

CS 241 with Gregor Richards

Eason Li

2025 W

Contents

| | | |
|----------|---|-----------|
| 1 | Data Representation | 6 |
| 1.1 | Bytes as Binary Numbers | 6 |
| 1.1.1 | Unsigned Integers in Binary | 7 |
| 1.1.2 | Signed Integers in Binary | 7 |
| 1.2 | ASCII Representation for English Text | 10 |
| 1.3 | Bitwise-Operators | 11 |
| 2 | Machine Language | 13 |
| 2.1 | CPU | 13 |
| 2.2 | MIPS Instruction Set | 14 |
| 2.3 | Fetch-Execute Cycle | 15 |
| 2.3.1 | Halt the Fetch-execute Cycle | 16 |
| 2.4 | Multiplication and division | 16 |
| 2.4.1 | Multiplication | 16 |
| 2.4.2 | Division | 16 |
| 2.5 | RAM | 16 |
| 2.6 | Assembly Language | 17 |
| 3 | Scanning and Regular Languages | 18 |
| 3.1 | Maximal Munch Scanning | 18 |
| 3.2 | Formal Languages | 18 |
| 3.2.1 | Membership in Languages | 19 |
| 3.3 | Regular Expressions & Regular Languages | 20 |
| 3.3.1 | Union | 20 |
| 3.3.2 | Concatenation | 20 |
| 3.3.3 | Kleene Star | 20 |
| 3.3.4 | Notation | 21 |
| 3.4 | Recognizing Regular Languages | 21 |
| 3.4.1 | From Regular Expressions to Programs | 21 |
| 3.4.2 | Deterministic Finite Automata | 23 |
| 3.4.3 | Generic DFA Recognition Algorithm | 24 |
| 3.5 | Maximal Munch Scanning: The Details | 24 |
| 3.5.1 | Simplified Maximal Munch | 25 |
| 4 | MIPS Assemblers & Assembly Language | 26 |
| 4.1 | Writing an Assembler: First Attempt | 26 |
| 4.1.1 | Scanning | 26 |
| 4.1.2 | Parsing | 27 |
| 4.1.3 | Semantics Analysis | 28 |
| 4.1.4 | Synthesis: Encoding, Output, & Bitwise Operations | 28 |
| 4.1.5 | Bitwise Operations | 28 |
| 4.1.6 | Encoding Instructions with Bitwise Operations | 30 |
| 4.1.7 | Producing Output | 30 |

| | | |
|----------|--|-----------|
| 4.2 | Advanced MIPS Assembly Programming | 31 |
| 4.2.1 | More Arithmetic: Multiplication & Division | 31 |
| 4.2.2 | Less-Than Comparison | 32 |
| 4.2.3 | Conditional Branching & Loops | 32 |
| 4.2.4 | Labels | 34 |
| 4.3 | Writing an Assembler with Label Support | 36 |
| 4.3.1 | The First Pass: Scanning Changes | 36 |
| 4.3.2 | The First Pass: Parsing and the Symbol Table | 37 |
| 4.3.3 | The Second Pass: Semantic Analysis and Synthesis | 38 |
| 4.4 | Input & Output in MIPS | 38 |
| 4.5 | Arrays in MIPS Assembly | 39 |
| 4.5.1 | Statically Allocated Arrays | 41 |
| 4.5.2 | Stack-Allocated Arrays | 42 |
| 4.6 | Procedures in MIPS Assembly | 44 |
| 4.6.1 | Protecting Data in Registers | 44 |
| 4.6.2 | Call & Return | 45 |
| 4.6.3 | Chaining Procedure Calls & Recursion | 45 |
| 4.6.4 | Passing Arguments & Returning Values | 46 |
| 5 | Context Free Languages and Parsing | 48 |
| 5.1 | Formal Language Theory | 48 |
| 5.2 | Regular Languages Revisited | 48 |
| 5.2.1 | Kleene's Theorem | 48 |
| 5.2.2 | Nondeterministic Finite Automata | 49 |
| 5.2.3 | From Regular Expressions to NFAs | 51 |
| 5.3 | The Limitations of Regular Languages | 52 |
| 5.4 | Context-Free Languages | 53 |
| 5.4.1 | Conventions | 54 |
| 5.4.2 | Derivations | 54 |
| 5.4.3 | The Language of a Grammar | 55 |
| 5.4.4 | Derivations and Parse Trees | 57 |
| 5.4.5 | Ambiguous Grammars | 57 |
| 5.4.6 | Parsing Algorithm | 60 |
| 6 | LL(1) Top-Down Parsing | 61 |
| 6.1 | Augmented Grammars | 61 |
| 6.2 | Informal Top-Down Parsing Algorithm | 61 |
| 6.3 | Formal Top-Down Parsing Algorithm | 63 |
| 6.4 | Constructing the Predict Table | 65 |
| 6.4.1 | Computing Nullable | 66 |
| 6.4.2 | Computing First | 66 |
| 6.4.3 | Computing Follow | 68 |
| 6.4.4 | Computing Predict Table | 68 |
| 6.5 | Parse Tree | 70 |

| | | |
|----------|---|------------|
| 6.6 | Limitations of LL(1) Grammars | 72 |
| 6.6.1 | Writing Right Recursive Grammars | 73 |
| 7 | Bottom-Up Parsing | 75 |
| 7.1 | Bottom-Up Parsing, Informally | 75 |
| 7.2 | Bottom-Up Parsing, Less Informally | 75 |
| 7.3 | ATrace Through The LR(0) Algorithm | 78 |
| 7.3.1 | Optional: LR(0) Formalism | 80 |
| 7.4 | Action Conflicts | 80 |
| 7.5 | Using Lookaheads | 82 |
| 7.5.1 | Optional: Formally defining LR(1) and SLR(1) | 83 |
| 7.6 | SLR(1), LALR(1) and LR(1) | 83 |
| 7.7 | But Where is my Parse Tree | 85 |
| 7.8 | Summary | 85 |
| 8 | Context-Sensitive Analysis | 86 |
| 8.1 | Intro to WLP4 | 86 |
| 8.2 | Context-Sensitive Analyses in WLP4 | 86 |
| 8.2.1 | Tree Traverse Examples | 87 |
| 8.3 | Catching Identifier Errors | 88 |
| 8.3.1 | More Examples in WLP4 | 89 |
| 8.3.2 | Checking Variable Use | 91 |
| 8.3.3 | Checking Procedure Calls | 92 |
| 8.4 | Catching Type Errors | 94 |
| 8.4.1 | Type Inference Rules | 96 |
| 8.4.2 | Type Checking Statements | 97 |
| 8.5 | Other Context-Sensitive Analyses | 99 |
| 9 | Code Generation | 102 |
| 9.1 | Accessing Variables | 102 |
| 9.1.1 | Initializing Variables | 105 |
| 9.1.2 | NULL Pointers | 105 |
| 9.2 | Expressions With Binary Operations | 106 |
| 9.3 | Unary Operations, Assignment, & Lvalues | 108 |
| 9.3.1 | Pointer Dereference | 109 |
| 9.3.2 | Address-of | 110 |
| 9.3.3 | Handling lvalue Summary | 110 |
| 9.4 | Pointer Arithmetic | 111 |
| 9.5 | Input / Output | 112 |
| 9.6 | If Statements and While Loops | 112 |
| 9.7 | Printing Integers & Calling External Procedures | 114 |
| 9.8 | Generating Code for WLP4 Procedures | 116 |
| 9.8.1 | Starting with <code>wain</code> | 116 |
| 9.8.2 | Calling Conventions | 116 |
| 9.8.3 | Calling Procedures | 118 |

| | | |
|-----------|---|------------|
| 9.8.4 | Generating The Procedure Itself | 118 |
| 9.8.5 | Label Names for Procedures | 120 |
| 10 | Runtime Support: Loading & Linking | 121 |
| 10.1 | Loader | 121 |
| 10.1.1 | The MERL Format | 124 |
| 10.1.2 | Loader Relocation Algorithm | 125 |
| 10.2 | Linking | 126 |
| 10.2.1 | Linking Algorithm | 130 |
| 10.2.2 | Tracing Through the Linking Algorithm | 133 |
| 10.2.3 | Practice | 133 |
| 11 | Memory Management | 135 |
| 11.1 | The Core Problem | 135 |
| 11.2 | The Free List Algorithm | 135 |
| 11.2.1 | Allocation Strategies | 136 |
| 11.2.2 | Deallocation and Coalescing | 136 |
| 11.3 | Garbage Collection | 138 |
| 11.3.1 | Reference Counting | 138 |
| 11.3.2 | Mark and Sweep | 139 |
| 11.3.3 | Copying Collector | 140 |
| 11.3.4 | Generational Garbage Collection | 140 |
| 12 | Closing Thoughts | 141 |

1 Data Representation

Lecture 1 - Tuesday, January 07

Definition 1.1: Bit

A **bit** is a binary digit. That is, a 0 or a 1 (off or on).

Definition 1.2: Nibble

A **nibble** is 4 bits.

Definition 1.3: Byte

A **byte** is 8 bits.

Definition 1.4: Word

A **word** is a machine-specific grouping of bytes. For us, a word will be 4 bytes (32 bits) as we'll using a 32-bit computer architecture. Note that 8-byte words (for 64-bit architectures) are more common in modern devices.

Definition 1.5: Hexadecimal Notation

A base 16 representation is called the **hexadecimal** system.

Example 1.1

The binary number 10011101 will be converted to 9d in hexadecimal.

Theorem 1.1: What do bytes represent

- Numbers;
- Characters;
- Garbage in memory;
- Instructions (Parts of instructions in our case. Words, or 4 bytes, will correspond to a complete instruction for our computer system).

1.1 Bytes as Binary Numbers

There are two types of numbers: unsigned (non-negative numbers), and signed numbers.

1.1.1 Unsigned Integers in Binary

This is a positional number system that works exactly how the unsigned (non-negative) decimal system works.

| | | | | | | | |
|-------|-------|-------|-------|-------|-------|-------|-------|
| b_7 | b_6 | b_5 | b_4 | b_3 | b_2 | b_1 | b_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|

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

$$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Example 1.2

For example,

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

or

$$11111111_2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255_{10}$$

1.1.2 Signed Integers in Binary

We will look at two ways of converting a decimal number to its binary representation.

