CS 241: Foundations of Sequential Programs

Chris Thomson

Winter 2013, University of Waterloo

Notes written from Gordon Cormack's lectures.

Contents

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 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 different interpretations 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 (interpret as 0101, so 0 is the two's complement sign bit). Similarly, 1011 would be $-2^3 + 2^2 + 2^0 = -8 + 4 + 1 = -3$.
- **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 last 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 symbols in Unicode, 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 $2^{16} = 65,536$ possible ways 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-bits ($2^{16} = 65,536$ possible values), 32-bits ($2^{32} \approx 4 \times 10^9$), or 64-bits ($2^{64} \approx 10^{19}$).

A "64-bit CPU" just means it's a CPU that uses 64-bit words.

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.

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 of which is 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 locations 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 (`xxd -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 in binary MIPS instructions, since $2^5 = 32$.
- In documentation, it's conventional to call S and T **source registers**, and D 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. You can convert the binary instructions into hex and place the instructions into an assembly language file as `.word` instructions.

3.2.3 A Few Important MIPS Instructions

1. **Load Immediate & Skip** (`lis`): loads the next word of memory into the D register. You specify a `lis` instruction followed by an arbitrary word next. The program counter will then skip past the arbitrary word that follows, to avoid executing it.
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 the value in the source register 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 that can be executed.

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. It provides you with a way to include arbitrary bits in the assembler's output. 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 (the next word) into $d, 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.

```
; $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 $1 by 1
.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}).

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.

3.2.8 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 use this information to simplify the programmer's job 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 line 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 (`beq` or `bne`) and in places where they're specified absolutely (`jr`), may become invalid if you add or remove any lines in 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 for that space in memory. 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 using 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 C

loop:
  add $3, $3, $1 ; C = C + N
  lis $2
  .word -1
  add $1, $1, $2 ; C = C + (-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 in the code above, so we have a way to jump to the `sum.asm` line (which is part of how procedures are called in MIPS). This is `sum10.asm`:

```
; 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
```

Aside: 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. Note that the order of the `cat` arguments matters – you must place your runner program before the procedures (by listing it first in the `cat` command) so a procedure isn't executed prior to your runner program.

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 $1, -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 from/to the user's keyboard/monitor. `getchar` is located at memory address `0xffff0004` and `putchar` is at address `0xffff000c`. If you load or store a byte at either of these addresses, you will retrieve or send the byte from/to 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 getchar()
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.

You must ensure you implement and test every little detail 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 *some* 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.. Scheme is recommended, especially since you'll be dealing with data structures of lengths that aren't predetermined, 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 a 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 identifiers in place of registers, etc. You must also ensure that there aren't too many or too few arguments, but to perform this check you should simply 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. In 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 efficient.

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 lexeme is the literal text of the token. For example, take the line `foo: beq $3, $6, -2`. The provided scanners will return a list with this data:

- (LABEL, "foo:", ?)
- (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 bitwise OR) will result in 11011100 and `a & b` (a bitwise 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 available utilities for performing bitwise operations. In C++, you can 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'd 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 for the Marmoset tests. 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 the 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.

Label definitions must be 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 40            c           4               00000028
.word 32            10          8               00000020
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)`.

8.3.1 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.

8.4 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."

8.5 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 notes 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 Introduction to 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).

10 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 a **finite state machine** or finite automaton to represent *any* machine with finite memory.

10.1 Deterministic Finite Automaton (DFA)

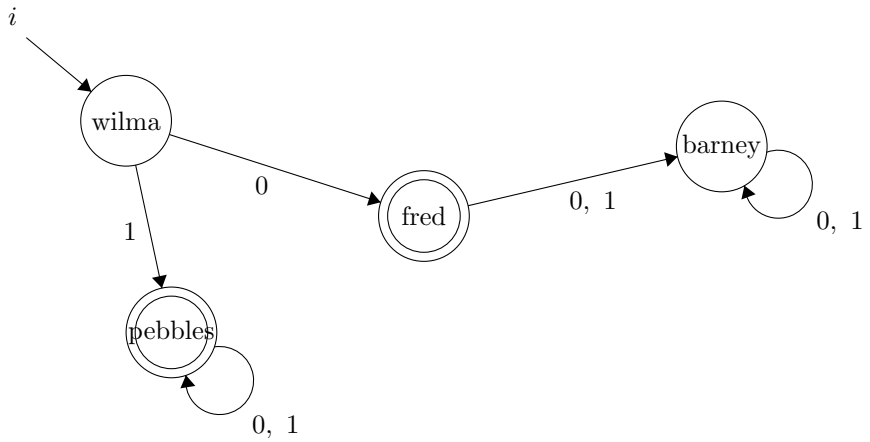
You have an alphabet Σ, a finite set of states S, an initial state $i \in S$, a set of final states $f \subseteq S$, and a transition function $T : S \times \Sigma \to S$.

For example: $\Sigma = \{0, 1\}$, $S = \{\text{wilma}, \text{fred}, \text{pebbles}\}$, $i = \text{wilma}$, $f = \{\text{fred}, \text{pebbles}\}$, 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:

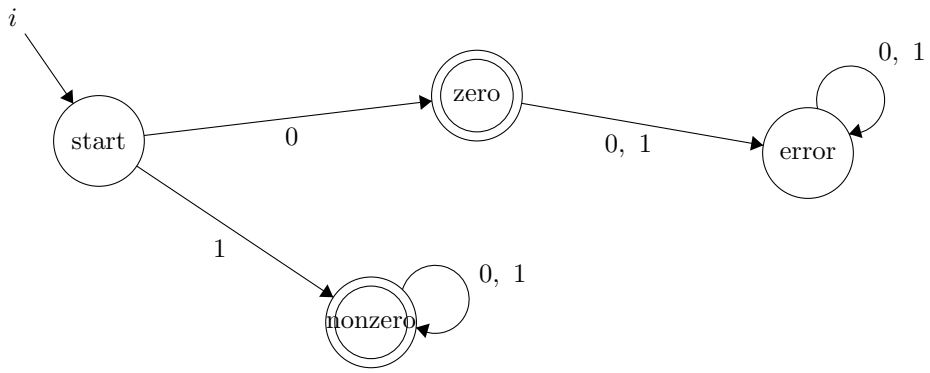
state = i;
for $a = x_0, x_1, x_2, \dots, x_{n-1}$ **do**
  state = T[state, a];
end
if *state* $\in f$ **then** accept;
else reject;

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 10.1. 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.

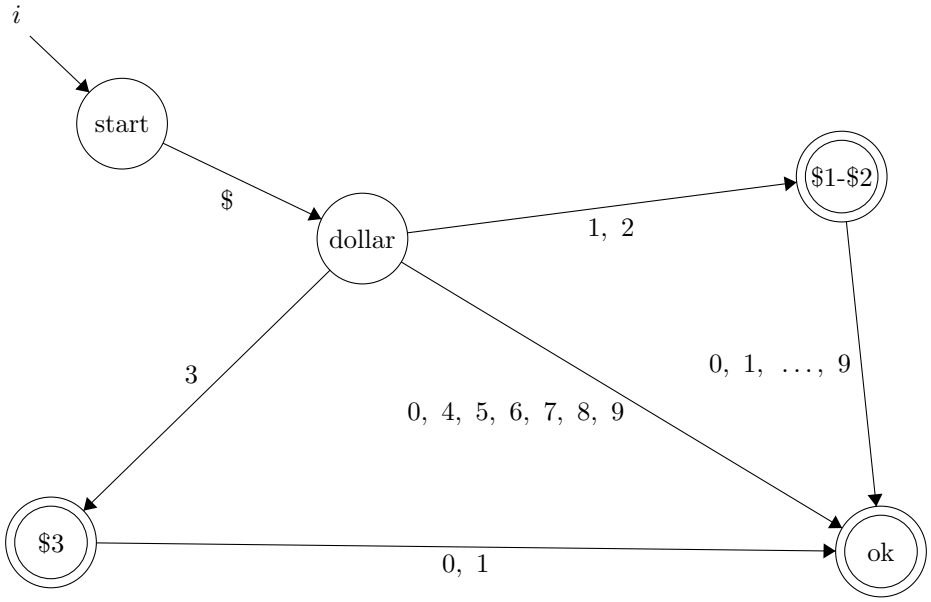
Example 10.2. Let's construct a finite automaton that represents the MIPS assembly notation for registers.

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.

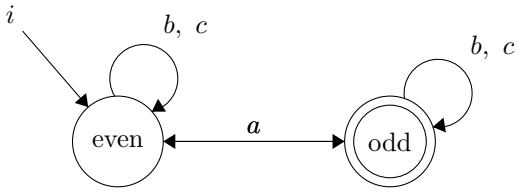


Every MIPS register must start with the dollar sign ($). Keep in mind that only $\$0, \ldots, \$31$ are valid MIPS registers. So, if the next number is a 1 or a 2, we can have any second digit (from 0-9) to form a valid MIPS register. If the number is 3, we can only follow that up by a 0 or 1 ($30 and $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 10.3. Let $\Sigma = \{a, b, c\}, L =$ any string with an odd number of a's. For example: $a, abaa, babbc, \ldots$.

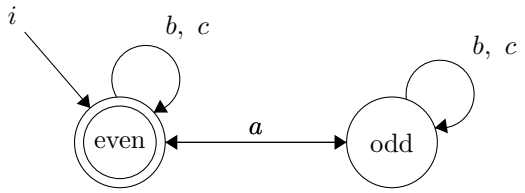


(Note that there are two arrows between those nodes: one for each direction.)

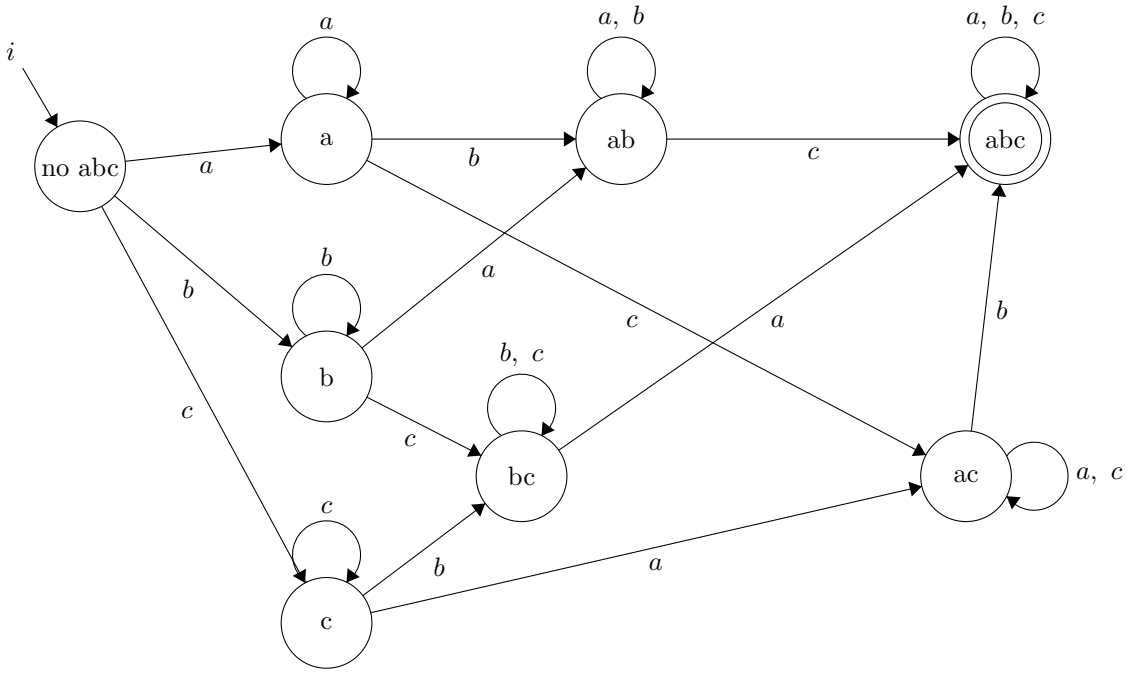
We can keep track of one binary bit of information with a state. In this case, that bit is "do I have an even number of a's?"

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 10.4. 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.

10.1.1 Searching and Scanning with DFAs

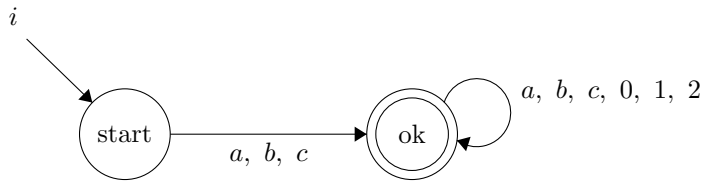
← February 11, 2013

- **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 10.5. 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.

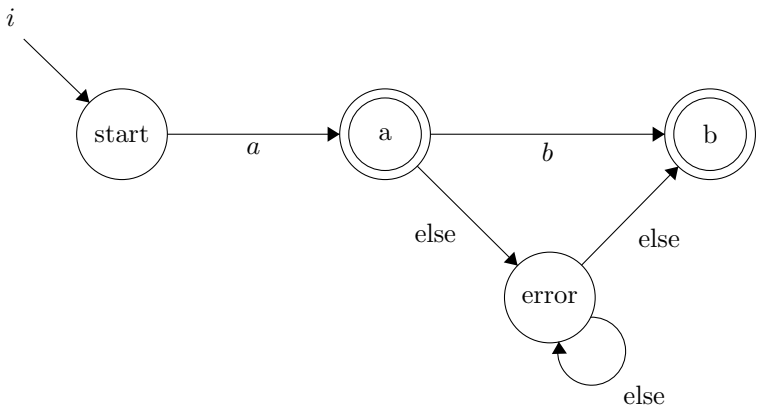
Definition. The **longest prefix problem** is when given x, find the longest prefix (of x), $y \in L$.

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 10.6. 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 10.7. Suppose $L = \{a\} \cup \{\text{any string beginning with } a \text{ and ending in } b\}$. 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.

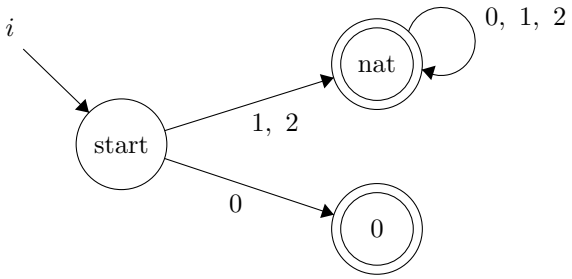
Here's the pseudocode for determining the longest prefix using a DFA, given an input $x = x_0x_1\ldots x_{|x|-1}$.

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

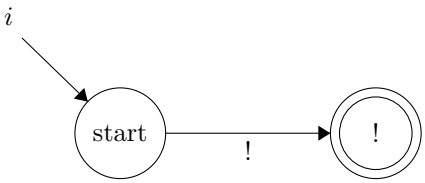
Next, let's take a deeper look at scanning. With scanning, we want to partition x using the longest prefix. This is also called leftmost longest, or more colloquially, the **maximal munch** scanning algorithm.

