Foundations of Sequential Programs

CS241

Jaiden Ratti

Prof. Chengnian Sun

Contents

1 Lecture 1 **2**

2 Lecture 2 **4**

3 Lecture 3 **6**

4 Lecture 4 **8**

5 Lecture 5 **11**

6 Lecture 6 **16**

7 Lecture 7 **20**

8 Lecture 8 **23**

9 Lecture 9 **28**

10 Lecture 10 **31**

11 Lecture 11 **33**

12 Lecture 12 **35**

13 Lecture 13 **38**

14 Lecture 14 **41**

15 Lecture 15 **44**

16 Lecture 16 **47**

17 Lecture 17 **52**

18 Lecture 18 **57**

19 Lecture 19 **62**

20 Lecture 20 **65**

21 Lecture 21 **72**

22 Lecture 22 **77**

23 Lecture 23 **82**

24 Lecture 24 **84**

1 Lecture 1

Question

What is a sequential program?

Question

What really happens when I compile and run a program?

Question

How does a computer take code and turn it into something it can utilize?

By the end of the course, there should be very little mystery left about computers or computer programs.

High level overview of compilers

Compiler: Scans (turns into substrings) → Parses → Optimizes → Codegen {backend}

Definition 1.1

A bit is a binary digit (0 or 1)

Definition 1.2

A nibble is 4 bits (1001)

Definition 1.3

A byte is 8 bits (10011101)

Definition 1.4

A word is a machine-specific grouping of bytes. For us, a word will be 4 bytes (32-bit architecture) though 8-byte (64-bit architecture) words are more common now.

Definition 1.5

The base-16 representation is called the hexadecimal system. It consists of the numbers from 0-9 and the letters a-f (convert the number from 10 to 15 in decimal).

The binary number 10011101 will convert to $9d$ in hexadecimal. Break it into 2 nibbles.

$\underbrace{1001}_{9}$ and $\underbrace{1101}_{d} = 9d$

$0x9d$, the $0x$ denotes a hexadecimal representation (or can do $9d_{16}$). A conversion table will be provided in the exam.

Bytes as binary numbers

- Unsigned (non-negative integers)
- Signed integers

Unsigned

The value of a number stored in this system is the binary sum, that is

$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0$ where b_n is either 0 or 1

For example,

$01010101_2 = 2^6 + 2^4 + 2^2 + 2^0 = 64 + 16 + 4 + 1 = 85_{10}$

$11111111_2 = 255_{10}$

A byte (unsigned) can represent 256 numbers

Converting from decimal to binary:

Approach 1: Take the largest power of 2 less than n, subtract and repeat.

Approach 2: Repeatedly divide by 2

The remainder (from bottom to top) is the binary representation

$\frac{N-b_0}{2} = b_1 + 2b_2 + 2^2 b_3 = 19$ (when $N = 38$). Remainder $b_0 = 0$

Signed Integers

Question

How do we represent negative integers?

Attempt 1: Make the first bit a signed bit. This is called the "sign-magnitude" representation.

0 represents + and 1 represents −, use the rest of the bits to create the number.

We get positive and negative zero.

$(+1) \quad 00000001 + (-1) \quad 10000001 = (2) \quad 10000010$

Attempt 2: Two's complement form

Similar to sign-magnitude in spirit. First bit is 0 if non-negative, 1 if negative (MSB is either $-b_n 2^n$)

Negate value by just subtracting from zero and letting it overflow.

A trick to get the same thing:

- Take the complement of all bits and then add 1.
- Or, locate the rightmost 1 bit and flip all the bits to the left of it.

Decimal to Two's Complement

Compute -38_{10} using one byte of space. First write 38 in binary.

$0010 \quad 0110_2$

Negate this number

$1101 \quad 1010_2$ in 2's complement.

Two's Complement to Decimal

To convert 11011010_2 to decimal, one method is to flip the bits and add 1 (or do the shortcut). Then compute and convert positive to negative number.

Another way is to treat as unsigned and subtract

$11011010_2 = -2^8 + 2^7 + 2^6 + 2^4 + 2^3 + 2^1 = 218 - 256 = -38$

2 Lecture 2

Warmup

-15 to twos complement

$15 \to 00001111$ (absolute value)

Convert to twos complement (shortcut)

$-15 \to 11110001$

-1 to twos complement

$1 \to 00000001$

Convert to twos complement (shortcut)

$-1 \to 11111111$

128 to twos complement

$128 \to 10000000$

$-128 \to 01111111$

Convert to absolute value using shortcut

$128 \to 10000000$

This is wrong. Cannot represent 128 this way (but can represent -128)

```
int abs(int a);
```

```
if (a >= 0) return a;
else return -a;
```

Bytes as Characters

ASCII uses 7 bits to represent characters.

Note that 'a' is different than 0xa. Former is the decimal number 97 in ACII, latter is the number 10 in decimal.

```
int main() {
    printf("%c", 48);
    return 0;
}
```

`Prints $0$.`

Bit-Wise Operators Suppose we have

`unsigned char a = 5, b = 3`

$a = 5 = 0000 \quad 0101$

$b = 3 = 0000 \quad 0011$

Bitwise not ~: negate bits (unary operator)

$c = \sim a = 1111 \quad 1010$

Bitwise and &: (binary operator)

$c = a \& b = 0000 \quad 0001$

Bitwise or |: (binary operator)

$c = a|b = 0000 \quad 0111$

Bitwise exclusive or ^: (binary operator)

$c = a\hat{}b = 0000 \quad 0110$

Returns 1 $\iff$ bits are different

Bitwise shift right or left >> and << (still dealing with unsigned)

$c = a >> 2 = 00000001$

Discard rightmost n bits and fill left n $0's$

$\frac{a}{2^n}$ where n is the number of bits shifted

$c = a << 3 = 00101000$

Discard leftmost n bits and fill right $n0's$

$2^n \cdot a$ where n is the number of bits shifted

Question

What is a Computer Program?

Programs operate on data.

Programs are data. This is a von Neumann architecture: Programs live in the same memory space as the data they operate on.

Programs can manipulate other programs.

We will use 32-bit MIPS in this course

CPU

- 32 general purpose registers
- Control Unit
- Memory
  - Registers
  - L1 cache
  - L2 cache
  - RAM
  - Disk
  - Network memory
- ALU

Registers are very fast memory.

Some general-purpose registers are special:

- `$0` is always 0
- `$31` is for return address
- `$30` is our stack pointer
- `$29` is our frame pointer

Code is just data: it is stored in RAM.

MIPS takes instructions from RAM and attempts to execute them.

Recall Problem: We only know from context what bits have what meaning, and in particular, which are instructions.

Solution: Convention is to set memory address 0 in RAM to be an instruction.

Question

Problem: How does MIPS know what to do next?

Solution: Have a special register called the Program Counter (PC) to tell us what instruction to do next.

Question

Problem: How do we put our program into RAM?

Solution: A program called a loader puts our program into memory and sets the PC to be the first address.

Fetch-Execute Cycle

```
PC = 0
while true do
    IR = MEM[PC] // loading the first word/instruction into IR
    PC += 4 // update PC to the next instruction
    Decode and execute instruction in IR // note we increment PC first
end while
```

This is the only program a machine really runs. This is the essence of the CPU cycle.

First Example: Addition

Write a program in MIPS that adds the values in registers `$8` and `$9` and stores the result in register `$3`

`add:`Add

`000000ssssstttttddddd00000100000`

`$s = source`, `$t = second source`, `$d = destination`

000000|01000|01001|00011|00000100000

Putting values in Registers

`lis`: Load Immediate & Skip

`0000000000000000ddddd00000010100`

Load value 1 into register `$3`

`lis: (000 ... 00011 ...)` load and escape to the next instruction `value: 000 ... 1` (which is 1)

`$d = MEM [PC]; PC += 4`

3 Lecture 3

Recall Fetch-Execute Cycle

```
PC = 0 // mem address of next instruction
while true {
   IR = MEM[PC]
   PC = PC+4
   Decode IR then execute
}
```

`add`: Add

000000 $\underbrace{sssss}_{\text{operand reg}}$ $\underbrace{ttttt}_{\text{operand reg}}$ $\underbrace{ddddd}_{\text{result}}$ 00000 10000

`s = source`, `t = second source`, `d = destination`

`lis`: Load Immediate & Skip, for putting values directly into registers.

$00\ldots0$ $ddddd$ $00\ldots1..0$

```
$3 = 1
0x0: 000... 00 00011 0....1..0 (d = 3)
0x4: 0000000000000000000000000000001 (value 1)
```

`$d = MEM[PC]; PC + 4`

```
$d = MEM[PC] // load
$3 = MEM[0x4]
PC becomes 0x8
```

Add values 11 and 13 and store the result in register 3.

```
0x0: lis 01000 //
0x4: 11 as a 32 bit word
0x8: lis 01001
0xc: 13 as a 32 bit word
0x10: add 01000 and 01001 and store in 00011
```

Question

How do we stop?

Operating system provides a return address in register `$31`.

We get to that return address via `jr` (Jump Register).

Multiplying two words together might give a word that requires twice as much space.

To deal with this, we use two registers: `hi` and `lo`.

`hi` has the leftmost 32 bits, and `lo` has the rightmost 32 bit.

Division performs integer division and stores the quotient in `lo` and remainder in `hi`.

`hi` and `lo` are special purpose registers: they don't have register numbers.

`mfhi` and `mflo` are general purpose registers.

`mfhi: $3 11 * $3 = hi`

Larger amount of memory is stored off the CPU.

RAM access is slower than register access (but is larger).

Data travels between RAM and CPU via the bus.

Words occur every 4 bytes, starting with byte 0. Indexed by $0, 4, 8, \ldots n - 4$.

Cannot directly use the data in the RAM. Must transfer first to the registers.

Operations on RAM.

Load word takes a word from RAM and places it into a register. Specifically, load the word in `MEM[$s + i]` and store in `$t`.

`load $t $s(i)` where `$t: result reg; $s: mem address; i: 16-54 bit signed offset (number)`

```
0x4444
lis $3 <- mem address
binary word for 4
load $3, $3(0)
// register 3 has the value in memory address 4
```

Load ≡ Read and Store ≡ Write

Store word takes a word from a register and stores it into RAM. Specifically, load the word in `$t` and store it in `MEM[$s + i]`.

```
store $0, $3(0)$
MEM[$3 + 0] = 0
```

Machine code is difficult to memorize and edit.

Assembly language is a text language which there is a 1-to-1 correspondence between assembly instructions and machine code instructions. This is more human-readable. Higher-level languages have a more complex mapping to machine code.

4 Lecture 4

A string like `add $3, $2, $1` is stored as a sequence of characters.

If we can break this down to meaningful chunks of information, it would be easier to translate.

`[add][$3][$2][$1]`

A scanner breaks strings down into tokens. Not as simple as looking for spaces, the words need to make sense.

Scanner extracts the Kind and Lexeme

`Indentifier: "sub"`

```
int a = 241;
INT: int
Identifier: "a"
EQ: =
NUM: 241
SEMI: ";"
```

The string “241” has four ways to be split up.

In the C statement `int x = 241;` we want to interpret “241” as a single number.

Definition 4.1

Maximal munch: Always choose the longest meaningful token while breaking up the string

Question

Then how do we know when to stop? There's no limit to how long a number can be.

We need a way to wrangle infinitely many possible strings.

Definition 4.2

An alphabet is a non-empty, finite set of symbols, often denoted by Σ

Definition 4.3

A string (or word) w is a finite sequence of symbols chosen from Σ.

Definition 4.4

The set of all strings over an alphabet Σ is denoted by $\Sigma*$

Definition 4.5

A language is a set of strings

Definition 4.6

The length of a string w is denoted by $|w|$

Alphabets:

$\Sigma = \{a, b, c, \ldots, z\}$ the Latin alphabet

$\Sigma = \{0, 1\}$ the alphabet of binary digits

$\Sigma = \{0, 1, 2, \ldots, 9\}$ the alphabet of base 10 digits

$\Sigma = \{0, 1, 2, \ldots, 9, a, b, c, d, e, f\}$ hexadecimal

Strings:

ε is the empty string. It is in $\Sigma*$ for any Σ. $|\varepsilon| = 0$

For $\Sigma = \{0, 1\}$, strings include $w = 011101$ or $x = 1111$. Note $|w| = 6$ and $|x| = 4$.

Languages:

$L = \emptyset$ or $\{\}$, the empty language

$L = \{\varepsilon\}$, the language consisting of (only) the empty string

$L = \{ab^n a : n \in \mathbb{N}\}$, the set of strings over the alphabet $\Sigma = \{a, b\}$ consisting of an a followed by 0 or more b characters followed by an a.

The objective of our scanner is to break a string into words in a given language.

Simpler objective: Given a language, determine if a string belongs to the language.

How hard is this? Depends on the language.

$L =$ any dictionary: Trivial

$L = \{ab^n a : n \in \mathbb{N}\}$: Very easy

$L = \{$Valid MIPS assembly programs $\}$: Easy

$L = \{$Valid Java/C/C++ programs $\}$: Harder

$L = \{$Set of programs that halt $\}$: Impossible

Memberships in Languages

In order of relative difficulty:

- Finite
- Regular
- Context-free
- Context-sensitive
- Recursive
- Impossible languages

Consider the language {bag, bat, bit}.

We can just check whether a string is in the list.

Question

But can we do this in a more efficient way?

Hash Map/Set

```
for each w in L {
    if w' = w
        Accept
}
```

Still not the most efficient

Most efficient is to check each letter at a time and reject if it can't be possible.

In our example, all our words start with b. If our first symbol for input is not b, we can instantly reject. If the first symbol is b and our second is a, we can reject "bit" and keep going. Similar to prefix tree.

Important Features of Diagram

- An arrow into the initial start state
- Accepting states are two circles
- Arrows from state to state are labelled
- Error state(s) are implicit

Definition 4.7

A regular language over an alphabet Σ consists of one of the following:
1. The empty language and the language consisting of the empty word are regular
2. All languages $\{a\}$ for all $a \in \Sigma$ are regular.
3. The union, concatenation or Kleene star of any two regular languages are regular.
4. Nothing else

Let L, L_1, L_2 be three regular languages. Then the following are regular languages

Union: $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$

Concatenation: $L_1 \cdot L_2 = L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$

Kleene star: $L^* = \{\varepsilon\} \cup \{xy : x \in L^*, y \in L\} = \cup_{n=0}^{\infty} L^n$

Suppose that $L_1 = \{\text{up, down}\}$, $L_2 = \{\text{hill, load}\}$ and $L = \{a, b\}$ over appropriate alphabets. Then

- $L_1 \cup L_2 = \{\text{up, down, hill load}\}$
- $L_1 \cdot L_2 = \{\text{uphill, upload, downhill, download}\}$
- $L^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \ldots\}$

Let $\Sigma = \{a, b\}$. Explain why the language $L = \{ab^n a : n \in \mathbb{N}\}$ is regular.

Solution: $\{a\}$ and $\{b\}$ are finite, and so regular. $\{b\}^*$ is also regular, regular languages are closed under Kleene star. Then, the concatenation $\{a\} \cdot \{b\}^* \cdot \{a\}$ must also be regular.

In tools like `egrep`, regular expressions are often used to help find patterns of text. Regular expressions are just a way of expressing regular languages.

The notation is very similar, except we drop the set notation. As examples

- $\{\varepsilon\}$ becomes ε
- $L_1 \cup L_2$ becomes $L_1 | L_2$
- Concatenation is never written with the explicit $\cdot$
- Order of operations: $^*, \cdot, |$
- ? means optional

Extending the Finite Languages Diagram

We can allow our picture to have loops.

Definition 4.8

Deterministic Finite Automata (DFA). A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$

- Σ is a finite non-empty set (alphabet)
- Q is a finite non-empty set of states
- $q_0 \in Q$ is a start state
- $A \subseteq Q$ is a set of accepting states
- $\delta : (Q \times E) \to Q$ is our total transition function (given a state and a symbol of our alphabet, what state should we go to?).

5 Lecture 5

Creating Binary in C++

How do we write the binary output

`0001 0100 0100 0000 1111 1111 1111 1101`

`bne $2, $0, -1`

Convert the registers to binary bits

We can use bit shifting to put the information into the correct position.