Definition 1.6: Sign-magnitude Representation

An intuitive approach is to use **sign-magnitude representation**: Use the most significant bit to indicate the sign. A 0 bit indicates a positive integer, and a 1 bit indicates a negative integer. The remaining bits represent the magnitude (absolute value) of the number in unsigned representation. This approach has multiple problems and is not used in practice.

Discovery 1.1

Notice that we have two problems:

- Two representations for 0, which is awkward and wasteful.
- Arithmetic is tricky.

Definition 1.7: Two's Complement Representation

A more common approach to representing signed integers, used in this course and in modern computers, is **two's complement representation**.

- It is similar to sign-magnitude in spirit; first bit is 0 if non-negative, 1 if negative.
- To negate a value: Take the complement of all bits (flip the 0 bits to 1 and 1 bits to 0) and add 1.

Example 1.3

To convert 11011010_2 , a number in two's complement representation, to decimal, one method is to flip the bits and add 1:

$$00100110_2 = 2^5 + 2^2 + 2^1 = 38.$$

Thus, the corresponding positive number is 38 and so the original number is -38 .

Another way to do this computation is to treat the original number 11011010_2 as an unsigned number, convert to decimal and subtract 2^8 from it (since we have 8 bits, and the first bit is a 1 meaning it should be a negative value). This also gives -38 :

$$\begin{aligned} 11011010_2 &= 2^7 + 2^6 + 2^4 + 2^3 + 2^1 - 2^8 \\ &= 128 + 64 + 16 + 8 + 2 - 256 \\ &= 218 - 256 = -38 \end{aligned}$$

Proof. We can justify that this works using modular arithmetic:

- The range for unsigned integers is 0 to 255. Recall that 255 is 11111111_2 . If we add 1 to 255, then, after discarding overflow bits, we get the number 0.
- Thus, let's treat 2^8 as 0, i.e., let's work modulo $2^8 = 256$. In this vein, we set up a correspondence between the positive integer k and the unsigned integer $2^8 - k$. Since we are working modulo 2^8 , subtracting a positive integer k from 0 is the same as subtracting it from 2^8 .
- In this case, note that $2^8 - 1 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ and in general:

$$2^n - 1 = \sum_{i=0}^{n-1} 2^i$$

Equivalently, consider overflow. If we take binary number 00000000 and subtract 00000010 (2), we'll get 11111110, ignoring the overflowing carry bit. 11111110 is the 8-bit two's complement representation of -2 , which is, of course, $0 - 2$. 11111110 is also the value we find using the approach above; in fact, taking the complement of the bits and adding 1 is equivalent to subtracting from 0. Two's complement can be viewed as a way of using the results of overflowing (unsigned) arithmetic to represent signed numbers. \square

Example 1.4

As an explicit example (which can be generalized naturally) take a number, say $38_{10} = 00100110_2 = 2^5 + 2^2 + 2^1$. What should the corresponding negative number be? Well,

$$2^8 - 1 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$$

$$2^8 - 1 = 38 + 2^7 + 2^6 + 2^4 + 2^3 + 2^0$$

$$2^8 - 38 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0 + 1 \quad (\text{flip the bits and add 1})$$

Definition 1.8: MSB

The **most significant bit (MSB)** in a binary sequence is the left-most bit.

Definition 1.9: LSB

The **least significant bit (LSB)** in a binary sequence is the right-most bit.

Exercise: What is the decimal value for 11111111_2 in two's complement notation?

Proof. The answer is -1. □

Exercise: What is the decimal value for 10000000_2 in two's complement notation?

Proof. The answer is -128. □

Exercise: What is the range of numbers expressible in one-byte two's complement notation?

Proof. The answer is -128 to 127 (-2^{8-1} to $2^{8-1} - 1$). □

1.2 ASCII Representation for English Text

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|-------|-----|-----|------|-----|-----|------|
| 0 | 00 | Null | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | SOH | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | STX | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | ETX | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | EOT | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | ENQ | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | ACK | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | BEL | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | BS | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | HT | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | LF | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | VT | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | FF | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | CR | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | SO | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | SI | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | DLE | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | DC1 | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | DC2 | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | DC3 | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | DC4 | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | NAK | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | SYN | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | ETB | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | CAN | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | EM | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | SUB | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | ESC | 59 | 3B | ; | 91 | 5B | [| 123 | 7B | { |
| 28 | 1C | FS | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| 29 | 1D | GS | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| 30 | 1E | RS | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | US | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | □ |

Definition 1.10:

Highlights:

- Characters 0-31 are control characters
- Characters 48-57 are the numbers 0 to 9
- Characters 65-90 are the letters A to Z
- Characters 97-122 are the letters a to z
- Note that 'A' and 'a' are 32 away from each other

Example 1.5

```
#include <stdio.h>

int main(void){
    unsigned char a = 88;           // stored as unsigned binary 01011000
    unsigned char b = 55;           // stored as unsigned binary 00110111

    printf("%c\n", a);              // prints character X (ASCII value 88)
    printf("%d\n", a);              // prints number 88

    printf("%c\n", b);              // prints character 7 (ASCII value 55)
    printf("%d\n", b);              // prints number 55
}
```

Lecture 2 - Thursday, January 09

Exercise: Convert the following into two's complement: -15, 24, -1, and 0xffffffff.

Proof. For -15, we know that

$$15 = 00001111_2$$

hence we negate each bit and add 1 to obtain that

$$-15 = 11110001_2$$

Recall that $-1 = 11111111_2$. □

1.3 Bitwise-Operators

Example 1.6

Suppose we have unsigned char $a = 5, b = 3$,

- Bitwise not \sim , for example $c = \sim a$; gives $c = 11111010$
- Bitwise and $\&$, for example $c = a \& b$; gives $c = 00000001$
- Bitwise or $|$, for example $c = a | b$; gives $c = 00000111$
- Bitwise exclusive or \wedge , for example $c = a \wedge b$; gives $c = 00000110$
- Bitwise shift right or left $>>$ and $<<$, for example
 - $c = a >> 2$; gives $c = 00000001$ and
 - $c = a << 3$; gives $c = 00101000$

Comment 1.1

High-level aside: bitshift ambiguity with signed characters: “arithmetic” left and right shift.

2 Machine Language

Comment 2.1

In CS241, we use a machine language named MIPS (Microprocessor Without Interlocked Pipeline Stages), and we use a 32-bit version of it.

Definition 2.1: Computer Program

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

Comment 2.2

Programs can operate on programs.

Definition 2.2: Instructions

Programs are written as instructions.

2.1 CPU

Definition 2.3: CPU

CPU is the brain of a computer. It contains

- Registers (see more at definition 2.5);
- Control Units, decodes instructions and dispatches
 - PC (program counter);
 - IR (instruction register);
- Memory, MDR (memory data register) and MAR (memory address register)
- ALU (Arithmetic Logic Unit), does arithmetic.

Theorem 2.1

Each register holds 32 bits. Despite the “general purpose” name, some of these registers have special uses dictated by MIPS. In particular, \$0 will always hold the value zero (attempts to modify \$0 are ignored by MIPS CPUs), \$30 will be used as a stack pointer, and \$31 will store return addresses.

Definition 2.4: Bus

Data travels along the **bus**.

Theorem 2.2

Memory of many kinds. From fastest to slowest:

- CPU/registers (fastest)
- L1 cache
- L2 cache
- RAM
- disk
- network memory [outside your local machine] (slowest)

Definition 2.5: Registers

Registers are memory in CPU, there are 32 of them, each of which is 32 bit. Besides, there's also registers **hi** and **lo**. Some general-purpose registers are special:

- \$0 is always 0
- \$31 is for return addresses
- \$30 is our stack pointer
- \$29 is our frame pointer

Definition 2.6: **hi** and **lo**

hi and lo Registers: These are two special registers used by the multiplication and division machine instructions (discussed in Module 4).

2.2 MIPS Instruction Set

Code 2.1

MIPS takes instructions from RAM and attempts to execute them.

Discovery 2.1


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

| Instruction | Opcode | \$s | \$t | \$d | unused | Function | Behavior |
|------------------------|--------|-------|-------|-------|--------|----------|-------------------------------------|
| Add | 000000 | sssss | ttttt | dddd | 00000 | 100000 | \$d = \$s + \$t |
| Subtract | 000000 | sssss | ttttt | dddd | 00000 | 100010 | \$d = \$s - \$t |
| Multiply | 000000 | sssss | ttttt | 00000 | 00000 | 011000 | hi:lo = \$s × \$t |
| Multiply Unsigned | 000000 | sssss | ttttt | 00000 | 00000 | 011001 | hi:lo = \$s × \$t |
| Divide | 000000 | sssss | ttttt | 00000 | 00000 | 011010 | lo = \$s / \$t, hi = \$s % \$t |
| Divide Unsigned | 000000 | sssss | ttttt | 00000 | 00000 | 011011 | lo = \$s / \$t, hi = \$s % \$t |
| Move from High | 000000 | 00000 | 00000 | dddd | 00000 | 010000 | \$d = hi |
| Move from Low | 000000 | 00000 | 00000 | dddd | 00000 | 010010 | \$d = lo |
| Load Immediate & Skip | 000000 | 00000 | 00000 | dddd | 00000 | 010100 | \$d = MEM[PC]; PC+ = 4 |
| Set Less Than | 000000 | sssss | ttttt | dddd | 00000 | 101010 | \$d = 1 if \$s < \$t, 0 otherwise |
| Set Less Than Unsigned | 000000 | sssss | ttttt | dddd | 00000 | 101011 | \$d = 1 if \$s < \$t, 0 otherwise |
| Jump Register | 000000 | sssss | 00000 | 00000 | 00000 | 001000 | PC = \$s |
| Jump & Link Register | 000000 | sssss | 00000 | 00000 | 00000 | 001001 | temp = \$s; \$31 = PC; PC = temp |


Table 1: MIPS Instruction Set

| Instruction | Opcode | \$s | \$t | i | Behaviour |
|---------------------|--------|-------|-------|---------------------|-----------------------------|
| Branch On Equal | 000100 | sssss | ttttt | iiii iiii iiii iiii | if (\$s == \$t) PC += 4 * i |
| Branch On Not Equal | 000101 | sssss | ttttt | iiii iiii iiii iiii | if (\$s != \$t) PC += 4 * i |
| Load Word | 100011 | sssss | ttttt | iiii iiii iiii iiii | \$t = MEM[\$s + i] |
| Store Word | 101011 | sssss | ttttt | iiii iiii iiii iiii | MEM[\$s + i] = \$t |


Table 2: Immediate Format Instruction Set

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

Question 2.1. How does MIPS know what to do next?