```
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 $x = yaz$.

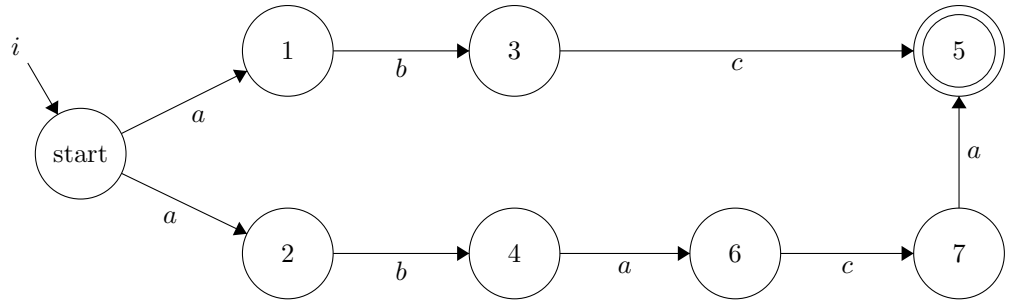
Suppose you have $L_{\text{int}} = \{0, 1, 2, 3, \ldots\}, L_{\text{id}} = \{x\}$, and $L_{\text{hex}} = \{0x1, \ldots\}$. If you have $x = 0x$, you can't determine if that's the integer zero followed by an identifier, or the beginning of a hex symbol. However, if you can look ahead and see $0x1$, you 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.

10.2 Nondeterministic Finite Automata (NFA)

← February 13, 2013

Example 10.8. 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 10.8, 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.

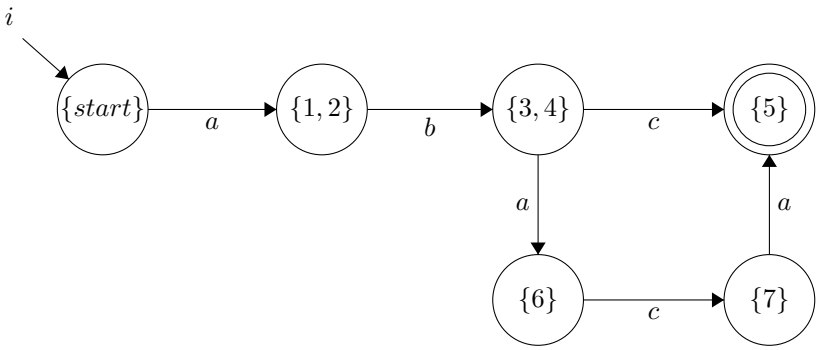
states = $\{start\}$;
for *a in* $x_0x_1x_2 \ldots x_{|x|-1}$ **do**
  states = $\bigcup_{s \in \text{states}}$ T[s, a];
end
if $states \cap f \neq \{\}$ **then** "accept";
else "reject";

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?

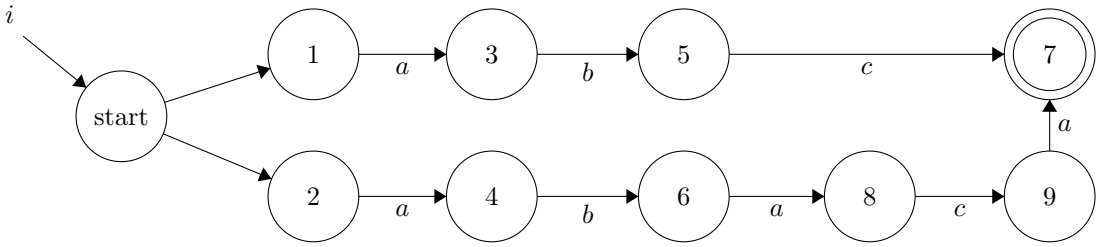
1. S_{NFA} is finite.
2. The number of subsets of a finite set is finite. There are $2^{|S|}$ subsets of a finite set S.
3. Each state in S_{DFA} represents a subset of the states in S_{NFA}.

From the NFA in Example 10.8, we can create this corresponding DFA:



10.3 Epsilon Nondeterministic Finite Automata (ϵ-NFA)

Example 10.9. This is an example of an ϵ-NFA:



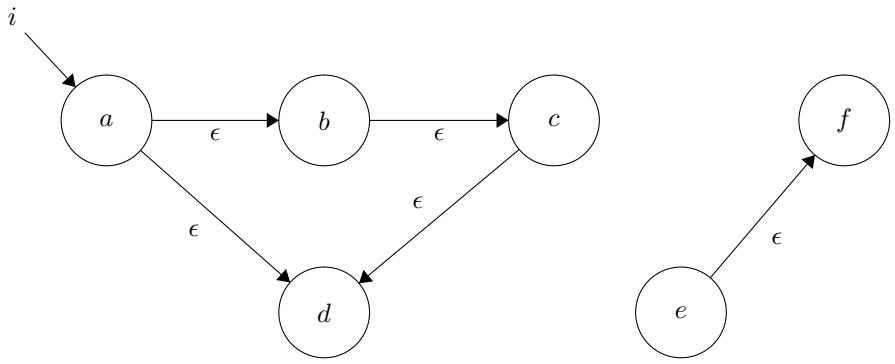
The difference between ϵ-NFAs and standard NFAs is that the transitions from the start state to states 1 and 2 have no symbol. That means you can spontaneously move your finger along those paths. We generally label those transitions as ϵ to prevent confusion between mistakes (omissions in labeling the transitions) and ϵ transitions. This recognizer is for the same language as the NFAs and DFAs in Example 10.8.

More formally, an ϵ-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 10.10. 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:

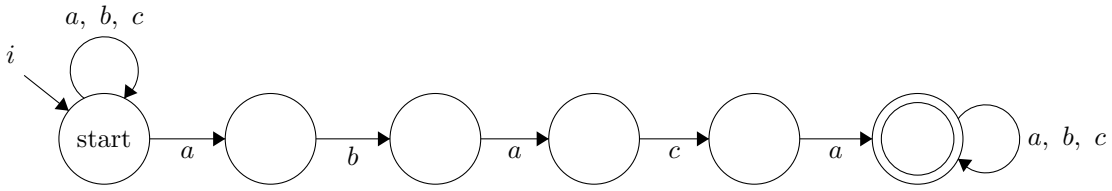
$$S' = S \cup \{s' | \exists s \in S', T[s, \epsilon] = s'\}$$

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 10.11. 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:



10.4 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 10.12. 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 10.13. Given the alphabet $\Sigma = \{a, b\}$, we're interested 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 10.14. 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 10.15. 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.

10.4.1 Extensions

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

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

(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).

10.4.2 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.

11 Context-Free Languages

← February 25, 2013

Context-free languages (CFLs) are a larger class of languages than regular languages. They capture recursion through matching brackets (`([()])`) and nesting structures (`if then if then else else`).

Every regular language is also a context-free language, however there are context-free languages that are not regular languages. Context-free languages that are not regular would be languages that require an infinite number of states in order to be expressed as a state machine. However, all regular languages can be expressed in a finite state machine, so those languages cannot be regular.

Example 11.1. An example of a context-free language that is not regular is the language defined by:

$$\Sigma = \{(,)\}$$
$$L = \{\epsilon, (), (()), ((())), \ldots\}$$

That is, L is the language of all strings containing n opening parentheses and n (matching) closing parentheses.

Definition. There are two definitions of **context-free languages**. A context-free language is:

- Any language generated by a context-free grammar.
- Any language recognized by a (nondeterministic) pushdown automaton (PDA).

Definition. A **pushdown automaton** (PDA) is a machine with finite memory and an unbounded (infinite) stack.

A pushdown automaton could be used to match parentheses and do similar recursive operations because it has an infinite stack.

Notice how a context-free language needs to be recognized by a nondeterministic pushdown automaton. The best running time of a pushdown automaton is approximately $O(n^{2.73}) \approx O(n^3)$ in the real world. This complexity makes nondeterministic pushdown automata only useful when we are dealing with small strings.

Some context-free languages can be recognized by a deterministic pushdown automaton (DPDA). This can be achieved in $O(n)$ time. These context-free languages are called **deterministic context-free languages**.

11.1 Context-Free Grammars

A context-free grammar (CTG) consists of:

- T: a finite set of terminal symbols (terminals), sometimes referred to as the alphabet Σ.
- N: a finite set of non-terminal symbols (nonterminals), sometimes referred to as variables.
- $S \in N$: a start symbol.
- R: a set of rules. $R = \{A \to \alpha | A \in N, \alpha \in (N \cup T)^\star\}$

Notation (by convention only):

- a, b, c, d are terminal symbols.
- A, B, C, D are nonterminal symbols.
- V is the vocabulary $V = N \cup T$.
- w, x, y, z are strings of terminals.
- $\alpha, \beta, \gamma, \delta$ are strings of vocabulary symbols (terminals and/or nonterminals).

Example 11.2. We have the context-free grammar G, where R is defined as:

$$S \to AhB$$
$$A \to ab$$
$$A \to cd$$
$$B \to ef$$
$$B \to g$$

We also have $T = \{a, b, c, d, e, f, g, h\}, N = \{S, A, B\}$, and $S = S$. We usually just write R and the rest of the components of the grammar can be inferred. The language generated by G is a finite (and regular) language $L(G) = \{abhef, abhg, cdhef, cdhg\}$.

$$\begin{aligned}
&\gamma_0 : && \implies S \\
&\gamma_1 : S \mapsto AhB && \implies AhB \\
&\gamma_2 : A \mapsto cd && \implies cdhB \\
&\gamma_3 : B \mapsto g && \implies cdhg
\end{aligned}$$

This is called a **derivation**.

Definition. $\alpha A \beta \implies \alpha\gamma\beta$. $\implies$ here roughly means "derives in one step." More precisely, $A \to \gamma \in R$.

Definition. $\alpha \overset{\star}{\implies} \beta$. $\overset{\star}{\implies}$ here roughly means "derives in zero or more steps." More precisely, it means either $\alpha = \beta$ *or* $\exists\gamma.\alpha \implies \gamma$ and $\gamma \overset{\star}{\implies} \beta$.

$\overset{\star}{\implies}$ is the reflexive, transitive closure of $\implies$.

Definition. A **derivation** of some string x with respect to some context-free grammar G is the sequence $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n$ where:

$$\begin{aligned}
&\gamma_0 = S \\
&\gamma_n = x \\
&\gamma_i \implies \gamma_{i+1} \text{ for } 0 \leq i < n
\end{aligned}$$

We could express $L(G)$ as $L(G) = \{x | S \overset{\star}{\implies} x\}$.

Definition. x is a sequence of terminals. x is sometimes called a **member** in $L(G)$, a **word** in $L(G)$, or a **sentence** in $L(G)$. All of this terminology means the same thing.

Definition. γ_i where $S \overset{\star}{\implies} \gamma_i$ is called a **derivation step** or a **sentential form**.

11.2 Parsing

Given some $x \in L(G)$, find a derivation for x. Parsing and recognition is when if $x \in L(G)$, you find a derivation, otherwise you reject. It's your duty to reject $x \notin L(G)$, but it'd be nice if you could also be somewhat informative as to why $x \notin L(G)$.

There are two general approaches for parsing: top-down and bottom up.

11.2.1 Top-Down Parsing

Top-down parsing is where you start with $\gamma_0 = S$ and continue until you find $\gamma_n = x$. Our example from earlier was an example of this:

$$\begin{aligned}
&\gamma_0 : && \implies S \\
&\gamma_1 : S \mapsto AhB && \implies AhB \\
&\gamma_2 : A \mapsto cd && \implies cdhB \\
&\gamma_3 : B \mapsto g && \implies cdhg
\end{aligned}$$

This is a **leftmost derivation**.

11.2.2 Bottom-Up Parsing

Bottom-up parsing is where you start with $\gamma_n = x$ and continue until you find $\gamma_0 = S$. Here's an example of a bottom-up parse:

$$\begin{aligned} \gamma_n : & \qquad \Longrightarrow cdhg \\ \gamma_{n-1} : cd \mapsto A & \qquad \Longrightarrow Ahg \\ \gamma_{n-2} : g \mapsto B & \qquad \Longrightarrow AhB \\ \gamma_{n-3} : AhB \mapsto S & \qquad \Longrightarrow S \end{aligned}$$

This is a **rightmost derivation**.

By contention, we will rewrite this derivation in reverse order:

$$\begin{aligned} \gamma_0 : & \qquad \Longrightarrow S \\ \gamma_1 : S \mapsto AhB & \qquad \Longrightarrow AhB \\ \gamma_2 : B \mapsto g & \qquad \Longrightarrow Ahg \\ \gamma_3 : A \mapsto cd & \qquad \Longrightarrow cdhg \end{aligned}$$

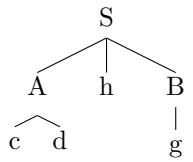
Note that the γ_2 for these two parsing examples is different. This is because we used leftmost derivation for the top-down example and rightmost derivation for the bottom-up example. Leftmost and rightmost derivations are **canonical forms**, since there are many other arbitrary decisions we could have made for the order of our derivation steps.

