ do
    find leftmost longest match of L;
    report result y, remove y from x;
end
```

In this algorithm, L is the language of all tokens and separators. That is, $L = L_{\text{id}} \cup L_{\text{int}} \cup L_{!}$, where L_{int} has this recognizer:



And $L_!$ has this recognizer:



There is a problem with maximal munch – efficiency.

Runtime of Maximal Munch: the loop executes $|x|$ times, and for each iteration, it uses the "find largest prefix" algorithm (which itself takes $O(|x|)$ time). $|x| \cdot O(|x|) \in O(|x|^2)$.

In order to improve efficiency and readability, we'll use a "total hack" called **simplified maximal munch**. This involves finding the longest prefix y of x such that ya could not possibly be a prefix of L where $x = yaz$.

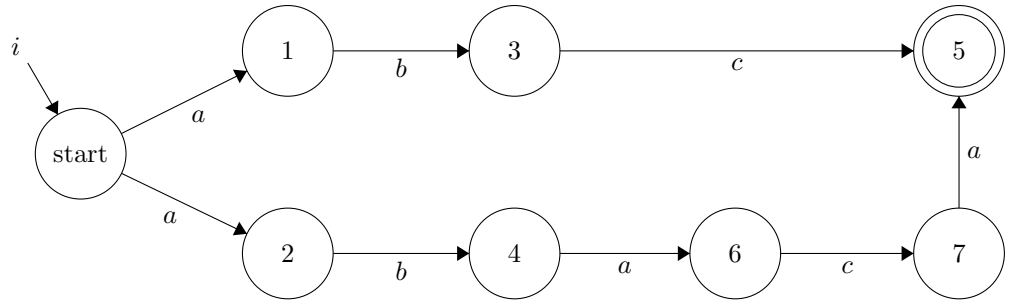
Suppose you have $L_{\text{int}} = \{0, 1, 2, 3, \ldots\}, L_{\text{id}} = \{x\}$, and $L_{\text{hex}} = \{0x1, \ldots\}$. If you have $x = 0x$, you can't determine if that's the integer zero followed by an identifier, or the beginning of a hex symbol. However, if you can look ahead and see $0x1$, you know it's a hex symbol. If you see something like $0xx$ (a hex prefix followed by an identifier), simplified maximal munch will not allow this.

The scanner we were given (which implements simplified maximal munch) will see $0xx$ as an error. It's an error because the scanner made the assumption that it was a hex constant, but that didn't work out (didn't see an integer after 0x), so it gave up and produced an error.

9.1.3 Nondeterministic Finite Automata (NFA)

← February 13, 2013

Example 9.8. Let's say we have the alphabet $\Sigma = \{a, b, c\}$ and the language $L = \{abc, abaca\}$. A recognizer for this NFA could be drawn as:



This is not a DFA because T[start, a] = 1 and T[start, a] = 2, which is not allowed in a DFA. It is acceptable for an NFA, though.

The transition function T no longer has to be a function – it can have multiple values for the same inputs. The rule is the same as before: if there is any path that is followed to a finish state, then the string is considered to be in the language. With DFAs, we used the "one-finger algorithm", but now we can't do that because there could be multiple paths we need to follow.

An **oracle** is a magic function that sees the future. In CS theory, if we have an oracle then you can resolve the proper path and use the same process as a DFA. However, we don't have an oracle. When we see multiple paths, we clone our finger and follow each path as far as we can. This is more operationally satisfactory than looking into the future.

More formally, an $NFA = (\Sigma, S, i, f, T)$, where Σ, S, i, and f follow the same definitions as with DFAs. However, $T : S \times \Sigma \to 2^S$ (where 2^S is the number of subsets of S). In Example 9.8, we have that T is defined by the table:

start, a	{ 1, 2 }
1, b	{ 3 }
2, b	{ 4 }
3, c	{ 5 }
4, a	{ 6 }
6, c	{ 7 }
7, a	{ 5 }
3, a	{ }

Additionally, we have $\Sigma = \{a, b, c\}, i = start, S = \{start, 1, 2, 3, 4, 5, 6, 7\}$, and $f = \{5\}$.

Note that we no longer need an error state because we are no longer required to make T a total function, since it is doesn't need to be a function at all.