`bne` opcode is 5

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | offset
```

Shift opcode 26 left, shift binary representation of register `$s` left 21 bits, shift binary representation of register `$t` by 16 bits.

We need to be careful with the offset.

Recall in C++, ints are 4 bytes. We only want the last two bytes. First we need to apply a “mask” to only get the last 16 bits.

`int offset = -1`

32-bit `000....1`

32-bit `111....0` bitwise or

32-bit `111....1`

This does not work since we are changing all 32 bits, we only need to change the last 16 bits.

We discard the leftmost 16 bits above.

`int offset = -1 & 0xffff`

Can declare the offset to be `int16_t` (still need to do the bit masking).

Just need to `cout << instr` right? Wrong.

This output would be 9 bytes, corresponding to the ASCII code for each digit of the instruction as interpreted in decimal. We want to put the four bytes that correspond to this number.

Printing Bytes in C++

```
int instr = (5 << 26) | (2 << 21) | (0 << 16) | (-1 & 0xffff);
unsigned char c = instr >> 24;
cout << c;
c = instr >> 16; cout << c;
c = instr >> 8; cout << c;
c = instr; cout << c;
```

You can also mask here to get the "last byte" by doing & `0xff` if worried about which byte will get copied over.

Rules for DFA

States can have labels inside the bubble, this is how we refer to the states in Q.

For each character, follow the transition. If there is none, go to the implicit error state.

Once the input is exhausted, check if the final state is accepting. If so, accept. Otherwise, reject.

Warm-up Problem

Write a DFA over $\Sigma = \{a, b\}$ that

1. Accepts only words with an even number of as
2. Accepts only words with an odd number of as and an even number of bs
3. Accepts only words where the parity of the number of as is equal to the parity of the number of bs
4. Write a DFA over $\Sigma = \{a, b\}$ that accepts all words ending with bba.

Start

1. Accepts only words with an even number of as

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1\}$

q_0 is our start state

$A = \{q_0\}$

$q_0 :=$ even as

$q_1 :=$ odd as

δ is defined by

- $\delta(q_0, b) = q_0$
- $\delta(q_0, a) = q_1$
- $\delta(q_1, b) = q_1$

- $\delta(q_1, a) = q_0$

2. Accepts only words with an odd number of as and an even number of bs

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1, q_2, q_3\}$

q_0 is our start state

$A = q_1$

$q_0 \coloneqq$ even as, odd bs

$q_1 \coloneqq$ odd as, even bs

$q_2 \coloneqq$ even as, odd bs

$q_3 \coloneqq$ odd as, odd bs

δ is defined by:

- $\delta(q_0, a) = q_1$
- $\delta(q_0, b) = q_2$
- $\delta(q_1, a) = q_0$
- $\delta(q_1, b) = q_3$
- $\delta(q_2, a) = q_3$
- $\delta(q_2, b) = q_0$
- $\delta(q_3, a) = q_2$
- $\delta(q_3, b) = q_1$

3. Accepts only words where the parity of the number of as is equal to the parity of the number of bs

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1, q_2, q_3\}$

q_0 is our start state

$A = q_0, q_3$

$q_0 \coloneqq$ even as, odd bs

$q_1 \coloneqq$ odd as, even bs

$q_2 \coloneqq$ even as, odd bs

$q_3 \coloneqq$ odd as, odd bs

δ is defined by:

- $\delta(q_0, a) = q_1$
- $\delta(q_0, b) = q_2$
- $\delta(q_1, a) = q_0$
- $\delta(q_1, b) = q_3$
- $\delta(q_2, b) = q_0$
- $\delta(q_3, b) = q_1$
- $\delta(q_2, a) = q_3$
- $\delta(q_3, a) = q_2$

This is not the optimal solution.

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1\}$

q_0 is our start state

$A = q_0$

$q_0 :=$ same parity

$q_1 :=$ different parity

δ is defined by:

- $\delta(q_0, a) = q_1$
- $\delta(q_0, b) = q_1$
- $\delta(q_1, a) = q_0$
- $\delta(q_1, b) = q_0$

4. Write a DFA over $\Sigma = \{a, b\}$ that accepts all words ending with bba.

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1, q_2, q_3\}$

q_0 is our start state

$A = q_3$

δ is defined by:

- $\delta(q_0, a) = q_0$ (self loop)
- $\delta(q_0, b) = q_1$
- $\delta(q_1, b) = q_2$
- $\delta(q_2, a) = q_3$
- $\delta(q_1, a) = q_0$ (need to start over)
- $\delta(q_2, b) = q_2$ (self loop)
- $\delta(q_3, a) = q_0$ (start over)
- $\delta(q_3, b) = q_1$ (start over but not at beginning)

Definition 5.1

The language of a DFA M is the set of all strings accepted by M, that is: $L(M) = \{w : M \text{ accepts } w\}$

```
w = a_1a_2...a_n
s = q_0
for i in 1 to n do
    s = \delta(s,ai)
end for
if s in A then
    Accept
else
    Reject
end if
```

You could also use a lookup table.

Theorem 5.2: Kleene

L is regular if and only if $L = L(M)$ for some DFA M. That is, regular languages are precisely the languages accepted by DFAs.

Question

Is C a regular language?

The following are regular

- C keywords
- C identifiers
- C literals
- C operators
- C comments

Sequences of these are also regular (Kleene star). Finite automata can do our tokenization.

Question

What about punctuation? Even simpler, set $\Sigma = \{(,)\}$ and $L = \{$strings with balanced parentheses$\}$. Is L regular?

```
w = ()
w = (())
w = ((((()))((())))
```

This language is not regular. It is a context-free language (more on this later).

Question

How does our scanner work?

Our goal is: Given some text, break up the text into tokens.

Some tokens can be recognized in multiple different ways.

`w = 0x12cc`. This could be a single hex, or could be an int followed by an (x) followed by another int (1) followed by another int (2) and followed by an id (cc).

Formalization of this problem.

Given a regular language L (say, L is all valid MIPS or C tokens), determine if a given word w is in $LL*$ (or in other words, is $w \in L * \setminus \{\varepsilon\}$) (we don't consider the empty program to be valid).

Consider the language L of just ID tokens in MIPS:

$Q = \{q_0, q_1\}$

q_0 is our start state

$A = \{q_1\}$

δ is defined by:

- $\delta(q_0, a - z, A - Z) = q_1$
- $\delta(q_1, a - z, A - Z, 0 - 9) = q_1$
- $\delta(q_1, \varepsilon/\text{output token}) = q_0$

`w = abcde`

Initially in $q_0 \to q_1$.

Then there are two options. Either stay in q_1, or return the empty token.

In this case, we would get `[ID, "a"]`

Go back to q_0. We can do the same thing again after reaching q_1 with `b`.

Now, we also get `[ID, "b"]`

We can keep doing this

This time, we want to stay in q_1. We will stay in q_1 for `d` and `e`. We have consumed all the symbols.

We would then output `[ID, "cde"]`. We now have a longer token.

6 Lecture 6

Given a regular language L, determine if a word w is in $LL*$.

Two algorithms:

- Maximal munch
- Simplified maximal munch

Idea: Consume the largest possible token that makes sense. Produce the token and then proceed.

Difference:

- Maximal Munch: Consume characters until you no longer have a valid transition. If you have characters left to consume, backtrack to the last valid accepting state and resume.
- Simplified Maximal Munch: Consume characters until you no longer have a valid transition. If you are currently in an accepting state, produce the token and proceed. Otherwise go to an error state.

DFA for now

$\Sigma = \{a, b, c\}$

$L = \{a, b, abca\}$

q_0 is our start state

$A = \{q_1, q_4, q_5\}$

δ is defined by:

- $\delta(q_0, a) = q_1$
- $\delta(q_0, b) = q_5$
- $\delta(q_1, b) = q_2$
- $\delta(q_2, c) = q_3$
- $\delta(q_3, a) = q_4$

Note there is a ε/output token from the accepting states to the start state.

Maximal Munch: $\Sigma = \{a, b, c\}, L = \{a, b, abca\}, w = ababca$

- Algorithm consumes a and flags this state as its accepting state. Then, b tries to consume a but ends up in an error state.
- Algorithm then backtracks to the first a since that was the last accepting state. Token a is output.
- Algorithm then resumes consuming b and flags this state as accepting. Then, it tries to consume a but ends up in an error state.
- Algorithm then backtracks to the first b since that was the last accepting state. Token b is output.

- Algorithm then consumes the second a, the second b, the first c, the third a, and runs out of input. This last state is accepting, so it outputs the last token $abca$ and accepts.

Simplified Maximal Munch: $\Sigma = \{a, b, c\}, L = \{a, b, abca\}, w = ababca$

- Algorithm consumes a, then b tries to consume a but ends up in an error state. Note there is no keeping track of the first accepting state.
- Algorithm then checks to see if ab is accepting. It is not (as $ab \notin L$).
- Algorithm rejects $ababca$.
- Note: This gave the wrong answer, but this algorithm is usually good enough and is used in practice.

Simplified demo

a: $q_0 \to q_1$ (accepting)

$b : q_1 \to q_2$

$a : q_2 \to$ ERROR (give error)

Consider the following C++ line:

```
vector<pair<string, int>> v;
```

Notice that at the end, there is the token $>>$!. This, on its own is a valid token. With either algorithm we would reject this declaration. To do this declaration, you needed a space:

```
vector<pair<string, int> > v;
```

Question

What was the point of scanning?

Machine language is hard to write: We want to use assembly language.

We need to scan assembly lines in order to compile assembly language.

All the machine language operations we've seen so far, as assembly.

```
add $d
lis $d
.word i
jr $s
mult $s, $t
dif $s, $t
mfhi $d
mflo $d
lw $t, i($s)
sw $t, i($s)
```

The order of `$s`, `$t`, and `$d` are different in assembly than machine code.

Suppose that `$1` contains the address of an array and `$2` takes the number of elements in this array (assume small enough that we don't have to worry about overflow). Place the number 7 in the last possible spot in the array.

```
lis $8
.word 0x7 // store 7 in $8
lis $9
.word 4 // store 4 in $9
mult $2, $9 // num of elements * 4
mflo $3 // move above product to $3
add $3, $3, $1 // add this offset to address
sw $8 {-}4($3) // store the value at the end
jr $31}
```

Question

Write an assembly language MIPS program that takes a value in register `$1` and stores the value of its last base-10 digit in register `$2`

```
lis $10
.word 10
div $1 $10
mfhi $2 // hi = remainder, lo = quotient
jr $31
```

MIPS also comes equipped with control statements.

`beq $s, $t, i`

```
if ($s == $t) {
    PC += i * 4;
}
```

`beq $s, $t, i`

```
if ($s != $t) {
    PC += i * 4;
}
```

```
beq $0, $0, 1 //skip 1 instruction
add $1, $2, $3
jr $31
```

If $i == 0$, doesn't skip next instruction. If $i == -1$, infinite loop.

Question

Write an assembly language MIPS program that places the value 3 in register `$2` if the signed number in register `$1` is odd and places the value 11 in register `$2` if the number is even.

```
lis $8
.word 2 ; $8 == 2
lis $9
.word 3 ;$9 == 3
lis $2
.word 11 ; assume even
div $1 $8
mfhi $3
beq $3 $0 1
add $2, $9, $0
jr $31
```

Inequality Command

`slt $d, $s, $t`

Set Less Than. Sets the value of register `$d` to be 1 provided the value in register `$s` is less than the value in register `$t` and sets it to be 0 otherwise.

```
if ($s < $t) {
    $d = 1
} else {
    $d = 0
}
```

Note: There is also an unsigned version of this command.

Question

Write an assembly language MIPS program that negates the value in register `$1` provided its positive.

```
slt $2, $1, $0 ; $1 < $0 -> $2 = 1. $1 > $0 -> $2 = 0.
bne $2, $0, 1 ; if $2 != 0, then $2 is negative
sub $1, $0, $1
jr $31
```

Question

Write an assembly language MIPS program that places the absolute value of register `$1` in register `$2`

```
add $2, $1, $0 ; assume $1 is positive
slt $3, $0, $1 ; 0 < $1 (if $1 > 0 -> $3 = 1) (else $3 = 0)
bne $3, $0, 1
sub $2, $0, $2 ; ($2 = -$2)
jr $31
```

With branching we can even do looping.

Question

Write an assembly language MIPS program that adds together all even numbers from 1 to 20 inclusive. Store the answer in register `$3`.

```
lis $2
.word 20
lis $1
.word 2
add $3, $3, $0
add $3, $3, $2 ; $3 = $3 + $2
sub $2, $2, $1 ; $2 = $2 - 2
bne $2, $0, -3 ; (if not == 0, will go back to first add)
jr $31
```

Hard coding the -3 above isn't good for the long run. We fix this by using a label.

`label: operation commands`

Explicit Example

```
0x0: sub $3, $0, $0
0x4: sample:
0x4: add $1, $0, $0
```

We can thus do

```
lis $2
.word 20
lis $1
.word 2
add $3, $0, $0
top:
    add $3, $3, $2
    sub $2, $2, $1
    bne $2, $0, top
jr $31
```

Assembler computes the difference between the program counter and `top`. PC is the line number after the current line.

7 Lecture 7

What if a procedure wants to use registers that have data already.

We could preserve registers: save and restore them.

We have lots of memory in RAM we can use. We don't want our procedures to use the same RAM.

Used RAM goes after Free RAM.

Calling procedures pushes more registers onto the stack and returning pops them off.

We call `$30` our stack pointer.

Template for Procedures

f : procedure modifies `$1` and `$2`.

Entry: preserve `$1` and `$2`

Exit: restore `$1` and `$2`

```
f:
    sw $1, -4($30) ; Push registers modified
    sw $2, -8($30)
    lis $2 ; Decrement stack pointer
    .word 8
    sub $30, $30, $2
    ; Code
    add $30, $30, $2 ; Assuming $2 is still 8
    lw $2, -8($30)
    lw $1, -4($30)
```

There is a problem with returning:

```
main:
lis $8
.word f ; Recall f is an address
jr $8 ; Jump to the first line of f
```

We get a new command `jalr $s`.

Jump and Link Register. Sets `$31` to be the PC and then sets the PC to be `$s`. Accomplished by `temp = $s` then `$31 = PC` then `PC = temp`.

`jalr` will overwrite register `$31`. How do we return to the loader from main after using `jalr`? What if procedures call each other?

We need to save this register first.

Question

How do we pass arguments?

Typically, we'll just use registers. If we have too many, we could push parameters to the stack and then pop them.

Sum Evens 1 to N (assume $N > 1$)

```
; sumEvens1toN adds all even numbers from 1 to N
; Registers:
; $1 Temp Register (Should save)
; $2 Input Register (Should save)
; $3 Output Register (Do not save)

sumEvens1ToN:
    sw $1, -4($30) ; Save $1 and $2
    sw $2, -8($30)
    lis $1
    .word 8
    sub $30, $30, $1 ; Decrement stack pointer
    add $3, $0, $0 ; Initialize $3
    lis $1
    .word 2
    div $2, $1 ; is N even?
    mfhi $1
    sub $2, $2, $1 ; Sub 1 if not
    lis $1
    .word 2 ; Restore 2
    top:
    add $3, $3, $2
    sub $2, $2, $1
    bne $2, $0, top
    lis $1
    .word 8
    add $30, $30, $1 ; Restore stack pointer
    lw $2, -8($30)
    lw $1, -4($30) ; Reload $1 and $2
    jr $31 ; Back to caller
```

Question

How do we print to the screen or read input?

We do this one byte at a time.

Output: Use `sw` to store words in location `0xffff000c`. Least significant byte will be printed.

Input: Use `lw` to load words in location `0xffff0004`. Least significant byte will be the next character from `stdin`.

Print `cs` to the screen followed by a newline character.