Definition. A **leftmost derivation** replaces the leftmost nonterminal first. A **rightmost derivation** replaces the rightmost nonterminal first.

Note that in the bottom-up example, we still appear to have replaced the leftmost nonterminal first, but since the order of our steps gets reversed, it seems like we performed a rightmost derivation instead.

11.2.3 Parse Tree

A parse tree is another canonical parsing form. You can draw a tree to represent the string:



Using a parse tree has the advantage of not having to rewrite the entire sentential form at each derivation step. It's also easier to visually traverse what has changed at each step.

Parse trees remove irrelevant differences, however we could still have some ambiguity as with the other parsing methods. Ambiguity is acceptable for context-free grammars, but it's sometimes easier to make compilers that deal with unambiguous grammars.

11.3 Parsing WLPP

← February 27, 2013

WLPP is described in lexical syntax (splitting it up into tokens), context-free syntax (how to arrange the tokens to create a valid WLPP program), and context-sensitive syntax (other rules that cannot be expressed by a context-free grammar).

The start symbol is always `procedure`. We parse the tokens in order. When we encounter a nonterminal symbol, we look for a rule that defines it (on the following lines). We perform a substitution based on the rule we found, and then check the continual validity of token kinds in that order.

Note that declarations can be defined recursively in these grammars.

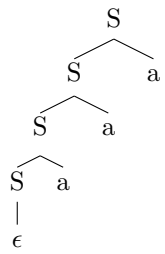
For more information, see the WLPP specification.

11.4 Some Parsing Examples

Example 11.3. We're given the alphabet $\Sigma = \{a, b\}$, the language $L = \{\epsilon, a, aa, aaa, \ldots\}$ (that is, $a^\star$), and the following rules R:

$$S \to \epsilon$$
$$S \to Sa$$

Let's derive $x = aaa$ using a parse tree.



This could also be expressed as a derivation like so:

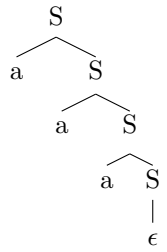
$$S$$
$$Sa$$
$$Saa$$
$$Saaa$$
$$aaa$$

That derivation is both a leftmost and rightmost derivation. This grammar is called **left-recursive** or **left-associative**.

Example 11.4. Given the same alphabet and language as before, we could have also had this set of rules R:

$$S \to \epsilon$$
$$S \to aS$$

Let's again look at deriving $x = aaa$, but with these new rules.



This could also be expressed as a derivation like so:

$$\begin{aligned} S \\ aS \\ aaS \\ aaaS \\ aaa \end{aligned}$$

This grammar is called **right-recursive** or **right-associative**.

Example 11.5. Given the same alphabet and language as before, we could have also had this set of rules R:

$$\begin{aligned} S &\to \epsilon \\ S &\to a \\ S &\to SS \end{aligned}$$

Let's again look at deriving $x = aaa$, but with these new rules.

We have two different parse trees, which means this grammar is ambiguous. That is, there are two canonical forms.

We have two different leftmost derivations for $x = aaa$. The first leftmost derivation is:

$$\begin{aligned} S \\ SS \\ SSS \\ aSS \\ aaa \end{aligned}$$

The second leftmost derivation is:

$$
\begin{aligned}
&S \\
&SS \\
&aS \\
&aSS \\
&aaS \\
&aaa
\end{aligned}
$$

Note that these two derivations are different.

Similarly, there are two different rightmost derivations. The first rightmost derivation is:

$$
\begin{aligned}
S& \\
SS& \\
Sa& \\
SSa& \\
Saa& \\
aaa&
\end{aligned}
$$

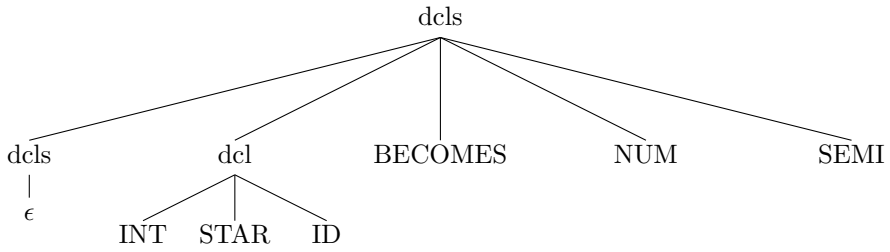
The second rightmost derivation is:

$$
\begin{aligned}
S& \\
SS& \\
SSS& \\
SSa& \\
Saa& \\
aaa&
\end{aligned}
$$

Once again, note that these two derivations are different.

Example 11.6. Assume the context-free grammar as specified in the WLPP specification.

Let's parse `int* x = 25;`. We get the following parse tree:



Technically, this shouldn't be valid, because type `int*` cannot be assigned the value 25. However, this context-free grammar does not enforce that – the context-sensitive syntax

will. It was decided that this grammar was sufficient to construct a parse tree, although if we wanted we could enforce that constrain in a context-free grammar.

It is sometimes the case that adding seemingly-simple constraints like these will make the grammar quite complicated, so it's sometimes better to just leave certain restrictions for enforcement through a context-sensitive syntax instead.

11.5 Formalizing Derivations

We'll start off with two formal definitions for leftmost derivations.

Definition. $xA\beta \implies x\gamma\beta := A \to \gamma \in R$. That is, you must substitute the first nonterminal symbol.

Definition. $\alpha \overset{\star}{\implies} \beta := \alpha = \beta$ OR $\exists \gamma \cdot \alpha \implies \gamma, \gamma \overset{\star}{\implies} \beta$.

We have similar definitions for rightmost derivations, as follows.

Definition. $\alpha Ay \implies \alpha\gamma y := A \to \gamma \in R$.

Definition. $\alpha \overset{\star}{\implies} \beta := \alpha = \beta$ OR $\exists \gamma \cdot \alpha \implies \gamma, \gamma \overset{\star}{\implies} \beta$.

We will now formally define ambiguity in a context-free grammar.

Definition. A grammar G is **ambiguous** if there exists $x \in L(G)$ such that:

- There are two or more leftmost derivations for x, or equivalently,
- There are two or more rightmost derivations for x, or equivalently,
- There are two or more parse trees for x.

Often, there is a different grammar G' where $L(G') = L(G)$ and G' is unambiguous. But, G' may not be easy to find, or G' may not exist at all. If G' does not exist, we say $L(G)$ is **inherently ambiguous**.

We're interested in unambiguous grammars because they allow us to do syntax-directed translation.

11.6 Syntax-Directed Translation

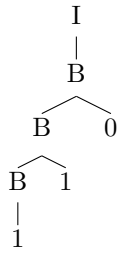
Syntax-directed translation involves computing the "meaning" for all subtrees in the parse.

Example 11.7. Let $T = \{0, 1\}, N = \{I, B\}, S = I$, and R be the set of rules:

$$\begin{aligned} I &\to 0 \\ I &\to B \\ B &\to 1 \\ B &\to B0 \\ B &\to B1 \end{aligned}$$

We're interested in the language of binary integers without leading zeros, that is, $\{0, 1, 10, 11, 100, 101, \ldots\}$.

Let's parse the big-endian value of $x = 110$. We have the parse tree:



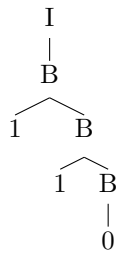
From the bottom of the tree, the meaning of B is 1. The meaning of the next B is 3, since $Meaning(B) = 2 \times 1 + 1$. The meaning of the uppermost B, and I, are 6. The meaning of a particular element is determined by its subtrees.

Suppose instead we wanted the little-endian binary unsigned value. That's reasonably difficult with this grammar. Instead, we'll use a right-associative grammar for the same language. Let's define the rules for this new grammar to be:

$$
\begin{aligned}
I &\to 0 \\
I &\to B \\
B &\to 1 \\
B &\to 0B \\
B &\to 1B \\
B &\to 0
\end{aligned}
$$

Actually, this example is broken, since the number 110 has a useless leading zero when interpreted as a little-endian binary value. So, we added the last rule, $B \to 0$ to allow leading zeroes.

We have the following parse tree:



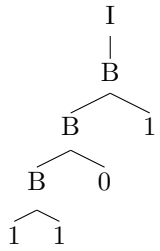
From the bottom of the tree, the meaning of B is 0, the meaning of the second B is 1, and the meaning of the uppermost B (and I) is 3.

← March 1, 2013

Example 11.8. Let's change the rules R, to be:

$$I \to 0$$
$$I \to B$$
$$I \to 1$$
$$I \to B0$$
$$I \to B1$$

We're interested in the meaning of $x = 1101 \in L(G)$ as a binary big-endian number. x means 13 in big-endian. We get this parse tree:

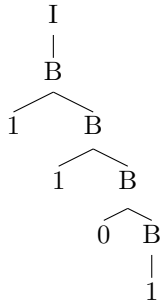


The meaning of an individual node is called its **attribute**. The attribute of the lowest B is 3, then 6, then 13. We denote the attribute of I as $m(I) = 13$. To determine the meaning, you determine the meaning of the subtrees first by doing a postfix traversal.

What about the little-endian meaning? The best way to determine this is to write another grammar that is right-associative. We'll define the rules R as:

$$I \to 0$$
$$I \to B$$
$$B \to 1$$
$$B \to 0B$$
$$B \to 1B$$

This is the parse tree for $x = 1101$ as a little-endian binary number:

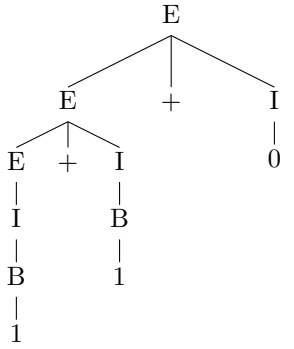


11.7 Extended Grammars

Let's extend the grammar from before to add these rules to R:

$$E \to I$$
$$E \to E + I$$

Now, let's parse `1 + 1 + 0`. We get the parse tree:

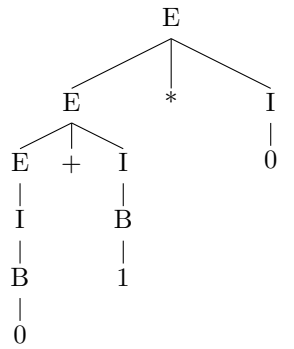


The meaning of the $E + I$ expression at the second level is $m(E) = 2$ and $m(I) = 0$, so the resulting meaning for the uppermost E is $m(E) = 2$.

Now, let's extend the grammar once more to include these rules for E, in addition to the rules for I that were defined earlier:

$$E \to I$$
$$E \to E + I$$
$$E \to E * I$$

Now, let's try to parse `1 + 1 * 0`. We get the parse tree:

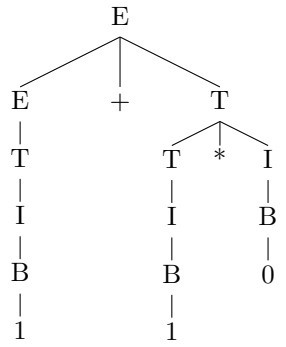


If you examine the attributes, you'll see that the uppermost `E * I` expression has $m(E) = 2$ and $m(I) = 0$, which gives the attribute of the uppermost E to be $m(E) = 0$. This is mathematically incorrect.

To get around this, we'll introduce the notion of a term T. We'll re-define the rules for E once more:

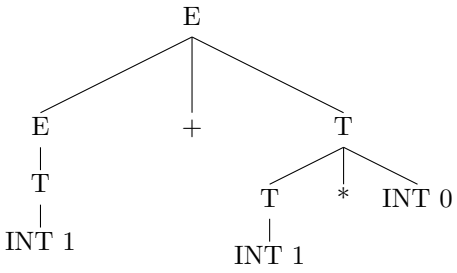
$$E \to T$$
$$E \to E + T$$
$$T \to I$$
$$T \to T * I$$

Now, when we parse `1 + 1 * 0`, we get this parse tree:



This yields the proper values for the uppermost `E + T` expression, $m(E) = 1$ and $m(T) = 0$. This works as we would mathematically expect.

You can see this general pattern of evaluation ordering in the WLPP spec, but instead of I and B, we will have specific INT tokens. Here's an example of a WLPP parse tree, parsing `1 + 1 * 0`.



11.8 Parsing Algorithms

There are several different parsing algorithms, some of which are top-down and some of which are bottom-up.

11.8.1 Generic Top-Down Parser

Given $x \in L(G)$, find a derivation $\gamma_0\gamma_1 \dots \gamma_n$.
$\gamma = S$;
while $\gamma \neq x$ **do**
  select α, A, β where $\gamma = \alpha A \beta$ (find a nonterminal A in γ);
  select δ where $A \to \delta \in R$ (find a rule with A as its left side, may fail if $x \notin L(G)$);
  $\gamma = \alpha\delta\beta$;
end

Example 11.9. We're given the following set of rules R:

$$S \to AhB$$
$$A \to ab$$
$$B \to ef$$
$$B \to g$$

Let's parse $x = abhg$. We get:

$$\gamma = S, A = S, \alpha = \beta = \epsilon, \delta = AhB$$
$$\gamma = AhB, A = A(\text{or} A = B), \alpha = \epsilon, \beta = hB, \delta = ab \text{ by oracle}$$
$$\gamma = abhB$$
$$\gamma = abhg$$

11.8.2 Stack-Based Top-Down Parser

Given input $x = x_0x_1x_2 \dots x_{|x|-1}$, we'll read x from left to right. At any step, we have:

$$x = \underbrace{x_0x_1x_2}_{\text{input read so far}} \underbrace{\dots x_ix_{i+1} \dots x_{|x|-1}}_{\text{input yet to be read}}$$

We also have a stack of vocabulary symbols:

$$\text{stack} = X_0X_1 \dots X_{|\text{stack}|-1}$$

(where X_0 is the top of the stack and $X_{|\text{stack}|-1}$ is the bottom of the stack.