NFA recognizer: given an input string x, we expect output to be:

$$\text{output} = \begin{cases} \text{“accept”} & x \in L \\ \text{“reject”} & x \notin L \end{cases}$$

The pseudocode for an NFA recognizer is as follows.

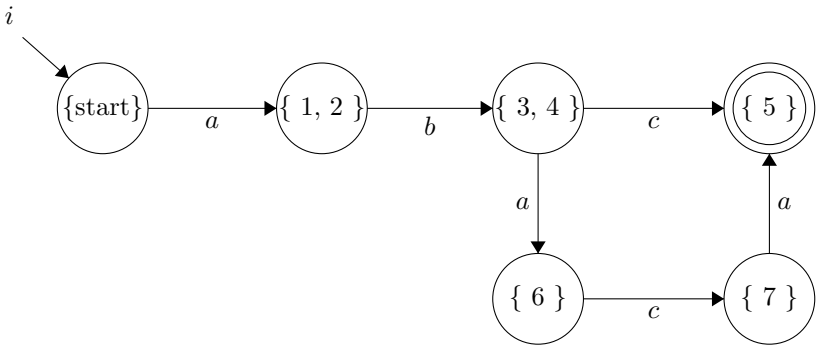
states = { start };
for *a in* $x_0x_1x_2 \ldots x_{|x|-1}$ **do**
  states = $\bigcup_{s \in \text{states}}$ T[s, a];
end
if *states* $\cap f \neq \{\}$ **then** "accept";
else "reject";

Theorem. Any language recognized by an NFA can be recognized by a DFA.

Proof outline: subset construction. For an $NFA = \{\Sigma, S_{NFA}, i_{NFA}, f_{NFA}, T_{NFA}\}$, we can construct a $DFA = \{\Sigma, S_{DFA}, i_{DFA}, f_{DFA}, T_{DFA}\}$. Why does this work?

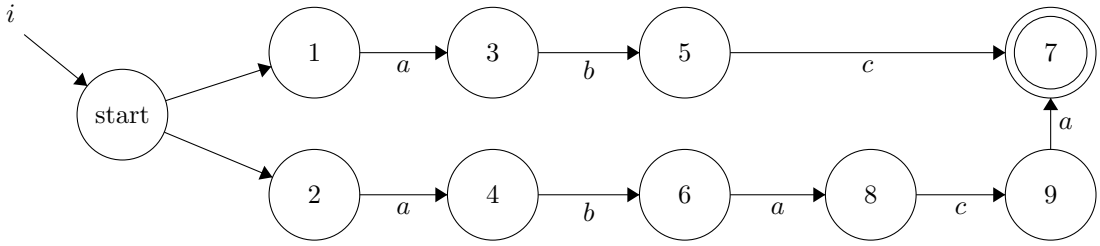
1. S_{NFA} is finite.
2. The number of subsets of a finite set is finite. There are $2^{|S|}$ subsets of a finite set S.
3. Each state in S_{DFA} represents a subset of the states in S_{NFA}.

From the NFA in Example 9.8, we can create this corresponding DFA:



9.1.4 Epsilon Nondeterministic Finite Automata (ϵ-NFA)

Example 9.9. This is an example of an ϵ-NFA:



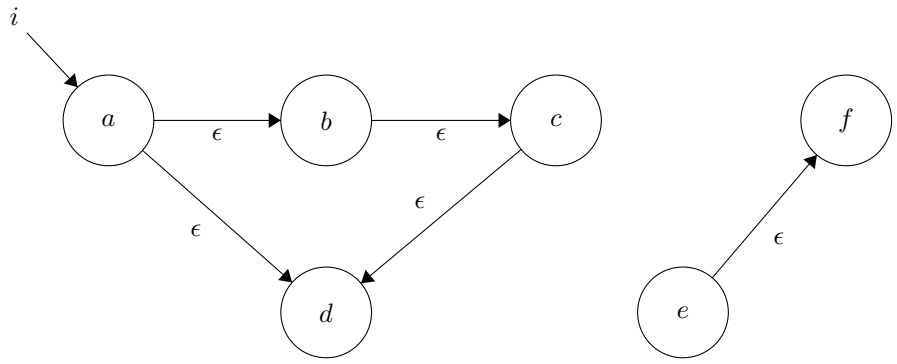
The difference between ϵ-NFAs and standard NFAs is that the transitions from the start state to states 1 and 2 have no symbol. That means you can spontaneously move your finger along those paths. We generally label those transitions as ϵ to prevent confusion between mistakes (omissions in labeling the transitions) and ϵ transitions. This recognizer is for the same language as the NFAs and DFAs in Example 9.8.