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

Question 2.2. 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. 

2.3 Fetch-Execute Cycle

Code 2.2

Algorithm 1: Algorithm 2: Fetch-Execute Cycle

```

1 PC ← 0;
2 while true do
3   IR ← MEM[PC];
4   PC ← PC + 4;
5   Decode and execute instruction in IR;
```

Comment 2.3

Abstraction!

Example 2.1

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

```
00000001000010010001100000100000
```

Example 2.2

Write a MIPS program that adds the values 11 and 13 and stores the result in register 3

```
0000000000000000000010000000010100
000000000000000000000000000001011
00000000000000000000100100000010100
000000000000000000000000000001101
00000001000010010001100000100000
```

2.3.1 Halt the Fetch-execute Cycle

Theorem 2.3

To halt the fetch-execute cycle, recall that operating system provides a return address in register \$31, so we simply enter

```
00000011111000000000000000001000
```

Lecture 3 - Tuesday, January 14

2.4 Multiplication and division

2.4.1 Multiplication

Multiply

```
000000sssstttt0000000000011000
```

Performs the multiplication and places the most significant word (largest 4 bytes) in hi and the least significant word in lo.

2.4.2 Division

Divide

```
000000sssstttt0000000000011010
```

Performs integer division and places the quotient $\$s / \t in lo and the remainder $\$s \% \t in hi. Note the sign of the remainder matches the sign of the divisor stored in \$s.

2.5 RAM

Definition 2.7:

Large[r] amount of memory stored off the CPU. RAM access is slower than register access (but is larger, as a tradeoff).

Comment 2.4

Instructions occur in RAM starting with address 0 and increase by the word size (in our case 4).

2.6 Assembly Language

Definition 2.8: Assembly Language

An **assembly language** is a text language in which there is a 1-to-1 correspondence between assembly instructions and machine code instructions.

Example 2.3

For instance, this machine language fragment:

```
0000000000000000100000000010100
00000000000000000000000000001011
0000000000000000100100000010100
00000000000000000000000000001101
00000001000010010001100000100000
```

In assembly:

```
lis $8
.word 11
lis $9
.word 13
add $3, $8, $9
```

Definition 2.9: Compiler

A **compiler** convert from human-readable(-ish) assembly into machine code.

Comment 2.5

A compiler for assembly is also called an “assembler”.

3 Scanning and Regular Languages

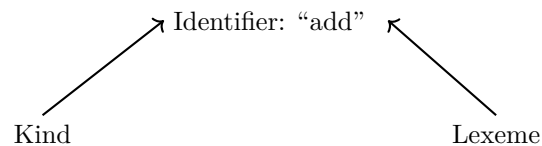
Discovery 3.1

A string like “add \$3, \$2, \$1” is stored as a sequence of characters: ‘a’ ‘d’ ‘d’ ‘ ’ ‘\$’ ‘3’ ‘,’ ‘ ’ ‘\$’ ‘2’ ‘,’ ... If we could break down the string into meaningful chunks of information, it would be easier to translate:

[add] [\$3] [\$2] [\$1]

Definition 3.1: Scanner (Tokenizer)

A **scanner** (or **tokenizer**) breaks down a string into tokens. Tokens are the “words” of the language of assembly.



3.1 Maximal Munch Scanning

Theorem 3.1: Maximal Munch

Always choose the longest meaningful token while breaking up the string.

3.2 Formal Languages

Definition 3.2: Alphabet

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

Example 3.1

Here are some examples of alphabets:

- $\Sigma = \{a, b, \dots, z\}$, the Latin alphabet;
- $\Sigma = \{0, 1\}$, the alphabet of binary digits;
- $\Sigma = \{0, 1, \dots, 9\}$, the alphabet of base 10 digits.

Definition 3.3: String

A **string** (or word), w , is a finite sequence of symbols chosen from Σ . The set of all strings over an alphabet Σ is denoted by Σ^* .

Example 3.2

Here are some examples of strings:

- ε is the empty string. Note that $\varepsilon \in \Sigma^*$ for all Σ ;
- For $\Sigma = \{0, 1\}$, a string might be 011101 or 1111.

Definition 3.4: Language

A **language** is a set of strings.

Example 3.3

Here are some examples of Languages:

- $L = \emptyset$;
- $L = \{\varepsilon\}$;
- $L = \{ab^n a : n \in \mathbb{N}\}$, a particular set of strings over alphabet $\Sigma = \{a, b\}$.

Definition 3.5: Length

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

Lecture 4 - Thursday, January 16

We wish to check if a string is in a language.

3.2.1 Membership in Languages

Theorem 3.2

In order of relative difficulty:

- | | |
|------------------|-------------------------|
| 1. finite; | 4. context-sensitive; |
| 2. regular; | 5. recursive; |
| 3. context-free; | 6. impossible language. |

3.3 Regular Expressions & Regular Languages

Definition 3.6: Regular

A language is regular if:

1. It is an empty set. The empty set is denoted $\{\}$ in this course (\emptyset is another common notation).
2. It is a set containing only the empty string, i.e., $\{''\}$.
3. It is a set containing only one string, and this string has only one character.
4. It is the union of two regular languages.
5. It is formed from two regular languages by (pairwise) concatenation.
6. It is formed by applying the Kleene star operator to a regular language.

3.3.1 Union

Definition 3.7: Union

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

3.3.2 Concatenation

Definition 3.8: Concatenation

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


3.3.3 Kleene Star

Definition 3.9: Kleene Star

If L is a regular language, then $L^* = \bigcup_{n=0}^{\infty} L^n$ is regular, where

$$L^n = \begin{cases} \{''\}, & n = 0 \\ LL^{n-1}, & \text{otherwise} \end{cases}$$

Question 3.1. What does $L \cdot \{\}$ equal?

Solution: It is the empty language, $\{\}$. Recall the definition of concatenation. In this example, L_2 is the empty language, and the concatenation of any language to the empty language will produce an empty language. 

Question 3.2. ab...ba

Consider strings of the following form:

$$\{ "aa", "aba", "abba", "abbba", "abbbba", "abbbbba", \dots \}$$

Explain why this set of strings is a regular language.

Solution: $\{ 'a' \}$ is regular. $\{ 'b' \}^*$ is also regular since $\{ 'b' \}$ is regular and the Kleene star of a regular language is a regular language. Therefore, the concatenation

$$\{ 'a' \} \cdot \{ 'b' \}^* \cdot \{ 'a' \}$$

must also be regular.



3.3.4 Notation

Result 3.1

- The curly braces $\{ \dots \}$ used in set notation are dropped.
- Union is often written with a vertical bar $|$ instead of the union symbol \cup .
- Concatenation is usually not written using a centered dot \cdot and instead expressions are just placed beside each other to concatenate them.
- In computer implementations, Kleene star typically uses the asterisk $*$ character.

Theorem 3.3: order of operations

The order of operations is: Kleene star first, then concatenation, then union. Parentheses (...) can be used to group expressions, overriding the order of operations.

3.4 Recognizing Regular Languages

Definition 3.10: Recognition Problem

The **recognition problem** for a language is to determine, given a string as input, whether the string belongs to the language.

3.4.1 From Regular Expressions to Programs

Our goal in this section is to show that for every possible regular expression, we can write a program that solves the recognition problem for the corresponding regular language.

Case 1 (Empty Set) Our recognition program just rejects all inputs.

Case 2 (Set Containing Only The Empty String) Our recognition program rejects all non-empty inputs, and accepts the empty string.

Case 3 (Set Containing A Single-Character String) Our recognition program rejects all inputs that are not single-character strings. For single-character strings, compare the input against the expected character, and accept or reject accordingly.

Case 4 (Union)

Algorithm 2: Recognition program for regular expression $R \mid S$ (union)

Input: String x

- 1 Run recognition program for R on x ;
 - 2 Run recognition program for S on x ;
 - 3 **return** *True* if either program accepted x , *False* if both rejected x ;
-

Case 5 (Concatenation)

Algorithm 3: Recognition program for regular expression RS (concatenation)

Input: String x

- 1 **for** each pair of strings (y, z) such that $x = yz$ **do**
 - 2 Run recognition program for R on y ;
 - 3 Run recognition program for S on z ;
 - 4 **return** *True* if both programs accept
 - 5 **return** *False*
-

Case 6 (Kleene Star)

Algorithm 4: Recognition program for regular expression R^* (Kleene star)

Input: String x of length n

- 1 **if** x is empty **then**
 - 2 **return** *True*;
 - 3 **else**
 - 4 **for** $i = 1$ **to** n **do**
 - 5 **for** each i -tuple of strings (x_1, \dots, x_i) such that $x = x_1 \dots x_i$ **do**
 - 6 **for** $j = 1$ **to** i **do**
 - 7 Run recognition program for R on x_j ;
 - 8 **return** *True* if x_1, \dots, x_i were all accepted;
 - 9 **return** *False*;
-

Result 3.2

We will now give up on trying to implement regular expressions. Instead, we will look at an entirely

different method of specifying regular languages.

3.4.2 Deterministic Finite Automata

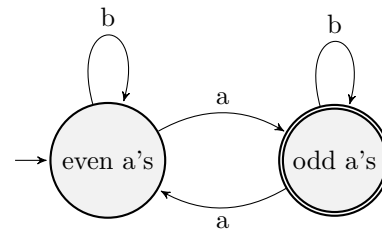
Definition 3.11: DFA

A DFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ is a finite set of 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
- $\delta : [Q \times \Sigma] \rightarrow Q$ is our [total] transition function (given a state and a symbol of our alphabet, what state we should go to)

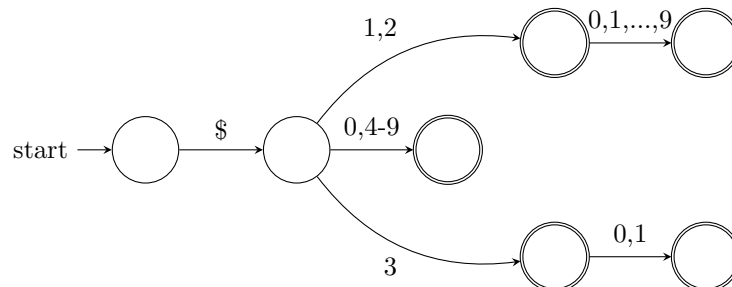
Example 3.4

This DFA describes a recognition program for strings which contain an odd number of a 's and an arbitrary number of b 's.