A key observation: at all steps, we have:

$$\gamma = (\text{input read so far})(\text{stack})$$

Example 11.10. Let's parse $x = abhg$.

γ	input read	input yet to be read	stack (top ... bottom)
S	ϵ	abhg	S
AhB	ϵ	abhg	AhB
abhB	ϵ	abhg	abhB
abhB	a	bhg	bhB
abhB	ab	hg	hB
abhB	abh	g	B
abh	abh	g	g
abhg	abhg	ϵ	ϵ

We're popping a terminal from the stack and comparing it with the input. When both the input to be read and the stack are empty, we're done.

Here's some pseudocode for a stack-based top-down parser:

```
push S onto stack;
for c = x_0, x_1, x_2, ..., x_{|x|-1} do
    while top of stack is A ∈ N do
        consult oracle to determine δ where A → δ ∈ R;
        pop stack;
        push δ_{|δ|-1};
        push δ_{|δ|-2};
        ⋮                                                    ← March 4, 2013
        push δ_0;
    end
    assert c = top of stack;
    pop stack;
end
assert stack = ϵ;
```

The crux of a stack-based top-down parser is in each derivation step, we have this invariant:

$$\gamma = (\text{input read so far})(\text{stack: top} \ldots \text{bottom})$$

11.8.3 LL(1) Parsing

The name "LL(1)" comes from the fact that this parsing algorithm reads input from left-to-right, and it uses leftmost derivations. An oracle uses one symbol of lookahead – that is, it looks at A (the nonterminal to expand) and c (the next input symbol) in order to choose $A \to \alpha \in R$.

The oracle is represented as a Predict function, Predict: $N \times T \to 2^R$ (that is, it yields a subset of the set of rules R).

As a convenience, we'll introduce the notion of an augmented grammar.

Augmented Grammar: given G, the augmented grammar G' is:

$$\begin{aligned}
N' &= N \cup \{S'\} \text{ where } S' \text{ is the new start symbol} \\
S' &= S' \\
T' &= T \cup \{\vdash, \dashv\} \text{ where } \vdash \text{ and } \dashv \text{ are the beginning and end string markers} \\
R' &= R \cup \{S' \to \vdash S \dashv\}
\end{aligned}$$

Let's introduce a specific augmented grammar with the following set of rules R':

$$S' = \vdash S \dashv \tag{1}$$
$$S = AhB \tag{2}$$
$$A = ab \tag{3}$$
$$A = cd \tag{4}$$
$$B = ef \tag{5}$$
$$B = g \tag{6}$$

A derivation for some input x would look like:

$$\gamma_0 = \vdash S' \dashv$$
$$\vdots$$
$$\gamma_n = \vdash x \dashv$$

Let's get back to our Predict function. We have the following table which represents the logic the oracle uses:

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	$\vdash$	$\dashv$
S'	{}	{}	{}	{}	{}	{}	{}	{}	{1}	{}
S	{2}	{}	{2}	{}	{}	{}	{}	{}	{}	{}
A	{3}	{}	{4}	{}	{}	{}	{}	{}	{}	{}
B	{}	{}	{}	{}	{5}	{}	{6}	{}	{}	{}

The (B, e) position means if e is the next symbol, rule 5 must be used. We could have drawn this table without empty sets all over the place – blank spaces indicate empty sets of applicable rules.

Here's the pseudocode for an LL(1) parser for input x:

push $\dashv$;
push S;
push $\vdash$;
for c *in* $\vdash x \dashv$ **do**
  while *top of stack is some* $A \in N$ **do**
    assert Predict$[A, c] = \{A \to X_0 X_1 \dots X_m\}$;
    pop stack;
    push X_m;
    push X_{m-1};
    $\vdots$
    push X_0;
  end
  assert top of stack = c;
  pop stack;
end
assert stack = ϵ;

Aside: if we added the rule (7) $A \to abdc$, we would have a set of two elements for $[A, a]$, which means the grammar would no longer be an LL(1) grammar. LL(1) is a deterministic

parser. If we wanted to support multiple rules, like by adding that rule (7), we could build a non-LL(1) nondeterministic parser.

Here's a sample LL(1) parse:

γ	input read	not read	stack (top ... bottom)	action
$\vdash S \dashv$	ϵ	$\vdash abhg \dashv$	$\vdash S \dashv$	assert $\vdash = \vdash$
$\vdash S \dashv$	$\vdash$	$abhg \dashv$	$S \dashv$	read $\vdash$
$\vdash AhB \dashv$	$\vdash$	$abhg \dashv$	$AhB \dashv$	Predict[S, a] = $\{S \to AhB\}$
$\vdash abhB \dashv$	$\vdash$	$abhg \dashv$	$abhB \dashv$	Predict[A, a] = $\{A \to ab\}$
$\vdash abhB \dashv$	$\vdash a$	$bhg \dashv$	$bhB \dashv$	assert $a = a$, read a
$\vdash abhB \dashv$	$\vdash ab$	$hg \dashv$	$hB \dashv$	assert $b = b$, read b
$\vdash abhB \dashv$	$\vdash abh$	$g \dashv$	$B \dashv$	assert $h = h$, read h
$\vdash abhg \dashv$	$\vdash abh$	$g \dashv$	$g \dashv$	Predict[B, g] = $\{B \to g\}$
$\vdash abhg \dashv$	$\vdash abhg$	$\dashv$	$\dashv$	assert $g = g$
$\vdash abhg \dashv$	$\vdash abhg \dashv$	ϵ	ϵ	assert stack = ϵ

Constructing the Predict Function: we'll need some helper functions.

- first$[\alpha] = \{a | \alpha \stackrel{\star}{\Longrightarrow} a\beta\}$. For example, first$(A) = \{a, c\}$, and first$(AbB) = \{a\}$.
- nullable$[\alpha] := \alpha \stackrel{\star}{\Longrightarrow} \epsilon$.
- follow$[A] := \{b | S' \stackrel{\star}{\Longrightarrow} \alpha Ab\beta\}$. That is, in *some* sentential form, A is followed by these elements b.

Now, we can define Predict:

$$\text{Predict}[A, a] = \{A \to \alpha \in R | a \in \text{first}(\alpha) \text{ OR } (\text{nullable}(\alpha) \text{ AND } a \in \text{follow}(A)\}$$

In the previous example, we didn't use the nullable rule because we did not have any rules leading to ϵ.

Observation: if for all A, a, $|\text{Predict}[A, a]| \leq 1$, we say that G is an LL(1) grammar. Otherwise, G is not LL(1). We *may* be able to write an LL(1) grammar for the same language.

Example 11.11. Let G be a grammar with the following set of rules R:

$$S' \to \vdash S \dashv$$
$$S \to ab$$
$$S \to aab$$

This grammar is not LL(1), because Predict$[S, a] = \{S \to ab, S \to aab\}$. In particular, $|\text{Predict}[A, a]| \not\leq 1$ for all A, a.

We can fix this (and make it an LL(1) grammar) by changing R to be the following set

of rules:

$$\begin{aligned} S' &\to \vdash S \dashv \\ S &\to aB \\ B &\to b \\ B &\to ab \end{aligned}$$

In this case, we have:

$$\begin{aligned} \text{Predict}[S, a] &= \{S \to ab\} \\ \text{Predict}[B, a] &= \{B \to ab\} \\ \text{Predict}[B, b] &= \{B \to b\} \end{aligned}$$

Now, $|\text{Predict}[A, a]| \leq 1$ for all A, a, so this is an LL(1) grammar for the same language.

Similarly, we can find yet another LL(1) grammar for the same language. Let R now be this set of rules:

$$\begin{aligned} S' &\to \vdash S \dashv \\ S &\to aBb \\ B &\to \\ B &\to a \end{aligned}$$

Note that the third rule is B mapping to nothing. That is, $B \to$ is nullable. We get these results from our Predict function:

$$\begin{aligned} \text{Predict}[S, a] &= \{S \to aBb\} \\ \text{Predict}[B, a] &= \{B \to a\} \\ \text{Predict}[B, b] &= \{B \to\} \end{aligned}$$

So, we have that $B \to$ is nullable and follow$(B) = \{b\}$.

If you use LL(1), you'll have to forget about left recursion, and therefore forget about syntax-directed translation as well.

There are some languages for which no LL(1) grammar is possible.

Example 11.12. Let's look at the language defined by the regular expression $a^\star(ab)^\star$. Here's a set of rules R for that regular expression:

$$\begin{aligned} S &\to \epsilon \\ S &\to aSb \\ S &\to aaSb \end{aligned}$$

Note that this grammar is not LL(1) and it is ambiguous.

11.8.4 Stack-Based Bottom-Up Rightmost Parser

← March 6, 2013

Example 11.13. We're given the following set of rules R:

$$S \to AhB$$
$$A \to ab$$
$$A \to cd$$
$$B \to ef$$
$$B \to g$$

A bottom-up rightmost derivation for $x = abhg$ is:

$$\gamma_3 = abhg$$
$$\gamma_2 = Ahg$$
$$\gamma_1 = AhB$$
$$\gamma_0 = S$$

Stack-Based Bottom-Up Rightmost Parse:

The key observation is that we have the stack written the opposite way:

$$\gamma = (\text{stack: bottom} \ldots \text{top})(\text{input yet to be read})$$

γ	stack	input read	input not read	comment
abhg	ϵ	ϵ	abhg	
abhg	a	a	bhg	shift a
abhg	ab	ab	hg	shift b
Ahg	A	ab	hg	reduce $A \to ab$ (pop, pop, push A)
Ahg	Ah	abh	g	shift h
Ahg	Ahg	abhg	ϵ	shift g
AhB	AhB	abhg	ϵ	reduce $B \to g$
S	S	abhg	ϵ	reduce $S \to AhB$
				done

11.8.5 LR(1) Parsing

LR(1) parsing means we're analyzing input from left to right, using rightmost reversed derivations, and we have one symbol of lookahead.

The oracle, in this case, looks at the stack and the next symbol that has yet to be read. It tells you either:

- Goto[start, symbol]: new element to push onto the stack.
- Reduce[start, symbol]: a set of rules to use to reduce the expression. Hopefully, this is at most one rule.

You can build this oracle in a sneaky way.

In 1966, Donald Knuth made this key observation: when you do a stack-based rightmost derivation in reverse, the set of valid stacks is a regular language. That is, a valid stack can be recognized by a DFA. Assuming a stack is valid, if we push the next symbol onto the stack, will we still have a valid stack (will we still be in a final state)?

If we're planning to reduce, we have the right-hand side of the stack. We analyze more "what-if" scenarios. We check for the next sequence of symbols if reductions are possible that still lead to a valid stack.

Example 11.14. LR(1) uses an augmented grammar. We have this augmented set of rules R':

$$S \to \vdash S \dashv$$
$$S \to AhB$$
$$A \to ab$$
$$A \to cd$$
$$B \to ef$$
$$B \to g$$

We get this derivation:

$$\gamma_3 = \vdash abhg \dashv$$
$$\gamma_2 = \vdash Ahg \dashv$$
$$\gamma_1 = \vdash AhB \dashv$$
$$\gamma_0 = \vdash S \dashv$$

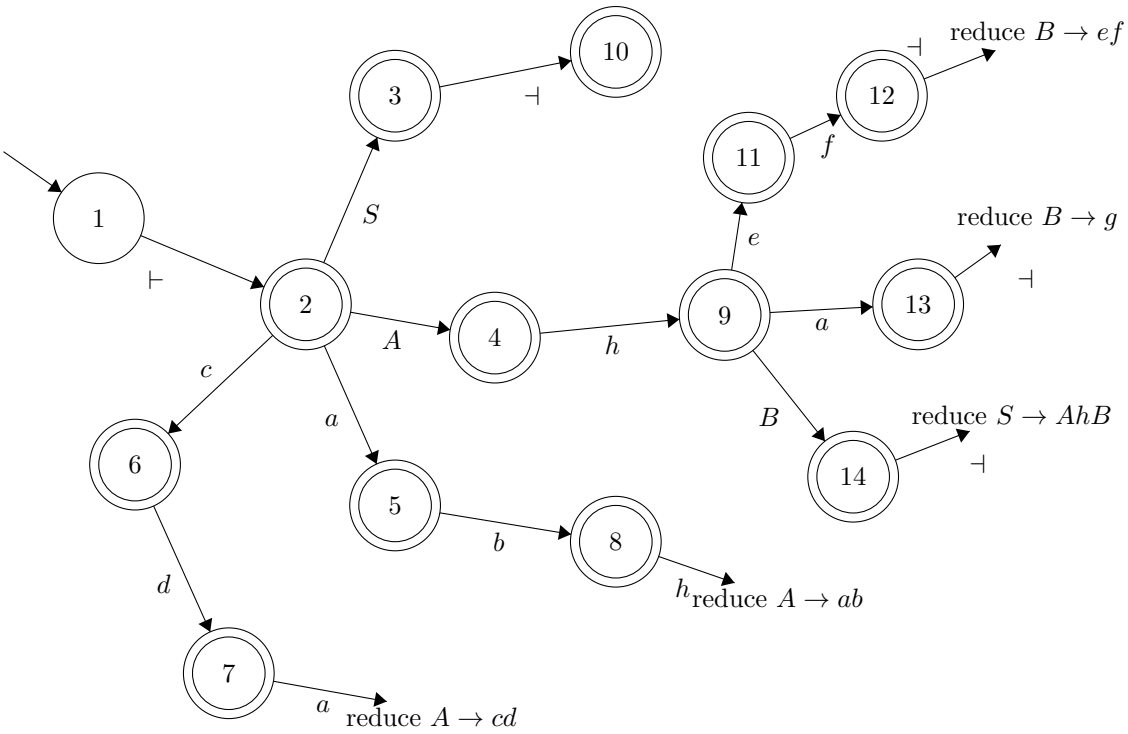
After this augmentation, we have:

γ	stack	input read	input not read	comment
$\vdash$ abhg $\dashv$	$\vdash$	$\vdash$	abhg $\dashv$	
$\vdash$ abhg $\dashv$	$\vdash$ a	$\vdash$ a	bhg $\dashv$	shift a
$\vdash$ abhg $\dashv$	$\vdash$ ab	$\vdash$ ab	hg $\dashv$	shift b
$\vdash$ Ahg $\dashv$	$\vdash$ A	$\vdash$ ab	hg $\dashv$	reduce $A \to ab$ (pop, pop, push A)
$\vdash$ Ahg $\dashv$	$\vdash$ Ah	$\vdash$ abh	g $\dashv$	shift h
$\vdash$ Ahg $\dashv$	$\vdash$ Ahg	$\vdash$ abhg	$\dashv$	shift g
$\vdash$ AhB $\dashv$	$\vdash$ AhB	$\vdash$ abhg	$\dashv$	reduce $B \to g$
$\vdash$ S $\dashv$	$\vdash$ S	$\vdash$ abhg	$\dashv$	reduce $S \to AhB$
$\vdash$ S $\dashv$	$\vdash$ S $\dashv$	$\vdash$ abhg $\dashv$	ϵ	shift $\dashv$

We don't need the actual symbols in the stack in this table. You may still want to keep them to make more intuitive sense, or if you're doing syntax-directed translation. But, if we wanted, we could label them just as numbers that will represent state numbers:

γ	stack	input read	input not read	comment
$\vdash$ abhg $\dashv$	2	$\vdash$	abhg $\dashv$	
$\vdash$ abhg $\dashv$	2,5	$\vdash$ a	bhg $\dashv$	shift a
$\vdash$ abhg $\dashv$	2,5,8	$\vdash$ ab	hg $\dashv$	shift b
$\vdash$ Ahg $\dashv$	2,4	$\vdash$ ab	hg $\dashv$	reduce $A \to ab$ (pop, pop, push A)
$\vdash$ Ahg $\dashv$	2,4,9	$\vdash$ abh	g $\dashv$	shift h
$\vdash$ Ahg $\dashv$	2,4,9,13	$\vdash$ abhg	$\dashv$	shift g
$\vdash$ AhB $\dashv$	2,4,9,14	$\vdash$ abhg	$\dashv$	reduce $B \to g$
$\vdash$ S $\dashv$	2,3	$\vdash$ abhg	$\dashv$	reduce $S \to AhB$
$\vdash$ S $\dashv$	2,3,10	$\vdash$ abhg $\dashv$	ϵ	shift $\dashv$

We can construct this DFA with the state identifiers being the numbers from the stack in the table above:



This DFA recognizes two different things: valid stacks *and* stacks ending in right-hand sides of rules.

Goto[state, symbol] $\to$ state: this is the same DFA transition function from before.

Reduce[state, symbol] $\to 2^R$: yields a set of rules to reduce. If |Reduce[s, a]| > 1, then we have a reduce conflict (the oracle can't decide between two rules to reduce) which means this is not an LR(1) grammar. If |Reduce[s, a]| > 0 and Goto[s, a] $\in$ f, then we have a shift/reduce conflict.

If you have no conflicts, then you have a deterministic LR(1) parser. With conflicts, you

could make a nondeterministic parser which would be inefficient and not LR(1).

Here's the pseudocode for an LR(1) parser for input x:

push $\vdash$;
for *c in x* $\dashv$ **do**
  while *Reduce[top of stack, c]* $= \{A \to \alpha\}$ **do**
    pop $|\alpha|$ times;
    push Goto[(new) top of stack, c];
  end
  assert Goto[top of stack, c] $\in$ f; // if not, then $x \notin L(G)$
end
accept if all assertions succeeded;

In an LR(1) parse, if you ever shift $\dashv$, you know for a fact that your parse succeeded.

11.9 Building a Parse Tree From a Bottom-Up Parse

← March 11, 2013

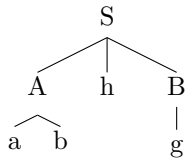
A parse tree is a much more general and flexible representation over a rightmost reversed derivation.

Key: in addition to the (parse) stack, use a parallel tree stack. For every symbol on the stack, we'll have a corresponding parse (sub)tree on the tree stack.

Example 11.15. We have an augmented grammar with this set of rules R:

$$\begin{aligned} S' &\to \vdash S \dashv \\ S &\to AhB \\ A &\to ab \\ A &\to cd \\ B &\to ef \\ B &\to g \end{aligned}$$

If we have input $x = abhg$, the outputted parse tree should be as follows.



The parse would be as follows.

tree stack	parse stack	input read	input not read	action
$\vdash$	$\vdash$	ϵ	abhg $\dashv$	initialization
$\vdash$ a	$\vdash$ a	a	bhg $\dashv$	shift a
$\vdash$ ab	$\vdash$ ab	ab	hg $\dashv$	shift b
$\vdash$ A a b	$\vdash$ A	ab	hg $\dashv$	reduce $A \to ab$
$\vdash$ A h a b	$\vdash$ Ah	abh	g $\dashv$	shift h
$\vdash$ A hg a b	$\vdash$ Ahg	abhg	$\dashv$	shift g
$\vdash$ A h B a b g	$\vdash$ AhB	abhg	$\dashv$	reduce $B \to g$
$\vdash$ S A h B a b g	$\vdash$ S	abhg	$\dashv$	reduce $S \to AhB$
$\vdash$ S $\dashv$ A h B a b g	$\vdash$ S $\dashv$	abhg $\dashv$	ϵ	shift $\dashv$

To construct the tree stack, you'll likely want to use a linked implementation to make reference to the structures you built up in previous steps.

The pseudocode for building a parse tree is as follows.

```
stack = Goto[stack, ⊢];
tree_stack = makeleaf(⊢);
for every c in x ⊣ do
    while reduce A → α do
        // LR(1) reduction:
        pop |α| symbols from stack;
        push Goto[top of stack, c];
        // Tree build:
        pop |α subtrees from the tree_stack;
        push makenode(root = A, subtrees = nodes that were just popped);
    end
    push Goto[top of stack, c];
    push makeleaf(c) onto tree_stack;
end
// Parse tree is now the second element on the tree_stack.
pop tree_stack;
// Parse tree is now on the top of the tree_stack.
```

To perform syntax-directed translation, you just need to traverse the parse tree to determine meaning for each subtree.

11.9.1 Notes about Assignment 8

Questions 1 and 2 are about reversed rightmost derivations, with the questions being similar to ones asked on assignment 7.

Question 3 asks you to write a LR(1) parser. You're given a context-free grammar (CFG) and an LR(1) DFA (including Reduce and Goto). If $x \in L(G)$, you'll have to output a derivation for x.

Question 4 asks you to write a parser for WLPP. The input will be the output from the scanners we used in assignment 6. The output must be a WLPPI file, which is very much like a CFG file – it contains a leftmost derivation. You're given the grammar and a DFA, but you're not allowed to read the DFA from a file. You may want to write a program to generate the relevant code for including the DFA in your parser.

Note that for question 4, you need to produce a leftmost derivation. However, you would have already built a LR(1) parser for question 3. So, use your parser to build a parse tree and then perform a pre-order traversal to get a leftmost derivation.

11.9.2 How to Construct the LR(1) DFA (SLR(1) Method)

There are three basic steps to constructing a LR(1) DFA using the SLR(1) method:

1. Define an NFA that recognizes valid stacks (it implements the shift operation).
2. Convert the NFA to a DFA with subset construction.
3. Add extra transition states indicating when to reduce.

For each rule in the grammar you're given, you must construct several rules each placing a dot in a unique position on the right-hand side of the rule. These will represent states in the DFA.

Here are the states for the augmented grammar presented earlier.

$$S' \to . \vdash S \dashv$$
$$S' \to \vdash . S \dashv$$
$$S' \to \vdash S. \dashv$$
$$S' \to \vdash S \dashv .$$

$$S \to .AhB$$
$$S \to A.hB$$
$$S \to Ah.B$$
$$S \to AhB.$$

$$A \to .ab$$
$$A \to a.b$$
$$A \to ab.$$

$$A \to .cd$$
$$A \to c.d$$
$$A \to cd.$$

$$B \to .ef$$
$$B \to e.f$$
$$B \to ef.$$

$$B \to .g$$
$$B \to g.$$

Whenever you have a dot preceding a symbol, you move the dot to the other side of that symbol and make *that* rule a final state.

Whenever you have a dot before a nonterminal, you also add an epsilon transition to rules that have the corresponding nonterminal as their left-hand side.

To convert the NFA to a DFA, use subset construction. Since we have ϵ transitions, you'll also need to take ϵ-closure into account. Once all the states are built up with subset construction, you'll also want transitions to reduction rules for A (for example) on a symbol h (for example) if $h \in \text{follow}(A)$.

It *might* be time to perform a reduction when you're on the very right side of the rule. Don't go down a blind alley, though – consult your follow function to ensure a reduction is desired.

12 The Big Picture: Building a Compiler

← March 13, 2013

At the end of the day, we want to be able to take a WLPP source file, send it into a compiler, and get MIPS assembly object code, which has equivalent meaning. We could then feed the produced object code and send it into an assembler to get MIPS binary machine code, if we wanted.

The compiler portion consists of a scanner (from A7), which produces some output that will be fed into a parser (from A8), which will produce a WLPPI file. The WLPPI file will be the input for context sensitive analysis (from A9), which will produce a symbol table and a decorated parse tree, both of which will be fed into synthesis (from A10), where code generation occurs.

12.1 Context-Sensitive Analysis

Context-sensitive analysis involves all syntax rules that are not conveniently expressed (or are not possible to express) in the scanner and parser. In WLPP, these are rules pertaining to symbols and their types.

Symbols have the following rules:

- Every symbol that is used must be declared.
- Every symbol that is declared must be declared at most once.

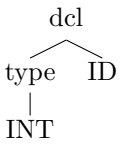
Types have the following rule:

- Every expression must have a valid type. In WLPP, types are `int` and `int *`.

12.1.1 Building a Symbol Table

We'll use syntax-directed translation to gather all symbols. During this process, we'll ensure each symbol is declared only once. We'll then create a multiset of symbols.

Let's say we have this parse (sub)tree:

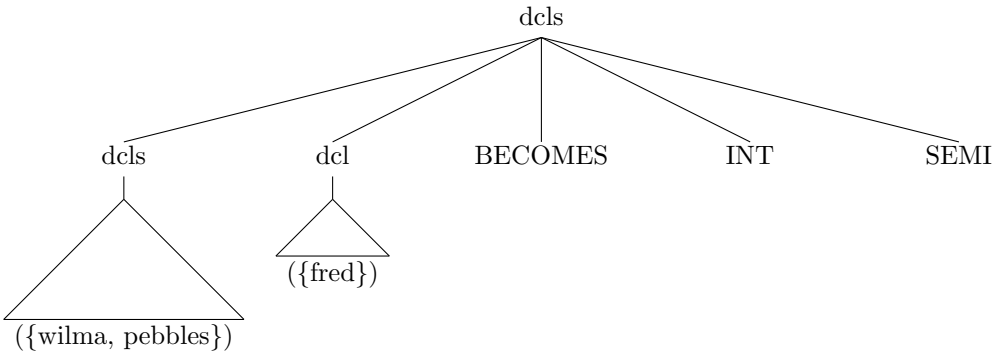


Each terminal symbol (ID) will be annotated with its lexeme, such as `fred`. We'll then get that symbols = {fred}.

Now, suppose we have the following production rule:

$$\text{dcls} \rightarrow \text{dcls dcl BECOMES INT SEMI}$$

This rule would give us the following parse tree:



We recursively determine the symbols for the subtrees, and then decorate the root element with the complete symbol list (the union). Since this is a multiset, you can check to ensure there were no duplicate declarations. This ensures the second rule for symbols was not violated.

Each expression gets a type evaluated for it:

$$\underbrace{\underbrace{1}_{\text{type = int}} + \underbrace{2}_{\text{type = int}}}_{\text{type = int}}$$

This gets slightly more complicated when we don't have constants. For instance:

$$\underbrace{1}_{\text{type = int}} + \underbrace{\text{fred}}_{\text{type TBD}}$$

Note that in this case, we need to check our symbol table for `fred`'s type, to determine if it's `int` or `int *`. So, the symbol table will need to keep track of each symbol's identifier and type (and value as well).

Suppose we have this expression:

$$\underbrace{\text{wilma}}_{\text{type TBD}} + \underbrace{\text{fred}}_{\text{type TBD}}$$

We could construct a table to represent all of the possibilities:

wilma	**fred**	**result**
int	int	int
int	int *	int *
int *	int	int *
int *	int *	error

Note that this last entry is an error because adding two pointers together does not make any sense.

Let's look at another example. Let's say we have this expression:

$$\underbrace{\text{betty}}_{\text{type TBD}} - \underbrace{\text{barney}}_{\text{type TBD}}$$

Once again, we could construct a table to represent all of the possibilities:

betty	barney	result
int	int	int
int*	int	int*
int	int*	error
int*	int*	int

In this example, the error is caused by the fact that no negative pointers exist. The `int* int*` case is valid in this example because subtracting one pointer from another results in an integer offset value.

12.2 Code Generation

← March 15, 2013

Let's say our compiler is given the following WLPP source code.

```
int wain(int a, int b) {
    return 1 + 2;
}
```

We expect the compiler to output object code. Ideally, we'd get object code like this:

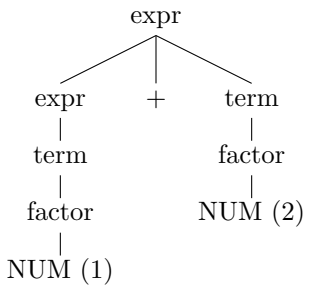
```
lis $3
.word 3
jr $31
```

Throughout the code generation process, we're going to obey our usual MIPS conventions. That is, $1 and $2 are input parameter registers, $3 is the register containing the returned value (or value to return), $30 is a stack, and $31 is a return address.