More formally, an $\epsilon - NFA = (\Sigma, S, i, f, T)$ where Σ, S, i, and f are defined the same way they were for DFAs and NFAs. The transition function is now defined as $T : S \times (\Sigma \cup \{\epsilon\})$.

ϵ-NFA recognizer:

We'll need a helper function called ϵ-closure, which accepts a parameter S (a set of states), and returns a set of all states that can be reached from a state in S by following ϵ-transitions. For instance, the ϵ-closure({start}) from earlier would return {start, 1, 2}.

Example 9.10. What is the ϵ-closure({a, e}) of the following ϵ-NFA?



ϵ-closure({a, e}) = {a, b, c, d, e, f}.

ϵ-closure(S) is actually a graph algorithm called **reachability**. The pseudocode of ϵ-closure(S) is as follows.

```
S' = S;
W = S; // work list
while w ≠ {} do
    select some element s ∈ W;
    W = W \ { s }; // remove s from work list
    for all s' ∈ T[s, ϵ] do
        if s' ∉ S then
            // New state we've never reached before.
            S' = S' ∪ {s'};
            W = W ∪ {s'};
        end
    end
end
```

More formally, ϵ-closure$(S) = S'$ where S' is the smallest solution to:

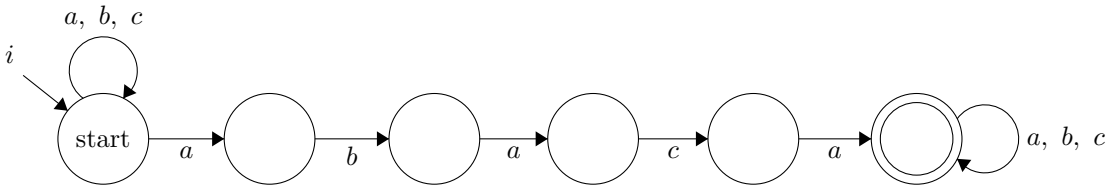
$$S' = S \cup \{s' | \exists s \in S', T[s, \epsilon] = s'\}$$

The inclusion of ϵ-closure is the only difference between the NFA and ϵ-NFA recognizers. Here's the pseudocode for an ϵ-NFA recognizer:

states = ϵ-closure({ start });
for *a in* $x_0x_1x_2 \ldots x_{|x|-1}$ **do**
  states = ϵ-closure($\bigcup_{s \in \text{states}}$ T[s, a]);
end
if *states* $\cap f \neq \{\}$ **then** "accept";
else "reject";

DFAs, NFAs, and ϵ-NFAs can all express recognizers for all regular languages, however sometimes one method is more straightforward than the others.

Example 9.11. Let L be the language containing all strings that contain "abaca" as a substring, where the alphabet is defined as $\Sigma = \{a, b, c\}$. This language can be recognized by a much simpler NFA recognizer than a DFA recognizer. Here's the NFA recognizer for this language L:



9.2 Regular Expressions

← February 15, 2013

Regular expressions are a textual notation to specify regular languages, based on a generative definition. Every regular expression can be translated into an ϵ-NFA. Any regular language can be expressed as a regular expression, but it's not always simple. However, it's a more intuitive way to express a language in typed form than using a typed bubble diagram.

Regular Expression R	Language $L(R)$	Comments / Examples
a	$\{a\}$	$a \in \Sigma$
ϵ	$\{\epsilon\}$	NULL string
R_1R_2	$L(R_1)L(R_2)$	Concatenation. e.g.: $L(ab) = \{ab\}$ $L(abc) = \{abc\}$ $L(\epsilon a) = \{a\}$
$R_1 \vert R_2$	$L(R_1) \cup L(R_2)$	e.g. $L(abc \vert def) = \{abc, def\}$
$R\star$	$L(R)^\star$	Kleene closure. e.g.: $(abc)\star = \{\epsilon, abc, abcabc, \ldots\}$
$\emptyset$	$\{\}$	Empty language.