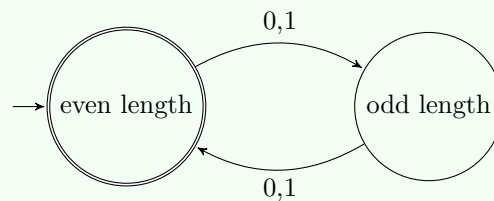


Example 3.5

This DFA describes a recognition program for strings that represent valid general-purpose registers in MIPS, ranging from 0 to 31, where useless leading zeroes are not allowed.



Question 3.3. DFA describing a recog program for the language of bin sequences of even length.



3.4.3 Generic DFA Recognition Algorithm

Algorithm 3.1

Algorithm 5: DFA Recognition

Input: Description of a DFA D , string x

```
1  $p \leftarrow$  initial state of  $D$ ;  
2 for each character  $a$  in  $x$  do  
3   if  $D$  has an arrow from  $p$  to  $q$  labelled with  $a$  then  
4      $p \leftarrow q$  ;                               // follow the arrow to the new state  
5   else  
6     return False ;                               // when there is no arrow for the current state and  
                                           character combination  
7 return True if  $p$  is an accepting state, False otherwise
```

3.5 Maximal Munch Scanning: The Details

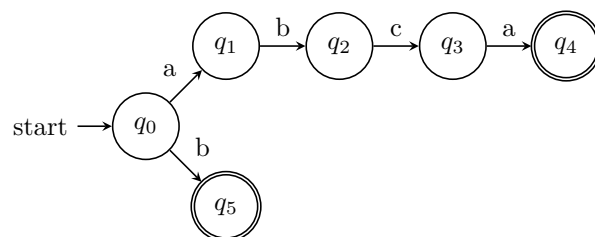
Recall our intuitive description of maximal munch from earlier. The set T is our set of valid token lexemes, and x is the string we want to scan.

Algorithm 3.2

- Find the longest prefix of x that is in T . If no prefix of x belongs to T , halt with an error (scanning failed).
- If a prefix was found, remove it from the front of x and generate a token corresponding to this prefix.
- Repeat the above steps until either an error occurs, or x becomes empty.

Theorem 3.4

In the case where we get stuck in a non-accepting state, we know that the prefix we have read is not a valid token, and cannot be extended to a valid token. The longest prefix of x that is in T actually occurred last time we saw an accepting state. If we did not pass through any accepting states before getting stuck, this means no prefix of x belongs to T , which is the case where we give up and return a scanning failure. If we did pass through an accepting state along the way, we actually need to **backtrack** to that accepting state in **both the input and the DFA**.



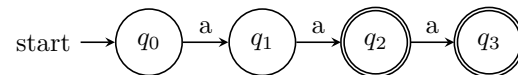
Question 3.4. Backtracking

Consider the above DFA for T . What would maximal munch output for the input string $x = \text{"baba"}$ and $T = \{ \text{'a'}, \text{'b'}, \text{'abca'} \}$?

Solution: The algorithm would output a 'b' token, followed by an 'a' token, then another 'b' token and finally another 'a' token. 🎵

Discovery 3.2

Maximal munch does not always find a tokenization, even if one exists. Consider the language of token lexemes $T = \{ \text{'aa'}, \text{'aaa'} \}$ and the corresponding DFA:



Now consider the input $x = \text{'aaaa'}$.

Result 3.3

The maximal munch algorithm, as presented above, has quadratic time complexity due to backtracking. A more complex implementation that uses memoization and dynamic programming is available that gives linear time complexity.

Question 3.5. Quadratic Time Complexity

Can you think of a language and input that would demonstrate this quadratic time complexity?

3.5.1 Simplified Maximal Munch

Definition 3.12: Simplified Maximal Munch

The Maximal Munch algorithm often still works well in practice even with the backtracking eliminated. We call this version of the algorithm **simplified maximal munch**.

Question 3.6. I

Is it possible that a string can be scanned into tokens, but both simplified maximal munch and ordinary maximal munch reject the string?

Solution: Maximal Munch is stronger than Simplified Maximal Munch, so we still consider the example in the above discovery (3.2). 🎵

4 MIPS Assemblers & Assembly Language

4.1 Writing an Assembler: First Attempt

Discovery 4.1

For our first MIPS assembler, we'll make a simplifying assumption: *every line consists only of a single assembly instruction or directive.*

Code 4.1

We will remove this simplification in the second version of our assembler.

Theorem 4.1

An assembler must make sense of each line in the assembly language program and determine that there are no syntax errors or violations of assembly language rules. This is the **Analysis** phase. Once the assembler has determined that the input is a legal program, the **Synthesis** phase generates the output, the machine language equivalent of the program.

Definition 4.1: Analysis

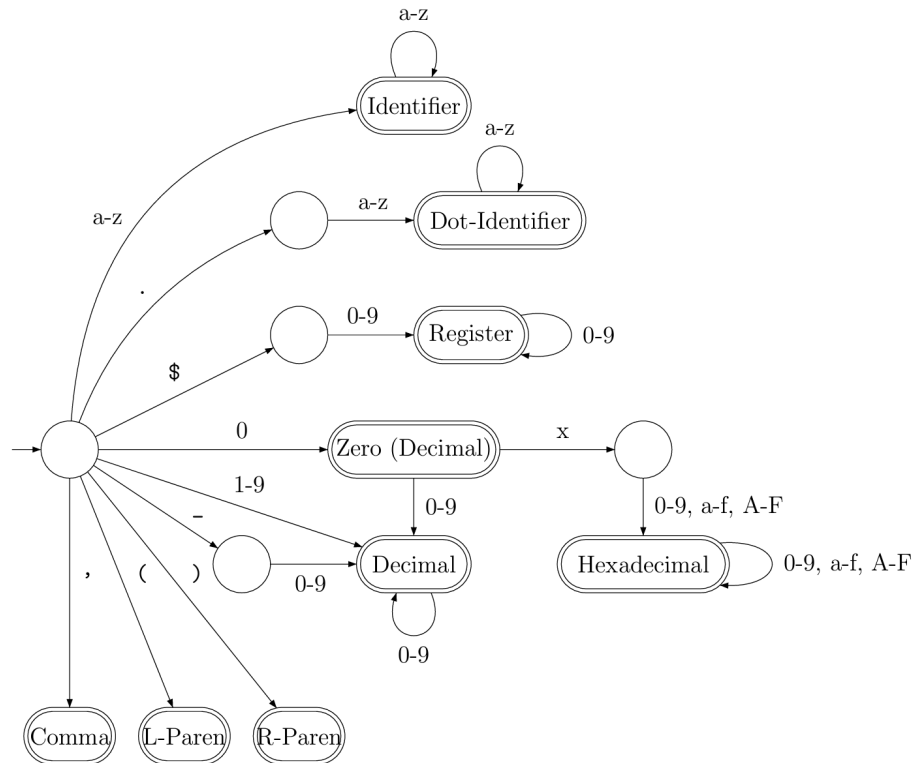
The Analysis phase itself is a multi-step process which includes:

- **Scanning:** Splitting each line into tokens.
- **Parsing:** Processing each line tokens to check for syntax errors.
- **Semantic Analysis:** Catching errors related to semantics (meaning) as opposed to syntax.

4.1.1 Scanning

The goal of this step is to turn a string of individual ASCII characters like “add \$1, \$2, \$3” into a sequence of tokens:

```
[Identifier, "add"] [Register, "$1"] [Comma, ","] [Register, "$2"] [Comma, ","] [Register, "$3"]
```



Discovery 4.2

The process of combining the DFAs is mostly straightforward, but there is one tricky part. Notice that if we are in the initial state and read the character 0, this could either represent the decimal integer 0, or the start of the sequence "0x" indicating a hexadecimal integer.

Question 4.1. See above discovery

Solution: We deal with this by having a separate state for "Zero". The scanner will still output a Decimal token for this "Zero" state, but it is separate from the main Decimal state. 🎵

4.1.2 Parsing

Theorem 4.2

In compilation, parsing has two purposes: to check for syntax errors, and to gather information about the structure of the program.

Comment 4.1

- For high-level programming languages, parsing is a very complex process that we will spend multiple modules discussing.

- For MIPS assembly, it is fairly straightforward, particularly with our simplifying assumption that each every line consists only of a single assembly instruction or directive. The first token on a line therefore must be an Identifier or Dot-Identifier; otherwise it is a syntax error.

4.1.3 Semantics Analysis

Parsing typically only catches some errors related to syntax (structure) rather than semantics (meaning). Semantic Analysis is used to catch any remaining errors not caught by parsing, and sometimes to compute additional information about the program that requires a broader context than what is available during parsing.

Comment 4.2

In MIPS assembly language, the only errors we need to detect in the Semantic Analysis step are related to **labels**, an assembly language feature we have not introduced yet. So right now, we will ignore this step.

4.1.4 Synthesis: Encoding, Output, & Bitwise Operations

Once the Analysis stage is done, and the input program is known to be correct, the Synthesis stage generates the machine code equivalent for each assembly instruction. There are two steps: encoding the instruction (that is, constructing the appropriate sequence of 32 bits) and writing the encoding to standard output.

Example 4.1

Let's consider how to generate an encoding for `add $3, $2, $4`. From the table: `add $3, $2, $4` should produce the following (components are highlighted):

$\underbrace{000000}_{opcode} \underbrace{00010}_{\$s} \underbrace{00100}_{\$4} \underbrace{00011}_{\$d} \underbrace{00000}_{padding} \underbrace{100000}_{func}$

What does this binary sequence correspond to as a decimal number? The answer is:

$$2^{22} + 2^{18} + 2^{12} + 2^{11} + 2^5 = 4462624$$

Notice we can rewrite this number as $(2 \cdot 2^{21}) + (4 \cdot 2^{16}) + (3 \cdot 2^{11}) + 2^5$.

4.1.5 Bitwise Operations

Bitwise AND (&) : Computes the value given by applying the logical AND operation to each pair of bits in the operands.

Bitwise OR (|) : Computes the value given by applying the logical OR operation to each pair of bits in the operands.

Left Bit Shift (<<) : The left bit shift operator, $x \ll n$, is equivalent to $x \cdot 2^n$.

Right Bit Shift (>>) : There are actually two kinds of right bit shift: arithmetic right shift and logical right shift. Logical right shift, just like left bit shift, pads the number with 0 bits on the left.

Discovery 4.3

Notice that for a negative two's complement number, this will actually change the sign to positive!