Generating the object code described above is difficult. Instead, our compilers will likely produce much more verbose (but equivalent) code like this:

```
lis $4
.word 4
lis $3  ;; expr -> term -> factor -> NUM (1) subtree
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $5 ;; term -> factor -> NUM (2) subtree
.word 2
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
jr $31
```

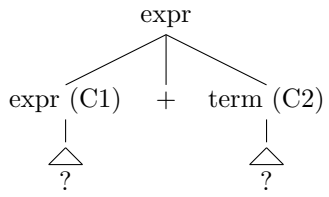
Note that the annotated lines above correspond to the subtrees in this (partial) parse tree:



Conventions Used in this Object Code:

- $4 always contains 4.
- Every expr, term, and factor puts its result into $3.
- The stack is available as usual.

In reality, we don't care about sub-subtrees. We only really care about this subtree, with the expr and term decorated with the associated code from the sub-subtree which is now hidden:



In the above tree, C1 decorating that instance of expr with:

```
lis $3
.word 1
```

Similarly, C2 is decorating that instance of term with:

```
lis $3
.word 2
```

The code produced follows this general format:

```
;; prologue code goes here (set $4, save registers, etc.)
lis $3
.word 1
sw $3, -4($30) ;; save value on stack
sub $30, $30, $4

lis $3
.word 2
add $30, $30, $4
lw $5, -4($30)
```

```
add $3, $3, $5
;; return housekeeping code goes here
```

We'll define the & symbol to mean concatenation, in this case.

Code ($\text{expr}_0 \rightarrow \text{expr}_1 + \text{term}$) =

```
code(expr_1) &
"sw $3, -4($30)
sub $30, $30, $4" & ;; push $3
code(term) &
"add $30, $30, $4
lw $5, -4($30)" &   ;; pop $3
"add $3, $5, $3"    ;; code(+)
```

Similarly, Code(procedure $\rightarrow \ldots$) =

```
"lis $4
.word 4" & ;; prologue
code(dcl_1) &
code(dcl_2) &
code(dcls) &
code(statements) &
code(expr) &
"jr $31" ;; epilogue
```

An alternate way of generating Code(expr $\rightarrow$ expr + term) =

```
code(term) &
"sw $3, -4($30)
sub $30, $30, $4" &
code(expr) &
"add $30, $30, $4 ;; pop stack
lw $5, -4($30)
add $3, $3, $5"   ;; perform addition
```

Let's examine the following WLPP code:

```
int wain(int a, int b) {
    return a + b;
}
```

Let's look at more conventions, specifically how to represent parameters. For now, (and we'll probably change this later when we represent variables), $1 is the first parameter and $2 is the second parameter. We're going to need to expand our symbol table to include how each symbol is represented:

Symbol	Type	Representation
a	int	$1
b	int	$2

Now, we can look at Code(factor → ID) =

```
"add $3, $0, " & representation(lexeme(ID)) ;; lookup in symbol table
```

For example, for variable a, Code(a) = `add $3, $0, $1`, and for variable b, Code(b) = `add $3, $0, $2`.

This assumes our variables are stored in registers. This is only the case for now, and only for `wain`'s parameters. Later, once we start storing all variables on the stack, we'll need to perform a `lw` to fetch the data from the stack, placing the result in $3.

So, the object code equivalent to the WLPP program above is:

```
lis $4
.word 4
add $3, $0, $1
sw $3, -4($30)
sub $30, $30, $4
add $3, $0, $2
add $30, $30, $4
lw $5, -4($30)
add $3, $5, $3
jr $31
```

Code Generation Conventions So Far: ← March 18, 2013

- $1 – first parameter (to be changed).
- $2 – second parameter (to be changed).
- $3 – result from every code fragment.
- $4 – contains 4 (for use when we're constantly pushing/popping the stack).
- $30 – stack. Use as necessary, but reset $30 after every code fragment.
- $31 – return address. Do not clobber.
- Every result is an int (to be changed since int* is also a type in WLPP).

The three conventions that I said we'll change were simplifying assumptions we made earlier. We're going to change those assumptions a bit now.

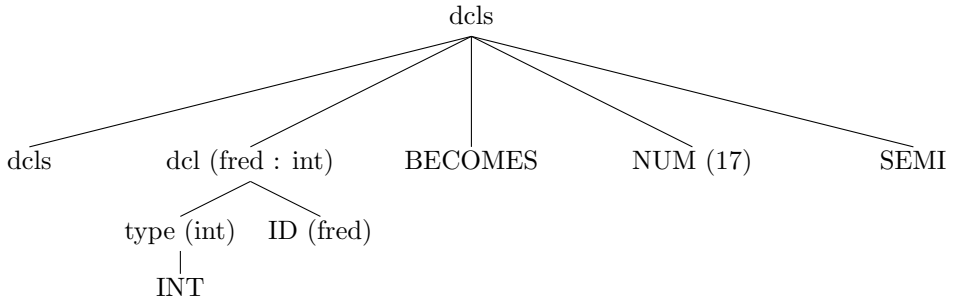
12.2.1 Variables

Variables have declarations and uses, which need to be handled in different ways.

Take this code, for instance:

```
int fred = 17;
int* barney = NULL;
```

The decorated parse tree for the first line in the code snippet above is:



We will have constructed our symbol table to be:

Symbol	**Type**
fred	int
barney	int*

First, we perform the context-sensitive analysis. This ensures that (int, 17) and (int*, barney) are valid pairings.

Next, we need to pick a representation for the variables. Note that we can't use registers only because there are a finite number of them. Instead, we should use RAM. We could use both, but we won't because it's simpler to keep all of our variables in one place.

There are two choices for selecting locations in RAM to store variables:

- Declare a garbage `.word`. However, these variables will act as static variables, which is not always desired (especially when handling recursive programs).
- Use the stack. We're going to use the **stackframe storage allocation** algorithm.

At entry to the prologue of our code, we have a stack pointer, \$30. We'll reserve a large block of storage – enough for every variable – on the stack.

To do this, we decrement the stack pointer (\$30) to below the stackframe and we store a **frame pointer** in a different register (say, \$29).

Suppose there are n declared variables in our program. We must arbitrarily assign each variable a distinct number $0 \leq i < n$. The address in RAM of variable i is $4i + \$29$. Let's say our symbol table now contains:

Symbol	**Type**	i
fred	int	0
barney	int*	1

The address of fred in RAM would be $4(0) + \$29 = \$29$. The address for barney would be $4(1) + \$29 = 4 + \$29$.

The MIPS assembly code to get the value of a variable would be `lw $3, 4i($29)`. Similarly,

the assembly code to place a variable into RAM would be `sw $5, 4i($29)`, where $5 is a register holding the value of the variable presently.

There is a gotcha with this approach, however. The value of 4i in the `lw` instruction must be a 16-bit number, which means we can only create $\approx 8,000$ variables using this method. Just between you, me, and Marmoset, there will not be more than 8,000 variables on this assignment, so this approach is okay for our use.

What kind of code do we need for all of this? If we have the line `int fred = 17;`, we will construct a symbol table containing just "fred" (int) = 0. Once the context-sensitive analysis determines that int and 17 are a valid type match, we can generate the code which will be similar to this:

```
lis $3
.word 17
sw $3, 0($29)
```

Now, suppose instead we have `int* barney = NULL;`. After context-sensitive analysis, we could generate code like this:

```
add $3, $0, $0 ;; 0 represents NULL in this case
sw $3, 4($29)
```

Note that the representation for NULL is conventionally zero.

When representing variables of type int*, there are a few possibilities. You could store a word number, or an offset in an array, but what's highly recommended is that you store the RAM address itself.

What about `wain`'s parameters? We have the following WLPP code:

```
int wain(dcl, dcl) {
    // ...
}
```

Each `dcl` expands into a parse subtree like these:

Earlier, we represented `homer` as $1 and `marge` as $2. However, we want to treat these just like the other variables, by storing them in RAM. Just like normal variable declarations, they are assigned i values and are placed in RAM accordingly. For example, if the i values were 2 and 3 for `homer` and `marge`, respectively, then the following MIPS code would place them in RAM.

```
sw $1, 8($29)  ;; homer
sw $2, 12($29) ;; marge
```

Now, let's look at another simplification we made earlier: the plus operation. Suppose we had the code `homer + marge`. The MIPS assembly code that could be produced is Code(homer + marge) =

```
lw $3, 8($29)  ;; homer
sw $3, -4($30) ;; push $3 onto stack
sub $30, $30, $4
lw $3, 12($29)
add $30, $30, $4
lw $5, 0($30)  ;; pop into $5
add $3, $3, $3 ;; multiply
add $3, $3, $3 ;; by 4
add $3, $5, $3
```

When generating MIPS assembly code for `expr + term`, you must perform many comparisons to determine how to handle various types. The algorithm for this would be something like this:

if *type(expr) == int** **then**
| **if** *type(term) == int* **then**
| | // ...
| **end**
else if *type(expr) == int and type(term) == int* **then**
| // ...
else if *...various other cases...* **then**
| // ...
else
| ERROR
end

This is an important step because the code you generate will actually differ depending on the combination of types.

When performing context-sensitive analysis, don't recompute the type all over the place (such as in recursive calls). Decorate your parse tree instead. Also, make sure you're passing around reference parameters whenever possible. Otherwise, you'll end up with a program with worse complexity.

← March 20, 2013

We have (at least) two choices when generating Code(expr + term). The first approach evaluates expr first:

```
code(expr)
push $3
code(term)
pop $5
add $3, $5, $3
```

Or, we could evaluate the second operand (term) first:

```
code(term)
push $3
code(expr)
```

```
pop $5
add $3, $3, $5
```

This applies similarly for subtraction and multiplication. You shouldn't assume commutability for the `add` instruction – pass the operands in their correct order, even if it does not matter in some cases. In general, for Code(foo OP bar), we have the choice between:

```
code(foo)
push
code(bar)
pop
operations for OP
```

And:

```
code(bar)
push
code(foo)
pop
operations for OP
```

Moving along, we also can trivially determine these code fragments:

- Code(expr → term) = Code(term)
- Code(term → factor) = Code(factor)
- Code(factor → LPAREN expr RPAREN) = Code(expr)

You might be able to be clever and make your compiler look ahead and realize that Code(bar) doesn't use $6 (for instance), so Code(foo) could put its result in $6 directly in order to skip the stack steps. This might be something to try for the bonus, but otherwise, use the $3 stack-based approach for simplicity to get things working.

12.2.2 Statements

Let's look at printing next. We don't want to reproduce all the print code every time we want to print something. Instead, we'll use a procedure. Recall the WLPP grammar rule for printing:

statement → PRINTLN LPAREN expr RPAREN SEMI

Also, recall the generalized code for an arbitrary procedure:

```
;; prologue
push $31
  .
  .
  .
;; epilogue
pop $31
jr $31
print: ...
```

So, Code(statement → PRINTLN LPAREN expr RPAREN SEMI) =

```
code(expr)
add $1, $3, $0 ;; set up the parameter for the print procedure
lis $5
.word print
jalr $5        ;; clobbers $31
```

Note that we changed $1! That was the parameter for wain. However, remember we earlier placed $1's value into a variable on the stack, so everything's okay.

`jalr $5` clobbers $31. This isn't a huge problem, all we have to do is push $31 onto our stack in the prologue and pop it just prior to the `jr $31` call in our prologue.

Let's generate the code for the rule: statements → statements statement. The safest choice for this would be to generate:

```
code(statements)
code(statement)
```

What if we wanted to generate the code differently such that the statement's code was placed prior to the rest of the statements? That is, what if we wanted to generate code like this:

```
code(statement)
code(statements)
```

Is that correct? It might be. We have no way of guaranteeing it in every case. The order of some object code matters, sometimes. It's generally unsafe to do this.

Let's look at this rule: statement → lvalue BECOMES expr SEMI. We first need to decide on a representation for an lvalue. Here are all the rules for what an lvalue can be:

lvalue → ID

lvalue → STAR factor

lvalue → LPAREN lvalue RPAREN

How should we represent an lvalue? Well, we can't just place them in registers, for a couple of reasons. First, there is the limitation on the number of registers we have available for our use. More importantly, however, is the fact that when we change the value (since an lvalue by definition is going to be changed), we want it to be updated in its permanent location in RAM.

We'll take the stupidly simple approach of making each lvalue an address in RAM.

Let's determine Code(statement → lvalue BECOMES expr SEMI):

```
code(expr)
push $3
code(lvalue) ;; places address of lvalue into $3
pop $5
sw $5, 0($3)
```

Alternatively, we can switch up the order of evaluation:

```
code(lvalue)
push $3
code(expr)
pop $5
sw $3, 0($5)
```

In this case, it doesn't matter whether we evaluate (or generate code) for the lvalue or the expr first, so both of these approaches are equally acceptable.

Let's look at Code(lvalue $\rightarrow$ ID). We know that variables are just offsets in the stack-frame. We can get the offset of the ID from the symbol table. We need the *address* of that location, though. So, we get Code(lvalue $\rightarrow$ ID) =

```
lis $3
.word offset(ID)
add $3, $3, $29
```

You may think it'd be easier to just return the offset, rather than a memory address. That isn't easier, however, because we still have the following rules to deal with:

- Code(lvalue $\rightarrow$ STAR factor) = Code(factor), with the assertion that type(factor) = int $\star$.
- Code(lvalue $\rightarrow$ LPAREN lvalue RPAREN) = Code(lvalue).
- Code(factor $\rightarrow$ AMP lvalue) = Code(lvalue). Note that factor must be of type int $\star$, lvalue must be of type int, and the Code(lvalue) will always contain an address, as needed for these types.