```
lis $1
.word 0xffff000c
lis $2
.word 67 ; c
sw $2 0($1)
lis $2
.word 83 ; s
sw $2, 0($1)
lis $2
.word 10 ; \n
sw $2, 0($1)
jr $31
```

Let's finish up the assembler. Language translation involves two phases: Analysis and Synthesis.

Analysis: Understand what is meant by the input source. Use DFAs and maximal munch to break into tokens. But there's more; valid ranges, labels.

Synthesis: Output the equivalent target code in the new format

The Biggest Analysis Problem

How do we assemble this code:

```
beq $0, $1, myLabel
myLabel:
add $1, $1, $1
```

The problem is that `myLabel` is used before it's defined: we don't know the address when it's used.

The best fix to this is to perform two passes.

Pass 1: Group tokens into instructions and record addresses of labels. (Note: multiple labels are possible for the same line).

Pass 2: Translate each instruction into machine code. If it refers to a label, look up the associated address and compute the value.

8 Lecture 8

Recall the definition of DFA

We can extend the definition of $\delta : (Q \times \Sigma^*) \to Q$ to a function defined over $(Q \times \Sigma^*)$ via:

$$\begin{aligned} \delta^* : (Q \times \Sigma^*) &\to Q \\ (q, \varepsilon) &\to q \\ (q, aw) &\to \delta^*(\delta(q, a), w) \end{aligned}$$

where $a \in \Sigma$ and $w \in \Sigma^*$. If processing a string, process a letter first then process the rest of the string.

Definition 8.1

A DFA given by $M = (\Sigma, Q, q_0, A, \delta)$ accepts a string w if and only if $\delta^*(q_0, w) \in A$

What if we allowed more than one transition from a state with the same symbol?

To make the right choice, we would need an oracle that can predict the future.

This is called non-determinism. We then say that a machine accepts a word w if and only if there exists some path that leads to an accepting state.

We can simplify the "ends with bba" example from previous lecture to an NFA.

$L = \{w : w \text{ ends with bba}\}$

$Q = \{q_0, q_1, q_2, q_3\}$

q_0 is our start state

$A = \{q_3\}$

Transitions

- $\delta(q_0, a) \to q_0$
- $\delta(q_0, b) \to q_0$
- $\delta(q_0, b) \to q_1$
- $\delta(q_1, b) \to q_2$
- $\delta(q_2, a) \to q_3$

Definition 8.2

Definition Let M be an NFA. We say that M accepts w if and only if there exists some path through M that leads to an accepting states.

The language of an NFA M is the set of all strings accepted by M, that is: $L(M) = \{w : M \text{ accepts } w\}$

Definition 8.3

Definition An NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:
- Σ is a finite non-empty set
- Q is a finite non-empty set of states
- $q_0 \in Q$ is a start state
- $A \subseteq Q$ is a set of accepting states
- $\delta : (Q \times \Sigma) \to 2^Q$ is our total transition function. Note that 2^Q denotes the power set of Q, that is, the set of all subsets of Q. This allows us to go to multiple states at once

We can extend the definition of $\delta : (Q \times \Sigma) \to 2^Q$ via:

$$\begin{aligned} \delta^* : (2^Q \times \Sigma^*) &\to 2^Q \\ (S, \varepsilon) &\mapsto S \\ (S, aw) &\mapsto \delta^* \left(\bigcup_{q \in S} \delta(q, a), w \right) \end{aligned}$$

where $a \in \Sigma$. We also have:

Definition 8.4

Definition An NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ accepts a string w if and only if $\delta^*(\{q_0\}, w) \cap A \neq \emptyset$

Using the NFA defined earlier (bba) process `abbba`

$w =$ `abbba`

$S = \{q_0\}$

Process `a`

$S = \bigcup_{q \in \{q_0\}} \delta(q, a)$

$= \delta(q_0, a) = \{q_0\}$ (self-loop is the only option) $S = \{q_0\}$

Process `b`

$S = \bigcup_{q \in \{q_0\}} \delta(q, b)$

$= \delta(q_0, b) = \{q_0, q_1\}$ (two options) $S = \{q_0, q_1\}$

Process `b`

$S = \bigcup_{q \in \{q_0, q_1\}} \delta(q, b) = \delta(q_0, b) \cup \delta(q_1, b)$

$= \{q_0, q_1\} \cup \{q_2\}$

$= \{q_0, q_1, q_2\}$ $S = \{q_0, q_1, q_2\}$

Process `b`

$S = \bigcup_{q \in \{q_0, q_1, q_2\}} \delta(q_0, b) \cup \delta(q_1, b) \cup \delta(q_2, b)$

$= \{q_0, q_1\} \cup \{q_2\} \cup \emptyset$

$= \{q_0, q_1, q_2\}$ $S = \{q_0, q_1, q_2\}$

Process `a`

$S = \bigcup_{q \in \{q_0, q_1, q_2\}} \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$

$= \{q_0\} \cup \emptyset \cup \{q_3\}$

$= \{q_0, q_3\}$ $S \cap A = S \cap \{q_3\} = \{q_0, q_3\} = \{q_3\}$

Thus the input string is accepted by the NFA.

Question

Why are NFAs not more powerful than DFAs?

Even the power-set of a set of states is still finite. We can represent the set of states in the NFA as single states in the DFA.

Algorithm to convert from NFA to DFA.

- Start with the state $S = \{q_0\}$
- From this state, go to the NFA and determine what happens on each $a \in \Sigma$ for each $q \in S$. The set of resulting states should become its own state in your DFA.
- Repeat the previous step for each new state created until you have exhausted every possibility.
- Accepting states are any states that included an accepting state of the original NFA.

Previous NFA as a DFA.

$\Sigma = \{a, b\}$

$S_0 = \{q_0\}$

$A = \{q_3\}$

Transitions (for DFA)

- $\delta_D(S_0, a) = \{q_0\} = S_0$
- $\delta_D(S_0, b) = \{q_0, q_1\} = S_1$
- $\delta_D(S_1, a) = S_0$
- $\delta_D(S_1, b) = \delta(\{q_0\}, b) \cup \delta(\{q_1\}, b) = \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\} = S_2$
- $\delta_D(S_2, a) = \{q_0, q_3\} = S_3$
- $\delta_D(S_2, b) = S_2$
- $\delta_D(S_3, a) = S_0$
- $\delta_D(S_3, b) = S_1$

States in our DFA $S_0 = \{q_0\}$ $S_1 = \{q_0, q_1\}$ $S_2 = \{q_0, q_1, q_2\}$ $S_3 = \{q_0, q_3\}$

Example.

> Let $\Sigma = \{a, b, c\}$ Write an NFA such that $L = \{w : w \text{ does not contain ac}\}$

$\Sigma = \{a, b, c\}$

`not ending with a` $= q_0$ is our start state

$Q = \{q_0, q_1\}$

`ending with a` $= q_1$

$A = \{q_0, q_1\}$

Transitions:

- $\delta(q_0, b) \to q_0$
- $\delta(q_0, c) \to q_0$
- $\delta(q_0, a) \to q_1$
- $\delta(q_1, a) \to q_1$
- $\delta(q_1, b) \to q_0$

Example.

> Let $\Sigma = \{a, b, c\}$ Write an NFA such that $L = \{abc\} \cup \{w : w \text{ ends with cc}\}$

$\Sigma = \{a, b, c\}$

First element of union

$A = \{q_3\}$

$Q = \{q_0, q_1, q_2, q_3\}$

q_0 is our start state

Transitions

- $\delta(q_0, a) \to q_1$
- $\delta(q_1, b) \to q_2$
- $\delta(q_2, c) \to q_3$
- Second element of union

$A = \{q_2\}$

$Q = \{q_0, q_1, q_2\}$

q_0 is our start state

Transitions

- $\delta(q_0, a) \to q_0$
- $\delta(q_0, b) \to q_0$
- $\delta(q_0, c) \to q_0$
- $\delta(q_0, c) \to q_1$
- $\delta(q_1, c) \to q_2$

Question

How do we combine these?

$Q = \{q_0, , q_1, q_2, q_3, q_4, q_5, q_6\}$

$A = \{q_3, q_6\}$

Transitions

- $\delta(q_0, a) \to q_1$
- $\delta(q_1, b) \to q_2$
- $\delta(q_2, c) \to q_3$
- $\delta(q_0, a, b, c) \to q_4$
- $\delta(q_4, a, b, c) \to q_4$

- $\delta(q_4, c) \to q_5$
- $\delta(q_0, c) \to q_5$
- $\delta(q_5, c) \to q_6$

The DFA is complicated. Note: Combining two languages is non-obvious.

Summary: From Kleene's theorem, the set of languages accepted by a DFA are the regular languages. The set of languages accepted by DFAs are the same as those accepted by NFAs. Therefore, the set of languages accepted by an NFA are precisely the regular languages.

Question

What if we permitted state changes without reading a character.

These are known as ε transitions.

Definition 8.5

An ε-NFA is a 5-tuple$(\Sigma, Q, q_0, A, \delta)$

- Σ is a finite non-empty set that does not contain the symbol ε
- Q is a finite non-empty set of states
- $q_0 \in Q$ is a start state
- $A \subseteq Q$ is a set of accepting states
- $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \to 2^Q$ is our total transition function

These ε-transitions make it trivial to take the union of two NFAs.

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

$A = \{q_4, q_7\}$

Transitions:

- $\delta(q_0, \varepsilon) \to q_1$
- $\delta(q_0, \varepsilon) \to q_5$
- $\delta(q_1, a) \to q_2$
- $\delta(q_2, b) \to q_3$
- $\delta(q_3, c) \to q_4$
- $\delta(q_5, a, b, c) \to q_5$
- $\delta(q_5, c) \to q_6$
- $\delta(q_6, c) \to q_7$

Extending δ for an ε-NFA

Define $E(S)$ to be the epsilon closure of the set of states S, that is, the set of all states reachable from S in 0 or more ε transitions.

Note, this implies that $S \subset E(S)$

Again we can extend the definition of $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \to 2^Q$ to a function $\delta^* : (2^Q \times \Sigma^*) \to 2^Q$ via:

$$\begin{aligned} \delta^* : (2^Q \times \Sigma^*) &\to 2^Q \\ (S, \varepsilon) &\mapsto E(S) \\ (S, aw) &\mapsto \delta^* \left(\bigcup_{q \in S} E(\delta(q, a)), w \right) \end{aligned}$$

where $a \in \Sigma$. We also have

Definition 8.6

Definition An ε-NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ accepts a string w if and only if $\delta^*(E\{q_0\}, w) \cap A \neq \emptyset$

9 Lecture 9

Recall from last lecture:

Extending δ for an ε-NFA

S : a set of state $S \subseteq Q$

$a \in \Sigma, w \in \Sigma^*$

ϵ-NFA: $S : (Q \times (\Sigma \cup \{\varepsilon\})) \to 2^Q$

$\delta^*(2^Q \times \Sigma^*) \to 2^Q$

$\delta^*(S, \epsilon) = E(S)$

$\delta^*(S, aw) = \delta^*(\bigcup_{q \in S} E(\delta(q, a)), w)$

Simulating an ε-NFA

Let $E(S)$ be the epsilon closure of a set of states S. Recall $S \subset E(S)$

$w = a_1 a_2 \ldots a_n$

$S = E(\{q_0\})$

$S = \bigcup_{q \in S} \delta(q, a_i)$

if $S \cap A \neq \emptyset$ then Accept

Otherwise Reject

Example.

q_0 is the start state
$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$
$A = \{q_4, q_7\}$
Transitions:

- $\delta(q_0, \varepsilon) \to q_1$
- $\delta(q_0, \varepsilon) \to q_5$
- $\delta(q_1, a) \to q_2$
- $\delta(q_2, b) \to q_3$
- $\delta(q_3, c) \to q_4$
- $\delta(q_5, a, b, c) \to q_5$
- $\delta(q_5, c) \to q_6$
- $\delta(q_6, c) \to q_7$

Take the string $w = abcaccc$
$S = E(\{q_0\} = \{q_0, q_1, q_5\}$
$S = E(\delta(q_0, a) = \emptyset \cup \delta(q_1, a) = \{q_2\} \cup \delta(q_5, a) = \{q_5\})$
$S = \{q_2, q_5\}$

Example.

> q_0 is the start state
> $Q = \{q_0, q_1, q_2\}$
> $A = \{q_2\}$
> Transitions:
> - $\delta(q_0, a) \to q_1$
> - $\delta(q_1, a, b) \to q_1$
> - $\delta(q_1, \varepsilon) \to q_2$
> - $\delta(q_2, c) \to q_0$
>
> This machine represents the regular language $a(a|b)^*$
> Take the string $w = abca$
> $S = E(\{q_0\}) = \{q_0\}$ "a"
> $S = E(\delta(q_0, a)) = E(\{q_1\}) = \{q_1, q_2\}$
> Next symbol
> $S = \{q_1, q_2\}$ "b"
> $S = E(\delta(q_1, b) \cup \delta(q_2, b)) = E(\{q_1\} \cup \emptyset) = \{q_1, q_2\}$ since q_2 is available form q_1 via ε transitions.
> Next symbol
> $S = \{q_1, q_2\}$ "c"
> $S = E(\delta(q_1, c) \cup \delta(q_2, c)) = E(\emptyset \cup \{q_0\}) = \{q_0\}$
> Last symbol
> $S = \{q_0\}$ "a"
> $S = E(\delta(q_0, a)) = E(\{q_1\}) = \{q_1, q_2\}$
> Since $\{q_1, q_2\} \cap \{q_2\} \neq \emptyset$, accept.

DFA $\equiv$ NFA $\equiv$ ε-NFA $\equiv$ Regular $\equiv$ Regular Expression

If we can show that an ε-NFA exists for every regular expression, then we have proved one direction of Kleene's. We can do this by structural induction.

Regular Language

NFA that recognizes $\emptyset$

$A = \{\}$

q_0 is our start state

$Q = \{q_0\}$

NFA that recognizes $\{\varepsilon\}$

$A = \{q_0\}$

q_0 is our start state

$Q = \{q_0\}$

NFA that recognizes $\{a\}$

$A = \{q_1\}$

q_0 is our start state

$Q = \{q_0, q_1\}$

Transitions:

- $\delta(q_0, a) \to q_1$

NFA that recognizes union $L_1 \cup L_2$

Connect start state q_0 with first state of L_1 and L_2 via ε

NFA that recognizes concatenation $L_1 L_2$

Connect start state q_0 to start state of L_1 with ε transition. Connect the final state in L_1 with the first state in L_2 via ε transition.

NFA that recognizes L^*

From the accepting state of L, draw a ε transition back to start state and (accepting state) q_0. q_0 has a ε transition to the beginning of L.

We have completed Scanning. Now onto syntax.

Motivating Example:

Consider $\Sigma = \{(,)\}$ and $L = \{w : w \text{ is a balanced string of parentheses}\}$

Question

Is this language regular? Can we build a DFA for L?

No.

Consider the regular attempt:

$(???)|\varepsilon$

The ??? is the problem. What goes inside the parentheses is the entire language of matched parentheses. What if we could recurse in regular expressions?

$L = (L)|\varepsilon$ (Note: this just covers $(((\ldots)))$, not e.g., $()()(()))$

In terms of power, context-free languages are exactly regular languages plus recursion. In terms of expression, rather than extend regular expressions, we have a different form called grammars.

Definition 9.1

Definition Grammar is the language of languages.

Grammars help us describe what we ware allowed and not allowed to say. Context-free grammars are a set of rewrite rules that we can use to describe a language.

CFG for C++ has a lot of recursion.

Definition 9.2

Definition A Context Free Grammar (CFG) is a 4 tuple (N, Σ, P, S) where

- N is a finite non-empty set of non-terminal symbols
- Σ is an alphabet; a set of non-empty terminal symbols.
- P is a finite set of productions, each of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup \Sigma)^*$
- $S \in N$ is a starting symbol

Note: We set $V = N \cup \Sigma$ to denote the vocabulary, that is, the set of all symbols in our language.

Conventions:

Lower case letters from the start of the alphabet, i.e., a, b, c, ..., are elements of Σ.

Lower case letters from the end of the alphabet, i.e., w, x, y, z, are elements of Σ^* (words)

Upper-case letters, i.e., A, B, C, ..., are elements of N (non-terminals)

S is always our start symbol.

Greek letters, i.e., $\alpha, \beta, \gamma, \ldots$, are elements of V^* (recall this is $(N \cup \Sigma)^*$)

In most programming languages, the terminals of the context-free languages are the tokens, which are the words in the regular language.

This is why scanners categorize tokens (e.g. all infinity IDs are "ID"): so that the CFL's alphabet is finite.

It is possible to define CFGs directly over the input characters: this is called scannerless.

Let's revisit $\Sigma = \{(,)\}$ and $L = \{w : w \text{ is a balance string of parentheses}\}$

$S \to \varepsilon, S \to (S), S \to SS$

We can also write this using a shorthand: $S \to \varepsilon|(S)|SS$

Find a derivation of $(()())$. Recall our CFG above.

Definition 9.3

Over a CFG (N, Σ, P, S), we say that...

- A derives γ and we write $A \implies \gamma$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha A \beta \implies \alpha\gamma\beta$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha \implies \gamma$ if and only if a derivation exists, that is, there exists $\delta_i \in C^*$ for $0 \leq i \leq k$ such that $\alpha = \delta_0 \implies \delta_1 \implies \ldots \implies \delta_k = \gamma$. Note that k can be 0.

Solution:

$S \implies (S) \implies (SS) \implies ((S)S) \implies (()S) \implies (()(S)) \implies (()())$ Hence, $S \implies (()())$

Question

Why Context-Free?

Context-free languages actually add a sort of context to regular languages, so why are they "context free"?

They're free of a different sort of context. For instance, a context-free language can't catch this:

```
int a;
(*a) + 12;
```

Need the context that `a` is an int to know that this isn't allowed.

Definition 9.4

Define the language of a CFG (N, Σ, P, S) to be $L(G) = \{w \in \Sigma^* : S \implies w\}$

Definition 9.5

A language is context-free if and only if there exists a CFG G such that $L = L(G)$

Every regular language is context-free.

1. $\emptyset : (\{S\}, \{a\}, \emptyset, S)$
2. $\{\varepsilon\} : (\{S\}, \{a\}, S \to \varepsilon, S)$
3. $\{a\} : (\{S\}, \{a\}, S \to a, S)$
4. Union: $\{a\} \cup \{b\} : (\{S\}, \{a, b\}, S \to a|b, S)$
5. Concatenation: $\{ab\} : (\{S\}, \{a, b\}, S \to ab, S)$
6. Kleene Star: $\{a\}^* : (\{S\}, \{a\}, S \to Sa|\varepsilon, S)$

10 Lecture 10

Practice

Let $\Sigma = \{a, b\}$. Find a CFG for each of the following

- $\{a^n b^n : n \in \text{natural numbers}\}$

CFG: $S \to \varepsilon | aSb$

Derivation:

$S \implies aSb \implies aaSbb \implies aaaSbbb \implies aaabbb$

- Palindromes over $\{a, b, c\}$

$S \to \varepsilon|a|b|aSa|bSb|cSc$

- $a(a|b) * b$

$S \to aTb$

$T \to MT|\varepsilon$

$M \to a|b$

A fundamental example

Let's consider arithmetic operations over $\Sigma = \{a, b, c, +, -, *, /, (,)\}$ Find

- A CFG for L_1: arithmetic expressions from Σ without parentheses, and a derivation for $a - b$.
- A CFG for L_2: Well-formed arithmetic expressions from Σ with balanced parentheses, and a derivation for $((a) - b)$.

L_1: Expression without parentheses

$S \to a|b|c|SRS$

$R \to +| - | * |/$

$w = a - b$

$S \implies SRS \implies aRS \implies a - S \implies a - b$

L_2: Expression with parentheses

$S \to a|b|c|(SRS)$

$R \to +| - | * |/$

$w = ((a)) - b)$

$S \implies (SRS) \implies ((SRS)RS) \implies ((aRS)RS) \not\Longrightarrow ((a) - b)$ This does not work. Instead, we need

$S \to a|b|c|SRS|(S)$

$R \to +| - | * |/$

$S \implies (S) \implies (SRS) \implies ((S)RS) \implies ((a)RS) \implies ((a) - S) \implies ((a) - b)$ Notice, in these two derivations that we had a choice at each step which element of N to replace.

- $S \implies SRS \implies aRS \implies a - S \implies a - b$
- $S \implies (S) \implies (SRb) \implies (S - b) \implies (a - b)$

Leftmost derivation: In the first derivation, we chose to do a left derivation, that is, one that always expands from the left first.

Rightmost derivation: In the second derivation, we chose to do a right derivation, that is, one that always expands from the right first.

Parse Trees

$w = aaabbb$

$S \to \varepsilon|aSb$ $S \implies aSb \implies aaSbb \implies aaaSbbb \implies aaa\varepsilon bbb \implies aaabbb$

Question

Is it possible for multiple leftmost derivations (or multiple rightmost derivations) to describe the same string?

Consider two leftmost derivations for $w = a - b * c$.

$S \implies SRS \implies aRS \implies a - S \implies a - SRS \implies a - bRS \implies a - b * S \implies a - b * c$

$S \implies SRS \implies SRSRS \implies aRSRS \implies a - SRS \implies a - bRS \implies a - b * S \implies a - b * c$

Definition 10.1

A grammar for which some word has more than one distinct leftmost derivation/rightmost derivation/parse tree is called ambiguous.

Question

Why do we care about this?

As compiler writers, we care about where the derivation came from: parse trees give meaning to the string with respect to the grammar.

Post-order traversal of trees pseudocode.

```
t: TreeNode
int evaluate(t: Tree) {
    for each child in t {
        v_i = evaluate(child)
    }
    values of children
    return compute(values)
}
```

We use some sort of precedence heuristic to guide the derivation process.

Or, make the grammar unambiguous. This is what we did without first (incomplete) L_2 (by adding parentheses).

11 Lecture 11

There is a better way to eliminate ambiguity.

In a parse tree, we evaluate the expression in a depth-first, post-order traversal.

We can make a grammar left/right associative by crafting the recursion in the grammar.

Forcing Right Associative

$S \to LRS|L$

$L \to a|b|c$

$R \to +|-|*|/$

This forces a right-associative grammar (for $a - b * c$, $b * c$ will evaluate first)

Forcing Left Associative

$S \to SRL|L$

$L \to a|b|c$

$R \to +|-|*|/$

This forces a left-associative grammar (recurses to the left).

We can use this to create a grammar that follows BEDMAS (by making $*, /$ appear further down the tree).

$S \to SPT|T$

$T \to TRF|F$

$F \to a|b|c|(S)$

$P \to +|-$

$R \to *|/$

Question

If L is a context-free language, is there always an unambiguous grammar such that $L(G) = L$?

No. This was proven by Rohit Parikh in 1961.

Question

Can we write a computer program to recognize whether a grammar is ambiguous?

No.

Question

Given two CFGs G_1 and G_2, can we determine whether $L(G_1) = L(G_2)$. What about determining whether $L(G_1) \cap L(G_2) = \emptyset$?

This is still undecidable.

We use parsers to handle CFLs and CFGs.

Formally: Given a CFG $G = (N, \Sigma, P, S)$ and a terminal string $w \in \Sigma^*$, find the derivation, that is, the steps such that $S \implies \ldots \implies w$ or prove that $w \notin L(G)$

We can find this with two broad ideas (top-down and bottom-up).

Top down parsing: Start with S and then try to get to w.

Bottom-up Parsing: Start with w and work our way backwards to S.

Top-Down Parsing

Start with S and look for a derivation that gets us closer to w. Then, repeat with remaining non-terminals until we're done.

The main trick is "look" for derivation. Thus, the core problem is to predict which derivation is right.

We present the $LL(1)$ algorithm (in practice, real compliers do not use this).

Here is the pseudocode for a top-down parsing algorithm (before having the knowledge of a predictor table)

```
push S
for each 'a' in input do
        while top of stack is A in N do
                pop A
```

```
                ask an oracle to tell you which production A -> \gamma to use
                push the symbols in \gamma (rtl)
        end while
        // TOS is terminal
        if TOS is not 'a' then
                Reject
        else
                pop 'a'
        end if
end for
Accept
```

This has some problems.

1. The oracle is not real
2. When we reach the end of input, we have no way of realizing we weren't done with the stack
3. The oracle should be able to tell us that no production matches at all

12 Lecture 12

We augment the grammar to begin and end with $\vdash$ and $\dashv$ to solve problem 2.

$S' \to \vdash S \dashv$

$S \to \ LRS|L$

$S \to \ a|b|c$

$R \to \ +|-|*|/$

We treat these symbols as BOF and EOF characters.

Now let's solve problem 1 and 3.

This oracle could try all possible productions (way too expensive).

Solution: Use a single symbol lookahead to determine where to go. (This is a heuristic, doesn't always work).

We construct a predictor table to tell us where to go: given a non-terminal on the stack and a next input symbol, what rule should we use?

Always looking at a terminal input symbol, so figuring out how they match is non-trivial.

But this is hard. Consider the following

0. $S' \to \vdash S \dashv$
1. $S \to \ LRS$
2. $S \to \ L$
3. $L \to \ a|b|c$
4. $R \to \ +|-|*|/$

How could we decide between 1 and 2 based on the next symbol? They both start with L, which is the same non-terminal, so either can start with a, b, or c.

This is a limitation of the top-down parsing algorithm.

We will look at a simpler grammar.

0. $S' \to \vdash S \dashv$
1. $S \to \ LRS$

2. $S \rightarrow \ L$

3. $L \rightarrow \ a|b$

4. $R \rightarrow \ +|-$

Let's look at $w = \vdash a + b + c \dashv$

0. $S' \rightarrow \vdash S \dashv$

1. $S \rightarrow \ LM$

2. $L \rightarrow \ I$

3. $M \rightarrow \ OS$

4. $M \rightarrow \ \varepsilon$

5. $I \rightarrow \ a$

6. $I \rightarrow \ b$

7. $O \rightarrow \ +$

8. $O \rightarrow \ -$

Predictor Table of the simple grammar above.

	$\vdash$	$\dashv$	a	b	$+$	$-$
S'	0					
S			1	1		
L			2	2		
M	4				3	
I			5			
O					7	8

Table 1: Predictor Table

If in the non-terminal on the left, and encounter the terminal symbol on the top, pick the option of the number (see grammar above). Pop the elements that match the input string and repeat until we have finished (or encounter an error).

Definition 12.1

A grammar is called $LL(1)$ if and only if each cell of the predictor table contains at most one entry.

For an $LL(1)$ grammar, don't need sets in our predict table.

Question

Why is it called $LL(1)$?

First L: Scan left to right

Second L: Leftmost derivations

Number of symbol lookahead: 1

Not all grammars work with top-down parsing. Our simple grammar from above is not $LL(1)$ since more than one value is in a cell in the predictor table.

Constructing the Lookahead Table

Our goal is the following function, which is our predictor table.

Predict(A, a) : production rule(s) that apply when $A \in N$ is on the stack, $a \in \Sigma$ is the next input character.

To do this, we also introduce the following function, $\text{First}(\beta) \subseteq \Sigma$: $\text{First}(\beta)$: set of characters that can be the first symbol of a derivation starting from $\beta \in V^*$.

$\beta \implies \ldots \implies ar$. Thus, $a \in \text{First}(\beta)$

More formally:

$\text{Predict}(A, a) = \{A \to \beta : a \in \text{First}(\beta)\}$

$\text{First}(\beta) = \{a \in \Sigma : \beta \implies ay, \text{for some } y \in V^*\}$

Example of First

Recall the grammar from above with 8 steps.

First $(\vdash S \dashv) = \{\vdash\}$

First $(LM) = \text{First}(L) = \text{First}(I) = \{a, b\}$. So to compute $\text{First}(LM)$ we first need $\text{First}(I)$.

First $(I) = \{a, b\}$

First $(OS) =$ First $(O) = \{+, -\}$

First $(\varepsilon) = \{\}$

First $(a) = \{a\}$

First $(b) = \{b\}$

$\text{First}(+) = \{+\}$

First $(-) = \{-\}$

The problem with $\text{Predict}(A, a) = \{A \to \beta : \beta \implies \ldots \implies ay$, for some $\beta, y \in V^*\}$ is that it is possible that $A \implies \ldots \implies \varepsilon$. This would mean that the a didn't come from A but rather some symbol after A.

Example $S' \implies \ldots \implies \vdash abc \dashv$

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

1. $S \to AcB$

2. $A \to ab$

3. $A \to ff$

4. $B \to def$

5. $B \to ef$

6. $B \to \varepsilon$

Notice that $S' \implies \vdash S \dashv \implies \vdash AcB \dashv \implies \vdash abcB \dashv \implies \vdash abc \dashv$

In the top-down parsing algorithm, we reach $\vdash abcB \dashv$ the stack is $\dashv B$ and remaining input is $\dashv$.

We look at $\text{Predict}(B, \dashv)$ which is empty since $\dashv \notin \text{First}(B)$. Thus we reach an error.

We augment our predict table to include the elements that can follow a non-terminal symbol, if it can reduce to ε.

In this case, we need to include that $\dashv \in \text{Predict}(B, \dashv)$

To correct this, we introduce two new functions.

$\text{Nullable}(\beta)$: boolean function; for $\beta \in V^*$ is true if and only if $\beta \implies \ldots \implies \varepsilon$

$\text{Follow}(A)$: for any $A \in N$, this is the set of elements of Σ that can come immediately after A in a derivation starting from S'

Definition 12.2

We say that a $\beta \in V^*$ is nullable if and only if Nullable(β) = true

Example of Follow

0. $S' \to \vdash S \dashv$
1. $S \to AcB$
2. $A \to ab$
3. $A \to ff$
4. $B \to def$
5. $B \to ef$
6. $B \to \varepsilon$

Follow(S') = {} (Always)

Follow(S) = $\{\dashv\}$

Follow(A) = $\{c\}$

Follow(B) = $\{\dashv\}$

Question

What happens with Predict(A, a) if Nullable(A) = false?

Follow(A) is still some set of terminals but it won't be relevant since we would need to consider what happens to First(A) first.

Thus, the Follow function only matters if Nullable is true.

This motivates the following correct definition of our predictor table:

Definition 12.3

Predict(A, a) = $\{A \to \beta : a \in \text{First}(\beta)\} \cup \{A \to \beta : \text{Nullable}(\beta) \text{ and } a \in \text{Follow(A)}\}$

This is the full, correct definition. Notice that this still requires that the table only have one member of the set per entry to be useful as a deterministic program.

Note that Nullable(β) = false whenever β contains a terminal symbol.

Further, Nullable(AB) = Nullable(A) $\wedge$ Nullable(B)

Thus, it suffices to compute Nullable(A) for all $A \in N$.

13 Lecture 13

Algorithm of Nullable(A)

```
Initialize Nullable(A) = false for all A \in N'
repeat
for each production in P do:
        if (P is A -> \varepsilon) or
        (P is A -> B_1 ... B_k
        and \wedge_{i=1}^{k} Nullable(B_i) = true) then
           Nullable(A) = true
```

```
8         end if
9 end for
10 until nothing changes
```

Example of Nullable

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

1. $S \to c$

2. $S \to QRS$

3. $Q \to R$

4. $Q \to d$

5. $R \to \varepsilon$

6. $R \to b$

Nullability Table

Iter	0	1	2	3
S'	F	F	F	F
S	F	F	F	F
Q	F	F	T	T
R	F	T	T	T

Table 2: Nullability Table

Thus, Nullable(S') = Nullable(S) = F and Nullable(Q) = Nullable(R) = T

Notes about First

Main idea: Keep processing $B_1, B_2, \ldots, B_k$ from a production rule until you encounter a terminal or a symbol that is not nullable. Then go to the next rule. Repeat until no changes are made during the processing.

Remember, ε, isn't a real symbol, and can't be in a First set.

For First, we will ignore trivial productions of the form $A \to \varepsilon$ based on the above observation.

Further, First(S') = $\{\vdash\}$ always.

We first compute First(A) for all $A \in N$ and then we compute First(β) for all relevant $\beta \in V^*$

Computing First

```
1 Initialize First(A) = {} for all A \in N'
2 repeat
3 for each rule A -> B_1B_2 ... B_k in P do
4         for i in {1,..., k} do
5                 if B_i in T' then
6                         First(A) = First(A) \cup {B_i}; break
7                 else
8                         First(A) = First(A) \cup First(B_i)
9                         if Nullable(B_i) == False then break
10                end if
11        end for
12 end for
13 until nothing changes
```

Example of First

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

1. $S \to c$

2. $S \to QRS$

3. $Q \to R$

4. $Q \to d$

5. $R \to \varepsilon$

6. $R \to b$

First Table

Iter	0	1	2	3
S'	$\{\}$	$\{\vdash\}$	$\{\vdash\}$	$\{\vdash\}$
S	$\{\}$	$\{c\}$	$\{b, c, d\}$	$\{b, c, d\}$
Q	$\{\}$	$\{d\}$	$\{b, d\}$	$\{b, d\}$
R	$\{\}$	$\{b\}$	$\{b\}$	$\{b\}$

Table 3: First Table

Recall, Nullable(S') = Nullable(S) = F and Nullable(Q) = Nullable(R) = T

Thus, First(S') = $\{\vdash\}$, First(S) = $\{b, c, d\}$, First(Q) = $\{b, d\}$, First(R) = $\{b\}$

Computing First (2)

```
result = {}
for i \in {1,...,n} do
        if B_i \in T' then
                result = result \cup {B_i}; break
        else
                result = result \cup First(B_i)
                if Nullable(B_i) == False then break
        end if
end for
```

Computing Follow