Result 4.1

For unsigned values, it works the same as logical right shift. But for signed values, if the most significant bit (leftmost bit) is 0, it pads with 0 bits, and if it is 1, it pads with 1 bits. Mathematically, this is the same as integer division by 2^n .

Bitwise NOT and Bitwise XOR : Two other commonly provided bitwise operations are bitwise NOT, denoted by $\sim x$, which applies bitwise NOT to each the bit of a value, and bitwise XOR, denoted $x \wedge y$, which applies logical XOR (exclusive or) to each pair of bits in two values.

Question 4.2.

If x is a two's complement number explain why $-x = \sim x + 1$

Solution: Recall that to negate a two's complement number, you “flip the bits and add 1”. Bitwise NOT is the same as flipping all the bits in a binary value! 

Question 4.3.

Suppose you have a binary sequence that represents an array of boolean values. Each bit is either 1 (“true”) or 0 (“false”). The least significant bit is index 0 of the array, the bit to the left is index 1, and so on. Write an expression using bitwise operations that sets index i of the array A to the boolean value 1 (“true”).


Solution:

$$A = A \mid (1 \ll i)$$

The bit 1 is shifted left to position i , then bitwise OR is used to overwrite the bit at position i in A . 

Question 4.4. What is printed to the screen?

```
#include <stdio.h>
int main(void) {
    unsigned char c = 1;
    unsigned char d = 25;
    c <<= 3;
    c |= 1;
    c = c & d;
    printf("%d", c);
    return 0; }
```

Solution: Answer is 9. Line 5 updates the value for `c` to be the bit string 1000. After line 6, `c` has the value 1001. Since `d` is 11001, after line 7, `c` is 1001. Line 8 prints this value. 

4.1.6 Encoding Instructions with Bitwise Operations

Let's return to the problem of encoding `add $3, $2, $4`:



Assuming that the assembler has readied the following int values: `s`, `t` and `d` with the corresponding register values, `op` (short for Opcode) with the value 0, and `func` (short for Function) with the value 32 (100000 in binary). In C++, int values are stored as 32-bit values in two's complement representation.

Value Decimal 32-bit int in binary (space for emphasis)

```
op:      0      000000000000000000000000 000000
s:       2      000000000000000000000000 00010
t:       4      000000000000000000000000 00100
d:       3      000000000000000000000000 00011
func:    32     000000000000000000000000 100000
```

Rather than multiplying by specific powers of two, a more intuitive method is to **use left bit shifts** to put these values in the correct positions.

Comment 4.3

In C++, we can use the `<<` operator, and in Racket the `arithmetic-shift` function:

```
op:      (arithmetic-shift 0 26)  0 << 26  000000 00000 00000 00000 000000
s:       (arithmetic-shift 2 21)  2 << 21  000000 00010 00000 00000 000000
t:       (arithmetic-shift 4 16)  4 << 16  000000 00000 00100 00000 000000
d:       (arithmetic-shift 3 11)  3 << 11  000000 00000 00000 00011 000000
func:    32 32 000000 00000 00000 00000 100000
```

Once the useful bits are in the right positions, we can use bitwise OR to combine the values.

```
int instr = (0 << 26) | (2 << 21) | (4 << 16) | (3 << 11) | 32;
(define instr (bitwise-ior (arithmetic-shift 0 26)
                           (arithmetic-shift 2 21)
                           (arithmetic-shift 4 16)
                           (arithmetic-shift 3 11)
                           32 ))
```

4.1.7 Producing Output

In C, there is a function called `putchar` that outputs a single byte to standard output, which is also available in C++ as `std::putchar` through the `<cstdio>` header. If you prefer to use stream-style I/O with `std::cout`, there is a simple solution: if you output a value that is stored in a `char` variable, then C++ will output the correspond byte directly, rather than formatting it as a decimal integer. In Racket, the `write-byte` function can be used to output a byte.

Discovery 4.4

There is still one problem. Our `instr` variable stores four bytes, and we want to output all four bytes, not just one byte. To get the rightmost byte, we could use our bit-masking trick with the value `0xFF`.

Code 4.2

What about the other bytes? This is where right bit shift comes in handy. For example, to get the second byte from the left, we can do a right shift by 16 before masking

4.2 Advanced MIPS Assembly Programming

4.2.1 More Arithmetic: Multiplication & Division

| Instruction | Assembly | Behaviour |
|-------------------|-----------------------------|---|
| Multiply | <code>mult \$s, \$t</code> | <code>hi:lo = \$s * \$t</code> (signed) |
| Multiply Unsigned | <code>multu \$s, \$t</code> | <code>hi:lo = \$s * \$t</code> (unsigned) |

Code 4.3

The two multiplication instructions only differ in behaviour as follows: `mult` interprets its operands as signed two's complement integers, and `multu` interprets its operands as unsigned integers.

Theorem 4.3

To avoid losing the most significant 32 bits of this 64-bit result, MIPS splits the result across the special registers `hi` and `lo`. The most significant 32 bits of the result are placed in `hi` and the least significant 32 bits are placed in `lo`.

Question 4.5.

Find two values for `$s` and `$t` such that `mult` and `multu` place different values in `hi`.

Solution: An example is `$s = 0xFFFFFFFF` (-1 in two's complement, $2^{32} - 1$ in unsigned) and `$t = 1`. 🎵

Question 4.6.

Is it possible to find two values for `$s` and `$t` such that `mult` and `multu` place different values in `lo`?

Solution: It is not possible. Recall that if you ignore overflowing bits, addition and subtraction behave the same way for two's complement and unsigned values. The same is true for multiplication! The only reason we need separate `mult` and `multu` instructions is because we do not ignore overflowing bits—we place overflowing bits in `hi`. 🎵

| Instruction | Assembly | Behaviour |
|-----------------|----------------------------|--|
| Divide | <code>div \$s, \$t</code> | <code>lo = \$s / \$t; hi = \$s % \$t</code> (signed) |
| Divide Unsigned | <code>divu \$s, \$t</code> | <code>lo = \$s / \$t; hi = \$s % \$t</code> (unsigned) |

Theorem 4.4

The division instructions also use hi and lo, but in a completely different way. When a division is performed, lo stores the quotient and hi stores the remainder.

Result 4.2

Formally, the quotient q and remainder r produced by a MIPS division can be defined as the unique solutions to the following equation:

$$\$s = q \cdot \$t + r \text{ where } |q \cdot \$t| \leq |\$s| \text{ and } |r| < |\$t|$$

In other words, when you divide $\$s/\t , find the largest (in absolute value) multiple of $\$t$ that fits into $\$s$. Whatever's left over is the remainder. This implies the remainder must have the same sign as $\$s$.

4.2.2 Less-Than Comparison

| Instruction | Assembly | Behaviour |
|------------------------|---------------------------------|---|
| Set Less Than | <code>slt \$d, \$s, \$t</code> | $\$d = 1$ if $\$s < \t ; 0 otherwise (signed) |
| Set Less Than Unsigned | <code>sltu \$d, \$s, \$t</code> | $\$d = 1$ if $\$s < \t , 0 otherwise (unsigned) |

Example 4.2

| | |
|---|---|
| <pre>slt \$3, \$1, \$2 sltu \$4, \$1, \$2 jr \$31</pre> | <p>Suppose the following program is assembled and executed on the MIPS machine with initial register values $\\$1 = 0xFFFFFFFF$ and $\\$2 = 0x7FFFFFFF$. What values are placed in $\\$3$ and $\\$4$?</p> |
|---|---|

Solution: The value in $\$2$ ($0x7FFFFFFF$ or $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$ in binary) represents the number $2^{31} - 1$ in both the two's complement and unsigned integer representations. However, the value in $\$1$ ($0xFFFFFFFF$, or $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$) represents -1 in two's complement and $2^{32} - 1$ in unsigned. Therefore, $\$3 = 1$ since $-1 < 2^{31} - 1$ is true. But $\$4 = 0$ since $2^{32} - 1 < 2^{31} - 1$ is false! 🎵

4.2.3 Conditional Branching & Loops

| Instruction | Assembly | Behaviour |
|---------------------|------------------------------|-------------------------------------|
| Branch on Equal | <code>beq \$s, \$t, i</code> | if $(\$s == \$t)$ PC += $i \cdot 4$ |
| Branch on Not Equal | <code>bne \$s, \$t, i</code> | if $(\$s != \$t)$ PC += $i \cdot 4$ |

Example 4.3

Calculate the value $13 + 12 + 11 + 10 + \dots + 1$ and store it in $\$3$.

Solution: One of the approaches would be to use a loop like the one shown in the pseudocode below:

```
$2 = 13
$3 = 0
repeat {
```



```

    $3 += $2
    --$2
} until ($2 == 0)

```

Here is a MIPS assembly version of the pseudocode above.

```

lis $2
.word 13      ; $2 = 13
add $3, $0, $0 ; initialize $3 to 0
add $3, $3, $2 ; loop starts here; $3 += $2
lis $1
.word -1      ; $1 = -1
add $2, $2, $1 ; --$2
bne $2, $0, -5 ; if $2 != 0, go back 5 words (remember PC is already at the next word!)
jr $31        ; exit

```



Discovery 4.5

An interesting observation at this point is that this is not the most efficient program we could write. In particular, we load -1 into $\$1$ inside the loop! Since $\$1$ is unchanged in each iteration, there is no reason for this to be in the loop. Let's pull the "load into $\$1$ " out of the loop, to the top of the program. However, if we do this, we must also make another change.

Question 4.7.

Can you think of what other change we might need to make if we are moving the two lines that load the value -1 into $\$1$ above the loop?

Question 4.8. Warm-up 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 $\$2$.

Solution: We have

```

lis $10
.word 0xa
div &1, &10
mfhi $2
jr $31

```



Question 4.9. Extended Exercise

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

Solution: We have

```
add $2, $1, $0
slt $5, $1, $0
beq $5, $0, 1
sub $2, $0, $1
jr $31
```



4.2.4 Labels

Code 4.4

One place where labels can be used is in branch instructions. The way they work is quite intuitive:

```
lis $2
.word 13          ; $2 = 13
lis $1
.word -1          ; $1 = -1
add $3, $0, $0    ; initialize $3 to 0
loop:             ; loop label is defined here
add $3, $3, $2    ; $3 += $2
add $2, $2, $1    ; --$2
bne $2, $0, loop  ; if $2 != 0, go back to the loop label!
jr $31            ; exit
```

Definition 4.2: Label definition

Notice that at the location where we want the loop to start, we wrote the text “loop:”. This is called a **label definition**; it consists of the name of the label followed by a colon.