12.2.3 Control Structures & Tests

← March 22, 2013

We want to be able to generate code for rules like this:

statement $\rightarrow$ WHILE LPAREN test RPAREN LBRACE statements RBRACE

We first need to be able to generate code for a test. Let's look at this rule: test $\rightarrow$ expr_1 LT expr_2. This allows us to write WLPP code like:

```
while(a < b) {
    a = a + 1;
}
```

A **test** is essentially a boolean value. We'll follow the convention that a boolean value is represented as 0 = false, 1 = true. Note that this is not exactly how booleans are represented in languages like C++ (where instead, any non-zero value is true).

Code(test $\rightarrow$ expr_1 LT expr_2):

```
code(expr_1)
push $3
code(expr_2)
pop $5
slt $3, $5, $3
```

Now that we've generated the code for a test, we can proceed to generate code different control structures like loops and branching.

Code(statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE):

```
loop:
    code(test)
    beq $3, $0, quit
    code(statements)
    beq $0, $0, loop

quit:
```

There are a couple of problems with the code we generated. If we have multiple loops in our program, we'll have multiple loop and quit labels. To solve this, you should affix a unique identifier to the labels, and for readability, you may want to use the same identifier for the loop/quit labels that correspond with each other.

The other problem is if our loop is over $\approx$ 32,000 instructions long, since the i value that `beq` accepts as its third operand must be a 16-bit integer. Here's one way we could solve this:

```
loop123:
    code(test)
    bne $3, $0, cont123
    lis $5
    .word quit123
    jr $5

cont123:
    code(statements)
    lis $5
    .word loop123
    jr $5

quit123:
```

You could make your code dynamically choose between these two methods, depending of the length of code(statements). All control structures are subject to this caveat.

Let's now look at Code(statement → IF LPAREN test RPAREN LBRACE statements_1 BRACE ELSE LBRACE statements_2 RBRACE):

```
    code(test)
    beq $3, $0, else123
    code(statements_1) ;; true part
    beq $0, $0, done123

else123:
    code(statements_2) ;; false part
```

```
done123:
```

Alternatively, we could've generated this code:

```
    code(test)
    bne $3, $0, true123

    code(statements_2)
    beq $0, $0, done123

true123:
    code(statements_1)

done123:
```

The choice between these two is somewhat arbitrary.

One other approach to this is to take advantage of places in your code that can contain code that will never be executed (such as after `jr $31`). You could have:

```
    code(test)
    bne $3, $0, true123
                                                  true123: ;; out of line, somewhere
                                                      statements_1
    statements_2                                      beq $0, $0, done123

done123:
```

If the true part very rarely occurs, you take zero branches on the else case, which is nice. This would save some execution time.

Let's take a look at more test rules.

- Code(test $\rightarrow$ expr$_1$ GT expr$_2$) $\equiv$ Code(test $\rightarrow$ expr$_2$ LT expr$_1$).
- Code(test $\rightarrow$ expr$_1$ GE expr$_2$) $\equiv$ not(Code(test $\rightarrow$ expr$_1$ LT expr$_2$)). According to our convention, 0 = false and 1 = true, so not$(x) = 1 - x$. So, we get:

  Code(test $\rightarrow$ expr$_1$ GE expr$_2$):

  ```
  code(expr_1 LT expr_2)
  lis $11 ;; not $3
  .word 1
  sub $3, $11, $3
  ```

  We should move the populating of \$11 into our prologue, and add \$11 = 1 to our list of conventions.

- Code(test $\rightarrow$ expr$_1$ LE expr$_2$) $\equiv$ Code(test $\rightarrow$ expr$_2$ GE expr$_1$).

- Code(test → expr_1 NE expr_2) ≡ Code(test → expr_1 LT expr_2) OR Code(test → expr_2 LT expr_1). We can generate this code like so:

  ```
  code(expr_1)
  push $3
  code(expr_2)
  pop $5
  slt $6, $5, $3
  slt $7, $3, $5
  add $3, $6, $7
  ```

  Note that the final line is an OR, except this implementation of OR does not work in general. It only works here because we know both conditions won't be true simultaneously. In other cases, we would need to handle the case where both conditions are true, and then our possible truth values would be 1 and 2.

- Code(test → expr EQ expr_2) ≡ not(Code(test → expr_1 NE expr_2)). The code for this is:

  ```
  code(expr_1 NE expr_2)
  sub $3, $11, $3
  ```

12.2.4 Dynamic Memory Allocation

← March 25, 2013

We only have two WLPP grammar rules left to discuss:

$$\text{factor} \rightarrow \text{NEW INT LBRACK expr RBRACK}$$
$$\text{statement} \rightarrow \text{DELETE LBRACK RBRACK expr}$$

You can download `alloc.asm` for use on the assignments. It contains procedures for `new` and `delete`. You may want to optimize (rewrite) them for the bonus question.

`alloc.asm` implements three procedures:

- `init` initializes the allocation code. This must be called prior to any allocations or deletions.
- `new` attempts to allocate n words of storage, where $n = \$1$. It returns a pointer in \$3 if memory was allocated, or NULL otherwise.
- `delete` marks the memory at the address given in \$1 as not needed. The procedure already knows how large the block of memory is. This essentially deletes a pointer that you had allocated earlier in your program.

"This is the Walmart approach to storage. You want to use it for awhile, so you go, you buy it. Then when you're tired of it, you tape it up again and take it back, and they put it back on the shelf for another customer." `new` and `delete` do the same thing as Walmart.

You'll have to remove the `.import` and `.export` statements from `alloc.asm` prior to including it in your outputted code. Those pseudo-instructions are meant to be used by a linker,

but we are not running a linker on our code. Additionally, `alloc.asm` *must* be physically last in your outputted code.

The code for each of the rules above is simply `code(expr)`, followed by a call to `new` or `delete` with n or p in \$1, respectively.

13 Dynamic Memory Allocation: Implementing Heaps

`alloc.asm` implements a **heap**, which is an abstract data type (ADT) (not to be confused with the priority queue implementation), which supports the following operations:

- `initialize`.
- `finalize`.
- `allocate(n)`: finds n available units of RAM, or fails.
- `free(p)`: free the previously allocated RAM that is referred to by p.

The first step to implementing a heap is to identify some available RAM. This RAM is typically in one continuous block, which we will call the **arena**.

Where do we put the arena? It'll have to use some space normally available for the stack. We could create a fixed size arena by dedicating a certain number of words directly above your program's code, and leave the rest of the space for the stack.

The given `alloc.asm` file implements a fixed-size arena. That's why it must be at the very end of our program. It has a label `end:` at the very bottom of it that indicates where your program ends, which is also where the arena begins.

Aside: up until this point, we've made the naïve assumption that the stack is not bounded. In reality, of course the stack is bounded, but we're not sure how large the stack needs to be. Also, you should really check for possible stack overflows, otherwise your program could get overwritten by a user and arbitrary code could be executed.

You could dedicate half of your stack space (conventionally denoted as the memory below the address stored in \$30) to the arena. The other half will be used for the stack. We will call this the **split the baby approach**.

Alternatively, you could start the arena right after your program and start the stack at \$30, and use the first-come, first-serve approach.

13.1 The Loaf of Bread Algorithm

The simplest algorithm for `allocate(n)` is what we will call the **loaf of bread algorithm**. It's simple: you just give out the memory in order. The chunks of memory you're allocating could potentially be chunks of different sizes.

With the loaf of bread algorithm, we can't implement `free(p)`. All sales are final, so

to speak.

If instead we wanted to implement the Walmart algorithm, we would need to find the most appropriate hole when we want to allocate some memory. What if there are many small holes but we need a large one? That is, the total amount of unused storage in your arena is large enough, but there's no continuous block of RAM available. This is known as **fragmentation**.

← March 27, 2013

In the simple case, `alloc()` will allocate k words, where typically $k = 2$. In Lisp or Scheme, `alloc()` is written as `(cons a b)`. A `cons` statement allocates two spaces in memory (`first`/`rest`, also known as `car`/`cdr`).

We could create an arena where we have two pointers: a `bread` pointer (the next slice of bread comes from where the `bread` pointer is pointing to), and an `end` pointer. The slice of bread algorithm for `alloc()` is as follows.

if *bread* < *end* **then**
  tmp = bread;
  bread++;
  return *tmp*
else
  fail
end

This algorithm is a bit hostile because it completely ignores `free`s. Storage is never reclaimed.

13.2 Available Space List

The available space list technique involves building up an array of elements that represent available blocks of memory, split into fixed-size pieces. Each block points to the next one, and we also have three pointers. The `avail` pointer points to the next available piece of memory. We also have `tmp` and `i` pointers, pointing to the end of our list.

`init()` runs in $O(N)$ time, where N is the size of the arena (which is quite slow). It follows this algorithm:

next(1st area) = NULL;
for *i = arena; i < end; i++* **do**
  next(i-th area) = avail;
  avail = i-th area;
end

`alloc()` runs in constant $O(1)$ time by following this algorithm:

if *avail* ≠ *NULL* **then**
  tmp = avail;
  avail = next(avail);
  return *tmp*
else
  fail
end

`free(p)` trivially runs in constant $O(1)$ time by following this algorithm:

next(p) = avail;
avail = p;

This is all pretty nice, except for `init`. We could take a hybrid approach to make `init` faster while maintaining the runtimes of the other operations.

13.3 Hybrid Loaf of Bread and Available Space List

We introduce a `bread` pointer into our available space list, with it initially being set to the beginning of the arena (the first element).

`init()` now follows this algorithm, which runs in constant $O(1)$ time:

bread = arena;
avail = NULL;

`alloc()` now follows this algorithm, which also runs in constant $O(1)$ time (as before):

if *avail* $\neq$ *NULL* **then**
  tmp = avail;
  avail = next(avail);
  return *tmp*
else if *bread* $<$ *end* **then**
  tmp = bread;
  bread++;
  return *tmp*
else
  fail
end

`free(p)` operates identically to before, in constant $O(1)$ time:

next(p) = avail;
avail = p;

This entire algorithm runs in constant time! However, it's important to remember that this algorithm only works for allocations of fixed size.

13.4 Implicit Freedom

Modern languages that aren't archaic all have implicit `free`. How does that work?

Key: find unreferenced storage units (areas that are not pointed to by anyone). We could instead find all of the referenced units, which might be easier.

We can use the stack frame (known variables) as **root pointers**. We can then find all *reachable* units in the arena, and mark them.

How do we mark them, exactly? Observe that the last two bits in a pointer must be zero, because every pointer is a multiple of four. We can use these two bits for a sticky note (a "taken?" sticky note).

This is known as the **mark and sweep algorithm**.

Mark(n) takes $O(n)$ time, where n is the number of reachable units. Sweep(N) takes $O(N)$ time, where N is the size of the arena.

It would be nice if the runtime of this algorithm was proportional only to the memory we're *actually* using, and not the entire arena.

The *average* runtime of this is still $O(1)$, however. This is because we have a lot of really quick operations followed by one occasional *huge* operation.

The mark and sweep algorithm is akin to the procrastination algorithm of letting dirty dishes pile up until there are no clean ones left. Better algorithms, such as those used in incremental garbage collectors, do one piece of the garbage collection (marking or sweeping) in each `alloc()`, to distribute the longer time into all of the smaller $O(1)$ `alloc()` calls.

Garbage collection could also be concurrent. "It's hard to buy a computer that doesn't have 16 cores these days. Let one of your cores be a garbage collector and clean up after you, like a maid or a mother."

13.5 Use Counts

Another approach to garbage collection is **use counts**. We'll start by appending a use count to each piece of allocated storage.

In `alloc()`, we set this use count to 1. When we assign a variable away from a piece of memory, we decrement the use count. When we assign a variable to a piece of memory, we increment its use count. Calls to `delete` will decrement this count.

There are two gotchas with this approach:

- We have to store use counts.
- It doesn't work for cyclic structures. If we have a data structure that points to another data structure that points back to the first one, the use counts are both at 2, and then when we call `delete` they both become $1 \neq 0$, so they are not freed.

Use counts are still a perfectly valid idea in cases where we know we won't have cyclic data structures.

13.6 Copying Collector

As you're using a piece of memory, you copy all of it to a space dedicated for memory that's currently in use. Note that this has the consequence of only letting you effectively use half of the RAM available, maximum.

This copying is also relocating code. That might be tough in languages like C++ where memory addresses are part of the language. You could always convert pointers to integers, strings, or do other crazy things (like reverse the bits in a pointer), and the compiler will not be aware of that use.

Andrew Appel argued that instead of using the stack (which would involve a push and pop), just use `alloc()` instead. It's equivalent to just pushing onto the stack, and removes the need for the popping code.

13.7 Holes

← April 3, 2013

We will refer to unallocated areas of memory as **holes**. That is, holes are areas of memory that are available to fill with some new data. In fixed-size memory allocation, all holes have the same size as each other.

13.8 Variable-Sized Dynamic Memory Allocation

When we start to discuss variable-sized dynamic memory allocation, all of the properties we discussed for fixed-size allocation go out the window. But what would happen if we *tried* to apply the same concepts to variable-sized allocation?

The approach of maintaining an available space list is not sufficient anymore. `alloc(n)` now takes a parameter, n, which represents the size of the chunk of memory we want to allocate. The available space list is a linked list connected in the order of deletions, but we don't know how big each hole in that list actually is.

We somehow need to find a hole that is at least n bytes large, in order to satisfy the `alloc(n)` request. In order to do this, we'll have to store the size of each hole.

In order to find an appropriate hole, we'll have to search through the available space list (the list of holes). There are a few approaches to this search:

- **First fit**: choosing the first hole that is big enough. You don't need to pick sides. This approach is easy to implement. This approach also leaves a variety of hole sizes available for future use.
- **Worst fit**: choosing the biggest hole we can find. This experiences hole erosion – that is, all holes will tend to become very similar sizes over time.
- **Best fit**: choosing the smallest hole that's big enough. Most of the time, you'll be able to use most of the hole you selected, but that'll leave a sliver of storage remaining. That sliver will be hard to fill later on. You waste less space now, but more space later (as the slivers accumulate).