```
Initialize Follow(A) = {} for all A \in N
repeat
    for each production A -> B_1 B_2 ... B_k in P' do
        for i \in {i, ..., k} do
            if B_i \in N then
                Follow(B_i) = Follow(B_i) \cup First(B_{i+1} ... B_k)
                if \wedge_{m=i+1}^{k} Nullable(B_m) == True or i == k then
                    Follow(B_i) = Follow(B_i) \cup Follow(a)
                end if
            end if
        end for
    end for
until nothing changes
```

Example of Follow

0. $S' \to \vdash S \dashv$
1. $S \to c$
2. $S \to QRS$
3. $Q \to R$
4. $Q \to d$
5. $R \to \varepsilon$
6. $R \to b$

Follow Table

$Iter$	0	1	2
S	$\{\}$	$\{\vdash\}$	$\{\vdash\}$
Q	$\{\}$	$\{b, c, d\}$	$\{b, c, d\}$
R	$\{\}$	$\{b, c, d\}$	$\{b, c, d\}$

Table 4: Follow Table

14 Lecture 14

Cheat sheet

Nullable:

- $A \to \varepsilon$ implies that Nullable(A) = true. Further Nullable(ε) = $true$
- If $A \to B_1 \dots B_n$ and each of Nullable(B_i) = $true$ then Nullable(A) = $true$

First:

- $A \to \alpha\alpha$ then $a \in$ First(A)
- $A \to B_1 \dots B_n$ then First(A) = First(A) $\cup$ First(B_i) for each $i \in \{1, \dots, n\}$ until Nullable(B_i) is false
- $A \to \alpha B \beta$ then Follow(B) = First(β)
- $A \to B\beta$ and Nullable(β) = $true$, then Follow(B) = Follow(B) $\cup$ Follow(A)

Primary Issue

With this grammar:

$S \to S + T$ $S \to T$ $T \to T * F$

$T \to F$

$F \to a|b|c|(S)$

The primary issue is that left recursion is at odds with $LL(1)$. In fact, left recursive grammars are always not $LL(1)$. Examine the derivations for a and $a + b$.

$S \implies S + T \implies T + T \implies F + T \implies a + T \implies a + bS \implies T \implies F \implies a$

Notice that they have the same first character but required different starting rules from S. That is $\{1, 2\} \subseteq$ Predict(S, a). Our first step is to at least make this right recursive.

To make a left recursive grammar right recursive; say

$A \to A\alpha|\beta$

where β does not begin with the non-terminal A, we remove this rule from our grammar and replace it with:

$A \to \beta A'$

$A' \to \alpha A' | \varepsilon$

The above solves our issue

$S \to TZ'$

$Z' \to +TZ' | \varepsilon$

$T \to FT'$

$T' \to *FT' | \varepsilon$

$F \to a|b|c|(S)$

we get a right-recursive grammar. This is $LL(1)$.

However: recall that we didn't want these grammars, because they were right associative. This is an issue we need to resolve with new techniques.

Not all right right recursive grammars are $LL(1)$. Consider

$S \to T + S$

$S \to T$

$T \to F * T$

$T \to F$

$F \to a|b|c|(S)$

we get a right-recursive grammar, but not $LL(1)$

$S \implies T + S \implies F + S \implies a + S \implies a + T \implies a + bS \implies T \implies F \implies a$

Again, we have $\{1, 2\} \subseteq \text{Predict}(S, a)$. There is still hope, we can apply a process known as factoring.

Left factoring

Idea: If $A \to \alpha\beta_1 | \dots \alpha\beta_n | y$ where $\alpha \neq \varepsilon$ and y is representative of other productions that do not begin with α, then we can change this to the following equivalent grammar by left factoring:

$A \to \alpha B | y$

$B \to \beta_1 | \dots | \beta_n$

Recursive-Descent Parsing

Fixing the parse trees from right-recursive and left-factored grammars is the #1 thing that recursive-descent ad hoc solutions fix

The actual sequence of steps is $LL(1)$, but then they generate a different parse tree by changing it on the fly.

Bottom-up Parsing

Recall: Determining the α_i in $S \implies \alpha_1 \implies \dots \implies w$

Idea: Instead of going from S to w, let's try to go from w to S. Overall idea: look for the RHS of a production, replace it with the LHS. When you don't have enough for a RHS, read more input. Keep grouping until you reach the start state.

Our stack this time will store the α_i in reverse order (Contrast to top-down which stores the α_i in order)

Our invariant here will be Stack + Unread Input = α_i (Contrast to top-down where invariant was consumed input + reversed Stack contents = α_i)

```
1 for each symbol $a$ in the input from left to right do
2         // ask an oracle whether to shift, reduce, or reject,
3         // and with which production to reduce (if we reduce)
4         while the oracle tells us to reduce with some B -> \gamma do
```

```
5                 stack.pop symbols in \gamma
6                 stack.push $B$
7         end while
8         if the oracle told us to reject then
9                 reject
10        end if
11        stack.push a
12 end for
13 accept
```

Example.

> Recall our grammar:
> $S' \to \vdash S \dashv \quad (0)$
> $S \to AcB \quad (1)$
> $A \to ab \quad (2)$
> $A \to ff \quad (3)$
> $B \to def \quad (4)$
> $B \to ef \quad (5)$

We wish to process $w = \vdash abcdef \dashv$

Stack	Read	Processing	Action
	ε	$\vdash abcdef \dashv$	Shift $\vdash$
$\vdash$	$\vdash$	$abcdef \dashv$	Shift a
$\vdash a$	$\vdash a$	$bcdef \dashv$	Shift b
$\vdash ab$	$\vdash ab$	$cdef \dashv$	Reduce (2); pop b, a, push A
$\vdash A$	$\vdash ab$	$cdef \dashv$	Shift c
$\vdash Ac$	$\vdash abc$	$def \dashv$	Shift d
$\vdash Acd$	$\vdash abcd$	$ef \dashv$	Shift e
$\vdash Acde$	$\vdash abcde$	$f \dashv$	Shift f
$\vdash Acdef$	$\vdash abcdef$	$\dashv$	Reduce (4); pop f, d, e push B
$\vdash AcB$	$\vdash abcdef$	$\dashv$	Reduce (1); pop B, c, A push S
$\vdash S$	$\vdash abcdef$	$\dashv$	Shift $\dashv$
$\vdash S \dashv$	$\vdash abcdef \dashv$	ε	Reduce (0); pop $\dashv S, \vdash$ push S'
S'	$\vdash abcdef \dashv$	ε	Accept

Theorem 14.1

For any grammar G, the set of viable prefixes (stack configurations), namely $\{\alpha a : \alpha \in V^*$ is a stack $a \in \Sigma$ is the next character $\exists x \in \Sigma^*$ such that $S \implies \ldots \implies \alpha a x\}$ is a regular language, and the NFA accepting it corresponds to items of G. Converting this NFA to a DFA gives a machine with states that are set of valid items for a viable prefix.

We will show how to use this theorem to create a $LR(0)$, $SLR(1)$, and $LR(1)$ automata to help us accept the words generated by a grammar.

Consider the following context-free grammar:

$S' \to \vdash S \dashv \quad (0)$

$S \to S + T \quad (1)$

$S. \to T \quad (2)$

$T. \to d \quad (3)$

Definition 14.2

An item is a production with a dot $\cdot$ somewhere on the right hand side of a rule

Items indicate a partially completed rule. We will begin in a state labelled by the rule $S' \to \cdot \vdash S \dashv$

That dot is called the bookmark

$LR(0)$ Construction

From a state, for each rule in the state, move the dot forward by one character. The transition function is given by the symbol you jumped over.

For example, with $S' \to \cdot \vdash S \dashv$, we move the $\cdot$ over $\vdash$. Thus, the transition function will consume the symbol $\vdash$.

The state we end up in will contain the item $S' \to \vdash \cdot S \dashv$

In the new state, if the set of items we have $\cdot A$ for some non-terminal A, we then add all rules with A in the left-hand side of a production with a dot preceding the right-hand side.

In this case, this state will include the rules $S \to \cdot S + T$ and $S \to \cdot T$.

Notice now we also have $\cdot T$ and so we also need to include the rules where T is the left-hand side, adding the rule $T \to \cdot d$.

If we find ourselves at a familiar state, reuse it instead of remaking it.

We continue with these steps until there are no bookmarks left to move. Then we have the final DFA.

We skipped the $\varepsilon-$NFA step by putting all these items in the same rule. You may see versions of this algorithm that involve building an $\varepsilon-$NFA and then converting, but the result will be the same.

Back to the example

$S' \to \vdash S \dashv \quad (0)$

$S \to S + T \quad (1)$

$S. \to T \quad (2)$

$T. \to d \quad (3)$

This automaton is our oracle. Run the stack through the automaton, and:

- If you end up in a state with the bookmark at the right-hand side of an item, perform that reduction
- If you end up in a state with the bookmark elsewhere, shift
- Else (error state), reject

15 Lecture 15

$w = \vdash d + d + d \dashv$

Stack	Read	Processing	Action
	ε	$\vdash d + d + d \dashv$	Shift state 0
$\vdash$	$\vdash$	$d + d + d \dashv$	State 1 Shift
$\vdash d$	$\vdash d$	$+d + d \dashv$	State 5 reduce(3), pop d, push T
$\vdash T$	$\vdash d$	$+d + d \dashv$	State 4 reduce(2), pop T, push S
$\vdash S$	$\vdash d$	$+d + d \dashv$	State 2, shift
$\vdash S \dashv$	$\vdash d+$	$d + d \dashv$	State 6, shift
$\vdash S + d$	$\vdash d + d$	$d \dashv$	State 5, reduce (3), pop d, push T
$\vdash S + T$	$\vdash d + d$	$+d \dashv$	State 7, reduce(1) pop S and T, push S
$\vdash S$			

The stack is a stack, so the bottom of the stack (beginning of our input) doesn't usually change. We're rerunning the whole DFA even when the prefix of our stack is the same. Because of this, our algorithm is $O(n^2)$.

Remember how we moved through the DFA in a state stack, and push and pop to the state stack at the same time as the symbol stack. That way, we don't repeat getting to a state with a prefix that hasn't changed.

This brings us to $O(n)$.

Stack column above becomes Symbol Stack

$LR(0)$

```
stateStack.push q_0
for each symbol a in EOF x BOF from left to right do
        while Reduce[stateStack.top] is some production B -> \gamma do
                symStack.pop symbols in \gamma
                stateStack.pop |\gamma| states
                symStack.push B
                stateStack.push \delta[stateStack.top, B]
        end while
        symStack.push a
        reject if \delta[stateStack.top, a] is undefined
        stateStack.push \delta[stateStack.top, a]
end for
accept
```

State Stack	Symbol Stack	Read	Processing	Action
q_0		ε	$\vdash d+d+d\dashv$	Shift state 0
q_0, q_1	$\vdash$	$\vdash$	$d+d+d\dashv$	State 1 Shift
q_0, q_1, q_5	$\vdash d$	$\vdash d$	$+d+d\dashv$	State 5 reduce(3), pop d, push T
q_0, q_1, q_4	$\vdash T$	$\vdash d$	$+d+d\dashv$	State 4 reduce(2), pop T, push S
	$\vdash S$	$\vdash d$	$+d+d\dashv$	State 2, shift
	$\vdash S \dashv$	$\vdash d+$	$d+d\dashv$	State 6, shift
	$\vdash S+d$	$\vdash d+d$	$d\dashv$	State 5, reduce (3), pop d, push T
	$\vdash S+T$	$\vdash d+d$	$+d\dashv$	State 7, reduce(1) pop S and T, push S
	$\vdash S$	$\vdash abcdef$		

Possible Issues

Issue one (Shift-Reduce): What if a state has two items of the form:

- $A \to \alpha \cdot a\beta$
- $B \to \gamma\cdot$

Question

Should we shift or reduce?

Note, having two items that shift, e.g.:

- $A \to \alpha \cdot a\beta$
- $B \to \gamma \cdot b\delta$

is not an issue.

Definition 15.1

A grammar is $LR(0)$ if and only if after creating the automaton, no state has a shift-reduce or reduce-reduce conflict.

Recall that $LL(1)$ grammars were at odds with recursive languages.

Question

Are $LR(0)$ grammars in conflict with a type of recursive language?

Not usually. Bottom-up parsing can support left and right recursive grammars. However not all grammars are $LR(0)$ grammars.

Consider the grammar

$S' \to \vdash S \dashv$

$S \to T + S$

$S. \to T$

$T. \to d$

New $LR(0)$ Automaton

Conflict

State 4 has a shift-reduce conflict.

Suppose the input began with $\vdash d$. This gives a stack of $\vdash d$ and then we reduce in state 5, so our stack changes to $\vdash T$ and we move to state 4 via state 1.

Should we reduce $S \to T$? It depends.

We add a lookahead to the automation to fix the conflict. For every $A \to \alpha^*$, attack Follow(A). Recall:

$S' \to \vdash S \dashv$

$S \to T + S$

$S. \to T$

$T. \to d$

Note that Follow(S)= $\{\dashv\}$ and Follow(T) $= \{+, \dashv\}$. So state 4 becomes

$S \to T \cdot +S$ and $S \to T \cdot : \{\dashv\}$

Apply $S \to T \cdot +S$ if the next token is $+$, and apply $S \to T \cdot \{\dashv\}$ if the next token is $\dashv$.

With lookahead from Follow sets on reduce states, we call these parses $SLR(1)$ parsers.

$SLR(1)$ is not a simplified version of $LR(1)$, it's just different.

Building the Parse Tree

With top-down parsing, when we pop S from the stack and push B, y and $A : S$ is a node, make the new symbols the children.

With bottom-up parsing, when you reduce $A \to ab$ (from a stack with a and b). You then keep these two old symbols as children of the new node A.

A Last Parser Problem

Most famous problem in parsing: the dangling else.

```
if(a) if (b) S1: else S2
```

```
if (a) {
    if (b) {
        S1;
    }
} else {
    S2;
}
```

```
if(a) {
    if (b) {
        S1;
    } else {
        S2;
    }
}
```

16 Lecture 16

Where are we now?

We have finished syntactic analysis, are now on to type checking (semantic analysis).

Semantics: Does what is written make sense?

Not everything can be enforced by a CFG. Examples

- Type checking
- Declaration before use
- Scoping
- Well-typed expressions

To solve these, we can move to context-sensitive languages.

As it turns out, CSL's aren't a very useful formalism.

We already needed to give up many CFGs to make a parser handle CFLs; with CSLs, it would be even worse.

As such, we treat context-sensitive analysis as analysis (looking over the parse tree generated by CFL parsing) instead of parsing (making its own data structure).

- pre-order tree traversal
- post-order tree traversal
- hybrid of the two

We will traverse our parse tree to do our analysis.

We still need to check for:

- Variables declared more than once
- Variables used but not declared
- Type errors
- Scoping as it applies to the above

Declaration errors

Question

How do we determine multiple/missing declaration errors?

We've done this before. We construct a symbol table.

- Traverse the parse tree for any rules of the form dcl → type ID.
- Add the ID to the symbol table
- If the name is already in the table, give an error.

```
int foo(int a) {
        int x = 0;
        return x + a;
}

int wain(int x, int y) {
        return foo(y) + x;
}
```

Checking

To verify that variables have been declared. Check for rules of the form factor → ID, and lvalue → ID. If ID is not in the symbol table, produce an error. The previous two passes can be merged.

def: dcl → type ID (make sure not already in symbol table, and add to symbol table)

Use of variables:

factor → ID (check to see if it is in symbol table (return error if not))

lvalue → ID (check to see if it is in symbol table (return error if not))

Question

With labels in MIPS in the assembler, we needed two passes. Why do we only need one in the compiler?

We need to declare variables before using them. This is not true for labels.

Note that in the symbol table, we should also keep track of the type of variables. Why is this important? Just by looking at the bits, we cannot figure out what it represents. Types allow us to interpret the contents.

Good systems prevent us from interpreting bits as something we should not.

For example

```
int *a = NULL;
a = 7;
```

should be a type mismatch.

This is just a matter of interpretation.

In WLP4, there are two types: `int` and `int*` for integers and pointers to integers.

For type checking, we need to evaluate the types of expressions and then ensure that the operations we use between types corresponds correctly.

Question

If given a variable (in the wild), how do we determine it's type?

Use its declaration. Need to add this to the symbol table.

We can use a global variable to keep track of the symbol table:

```
enum class Type {INT, POINTER};
unordered_map<string, Type> symbolTabel; // name->type
```

Some things can go wrong. This doesn't take scoping into account. Also need something for functions/declarations.

Consider the following code. Is there an error?

```
int foo(int a) {
        int x = 0;
        return x + a;
}
int wain(int x, int y) {
        return foo(y) + x;
}
```

No. Duplicate variables in different procedures are okay.

Is the following an error?

```
int foo(int a) {
        int x = 0;
        return x + a;
}
int wain(int a, int b) {
        return foo(b) + x;
}

```

Yes. The variable `x` is not in scope in `wain`.

Is the following an error?

```
int foo(int a) {
        int x = 0;
        return x + a;
}
int foo(int b) {return b;}
int wain (int a, int b) {
        return foo(b) + a;
}
```

Yes. We have multiple declarations of `foo`.

We resolve this with a separate symbol table per procedure. We also need a global symbol table.

Question

A symbol table is a map, what are we mapping each symbol to?

For type checking, we need to know the type of each symbol. In WLP4, a string could be sufficient, but you should use a data structure so you can add more later.

Question

What about procedures?

Procedures don't have types, they have signatures.

In WLP4, all procedures return `int`, so we really just need the argument types: an array of types.