In the bne instruction, the branch offset (the *i* value) has been replaced by loop. If the branch condition is true, the branch instruction will jump back to the location where the loop label was defined.

Theorem 4.5

Labels can also be used with .word directives. In this case, .word label will be equal to the memory address where the label is defined.

Example 4.4

Consider the following program:

```
lis $1
.word 241
lis $3
.word secretStash
sw $1, 0($3)
jr $31
secretStash: .word 240
```

Comment 4.4

The program loads the 32-bit encoding of 241 into \$1, and then loads the value of `.word secretStash` into \$3. The value loaded into \$3 is actually the memory address of the “240” word. For example, if this program was loaded at address zero, it would look like this in memory:

| Assembly Source | Address | Actual Contents of Memory |
|------------------------|---------|--|
| lis \$1 | 0x00 | 00000000 00000000 00000000 00010000 (0x00000814) |
| .word 241 | 0x04 | 00000000 00000000 00000000 11110001 (0x000000F1) |
| lis \$3 | 0x08 | 00000000 00000000 00010000 00010000 (0x00001814) |
| .word secretStash | 0x0C | 00000000 00000000 00010000 00000000 (0x00000018) |
| sw \$1, 0(\$3) | 0x10 | 10101100 01100001 00000000 00000000 (0xac610000) |
| jr \$31 | 0x14 | 00000011 11000000 00000000 00001000 (0x03e00008) |
| secretStash: .word 240 | 0x18 | 00000000 00000000 00000000 11110000 (0x000000F0) |

Notice that `.word secretStash` was encoded as 0x00000018, or in other words, 0x18, the memory address corresponding to `.word 240`, where the `secretStash` label was defined in the original source code.

Result 4.3

The result of this program is that the word at 0x18, which initially contains 240 (0xF0), will be overwritten with 241 (0xF1) by the `sw` (Store Word) instruction.

Question 4.10.

Consider this MIPS assembly code:


```
endless: beq $0, $0, endless
        beq $0, $0, end
        end:
```

What would an equivalent program that doesn't use labels look like?

Solution: An equivalent program would be:

```
beq $0, $0, -1
```

```
beq $0, $0, 0
```

This might be unintuitive given the position of the labels! Remember, PC is already at the next word when a branch happens. 

Question 4.11.

Write a MIPS assembly language program that stores into register \$3 the sum of all even numbers from 1 to 20 inclusive. Use labels instead of hardcoding offsets.

Solution: Here is one possible solution:

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



4.3 Writing an Assembler with Label Support

Comment 4.5

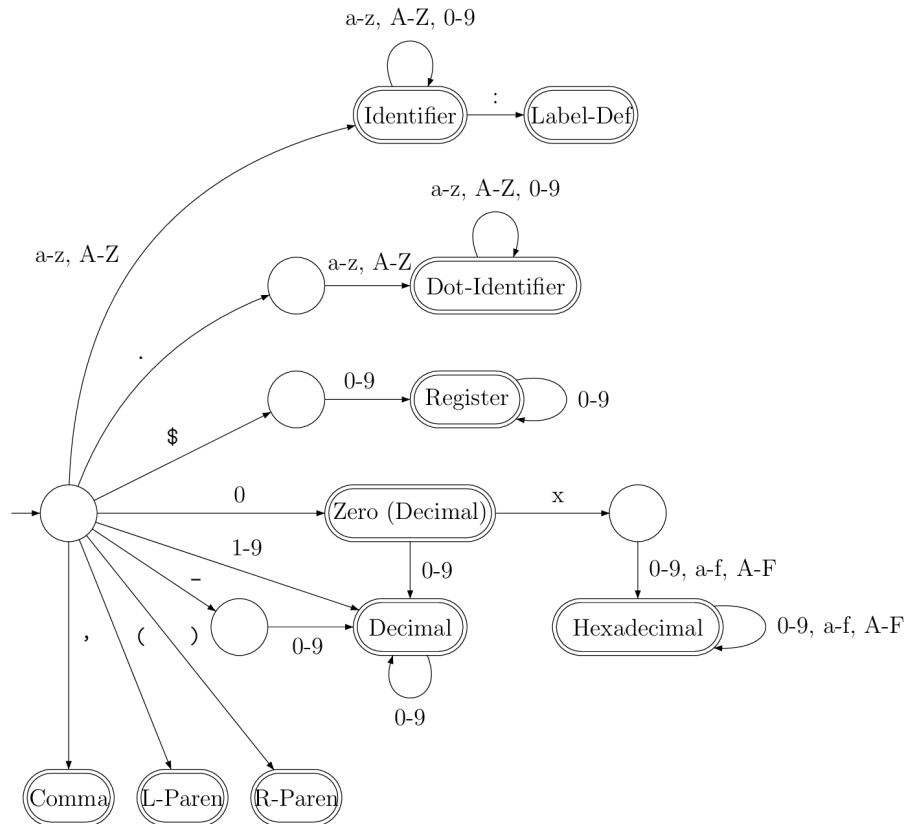
The fundamental problem we need to solve to support labels is that of forward references. In cases where we want to branch forward in the program, the label is used as an operand to the branch instruction before it has been defined. If we read through the program once, in the usual forward order, we won't know how far the branch instruction should branch—because we haven't seen the label it's branching to yet! We also have cases where labels are defined before they are used (when branching backwards, like in a loop) so reading the program backwards isn't a solution.

Result 4.4

The solution is to implement the assembler in two passes. On the first pass, we read the input and perform scanning and parsing, and gather information about labels. The parsed lines can be stored in a data structure, which we loop over for the second pass. The label information we gathered in the first pass is used to perform semantic analysis and synthesis.

4.3.1 The First Pass: Scanning Changes

We need to update our scanning DFA a little.



4.3.2 The First Pass: Parsing and the Symbol Table

We would like to be more flexible and drop the assumption that every line consists only of one instruction or directive. The general format of a line in MIPS assembly has three components, all of which are optional, but they must appear in the order below.

[labels] [instruction] [comment]

- The [labels] component consists of one or more label definitions.
- The [instruction] component consists of an instruction (or `.word` directive).
- The [comment] component consists of a comment, starting with a semicolon and running to the end of the line.

Comment 4.6

Since all three components are optional, a line can be completely blank, it could have only one component, it could have a mix of two, or it could have all three.

However, the labels component must appear before the instruction component, i.e., you cannot have labels after an instruction on the same line.

Discovery 4.6

And of course, if a comment appears, everything else on the line is part of the comment, so it is not possible to have other components appear after a comment.

4.3.3 The Second Pass: Semantic Analysis and Synthesis


Definition 4.3: Semantic Analysis

In a compiler, the purpose of **semantic analysis** is to check for errors in semantics or meaning, as opposed to the syntax or structural errors caught during parsing.

Code 4.5

There are two kinds of semantic errors related to labels:

- Duplicate label definitions, which are most easily caught during parsing.
- Use of an undefined label, which is most easily caught while converting labels to numeric values for synthesis.

Solution: To encode an instruction in the synthesis phase, we need to look up the label's address in the symbol table. If this lookup fails, we produce an **undefined label error**. 

4.4 Input & Output in MIPS

Definition 4.4: Input in MIPS

Using `lw $t, i($s)` to load from address `0xffff0004` will consume a byte from standard input and store it in the lower (rightmost) 8 bits of register `$t`. If a byte is successfully read, the upper 24 bits are filled with zeroes.

Definition 4.5: Output in MIPS

Using `sw $t, i($s)` to store to address `0xffff000c` will write the lower (rightmost) 8 bits of register `$t` to standard output. This is effectively the same as the C/C++ function `putchar`.

Example 4.5

Assume standard input contains n characters, and `$2` contains the value of n . Write a program that reads all characters from standard input, then prints the characters to standard output twice. For example, if standard input contains “cat”, then `$2` will contain 3, and the program should output “catcat”.

Solution: We cannot read input twice in MIPS, so to print the characters twice, we need to store them in memory as we read them. First we set up constants and allocate an array of size n on the stack.

```

lis $20
.word 0xffff000c    ; output address
lis $21
.word 0xffff0004    ; input address
lis $11
.word -1            ; end of input
lis $4
.word 4
mult $2, $4
mflo $5              ; $5 contains 4 * n
sub $30, $30, $5
add $1, $30, $0      ; $1 contains initial address of array


```

Next, we read input in a loop. As we read each character, we write it to standard output, but also store it in our array on the stack. After this, we do a second loop over our stack-allocated array to print the contents.

```

add $10, $1, $0      ; make a backup copy of $1 in $10
loop:
    lw $3, 0($21)     ; read from stdin into $3
    beq $3, $11, end   ; end the loop when end of input is reached
    sw $3, 0($20)     ; write the character just read to stdout
    sw $3, 0($1)       ; store character in A[i]
    add $1, $1, $4     ; $1 = address of A[i+1]
    beq $0, $0, loop   ; back to top of loop
end: add $1, $10, $0    ; restore original $1 from the backup
loop2:
    beq $2, $0, end2   ; end the loop when n = 0
    lw $3, 0($1)       ; load character from A[i] into $3
    sw $3, 0($20)     ; write the character just loaded to stdout
    add $1, $1, $4     ; $1 = address of A[i+1]
    add $2, $2, $11     ; n-= 1 (remember $11 is -1)
    beq $0, $0, loop2  ; back to top of loop
end2:
add $30, $30, $5      ; deallocate array
jr $31

```

Note that before returning, we deallocate the array by incrementing the stack pointer (we add the value $4 \cdot n$, which we stored in \$5 earlier). 

4.5 Arrays in MIPS Assembly

Since our MIPS instruction set can only load and store entire words at a time, we will focus on arrays where each element is a word, 4 bytes.

Example 4.6

\$1 contains the starting address of an array of 32-bit integers, `arr`, stored in memory. \$2 contains the number of elements in `arr`. Retrieve `arr[3]` into \$3.

Solution: Once we have understood how the array is situated in memory and how offsets into the array are calculated, writing the solution is straightforward:

```
lis $5
.word 3          ; index of element to retrieve
lis $4
.word 4
mult $4, $5      ; multiply index by 4 to compute offset
mflo $5          ; retrieve result of multiplication from lo register
add $5, $5, $1   ; starting address of element at index 3
lw $3, 0($5)     ; $3 = MEM[$5 + 0]
jr $31
```

While the program above solves the question, it is not the simplest. We presented it to demonstrate using multiplication to compute offsets. A simpler solution is:

```
lw $3, 12($1)    ; $3 = MEM[$1 + 12]
jr $31
```



Example 4.7

\$1 contains the starting address of an array and \$2 contains the number of elements. Write a MIPS program that sets each value in the array to zero.