The choice between these three approaches is not clear cut because there is no analytical way to compare these.

With fixed-size allocation, you'll fail only if all memory is full, because all holes are the same size. With variable-size allocation, you'll fail if there is no hole that is big enough.

In archaic languages like C and C++ (that don't have implicit `free`s), `free(p)` frees the storage area that pointer p points to. How do we know how big that area is, though? We'll need to keep track of the length of those areas as well.

So, we'll need to keep track of the length of holes, and the length of allocated areas. We might as well standardize on a way to store these lengths. When you allocate n bytes of storage, you'll actually allocate a few extra bytes that you won't expose to the user. These bytes will act as a header that can contain the length of the block as well as a pointer to the next block (and a footer, which will be discussed later).

Fragmentation occurs when small holes collectively have enough space to satisfy an `alloc(n)` request, but no single hole is big enough to satisfy that request. This occurs with the best fit approach because the holes that turn into sliver-holes become smaller and smaller over time.

13.8.1 Coalescing Adjacent Holes

When you `free` an area, combine it with adjacent hole(s) if there are any. This has the consequence of no two holes ever being beside each other anymore.

The process of coalescing adjacent holes does not guarantee that you'll solve the problem of fragmentation, it only guarantees that you aren't destined to fail.

There are a few cases to consider when you're trying to `free` an area pointed to by p:

- p is surrounded by two holes. This results in one single hole as p is freed, which means we eliminated one other hole.
- p has other allocated memory on its left, and a hole on the right. We expand the hole as p is freed. This causes no change in the number of holes.
- p has other allocated memory on its right, and a hole on the left. We expand the hole as p is freed. This causes no change in the number of holes.
- p is surrounded by allocated memory on both sides. This results in a new hole as p is freed.

We need some way of indicating which blocks of memory are holes and which are allocated. We could add another bit to the header. This introduces a problem, though: how do we find the header of the previous area, if we don't know its size? We could add the length of the block to the end of each area (adding a footer), as well as keeping the length in the header.We also need a pointer backwards to the previous area. This makes our linked list a doubly linked list.

At the end of the day, each block of memory will have:

- **Header**: length of the block, pointer forward, pointer backward, and a `hole?` bit.
- **Footer**: length of the block.

13.8.2 The Buddy System

The `alloc.asm` file we were given for assignment 11 implements the buddy system approach. This approach works by allocating a number of words that is a power of two. This has the consequence of there only being $\lg n$ possible sizes, which means up to 50% of memory could

be wasted in the worst case.

If the size of a hole is double the size you need, simply split the hole in two and use half of it. Otherwise, use the entire hole.

The buddy system approach doesn't perform better, and it's not easier to implement than the other approaches. It's just an alternative.

The bottom-line of variable-sized allocations is there's no perfect solution. It sucks. It gets particularly messy if you can't do relocation, such as in C/C++.

13.9 The 50% Rule

Donald Knuth has stated many important results in computer science. One of them applies here, called The 50% Rule.

Theorem (The 50% Rule)**.** At equilibrium, the number of holes tends to be approximately half the number of allocated areas.

There's also an interesting corollary to this.

Theorem (Corollary)**.** If there is as much free space as allocated space in the arena, the average hole size is double the average allocated area.

In the first fit approach, the size will on average be twice the size you actually need. You'll find an appropriate hole pretty quickly as a result of this. We can even go so far as to say that finding an appropriate hole with the first fit approach operates in constant $O(1)$ time.

13.10 Beyond Memory

These principles apply beyond memory. Allocations on permanent filesystems have the same key issues. Most filesystems do a pretty good job at reducing fragmentation, but Windows' filesystem does not and sometimes users have to manually de-fragment Windows machines.

14 Compiler Optimization

← April 5, 2013

Optimizing a compiler involves seeing into the future. Compiler optimization is not strictly a mathematical problem. The combinatorics and optimization people get pissed when we call it compiler "optimization", because that implies that there's an optimal solution, but there is not.

There's no efficient algorithm to optimize a compiler. In fact, there is no algorithm at all, not even an inefficient one. Measuring how well you optimized your compiler is also not well-defined, because ideally we'd have to measure how well our compiler does on all possible programs (which cannot be enumerated).

The halting problem also has an effect here. If we could determine if a piece of code would result in an infinite loop, we could make further improvements with that in mind.

We'll aim to *improve* our compilers as much as possible, using a proxy measurement technique of some sort.

14.1 Folding

Folding is the act of evaluating constant expressions at compile-time. For example, `x = 1 + 2 + 3;` could be replaced by `x = 7;`. Similarly, `y = x + 2;` could be replaced by `y = 9;`, but only if we know that x is not changed between the two expressions involving it.

Code(expr) implicitly returns a tuple containing (encoding, value). Up until this point, we've always returned (register, 3), because we've always placed the result of the expression in $3.

If we implement folding, we would return something like (register, 3) or (constant, 7) in some cases (where 3 could be any arbitrary register, and 7 is the arbitrary value of the constant).

Code(expr + term) will return the result of this addition if the result of the expression and the result of the term are both constant.

14.2 Common Subexpression Elimination

Let's say you had the following code:

```
x = y + z * 3;
a = b + z * 3;
```

Notice that `z * 3` occurs in both of these expressions. `x * 3` is called a **common subexpression**. We can eliminate common subexpressions like these by only performing that computation once, like this:

```
t = z * 3;
x = y + t;
a = b + t;
```

However, it's important to note that we can only do this if the value of z does not change between the two uses.

Similarly, when we have `x = x + 1`, we can avoid computing the address of x twice. You can instead just compute the address for the lvalue x, and then use that to get the data value of x, without computing the address twice.

14.3 Dead Code Elimination

Sometimes source code contains dead code. For instance:

```
if (0 == 1) {
code_A
} else {
code_B
}
```

Notice that in this code sample, code$_A$ will never be executed. After we perform folding, we can often determine if one path of a branch will never be taken, like in this case. So, we can remove all the code for code$_A$ and we can also remove all the code for the if statement itself, leaving only code$_B$.

We could keep track of the values of variables (as much as possible) in our compiler. Then, we'd be able to determine if a condition like `x < 0` will ever be truthful or not.

14.4 Partial/Abstract Evaluation

We could keep track of whether a given variable x is $< 0, = 0$, or > 0 at any given point. We can determine this for variables recursively in an expression like `x = y * z` as well, by knowing the abstract evaluation of y and z.

You may even be able to determine a bounded set of values that these variables could possibly have. Alternatively, you may be able to come up with a regular expression that represents all values that x could take on.

14.5 Loop Optimization

Loop optimization is done typically to improve the run-time speed of the programs you produce. It doesn't necessarily make your code smaller. In fact, in many cases optimized loops will actually produce more code than unoptimized loops.

14.5.1 Lifting

Let's say you have code like this:

```
while(test) {
   .
   .
   .
  x = y + z * 3; // code to lift
   .
   .
   .
}
```

Let's assume y and z don't change inside the loop. We could lift this statement out of the loop so it'll only be executed once, as needed. Basic lifting would produce (incorrect) code like this:

```
x = y + z * 3; // lifted code
while(test) {
   .
   .
   .
}
```

Lifting will actually make the loop perform worse if the loop executes zero times. It would also produce incorrect code, since for example, if the loop above wasn't supposed to be executed at all, x should not take on the value `y + z * 3`. Instead, we would produce code like this:

```
if (test) {
  x = y + z * 3; // lifted code
  do {
  .
  .
  .
  } while(test);
}
```

14.5.2 Induction Variables

Suppose you had code like this:

```
for(i = 0; i < 10; i += 1) {
  x[i] = 25;
}
```

Note that `x[i] = 25;` is equivalent to `*(x + i) = 25;`, which implicitly multiplies i by 4. We want to avoid performing that multiplication on every iteration of the loop. Instead, we'll produce code like this:

```
for(I = 0; I < 40; I += 4) {
  (addr(x) + I) = 25;
}
```

14.6 Register Allocation

Register allocation is the mother of all variable optimizations. We want to use registers for storing variables, and for storing temporary results of subexpressions.

There is a problem, however. Recall that we only have a finite number of registers.

14.6.1 Register Allocation for Variables

Throughout our program, we'll keep track of **live ranges** of each variable. A live range is the point at which a variable is assigned to the last point where the variable is used with that assignment.
For example:

```
int x = 0;
int y = 0;
int z = 0;
x = 3;
y = 4;
x = x + 1;
```

```
println(x);
println(y);
z = 7;
println(z);
```

In this code, the live range for x is from `x = 3;` to `println(x);`. The live range for y is from `y = 4;` to `println(y);`. Finally, the live range for z is from `z = 7;` to `println(z);`.

This follows the great philosophical discussion where if a tree falls in a forest and no one is there to hear it, does it make a sound? If we set a variable to a value that is never used, do we have to set that variable to that value? The answer is no.

Two variables are said to **interfere** is their live ranges intersect. An **interference graph** is a graph where every variable is a vertex and edges are defined by intersecting variables.

We can re-use registers for two different variables that don't interfere. This is called **register assignment**.

Register assignment is an example of graph coloring. Graph coloring is a well known NP-complete problem.

14.6.2 Register Allocation for Subexpressions

Let's now look at using registers to store the temporary results of subexpressions. Recall from earlier, we had Code(expr + term) being:

```
code(expr)
push $3
code(term)
pop $5
add $3, $5, $3
```

Alternatively, we could have also generated:

```
code(term)
push $3
code(expr)
pop $5
add $3, $3, $5
```

Both of these approaches do a lot of stupid work, all because of our convention that all results are to be placed in $3. Since we're placing all of our results in $3, we have to constantly store that result somewhere else (i.e. on the stack). This is called **register spilling**.

It'd be nice to avoid register spilling wherever possible. It'd be nice if we could specify where we want the result to be placed, or to have some way to indicate that the result was placed in an alternate location. We have to tell our code generator which registers it's allowed to use, in order to indicate to it where it can place its result, as well as which registers it is allowed to clobber.

For all code generators, we now have Code(tree, avail) (where avail is the set of available registers) which will return a tuple containing the code and the representation of the result. For example, we'll have Code(expr + term, avail):

```
code(expr, avail) ;; let's say this returned that it placed its result in $r.
code(term, avail\{$r$}) ;; set difference
add $t, $r, $s ;; where $t is a member of avail.
```

We could've produced similar code for the case where we executed code(term) first:

```
code(term, avail) ;; let's say this returned that it placed its result in $r.
code(expr, avail\{$r$}) ;; set difference
add $t, $s, $r ;; where $t is a member of avail.
```

Ultimately, we need to determine how many registers a given expression needs, and then we choose to evaluate the one that needs the maximum number of registers first.

This approach works remarkably well. The number of registers you need is equal to the depth of the expression tree, at worst.

14.7 Static Single Assignment

Most compilers in the real world use a flow representation of the program their given. A flow representation analyzes the control flow of the program, rather than the in-order flow of the given program.

If you're interested in learning more about this, check out the page about it on Wikipedia.

15 Course Overview and Other Courses

← April 8, 2013

15.1 Course Overview & Final Coverage

The course started off discussing using abstractions that were implemented by other people, including:

- Binary representation. The meaning is in the eye of the beholder.
- Stored program computers. We programmed MIPS programs in machine code.
- MIPS assembly language. We examined the meaning of assembly language and we programmed MIPS programs using assembly language.

We then proceeded to implement some of our own abstractions:

- Assemblers.
- Formal languages.
  - Regular languages.
  - Finite automata (DFAs, NFAs, etc.). DFAs may show up in some other context on the final.

  - Regular expressions.
  - Scanning and searching.

All of the topics discussed up to this point were covered by the midterm. They may also appear on the final, but with less weight than the remainder of the course. The final will mostly consider the following topics:

- Context-free languages and their grammars.
- Derivations.
- Top-down (LL(1)) and bottom-up parsing (LR(1)).
  - Note that it's not easy to construct the DFA needed by a LR(1) parser by hand (we used an automated tool for this).
- Context-sensitive specification.
- Syntax-directed translation.
- Compilers.
  - Code generation.
  - The need for conventions.
  - Code generation for each WLPP grammar rule.
- Dynamic storage allocation.
  - Differences and similarities between fixed- and variable-sized allocation.
  - Copying and relocation.
  - Garbage collection.
  - Note that we didn't have any assignments on dynamic memory allocation, so we won't be expected to write actual code for this. We might be asked for a diagram or pseudocode, though.
- Compiler optimization.
  - Be aware of the key issues.
  - We aren't expected to write code that optimizes a compiler.

15.2 Related Waterloo CS Courses

There are several other computer science courses offered at Waterloo that are relevant to CS 241 and its content.

- **CS 251** (computer organization and design). The course starts with a stored program computer, but asks "how do we build them?" This is useful if you're stranded on a desert island with a bucket of parts from an electronics shop and you wish to build a stored program computer, without the help of an engineer.

- **CS 240** (data structures and algorithms). This course involves the RAM method of computation. Most of the data structures and algorithms discussed in this course assume that your data lies in RAM.
- **CS 245** (logic and computation). This is a course about logic. The course varies a bit based on who's teaching it. That's all you really need to know.
- **CS 360 / CS 365** (introduction to the the theory of computing). This is a hard-core theory course. This course analyzes Turing machines, which are an alternative model of computation. They were one of the first abstractions of computing, invented by Alan Turing. He used them to prove that some problems are unsolvable, such as the halting problem. The course also looks at context-free parsers.
- **CS 444** (compiler construction). This is the hard-core compilers course. If you just can't get enough of assignments 8 through 11, you'll love this course. You'll write a compiler for a non-trivial language.
- **CS 462** (formal languages and parsing). This course involves strings and string matching, which is a subtle problem with complex theory behind it.