```
int foo(int x, int y)
// (int, int) -> int
int bar(int *x, int y, int c)
// (int*, int, int) -> int
```

Computing the signature.

Simply need to traverse nodes in the parse tree of these forms.

- params $\rightarrow$
- params $\rightarrow$ paramlist
- paramlist $\rightarrow$ dcl
- paramlist $\rightarrow$ dcl COMMA paramlist

This can be done in a single pass.

Consider

```
int foo(int a) {
        int x = 0;
        return x + a;
}
int wain(int *a, int b) {
        return foo(b) + a;
}
```

Global symbol table:

- `foo: [int], wain: [int*, int]`

Local symbol tables:

- `foo: a: int, x: int`
- `wain: a: int*, b: int`

Type errors

Question

What are type errors and how to find them?

Two separate issues:

- What are type errors? (Definition)
- How to find them? (Implementation)

Definition of Type (Errors)

Need a set of rules to tell us

- The type of every expression
- Whether an expression makes sense with the types of its subexpressions
- Whether a statement makes sense with the types of its subexpressions

Detection of Type (Errors)

There's really only one algorithm with a tree: traverse the tree. Implement a (mostly) post-order traversal that applies defined rules based on which expressions it encounters.

Inference rules are Post rules (like in CS245)

If an ID is declared with type τ then it has this type:

$$\frac{(\text{id.name}, \tau) \in \text{declarations}}{\text{id.name} : \tau}$$

Numbers have type `int`: $\overline{\text{NUM: int}}$

NULL is of type `int*`: $\overline{\text{NULL: int*}}$

Inference rules for types

Inference rules are the true case. If no inference rule matches, that means the expression or statement doesn't type check: type error.

Look for good, not for bad: errors should always be the "else" case.

Parentheses do not change the type $\frac{E:\tau}{(E):\tau}$

The Address of an `int` is of type `int*` $\frac{E:\text{int}}{\&E:\text{int*}}$

```
int x = 0;
int *p = NULL;
p &x; // int*
```

Dereferencing `int*` is of type `int` $\frac{E:\text{int*}}{*E:\text{int}}$

```
x = *p // int
```

If E has type `int` then `new int[E]` is of type `int*` $\frac{E:int}{\text{new int[E]: int*}}$

```
p = new int[*p] // int*
```

Arithmetic Operations

Multiplication $\frac{E_1:\text{int} \quad E_2:\text{int}}{E_1 * E_2:\text{int}}$

Division $\frac{E_1:\text{int} \quad E_2:\text{int}}{E_1 / E_2:\text{int}}$

Module $\frac{E_1:\text{int} \quad E_2:\text{int}}{E_1 \% E_2:\text{int}}$

Addition

$$\frac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 + E_2 : \text{int}}$$

$$\frac{E_1 : \text{int*} \quad E_2 : \text{int}}{E_1 + E_2 : \text{int*}}$$

$$\frac{E_1 : \text{int} \quad E_2 : \text{int*}}{E_1 + E_2 : \text{int*}}$$

Subtraction

$$\frac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 - E_2 : \text{int}}$$

$$\frac{E_1 : \text{int*} \quad E_2 : \text{int}}{E_1 - E_2 : \text{int*}}$$

$$\frac{E_1 : \text{int*} \quad E_2 : \text{int*}}{E_1 - E_2 : \text{int}}$$

Procedure Calls:

$$\frac{(f, \tau_1, \ldots, \tau_n) \in \text{declarations} \quad E_1 : \tau_1 \;\; E_2 : \tau_2 \;\; \ldots \;\; E_n : \tau_n}{f(E_1, \ldots, E_n) : \text{int}}$$

The basic kind of statement type is an expression statement. An expression statement is okay as long as the expression has a type. We will need rules for all the other statements too. Statement's don't have a type, but can be "well typed".

17 Lecture 17

Control statements:

```
while (T) {S}
if (T) {S_1} else {S_2}
```

The value of `T` above should be a boolean. But WLP4 doesn't have booleans.

Our grammar forces it to be a boolean expression, so we don't need to check that.

But, we still need to check its subexpressions.

Inference rules for well-typed

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 < E_2)}$

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 > E_2)}$

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 == E_2)}$

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 <= E_2)}$

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 >= E_2)}$

$\frac{E_1:\tau \quad E_2:\tau}{\text{well-typed}(E_1 = E_2)}$

An if statement is well-typed if and only if all of its components are well typed

$$\frac{\text{well-typed}(T) \quad \text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(if(T)else\{S_2\})}$$

A while statement is well-typed if and only if all of its components are well-typed

$$\frac{\text{well-typed}(T) \quad \text{well-typed}(S)}{\text{well-typed}(while(T)\{S\})}$$

There is a final sanity check with the left- and right-hand sides of an assignment statement.

Given an expression, say $x = y$, notice the left-hand side and the right-hand side represent different things.

The left-hand side represents a place to store data; it must be a location of memory. The right-hand side must be a value; that is, any well-typed expression.

Anything that denotes a storage location is an lvalue.

Consider the following two snippets of code

```
int x = 0;
x = 5;
```

This is okay; the lvalue `x` is a storage location.

```
int x = 0;
5 = x;
```

This is not okay; the lvalue 5 is an integer and not a storage location.

For us, lvalues are any of variable names, dereferenced pointers and any parenthetical combinations of these. These are all forced on us by the WLP4 grammar so the checking is done for you.

The empty sequence is well-typed $\overline{\text{well-typed}()}$

Consecutive statements are well-typed if and only if each statement is well-typed

$$\frac{\text{well-typed}(S_1) \quad \text{well-typed}(S_2)}{\text{well-typed}(S_1;S_2)}$$

Procedures are well-typed if and only if the body is well-typed and the procedure returns an `int`.

$$\frac{\text{well-typed}(S) \quad E{:}int}{\text{well-typed}(intf(dcl_1,\ldots,dcl_n)\{dclsSreturnE;\})}$$

Type-checking recommendations

- Brush up on recursion. Everything from this point on is traversing a tree.

Example.

> Type-check a tree. We will use the code from last time.

```
int foo(int a) {
        int x = 0;
        return x + a;
}
int wain(int *a, int b) {
        return foo(b) + a;
}
```

```
void check(ParseTreeNode *)
post-order
```

Infer types for sub expressions using the rules.

Find three distinct one-character changes that make this code fail to type-check (while still passing parsing):

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

Change `int a` to `int* a`.

Change `return a + b;` to `return *a + b`.

Change `return a + b` to `return &a + b`.

There are infinitely many equivalent MIPS programs for a single WLP4 program.

Question

Which should we output?

Correctness is most important. We seek a simplistic solution. Efficiency to compile and efficiency of the code itself.

Real compilers have an intermediate representation (IR) that's close to assembly code but (at least) has infinite registers. This IR is good for optimization. We don't do optimization, we we won't use IR.

Our step of generating assembly will be (more-or-less) the "generate IR" step of a larger compiler. MIPS is our IR.

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

Recall our `mips.twoints` convention that registers $1 and $2 store the input values, and our usual convention to return in register $3.

```
add $3, $1, $0
jr $31
```

Question

How did we know where `a` was stored? What if we had >2 arguments? What if we had done something complex with intermediary results?

The parse tree will be the same if we'd done `return b` instead of `return a`.

The parse tree isn't going to be enough to determine the difference between the two pieces of code.

Question

How can we resolve this? How can we distinguish between these two codes?

We use the symbol table.

Symbol	Type	Location
a	int	$1
b	int	$2

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

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

We make the convention that `$4` always contains the number 4.

Instead of the symbol table storing registers, it should store the offset from the stack pointer.

Symbol	Type	Location
a	int	4
b	int	0

Offset from stack pointer will cause problems.

Variables also have to go on the stack but we don't know what the offsets should be until we process all of the variables and parameters.

For example

```
int wain(int a, int b)
{ int c = 0; return a; }
```

Code generated

```
lis $4
.word 4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30) ; For int c = 0
sub $30, $30, $4
lw $3, 8($30) ; Offset changed due to presence of c!
...
jr $31
```

As we process the code, we need to be able to compute the offset as we see the values. Also, we need to handle intermediate values of complicated arithmetic expressions by storing on the stack.

Symbol	Type	Location
a	int	8
b	int	4
c	int	0

How then do we arrange it so that when we see the variable, we know what the offset is? Remember that the key issue here is that \$30 (the top of the stack) changes.

Reference the offset from the bottom of the stack frame. We will store this value in \$29. This is called the "frame pointer".

If we calculate offsets from $29, then no matter how far we move the top of the stack, the offsets from $29 will be unchanged.

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
sw $0, -4($30)
sub $30, $30, $4
lw $3, 0($29) ; Offset always 0 from $29
...
jr $31
```

Symbol	Type	Location
a	int	0
b	int	-4
c	int	-8

What about a more complicated program?

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

Question

How do we handle this?

When `a` and `b` were in registers, we could just subtract them. Now we need to load them, then subtract them.

Load them into registers? We'll run out of registers again with more complicated behaviour.

We'll continue to use $3 for the result of any expression. Also use $5 for intermediate scratch values.

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30)
sub $30, $30, $4
sw $2, -4($30)
sub $30, $30, $4
lw $3, 0($29) ; a
add $5, $3, $0 ; Move a to $5
lw $3, -4($29) ; b
sub $3, $5, $3 ; a -b
... ; restore stack
jr $31
```

Question

Where does this approach break down?

Consider something like $a + (b - c)$. Would need to load a, load b, load c, compute $b - c$, then compute the answer. This would require a third register.

Question

Where should we store these values instead?

On the stack again.

Abstraction: We'll use some shorthand for our code

code(x) represents the generated code for x.

code(a): (where a is a variable) `lw $3, N($29)`

push($x):

```
sw $x, -4($30)
sub $30, $30, $4
```

pop($x):

```
add $30, $30, $4
lw $x, -4($30)
```

```
code (a-b):
    code(a) +
    push($3) +
    code(b) +
    pop($5) +
    sub $3, $5, $3
```

18 Lecture 18

Let's compute the MIPS code for

```
int wain(int a, int b) {
        int c = 3;
        return a + (b - c);
}
```

Solution

```
lw $3, 0($29) ; a
sw $3, -4($30) ; push($3)
sub $30, $30, $4
lw $3, -4($29) ; b
sw $3, -4($30) ; push($3)
sub $30, $30, $4
lw $3, -8($29) ; c
add $30, $30, $4 ; pop($5)
lw $5, -4($30)
sub $3, $5, $3 ; b - c
add $30, $30, $4 ; pop($5) l
w $5, -4($30)
add $3, $5, $3
```

We can generalize this technique so we only need two registers for any computation. (Divide and conquer).

Singleton grammar productions are relatively straightforward to translate:

`code(S -> BOF procedures EOF): code(procedures)`

`code(expr -> term): code(term)`

lvalues are odd. Recursion might do the wrong thing.

The basic idea of our code function is that it produces the code to put the value of the expression in `$3`.

lvalues shouldn't actually generate values.

If we have `{ int x = 0; x = 3; }`, having the code for `x` in `x=3` generate 0 would be useless.

We could have the code function do something different for lvalues

`code(lvalue -> anything)` produces an address in `$3` instead of its value.

Or, we could have the code function for expressions that include lvalues do something different based on the kind of lvalue.

`code(stmt -> (lvalue -> ID) BECOMES expr SEMI)` has to be distinct from `code(stmt->(lvalue -> STAR expr) BECOMES expr SEMI)`.

This code is less modular and less maintainable (not recommended).

We have two ways of outputting and one way of inputting: `println`, `putchar`, `getchar`.

Character I/O corresponds directly to memory-mapped I/O addresses:

```
code(putchar(expr);) =
    code(expr) +
    lis $5 +
    .word 0xffff000c +
    sw $3, 0($5)
```

```
code(getchar()) =
    lis $5 +
    .word 0xffff0004 +
    lw $3, 0($5)
```

`println` is more complex than `getchar` or `putchar`.

What do we generate for `stmt -> PRINTLN LPAREN expr RPAREN SEMI`.

We had to write MIPS code to do this, but that would be a lot of code to generate each time.

A compiler mixes the code it outputs with code from a runtime environment

Definition 18.1

A runtime environment is the execution environment provided to an application or software by the operating system to assist programs in their execution. Such an environment may include things such as procedures, libraries, environment variables etc.

MERL files. MIPS Executable Relocatable Linkable. Format for object files. MERL helps us store additional information needed by the loader and linker from output.

We will now need to do

```
./wlpgen < source.wlp4ti > source.asm
cs241.linkasm < source.asm > source.merl
cs241.merl source.merl print.merl > exec.mips
```

gcc and clang are not compilers, they are compiler drivers. They call other programs to compile, assemble, and link code.

To use print, we need to add `.import print` to the BOF.

After this we can use `print` in our MIPS code. It will print the contents of `$1`.

```
code(println(expr);) = push($1)
                     + code(expr)
                     + add $1, $3, $0
                     + push ($31)
                     + lis $5 + .word print
                     + jalr $5 + pop($31)
                     + pop($1)
```

We will write Baby's First Operating System

```
repeat:
    p <- next program to run
    read the program into memory at address 0x0
    jalr $0
    beq $0, $0, repeat
```

Where should this be stored? Could choose different addresses at assembly time, but how do we make sure they don't conflict.

A more flexible option is to make sure that code can be loaded anywhere.

Loader's job:

- Take a program P as input
- Find a location α in memory for P
- Copy P to memory, starting at α
- Return α to OS

Baby's First OS v2

```
repeat:
    p <-- choose program to run
    $3 <--- loader(p)
    jalr $3
    beq $0, $0, repeat

loader
a <-- findFreeRAM(N)
for (i = 0; i < codeLength; ++i) {
    mem[a+4i] = file[i];
}
$30 <-- a + N
return a to OS
```

Question

How did we assemble `.word label`?

It compiles to an address; the address of that label assuming that the program was loaded at 0. Loader needs to fix this.

More problems

`.word id`: need to add alpha to id

`.word constant`: do not relocate

`beq bne` (whether they use labels or not): do not relocate

We translate assembly code into machine code (bits). Given: `0x00000018`, is this `.word 0x18` or `.word id`? We can't know.

Thus, we need a way for our loader to know what does and what doesn't need to be relocated. We need to remember which .words were labels.

In our MERL files we need the code, but also the location of any `.word` ids.

```
lis $3
.word 0xabc
lis $1
.word A
jr $1
B: jr $31
A:
beq $0, $0, B
.word B
```

MERL would be

```
beq $0, $0, 2
.word end Module
.word endCode
---
lis $3
.word 0xabc
lis $1
reloc1: .word A ; different
jr $1
B:
jr $31
A:
beq $0, $0, B
reloc2: .word B ; different
---
endCode:
.word 1
.word reloc1
.word 1
.word reloc2
endModule:
```

Loading requires two passes:

Pass 1: Load the code from the file into the selected location in memory.

Pass 2: Using the relocation table, update any memory addresses that have relocation entries.

Even with this, it is possible to write code that only works at address 0:

```
lis $2
.word 12
jr $2
jr $31
```

We should never encode address as anything other than labels, so that your loader can update the references.

```
read_word // skip first word in MERL file
endMod ← read_word() // second word is address of end of MERL
codeSize ← read_word() - 12 // compute size of code
a ← findFreeRam(codeSize)

for (int i = 0; i < codeSize; i+=4) { // load program
        MEM[a + i] ← read_word()
}
i ← codeSize + 12 // start relocation of table
while (i < endMod) {
        format ← read_word()
        if (format == 1) {
                rel ← read_word() //relocate address
                MEM[a + rel - 12] += a - 12 // go forward by a but back by 12
        } else {
                ERROR // unknown format type
        }
```

```
18        i += 8 // update to next entry
19    }
```

19 Lecture 19

Most of our statements have been completed (except for if and while).

We need to handle boolean tests for conditionals. Convention is to store 1 inside of `$11`.

Code structure

```
; Prologue
lis $4
.word 4
lis $10
.word print
lis $11
.word 1
sub $29, $30, $4
; end Prologue

add $30, $29, $4
jr $31
```

Question

What is the code for the rule test $\rightarrow$ $\text{expr}_A < \text{expr}_B$?

```
code(expr_A)
+ push($3)
+ code(expr_B)
+ pop($5)
+ slt $3, $5, $3
```

For test $\rightarrow$ $\text{expr}_A > \text{expr}_B$ we change `slt $3, $5, $3` to `slt $3, $3, $5`.

Translate test $\rightarrow$ $\text{expr}_A \mathrel{!=} \text{expr}_B$.