Solution: Here is a roughly equivalent MIPS program:

```
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0    ; $5 holds i
for:
    slt $6, $5, $2 ; $6 is 1 if i < size, 0 otherwise
    beq $6, $0, end ; Go to end of loop if i < size is false
    mult $5, $4
    mflo $6        ; $6 = i * 4
    add $6, $1, $6  ; $6 = address of A + (i * 4) = address of A[i]
    sw $0, 0($6)    ; A[i] = 0
```



```

    add $5, $5, $11 ; i += 1
    beq $0, $0, for ; back to top of loop
end:
    jr $31

```

Comment 4.7

It is possible to simplify this:

- Instead of incrementing the index i by 1, multiplying it by 4, and then adding it to the starting address in \$1, we can just increment \$1 by 4 on each iteration.
- To detect the end of the loop, instead of comparing the index to the size in \$2, we can just decrement the size in \$2 by 1 on each iteration, and check if \$2 is 0 to end the loop.

Result 4.5

Hence we have a simplified code:

```

    lis $11
    .word 1
    lis $4 .word 4
    for:
        beq $2, $0, end      ; Go to end of loop if size == 0
        sw $0, 0($1)         ; Set A[i] = 0
        add $1, $1, $4       ; $1 = address of A[i+1]
        sub $2, $2, $11      ; size-= 1
        beq $0, $0, for      ; back to top of loop
    end:
        jr $31

```



4.5.1 Statically Allocated Arrays

Definition 4.6: Static Allocation

Static allocation refers to memory allocation done at compile time (or in this case, assembly time).

Code 4.6

To statically allocate an array, just reserve an area of memory in your program to store the array.

Example 4.8

A student wants to write a program that determines if \$7 contains the course code for a CS course they have previously taken. If it does, the program should return 1 in \$3. Otherwise, return 0 in \$3.

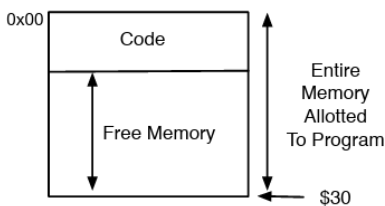
Solution: We have

```
lis $4
.word 4
lis $1
.word courseArray
lis $2
.word endArray
add $3, $0, $0
loop:
    beq $1, $2, end    ; go to end of loop if $1 is at endArray
    lw $6, 0($1)        ; load A[i] into $6
    add $1, $1, $4      ; $1 = address of A[i+1]
    bne $6, $7, loop    ; continue loop if $7 != A[i]
    lis $3
    .word 1             ; return 1 in $3 if $7 == A[i]
end:
    jr $31
courseArray:
.word 135
.word 136
.word 245
.word 246
endArray:
```



4.5.2 Stack-Allocated Arrays

Recall that (for now) we are assuming our programs are loaded into RAM at address 0. However, not all of RAM is available to our program; nor are we restricted only to the memory taken up by the program code itself. The loader reserves a block of memory, larger than the program code but smaller than the entirety of RAM, for our program to use.



Theorem 4.6

It then sets \$30 to the address just past the last word of memory allotted to the program.

Code 4.7

We can use \$30 as a pointer to allocate data; since \$30 is at the opposite end of allotted memory from our program code, we will not overwrite our code unless we allocate a lot of data.

To allocate memory, we store data at negative offsets from \$30. In terms of the diagram above, larger addresses are “farther down”, so we allocate memory “above” \$30. Now, once we have allocated something, we don’t want to overwrite it when we allocate again, so we update \$30 to mark the boundary between used and unused memory. In other words, we move \$30 upwards. When we want to deallocate memory, we move \$30 back downwards, to mark the memory we deallocated as “unused” again.

Discovery 4.7

In other words, we are using the area of memory above \$30 like a stack. Allocating means we “push to the stack”, moving \$30 upwards, and deallocating means we “pop from the stack”, moving \$30 downwards.

Definition 4.7: Stack Pointer

We refer to \$30 as the stack pointer, and refer to the area of memory opposite our program as “the stack”.

Question 4.12.

To push one word to the stack:

- Store the word at offset -4 from \$30.
- Subtract 4 from \$30.

Code 4.8

```
sw $3, -4($30)
lis $3
.word 4
sub $30, $30, $3
```

Question 4.13.

To pop one word from the stack:

- Add 4 to \$30.
- Load the word at offset -4 from \$30.

Code 4.9

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

4.6 Procedures in MIPS Assembly

4.6.1 Protecting Data in Registers

Code 4.10

In case multiple registers are being pushed (or popped), is to batch the stores (or loads) and then update the stack pointer in one go. You will often see procedures start with a section like this that **saves modified registers**:

Example 4.9

```
sw $1, -4($30)
sw $2, -8($30)
sw $5, -12($30)
sw $6, -16($30)
lis $1
.word 16
sub $30, $30, $1
```

Here four registers are stored on the stack. We then reuse one of the just-stored registers, \$1, to decrement the stack pointer by 16 bytes (4 words).

Comment 4.8

It's actually important that we used one of the four registers we stored to update the stack pointer.

Code 4.11

You'll often see procedures end with a similar section to the above, where registers are **restored** in a batch:

```
lis $1
.word 16
add $30, $30, $1
lw $1, -4($30)
lw $2, -8($30)
lw $5, -12($30)
lw $6, -16($30)
```

Theorem 4.7

An important convention is that, while procedures are free to make use of the stack, they must pop everything that they push, so that the stack is not unexpectedly changed by calling a procedure.

4.6.2 Call & Return

Definition 4.8: Procedure

A **procedure** in assembly is simply a label in front of assembly instructions that represent the code for the procedure.

Code 4.12

MIPS machine language has an instruction named Jump and Link Register, the last instruction we have yet to study in our simplified dialect of MIPS. In assembly language syntax, this is written as `jalr $s`. The Jump and Link Register instruction sets PC to \$s, but also places the *old value* of PC into \$31.

Discovery 4.8

The only issue is that when we used `jalr` we overwrote the value currently in \$31. Recall that the loader sets up \$31 with the address that MIPS programs should jump to in order to exit the program. We have overwritten that address. To prevent this from happening, anytime we use `jalr`, we must first preserve the address currently stored in \$31.

4.6.3 Chaining Procedure Calls & Recursion

The decisions we made regarding preserving registers when calling procedures and how we call and return from procedures mean that we have to do nothing special if a procedure calls another procedure.

4.6.4 Passing Arguments & Returning Values

There are two approaches to passing arguments to a procedure; we could use registers (often preferred when the number of parameters is relatively small) or we could use the stack (number of parameters bounded only by stack size!).

Example 4.10: Write a procedure to sum numbers 1 to N and store result in \$3.

Write a procedure to sum numbers 1 to N and store result in \$3.

```
; Sum1ToN adds all numbers from 1 to N
; Registers:
;  $1 constant-1 (original value should be preserved)
;  $2 input number N (original value should be preserved)
;  $3 return value (don't preserve original value)
Sum1ToN:
    sw $1, -4($30)      ; save previous value of $1
    sw $2, -8($30)      ; save previous value of $2
    lis $1
    .word 8
    sub $30, $30, $1    ; update stack pointer
    add $3, $0, $0      ; initialize return value to 0
    lis $1
    .word -1            ; initialize $1 to -1
loop:
    add $3, $3, $2
    add $2, $2, $1
    bne $2, $0, loop

    lis $1
    .word 8
    add $30, $30, $1    ; update stack pointer 34
    lw $1, -4($30)      ; restore original value of $1
    lw $2, -8($30)      ; restore original value of $2
    jr $31              ; return to caller
```

Question 4.14.

Write a MIPS procedure named `SumDigits` that takes a positive integer in register \$1 and stores the sum of the digits in register \$3. For example, if \$1 contains 123, \$3 should contain 6.

Solution: We keep divide the number and sum up the remainders until the quotient becomes zero:

| | | |
|----------------------|-----------------------|---------------------|
| SumDigits: | lis \$10 | done: |
| sw \$1, -4(\$30) | .word 10 | lis \$1 |
| sw \$2, -8(\$30) | loopOne: | .word 16 |
| sw \$10, -12(\$30) | beq \$1, \$0, done | add \$30, \$30, \$1 |
| sw \$11, -16(\$30) | divu \$1, \$10 | lw \$1, -4(\$30) |
| lis \$1 | mfhi \$11 | lw \$2, -8(\$30) |
| .word 10 | mflo \$1 | lw \$10, -12(\$30) |
| sub \$30, \$30, \$10 | add \$3, \$3, \$11 | lw \$11, -16(\$30) |
| add \$3, \$0, \$0 | beq \$0, \$0, loopOne | jr \$31 |

as desired.



Question 4.15.

Write a MIPS assembly language program that uses the procedure `SumDigits` to sum all the digits in \$1 and \$2. The result is stored in \$3. Values in \$1 and \$2 need not be preserved by the program.

Question 4.16.

Write a *recursive* MIPS procedure named `Sum` that computes the sum of an array of integers. The procedure takes the starting address of the array in \$1, and the number of elements in \$2. The result is stored in \$3.

Solution: We have

| | |
|---|--|
| Sum: | |
| sw \$2, -4(\$30) | lis \$4 ; retrieve a[N-1], |
| sw \$4, -8(\$30) | ; recall \$2 is already N-1 |
| sw \$5, -12(\$30) | .word 4 |
| sw \$31, -16(\$30) | mult \$2, \$4 |
| lis \$4 | mflo \$5 ; 4(N-1) |
| .word 16 | add \$5, \$1, \$5 ; address of a[N-1] |
| sub \$30, \$30, \$4 | lw \$5, 0(\$5) ; a[N-1] |
| add \$3, \$0, \$0 ; clear \$3 the result | |
| | add \$3, \$3, \$5 ; Sum(a, N-1) + a[N-1] |
| beq \$2, \$0, end ; N = 0 so done | |
| slt \$4, \$2, \$0 ; \$4 is 1 if N < 0 | end: ; pop all registers we pushed |
| lis \$5 | lis \$4 |
| .word 1 | .word 16 |
| beq \$5, \$4, end ; N < 0 so done | add \$30, \$30, \$4 |
| | lw \$2, -4(\$30) |
| ; if we get here we need a recursive call | lw \$4, -8(\$30) |
| sub \$2, \$2, \$5 ; N = N - 1 | lw \$5, -12(\$30) |
| lis \$5 | lw \$31, -16(\$30) |
| .word Sum | jr \$31 |
| jalr \$5 ; result of call will be in \$3 | |



5 Context Free Languages and Parsing

5.1 Formal Language Theory

Comment 5.1

The definition of *language* in Chapter 3 comes from an field of mathematics and theoretical computer science known as **formal language theory**. We will now present the typical definition of a language and its constituent parts used in formal language theory.