Example 9.12. Given the alphabet $\Sigma = \{0, 1\}$, we're interested in the language $L(R)$ of all binary integers. The regular expression R would be defined as $R = 0|1(0|1)^\star$.

Example 9.13. Given the alphabet $\Sigma = \{a, b\}$, we're interseted in the language $L(R)$ of all strings with an even number of a's. Note that $L = \epsilon, babbab, \ldots\}$. We could define the

regular expression for this language in several different ways:

$$\begin{aligned} R &= (b^\star ab^\star ab^\star)^\star \\ &= b^\star(b^\star ab^\star ab^\star)^\star \\ &= b^\star(ab^\star ab^\star)^\star \\ &= (b^\star ab^\star a)b^\star \\ &= b^\star(b^\star ab^\star ab^\star)^\star b^\star \\ &\vdots \end{aligned}$$

This illustrates an important point: there are many ways to express every regular language with regular expressions. There is not necessarily a unique minimal regular expression for a language. Also, we don't care if the regular expression is ambiguous or not.

Example 9.14. Let $\Sigma = \{0, 1, +\}$, and $L(R)$ be the sum of one or more integers. We can express this as the regular expression R:

$$R = 0|1(0|1)^\star(+0|1(0|1)^\star)^\star$$

Note that + is an extended regular expression, but we'll assume it isn't for now. We'll talk more about that shortly.

Example 9.15. Let $\Sigma = \{0, 1, +\}$ (as before), but now let $L(R)$ be the sum of two or more integers. We can express this as the regular expression R:

$$R = 0|1(0|1)^\star + 0|1(0|1)^\star(+0|1(0|1)^\star)^\star$$

As you can see, this becomes tedious. There are some regular expressions that grow exponentially as you make minor changes like these.

9.2.1 Extensions

Extensions give a more terse (sometimes exponentially) notation but they still only specify the regular languages.

Extended RE	Meaning
R^+	$RR^\star$
$\{R_1\|R_2\}$	$R_1(R_2R_1)^\star$
$\{R_1\|\|R_2\}$	$R_1(R_2R_1)^\star$

(the vertical bars in the above table are fatter than normal)

The R^+ case is where we would see an exponential increase in notation if we weren't using extensions, if there were nested expressions containing +.

We could have expressed the earlier expression as $\{0|1(0|1)^\star|+\}$.

In practice, many people call regexps "regular expressions", but they aren't true regular expressions because they can express more than just the regular languages. When *we* say "regular expressions", we really mean true regular expressions (expressions that can only represent regular languages).

9.2.2 Regular Expressions to Finite Automata

Any regular expression can be converted to an ϵ-NFA that recognizes the language. In fact, that ϵ-NFA will always have exactly one final state. In addition, recall that any ϵ-NFA can be converted to a DFA using subset construction.

If you have the regular expression a, or a regular expression ϵ, we can express that as an ϵ-NFA by creating a start state that is joined to the final state by either a or ϵ, respectively.

If we have a regular expression R_1R_2 (where R_1 and R_2 are also regular expressions), we will have ϵ-NFAs for R_1 and R_2 already. We simply make the final state from R_1 no longer a final state, and connect it with ϵ to the start state of R_2.

If we have a regular expression $R_1|R_2$ (where R_1 and R_2 are also regular expressions), we will have ϵ-NFAs for R_1 and R_2 already. We create a new final state, and we make the final states of R_1 and R_2 connected to the new final state through ϵ. The final states of R_1 and R_2 are no longer final states themselves. We also create a new start state that connects to the start states of both R_1 and R_2 through ϵ.

If we have a regular expression $R^\star$ (where R is a regular expression), we will have an ϵ-NFA for R already. We add a new start state and a new final state, and make the final state of R no longer a final state. We connect the new start state to both the start state of R and the new final state, both through ϵ. The previously-final state of R is also connected to the new final state through ϵ.

If we have $\emptyset$, we create a start state and a final state and provide no path between the two.