```
code(expr_A)
+ push($3)
+ code(expr_B)
+ pop($5)
; maybe store $6 and $7 if used
+ slt $6, $3, $5
+ slt $7, $5, $3
; Note 0 <= $6 + $7 <= 1
+ add $3, $6, $7
```

Now for test $\rightarrow$ $\text{expr}_A == \text{expr}_B$?

The key idea is $a == b$ is the same as $!(a \mathrel{!=} b)$.

We have ! by adding the line `sub $3, $11, $3` to flip 0 to 1 and vice versa.

For test $\rightarrow$ $\text{expr}_A \leq \text{expr}_B$ by using the fact that $a \leq b$ is the same as $!(a > b)$

Rule: `statement → IF (test) {stmts 1} ELSE {stmts 2}`

```
code (statement) = code(test)
                + beq $3, $0, else
                + code(stmts 1)
                + beq $0, $0, endif
                + else: code(stmts 2)
                + end if:
```

If we have multiple if statements, the label names will conflict.

We need a way of inventing totally unique label names.

We can keep track of how many if statements we have. Keep a counter `ifcounter`.

Each time we have an if statement, increment this counter.

Use label names like `else#` and `endif#` where `#` corresponds to the `ifcounter`.

Rule: `statement → WHILE (test) {statements}`

```
code(statement) = loop: code(test)
                + beq $3, $0, endWhile
                + code(stmts)
                + beq $0, $0, loop
                + endWhile:
```

Since we are generating MIPS code; we can also generate comments with MIPS code. Debugging code generators is hard.

Recap

- `$0` is always 0
- `$1` and `$2` are for arguments 1 and 2 in `wain`
- `$3` is always for output
- `$4` is always 4
- `$5` is always for intermediate computations
- `$6` and `$7` may be for intermediate computations
- `$10` will store `print`
- `$11` is reserved for 1
- `$29` is our frame pointer (fp)
- `$30` is our stack pointer (sp)
- `$31` is our return address (ra).

Prologue

At the beginning of the the code, we

- Load 4 into `$4` and 1 into `$11`
- Import print. Store in `$10`
- Store the return address on the stack
- Initialize the frame pointer hence creating a stack frame
- Store registers 1 and 2

Body

Need to store local variables in the stack frame. Contain MIPS code corresponding to the WLP4 program.

Epilogue

Need to pop the stack frame. Also need to restore the previous variables.

We have reached pointers. We need to support all the following

- NULL
- Allocating and deallocating heap memory
- Dereferencing
- Address-of
- Pointer arithmetic
- Pointer comparisons
- Pointer assignments and pointer access

NULL cannot be the value `0x0` it is a valid memory address. We want our NULL to crash in attempt to dereference. We pick a NULL that is not word-aligned (not a multiple of 4). So we can pick `0x1`.

```
code(factor → NULL) =
add $3, $0, $11
```

Question

What about dereferencing?

$\text{factor}_1 \to \text{STAR factor}_2$

The value in factor_2 is a pointer (otherwise a type error). We want to access the value at factor_2 and load it somewhere.

We load into `$3`. Since factor_2 is a memory address, we want to load the value in the memory address at `$3` and store in `$3`.

```
code(factor_1 → STAR factor_2) =
code(factor_2) + lw $3, $0($3)
```

Need to be careful of the difference between lvalues and pointers.

Recall that an lvalue is something that can appear as the LHS of an assignment rule. We can have a rule `factor → AMP lvalue`. Suppose we have an ID value `a`. How do we find out where `a` is in memory?

We use our symbol table. We can load the offset first and then use this to find out the location.

Comparisons of pointers work the same as with integers with one exception. Pointers cannot be negative, so `slt` is not what we want to use. We should use `sltu` instead.

Given `test → expr COMP expr` how can we tell which of `slt` or `sltu` to use? We check the type of exprs.

Recall for addition and subtraction we have several contracts. The code for addition will vary based on the type of its subexpressions. For `int + int` or `int - int`, we proceed as before. This leaves 4 contracts we need to consider that use pointers.

Addition `expr_1 → expr_2 + term` where `type(expr_2) == int*` and `type(term) == int`

```
code(expr_1) = code(expr_2)
             + push($3)
             + code(term)
             + mult $3, $4
             + mflo $3
             + pop($5)
             + add $3, $5, $3
```

`expr_1 → expr_2 + term` where `type(expr_2) == int` and `type(term) == int*`

```
code(expr_1) = code(expr_2)
             + mult $3, $4
             + mflo $3
             + push($3)
             + code(term)
             + pop($5)
             + add $3, $5, $3
```

Subtraction

`expr_1 → expr_2 - term` where `type(expr_2) == int*` and `type(term) == int`

```
code(expr_1) = code(expr_2)
             + push($3)
             + code(term)
             + mult $3, $4
             + mflo $3
             + pop($5)
             + sub $3, $5, $3
```

`expr_1 → expr_2 - term` where `type(expr_2) == int*` and `type(term) == int*`

```
code(expr_1) = code(expr_2)
             + push($3)
             + code(term)
             + pop($5)
             + sub $3, $5, $3
             + div $3, $4
             + mflo $3
```

20 Lecture 20

We need to handle calls such as `new` and `delete`

We can outsource this work to the runtime.

Prologue Additions

```
.import init
.import new
.import delete
```

The command `init` initializes the heap. Must be called at the beginning. Takes a parameter in `$2` and initializes data structure.

New

- Finds the number of new words needed as specified in `$1`
- Returns a pointer to memory of beginning of this many words in`$3` if successful (otherwise places 0 in `$3`)

```
code(new int[expr]) = code(expr)
                    + add $1, $3, $0
                    + call(new)
                    + bne $3, $0, 1
                    + add $3, $11, $0
```

Delete

- Requires that `$1` is a memory address to be deallocated

```
code(delete [] expr) = code(expr)
                    + beq $3, $11, skipDelete
                    + add $1, $3, $0
                    + call(delete)
                    + skipDelete:
```

We need to now deal with multiple function calls.

Question

Who should save which registers? The caller? The callee? What do functions need to update/initialize? How do we update our symbol table?

Question

What do we need to do for wain?

Import `print`, `init`, `new`, `delete`

Initialize `$4`, `$10`, `$11`

Call `init`

Save `$1`, `$2`

Reset stack

Call `jr $31`

For general procedures we don't need any imports. But we need to update `$29`, save registers, restore registers and stack and `jr $31`

Question

Who is responsible for saving and restoring registers?

Definition 20.1

The caller is a function f that calls another function g

Definition 20.2

The callee is a function g that is being called by another function f

Our current convention:

- Caller needs to save `$31`. Otherwise we lose the return address (to, e.g., the loader) once we compute our call to `jalr`
- Callee has been saving registers it will modify and restore at the end. The function f shouldn't be worried about which registers g might be using. This makes sense as well from a programming point of view.
- Note that we have only used registers from `$1` to `$7` (and registers `$4`, `$10`, `$11` are constant) as well as registers `$29`, `$30`, and `$31`.
- `$30` is preserved through symmetry

Question

Who should save `$29`?

Assume that we will require that the callee will save `$29`. Thus they will initialize `$29` first:

`g: sub $29, $30, $4`

and then g saves registers. Is this the right order to do things in?

If we save registers first, `$29` is supposed to point to the beginning of the stack frame, but `$30` has already changed to store all registers. This is fine for now, but `$29` might be pointing to somewhere in the middle of the stack frame.

Having `$29` in the middle is annoying since we will need it later. We choose to make `$29` the bottom.

Therefore, we do:

```
push($29)
add $29, $30, $0
;push other registers
```

This callee-save approach with `$29` will work.

Caller-save approach: We could have the caller save `$29`

```
push($29)
push($31)
lis $5
.word g
jalr $5
pop($31)
pop($29)
```

This is much easier (we are going to do this).

Must be careful where everything is relative to `$29`.

Procedures:

We need to store the arguments to pass to a function.

For `factor → ID(expr1, ..., exprn)`, we have

```
code(factor) = push($29) + push($31)
             + code(expr1) + push($3)
             + code(expr2) + push($3)
             + ...
             + code(exprn) + push($3)
             + lis $5
             + .word ID
             + jalr $5
             + pop n times (pop all regs)
             + pop($31) + pop($29)
```

For `procedure → int ID(params) {dcls stmts RETURN expr;}` we have

```
code(procedure) = ID: sub $29, $30, $4
                + ; Save regs here?
                + code(dcls); local vars
                + ; OR save regs here?
                + code(stmts)
                + code(expr)
                + pop regs ; restore saved
                + add $30, $29, $4
                + jr $31
```

Question

When do we save registers? Before `code(dcls)` or after?

Saving them before is strange.

Stack

$30		
	local vars of g	frame of g
	saved regs of g	frame of g
$29	args of g	frame of f
	$31	frame of f
	$29	frame of f

Symbol Table Revisited

`int g(int a, int b) { int c = 0; int d;}`

Symbol table for g looks like

Symbol	Table	Offset (from $29)
a	int	8
b	int	4
c	int	???
d	int	???

Let's try pushing the registers after pushing the declarations.

For `procedure -> int ID(params) {dcls stmts RETURN expr;}`

```
code(procedure) = ID: sub $29, $30, $4
                + code(dcls)
                + push regs
                + code(stmts)
                + code(expr)
                + pop regs
                + add $30, $29, $4
                + jr $31
```

New stack

$30		
	saved regs of g	frame of g
	local vars of g	frame of g
$29	args of g	frame of f
	$31	frame of f
	$29	frame of f

Symbol Table Revisited Revisited

`int g(int a, int b) {int c = 0; ind d;}`

Symbol	Table	Offset (from $29)
a	int	8
b	int	4
c	int	0
d	int	-4

Parameters should have positive offsets. Local variables should have non-positive offsets.

Symbol table should have added $4 \cdot num(\text{params})$ to each entry in the table.

This complicates pushing registers, because we're now generating some code before we preserve register values.

Does this matter for us? No, because our declarations are forced to be simple. If the language allowed complex expressions then it would matter.

Labels can introduce another annoying problem

Consider the code

```
int print(int a ) {
    return a;
}
```

Question

What is the problem here?

We already have a label called `print`. Since it is not a WLP4 procedure, it shouldn't interfere with WLP4.

We can ban WLP4 code that uses function names that match some of our reserved labels like `new`, `init`, etc. This is not very future proof.

Since MIPS labels don't have to be identical to WLP4 procedure names, we can change them.

We will prepend an 'F' to the front of labels corresponding to procedures. Then, so long as we don't create any labels with a 'F' at the beginning for any other purpose it should be okay.

The print function above would correspond to a label `Fprint`.

Revisiting Translation

`factor → ID(expr 1, ..., exprn)` we have,

```
code(factor) = push($29) + push($31)
             + code(expr1) + push($3)
             + code(expr2) + push($3)
             + ...
             + code(exprn) + push($3)
             + lis $5
             + .word FID
             + jalr $5
             + pop n times (pop all regs)
             + pop($31) + pop($29)
```

Compete example, with procedures:

```
int add(int a, int b) {
        int c = 0;
        c = a + b;
        return c;
}
int wain(int a, int b) {
        return add(a, b);
}
```

```
.import print/new, delete, init

lis $4
.word 4
lis $10
.word print
lis $11
.word 1
---
beq $0, $0, Fwain
Fadd: sub $29, $30, $4
    code(int c = 0);
        push($0) ; above translated
    ; save regs
    push($5)
    push($6)
    push($7)
    push($31)
    ; done with register saving (now stmts)
    code(stmts)
        code(c = a + b)
            code(lvalue/c)
            push($3)
            code(expr/a+b)
            pop($5)
            sw $3, 0($5)
    ; final expr
    code(c)
    ; restore regs
    ; pop/locals
    jr $31 ; need to restore the stack
Fwain:
    push($1) ; first param
    push($2) ; second param
    sub $29, $30, $4
    push($31)
    ; save register $2
    $2 = 0
    call init ($2)
    ; no stmts
    ; return expr
    code(add(a, b))
    pop($0) ; pop a
    pop($0) ; pop b
    jr $31
```

We have finished code generation.

Compiler Optimizations

The goal of optimizations is generally to make code run faster

We want to reduce code size and make code run faster. (For bonus on A8, we are only concerned with reducing code size).

Implementing these optimizations is not easy.

Constant Folding

If we want to generate the code for $1 + 2$:

```
lis $3 ; $3 = 1
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $2 ; $3 = 2
.word 2
add $30, $30, $4 ; pop($5)
lw $5, -4($30)
add $3, $5, $3 ; $3 = 1 + 2
```

Our code generator could produce the following code for $1 + 2$

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

If we notice that each element of the expression is a constant, we can add the constants at compile time and output the code for the final value (instead of doing it at runtime).

If the expression was $1 + x$, we would need to know the value of variable x. We cannot determine this at compile time.

Constant Propogation

Sometimes the value of a variable is known at compile time:

```
int x = 1;
return x + x;
```

We can replace x with its known value, so this is equivalent to:

```
int x = 1;
return 1 + 1;
```

We can then apply Constant Folding

```
int x = 1;
return 2;
```

Since `x` is not used anywhere, we can eliminate the variable declaration entirely:

```
return 2;
```

21 Lecture 21

Constant propagation is more difficult than constant folding.

```
int wain(int x, int y) {
        println(x + x); // Constant propagation can't be applied
        x = 1;
        println(x + x); // Can be applied
        x = y;
        return x + x; // Can't be applied
}
```

We can only apply it if we know the variable's value does not depend on the input during the part of the program we're processing.

Common Subexpression Elimination

Even if the value of x is not known, there is a simplification we can make when generating code for $x+x$.

Here is the naïve code (assuming x is at offset 0 from $29)

```
lw $3, 0($29) ; $3 = x
sw $3, -4($30) ; push($3)
sub $30, $30, $4
lw $3, 0($29) ; $3 = x
add $30, $30, $4 ; pop($5)
lw $5, -4($30)
add $3, $5, $5 ; $3 = x + x
```

Even if the value of x is not known, there is a simplification we can make when generating code for $x+x$.

Since we're adding the same variable twice, we can just do this.

```
lw $3, 0($29) ; $3 = x
add $3, $3, $3 ; $3 = x + x
```

We can do the same trick with larger expressions, e.g., if we have $(a * b - c) + (a * b - c)$:

```
; block of code that computes a * b - c
add $3, $3, $3
```

Question

Can we apply common subexpression elimination to this code?

```
1 int f(int x) {
2         println(x);
3         return 2*x;
4 }
5 int wain(int a, int b) {
6         return f(a) + f(a);
7 }
```

No, CSE must not eliminate side effects.

Dead Code Elimination

Sometimes the compiler can determine that certain code will never execute, and can eliminate this code

```
1 int wain(int a, int b) {
2         if (a < b) {
3                 if (b < a) {
4                         b = 0;
5                 } else { }
6         } else { b = 0; }
7         return a + b;
8 }
```

The code inside the innermost if can be ignored.

```
int wain(int a, int b) {
        if (a < b) {
                if (b < a) {
                        // dead code
                } else { }
        } else { b = 0; }
        return a + b;
}
```

Deleting this code has a size benefit, but no real performance benefit.

```
int wain(int a, int b) {
        if (a < b) {
        // if condition eliminated
        } else { b = 0; }
        return a + b;
}
```

Dead code elimination interacts with other optimizations.

```
int wain(int x, int y) {
        int releaseVersion = 0;
        if (releaseVersion == 1) {
                x = 1;
        } else {
                x = 0;
        }
        return x * y;
}
```

Normally we can't apply constant propagation to x in the return.

```
int wain(int x, int y) {
        int releaseVersion = 0;
        x = 0;
        return x * y;
}
```

Now constant propagation can be used on x as well.

DCE can allow constant propagation to occur. Conversely, constant propagation can allow the compiler to prove code is dead.

Register Allocation

We ran into the issue that for sufficiently complicated code, it is not possible to store all values in registers.

Our solution was to put everything on the stack because this makes generating code simpler and more consistent.

But using registers for storage is much faster than using RAM.

Real-world compilers try to use registers as much as possible.

A variable is live if the current value of the variable will be used at a later point in the program.

A variable should be in a register if and only if it is live.

If too many variables or values are live at the same time, we have to choose which ones to put in RAM vs registers.

```
x = 3;
y=10;
println(x);
z = 7;
y = y-x;
y = y-z;
println(z);
return z;
```

- x becomes live on line 1, and is last used on line 5
- y becomes live on line 2, and is last used on line 6
- z becomes live on line 4, and is last used on line 8

On lines 4 to 5, all three variables are live. If we only had two registers available, we would need to put one variable in RAM.

We can use live ranges to construct a graph indicating which ranges overlap, and use graph colouring algorithms to allocate registers (see MATH239)

If the live range graph can be $k-$coloured, where k is the number of available registers, we can allocate all variables to registers.

Graph colouring can be slow (NP-complete problem), so it is usually approximated.

If the address-of operator is used on a variable then this variable must go in RAM.

Significant gains are possible just by implementing a basic register allocator. Optimizations to eliminate pushes/pops or decrease the number of instructions for a push/pop are effective on A8.

A heuristic can be allocating variables and temporaries to registers on a "first-come, first-served" basis.

In this case we'd need to modify the offset table so that there are two kinds of variable locations: offsets from the frame pointer, or registers.

Allocate non-parameter local variables in registers whenever possible.

Strength Reduction

This optimization involves replacing costly operations with equivalent faster operations. For example, multiplication is slower than addition.

- $(x + y) \times 2$ can be replaced with $(x + y) + (x + y)$, which can then be optimized further using common subexpression elimination.

A more complex version involves optimizing loops which perform expensive operations involving the loop counter.

Peephole Optimization

This optimization happens after code generation is finished. This is used in LLVM.

Instead of directly outputting the generated code, the code is placed in a data structure and subject to further analysis. The analysis tries to find sequences of instructions that can be replaced with simpler sequences.

For example

```
add $3, $1, $0 ; $3 = a
add $7, $3, $0 ; copy $3 to temporary register
```

A peephole optimization could change this to `add $7, $1, $0`. This might be easier than making the code generation itself smarter.

We use a sliding window.

Inlining Functions

We replace a function call with the body of the function itself

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

This is equivalent to:

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

This removes the overhead of a function call.

Tail Recursion

A recursive function call is in tail position if it is the last thing the function executes before returning.

In this case, what happens normally is:

- Recursive call happens, pushes local variables to the stack
- Recursive call finishes, pops from the stack, returns
- Original call finishes, pops from the stack, returns

Tail call optimization is based on the observation that in this situation, the recursive call can reuse the stack frame of the original call instead of pushing its own stack frame. This saves a lot of space.

Original call pops the reused stack frame.

```
int fac(int n) {
        return fac_rec(1, n);
}
int fac_rec(int acc, int n) {
        if (n < 2) {
                return acc;
        }
        return fac_rec(n*acc,n-1);
}
```

Can be replaced by

```
int fac_rec(int acc, int n) {
        TOP:
```

```
        if (n < 2) return acc;
        acc = n * acc;
        n = n - 1;
        gotoTOP
}
```

22 Lecture 22

Why do we split programs into multiple files?

Modularity, team development, faster build time

Question

How do we resolve situations where we have labels in different files?

One option is to cat all such files together, and then compile?

- Duplicate labels defined in different files
- Accidental use of labels which should be private

Can we assemble files first and then cat?

Almost. Only one piece of code can be at `0x0` at a time. These assembled files need to be MERL files, not just MIPS.

Concatenating two MERL files does not give a valid MERL file.

We still haven't resolve the issue of labels in different files.

When we encounter a `.word` where the label is not in the file, we need to use a placeholder (an arbitrary value), and indicate that we cannot run this program until the value of the label is given.

```
a.asm
lis $3
.word label
jr $31
```

```
b.asm
label: sw $4, -4($30)
```

You cannot run `a.asm` without linking with `b.asm`.

We need to extend our MERL file to notify us when we need to assembly with multiple files.

But sometimes we make typos. Consider

```
lis $3
.word bananana
banana:
```

Question

Did we make a mistake? Did we mean `.word banana`, or dod we mean for bananana to be provided by another MERL file?

How do we recognize such errors? Without any other changes, our assembler will believe that a label banana exists somewhere and would load this.

`.import id` is the directive that tells the assembler which symbols to link in. This will not assemble to a word of MIPS. Errors occur if the label id is not in the current file and there is no `.import id` in the file.

We need to add entries in the MERL symbol table. Previously we used the code `0x1` for relocation entries, but this isn't a relocation entry.

New format code: `0x11` for External Symbol Reference (ESR).

Question

What needs to be in an ESR entry?

1. Where the symbol is being used
2. The name of said symbol

Format:

```
0x11 ; Format code
; location used
; length of name of symbol (n)
; 1st ASCII char of name of symbol
; 2nd ASCII char of name of symbol
; ...
; nth ASCII char of name of symbol
```

Question

What if labels are duplicated?

Suppose we have `c.asm` along with our two other files that has:

```
label: add $1, $0, $0
; more code
beq $1, $0, label
```

We want label to not be exported, it should be self-contained.

`.export label` will make `label` available for linking with other files. As with `.import`, it does not translate to a word in MIPS. It tells the assembler to make an entry in the MERL symbol table.

The assembler makes an ESD, or an External Symbol Definition, for these types of words. It follows this format:

```
0x05 ; Format code
; Address the symbol represents
; length of name of symbol (n)
; 1st ASCII char of name of symbol
; 2nd ASCII char of name of symbol
; ...
; nth ASCII char of name of symbol
```

Our linker now has everything it needs to do its job.

Linking Algorithm

```
// Check for duplicate export errors
for each ESD in m1.table {
```

```
        if there is an ESD with the same name in m2.table {
                ERROR (duplicate exports)
        }
}

// Combine the code segments for the linked file
// The code for m2 must appear after the code for m1

linked_code = concatenate m1.code and m2.code
```

```
// Relocate m2's table entries
reloc_offset = end of m1.code - 12 // length of m1.code
for each entry in m2.table {
        add reloc_offset to the number stored in the entry
}

// Relocate m2.code
// It is essential for this to happen after the last step
for each relocation entry in m2.table {
        index = (address to relocate - 12) / word size
        add relocation offset to linked_code[index]
}
```

```
// Resolve imports for m1
for each ESR in m1.table {
        if there is an ESD in m2.table with a matching name {
                index = (address of ESR - 12) / word size
                overwrite linked_code[index] with the exported label value
                change the ESR to a REL
        }
}

// Resolve imports for m2
// Repeat previous step for imoprts from m2 and exports for m1
```

```
// Combine the tables for the linked file
linked_table = concatenate modified m1.table and modified m2.table

// Compute the header information
endCode = 12 + linked_code size in bytes
endModule = endCode + linked_table size in bytes

// Output the MERL file
output merl cookie
output endModule
output endCode
output linked_code
output linked_table
```

Linking Example

```
m1.asm

.import b
.export f
f: .word f
    .word b
l: word l
```

```
;; HEADER of m1
0x00: beq $0, $0, 2 ; header: beq
0x04: 0x48 ; header: endModule
0x08: 0x18 ; header: endCode
;; CODE
0x0c: 0x0c ; f: .word f
0x10: 0x00 ; .word b ; placeholder
0x14: 0x14 ; l: .word l
;; FOOTER
0x18: 0x01 ; footer: relocation entry
0x1c: 0x0c ; relocation entry at 0x0c for f:
0x20: 0x01 ; footer: relocation entry
0x24: 0x14 ; relocation entry at 0x14 for l:
0x28: 0x11 ; footer: external symbol reference (ESR)
0x2c: 0x10 ; address where ESR is used, i.e., "b"
0x30: 0x01 ; length of the label "b"
0x34: 0x62 ; ASCII for "b"
0x38: 0x05 ; footer: external symbol definition (ESD)
0x3c: 0x0c ; address where ESD is defined, i.e., "f"
0x40: 0x01 ; length of the label "f"
0x44: 0x66 : ASCII for "f"
```

```
m2.asm

.import f
.export b
    .word f
b: .word b
l: .word l
```

```
;; HEADER of m2
0x00: beq $0, $0, 2 ; header: beq
0x04: 0x48 ; header: endModule
0x08: 0x18 ; header: endCode
;; CODE
0x0c: 0x00 ; .word f ; placeholder
0x10: 0x10 ; b: .word b
0x14: 0x14 ; l: .word l
;; FOOTER
0x18: 0x01 ; footer: relocation entry
0x1c: 0x10 ; relocation entry at 0x0c for b:
0x20: 0x01 ; footer: relocation entry
0x24: 0x14 ; relocation entry at 0x14 for l:
0x28: 0x11 ; footer: external symbol reference (ESR)
0x2c: 0x0c ; address where ESR is used, i.e., "f"
0x30: 0x01 ; length of the label "f"
0x34: 0x66 ; ASCII for "f"
0x38: 0x05 ; footer: external symbol definition (ESD)
0x3c: 0x10 ; address where ESD is defined, i.e., "b"
0x40: 0x01 ; length of the label "b"
0x44: 0x62 : ASCII for "b"
```

Linked code (after a lot of work)

```
;; HEADER
0x00: beq $0, $0, 2 ; header: beq
0x04: 0x74 ; header: endModule
0x08: 0x24 ; header: endCode
;; CODE from m1.
0x0c: 0x0c ; f: .word f
0x10: 0x1c ; .word b
0x14: 0x14 ; l: .word l
;; CODE from m2.
0x18: 0x0c ; .word f ; placeholder
0x1c: 0x1c ; b: .word b
0x20: 0x20 ; l: .word l
```

```
;; FOOTER of m1
0x24: 0x01 ; footer: relocation entry
0x28: 0x0c ; relocation entry at 0x0c for f:
0x2c: 0x01 ; footer: relocation entry
0x30: 0x14 ; relocation entry at 0x14 for l:
0x34: 0x01 ; footer: relocation entry
0x38: 0x10 ; relocation entry at 0x10 for b:
0x3c: 0x05 ; footer: external symbol definition (ESD)
0x40: 0x0c ; address where ESD is defined, i.e., "f"
0x44: 0x01 ; length of the label "f"
0x48: 0x66 : ASCII for "f"
;; FOOTER of m2
0x4c: 0x01 ; footer: relocation entry
0x50: 0x1c ; relocation entry at 0x0c for b:
0x54: 0x01 ; footer: relocation entry
0x58: 0x20 ; relocation entry at 0x14 for l:
0x5c: 0x01 ; footer: relocation entry
0x60: 0x18 ; relocation entry at 0x18 for f:
0x64: 0x05 ; footer: external symbol definition (ESD)
0x68: 0x1c ; address where ESD is defined, i.e., "b"
0x6c: 0x01 ; length of the label "b"
0x70: 0x62 : ASCII for "b"
```

23 Lecture 23

Heap Management

In this course we have a library to deal with all of the memory management features, including `init`, `new`, and `delete`.

This allows for data to exist in memory that is out of scope, that is, out of the boundaries of your stack frame.

Question

How do we manage this memory?

The memory not on the stack is either in code, or on the heap. The `init` procedure initializes a heap for us to use.

This is much more problematic than a stack to take care of. Stacks are nice and ordered. Calls can be made to heaps using `delete` or `new` in arbitrary orders, so we can't simply push and pop memory.

In our world:

Code
Read-only data
Global data
Heap Stack

Code is just data. You can store data in your code. In C, it's common to separate out static, global information from code, but it's all just the stuff before the dynamic heap.

Question

How does a heap work?

We have a variety of implementations.

Example 1: No reclamation of Memory and Fixed Blocks

After `init`, we get two pointers, one to the start of memory on the heap and one at the end.

Initialization is $O(1)$. Allocation is also $O(1)$. We never delete.

Clearly not the best choice. We will run out of memory quickly since we aren't reusing reclaimed memory.

Example 2: Explicit Reclamation and Linked List of Fixed-Size Blocks

Keep the fixed size idea the same, but keep track of a free list (linked list of free memory blocks) and we can allocate from this linked list.

Example 3: Variable-Sized Blocks

We once again used a linked list but here our linked list will store a number of bytes and the next node.

Init: Start with the entire heap being free.

Let's say we want to allocate 50 bytes. What we will do is allocate 54 bytes.

The first 4 bytes are the size of the block (integer), and the rest is the requested bytes. We need this bookkeeping because delete doesn't take a size. We return a pointer to the start of the 50 bytes.

Memory:

54	start 50 p	...	end 50	

Free List is one node with 970 and $0x4036$ (since we started with 1024 bytes at address $0x4000$).

Next, we allocate 28 bytes. We allocated 32 bytes and return a pointer to the start of the 28 bytes.

54	p	...	32	q	...	end 28

Free list is one node with 938 and $0x4056$

Freeing the 50 bytes results in

32	q	...

Free list is 54 $0x4000$ pointing to 938 $0x4056$.

Freeing the other 32 bytes results in a free list of 54 $0x4000$ pointing to 32 $0x4036$ pointing to 938 $0x4056$

We can do some consolidation. Notice that $54 + 0x4000 = 0x4036$ and so the first two nodes in our linked list can collapse to a single node.

We can further consolidate since $86 + 0x4000 = 0x4056$.

The biggest issue with this approach is fragmentation. Suppose we have 48 bytes and we make the following calls:

- Allocate $4 + 8$
- Allocate $4 + 16$
- Allocate $4 + 4$
- Free 20
- Allocate $4 + 4$

We can't allocate 16, despite having 24 bytes free.

We've been showing the free list as a separate data structure. But we're the ones allocating the data structures. So how do we allocate space for the free itself?

The free list goes in the space itself. It is free, so the user doesn't care what we put there.

24 Lecture 24

Dealing with Fragmentation

Heuristics:

- First fit: Put memory in the first available spot
- Best fit: Put the block in an exact match (or as close to it so there is less waste)
- Worst fit: Exact match if possible, otherwise put the block in the largest available space

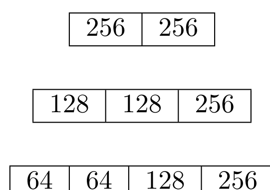
Problem: Best- and worst-fit involve looking over the whole free list (so they are slow).

Other ideas include `dmalloc` and the binary buddy system.

Binary Buddy System

Start with 512 bytes of heap memory.

Suppose we try to allocate 19 bytes. We need an extra one for bookkeeping (so 20). This fits in a block of size $2^5 = 32$. We split memory until we find such a block and reserve the entire block.



Binary buddy still creates fragmentation, just of a different sort. If most allocations are of nice powers of two, it defragments well.

Both the size and location of any block can be encoded using a list of “left” and “right” actions.

Binary Buddy Code

Start with a 1. The code for the whole of the heap is just 1

For each block division, append a 0 if the left block is selected, or a 1 if the right block is selected.

Since the first bit is always 1, we can tell the length of the code by looking for the first 1.

Since each bit represents a power of two, very short codes represent a lot of information.

One word is plenty for a code.

Some languages take care of deallocation.

```
int f() {
        Myclass ob = new MyClass();
} // ob no longer accessible
// garbage collector reclaims
```

Second example

```
int f() {
        MyClass ob2 = null;
        if (x == y) {
                MyClass ob1 = new MyClass();
                ob2 = ob1;
        } // ob1 goes out of scope
        // ob2 still holds the object
} // ob2 no longer accessible
```

32	32	64	128	256

In order for automatic memory management to happen, the compiler and the allocator need to coordinate in some way.

At the minimum, the compiler needs to tell the allocator when something goes out of scope.

Technique 1: Reference Counting

For each heap block, keep track of the number of pointers that point to it.

Must watch every pointer and update reference counts each time a pointer is reassigned. The compiler needs to call some procedure provided by the allocator every time a pointer goes into or out of scope.

If a block's reference count reaches 0, reclaim it.

```
struct List {
        List *next;
        int val;
};
l = new List;
l->next = new List;
// l->next is not a variable in scope
// it's a field of an object, and there is
// a reference to that object in scope
```

Question

What issues are there to this?

If a block points to another block and vice versa, then the cluster is unreachable and should be cleaned.

But both will retain a reference to each other.

Question

If reference counting is so bad, why do people use it?

It's very straightforward to implement, so it's an easy way to get some automatic memory management easily.

But it's expensive and flawed.

Question

Is it possible to fix this issue with reference counting?

Short answer: No. Long answer: What if we delayed counting references, so that we would never get these problematic clusters in the first place.

The remaining techniques are classed as garbage collection.

Technique 2: Mark and Sweep

Scan the entire stack (and global variables) and search for pointers. Mark the heap blocks that have pointers to them. If these contain pointers, continue following.

Then scan the heap, reclaim any blocks that aren't marked. Boils down to a graph traversal problem.

The compiler needs to tell the allocator

- Where the pointers are in the stack
- Where the pointers are in the heap

This means that the compiler and allocator need to agree on both stack layout and object layout.

This is a much greater degree of co-design than is needed for reference counting (so mark and sweep is rarely bolted onto a language post-hoc).

Technique 3: Copying Collector

Split the heap into two halves, say H_1 and H_2.

Allocate memory in H_1. Perform a mark-and-copy; but mark by coping objects from H_1 into H_2.

After the copy H_2 has all living objects stored contiguously.

Once finished copying, begin allocation to H_2 (reverse the roles).

Three major benefits

- Allocation is always extremely fast (filling a heap)
- The "sweep" phase is free
- We have no fragmentation in memory

If most objects aren't long-lived, we don't actually do much.

Since copying an object from one heap to an object is moving objects, the compiler needs to be prepared for the allocator to modify active memory.

In practice, most objects are short-lived (good for copy), but those objects that aren't short lived are nearly immortal.

Most practical garbage collectors are generational collectors: they use copying from H_1 to H_2, but then H_2 is mark-and-sweep.

With generational, we benefit from the free sweep for young objects, and the lack of copying for old objects.