Definition 5.1: Alphabet

An **alphabet** is a finite, non-empty set.

Comment 5.2

The elements of an alphabet are often called *symbols*, *letters*, or *characters*. Typically we use the Greek letter Σ (Sigma) to denote an alphabet.

Definition 5.2: String

A **string** over an alphabet Σ is a sequence of symbols from Σ .

Comment 5.3

Strings are often called words in the context of formal language theory. Note that the word "word" here is different from the word "word" we introduced before.

Definition 5.3: Empty String

In formal language theory, the empty string is typically denoted as either ε (epsilon) or λ (lambda).

Definition 5.4: Language

A **language** over a alphabet Σ is a set of strings over Σ .

5.2 Regular Languages Revisited

5.2.1 Kleene's Theorem

Theorem 5.1: Kleene's Theorem

A language is regular if and only if it is recognized by a DFA.

Proof. Take CS360 or CS365.

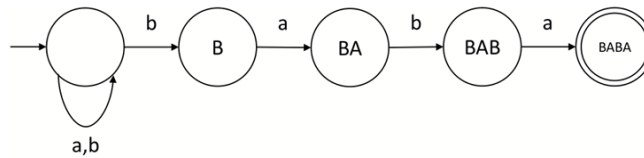
□

5.2.2 Nondeterministic Finite Automata

Definition 5.5: Nondeterministic Finite Automata

A state could have two arrows point outwards corresponding to a single character.

Example 5.1



Code 5.1

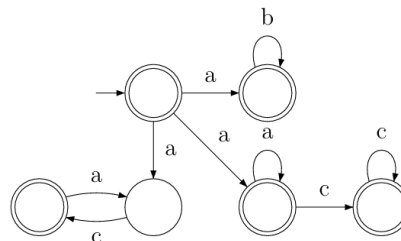
Choice Perspective: If there are multiple arrows leading out of the current state on the next symbol, the NFA needs to make a choice about which arrow to take. If there are no arrows leading out of the current state on the next symbol, the string is rejected. The NFA tries to make choices that result in the string being accepted; a string is only rejected if there is no possible sequence of choices that leads to acceptance.

Parallel Perspective: If there are multiple arrows leading out of the current state on the next symbol, the NFA tries all possibilities in parallel. When trying a possibility, if there are no arrows leading out of the current state, the possibility is discarded. The string is only rejected if all possibilities are discarded.

Example 5.2

Draw an NFA for the regular expression $ab^*|aa^*c^*(ac)^*$.

Solution: We start by creating simple DFAs for each component of the union: ab^* , aa^*c^* , and $(ac)^*$. Combining them we have



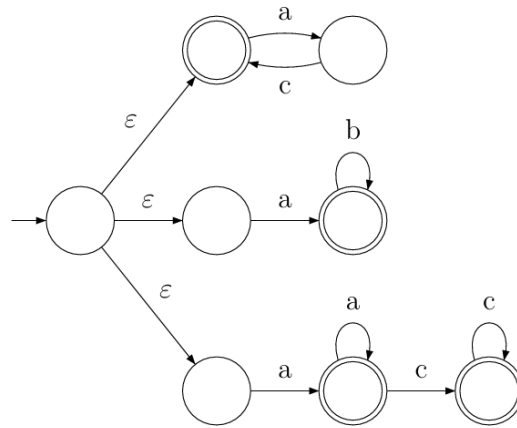
Comment 5.4

Although a DFA would have been worse, this NFA was still a little tricky to construct.

Fortunately, we have one more trick up our sleeves to make constructing NFAs simpler. In addition to allowing multiple arrows out of a state on the same symbol, we can also add optional arrows that can be taken without consuming a symbol.

Definition 5.6: ε -transitions

These optional arrows are called ε -transitions (epsilon transitions) and are denoted by labelling the arrow with the symbol ε , the empty string symbol.



Here is our solution.



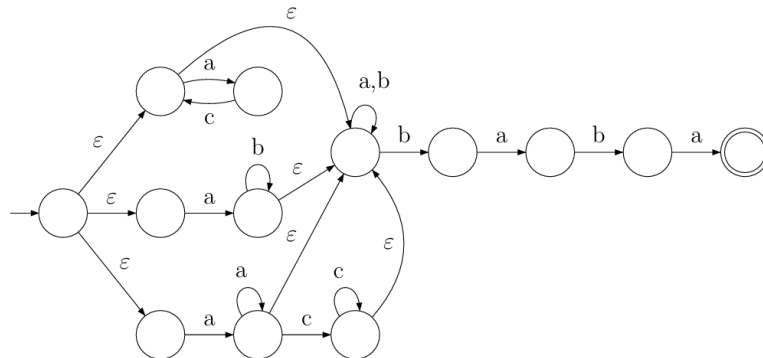
Result 5.1

In terms of the Parallel Perspective, the ε -transitions give us an easy way to explore all three DFAs simultaneously in parallel.

Example 5.3

Construct an NFA for the regular expression $(ab^*|aa^*c^*|(ac)^*)(a|b)^*baba$.

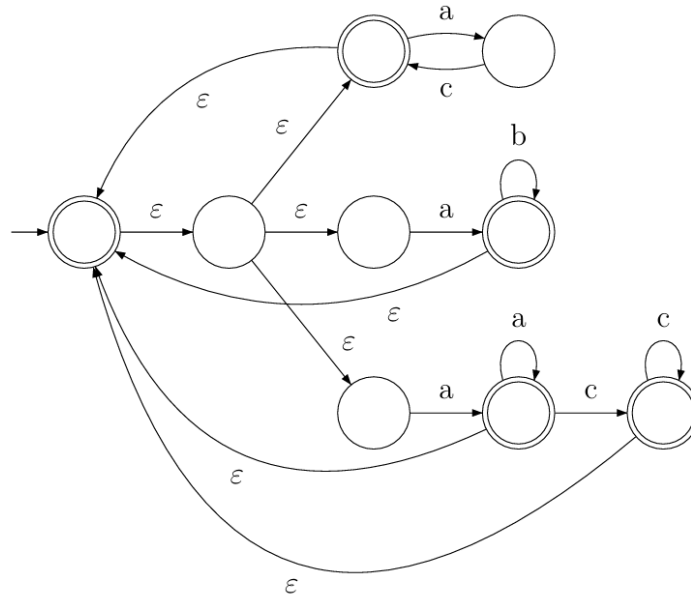
Solution: We have



Example 5.4

Construct an NFA for the regular expression $(ab^*|aa^*c^*|(ac)^*)^*$.

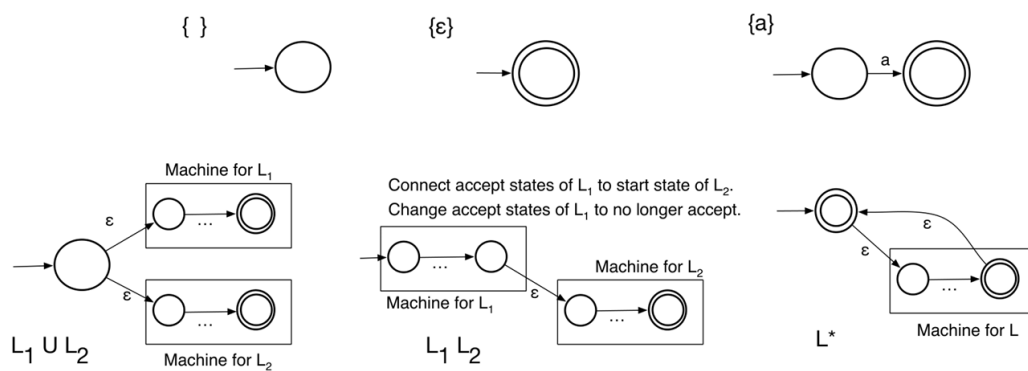
Solution: We have



5.2.3 From Regular Expressions to NFAs

Result 5.2

We have the following proof by picture:

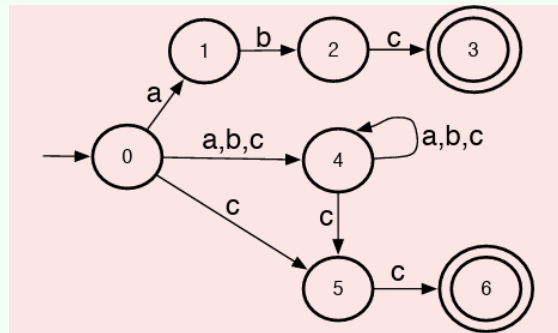


Question 5.1.

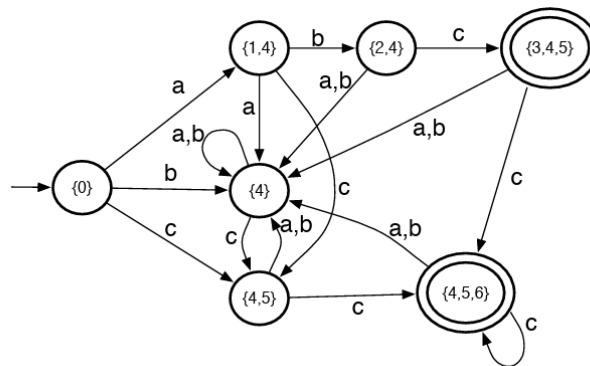
Using subset construction, convert the following NFA for the language:

$$L = \{abc\} \cup \{w : w \text{ ends with } cc\}$$

into a DFA.



Solution: We have



5.3 The Limitations of Regular Languages

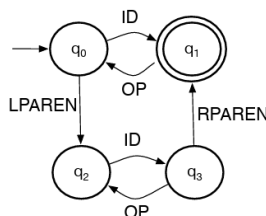
Consider we have a set of alphabet and a language:

$$\Sigma = \{ \text{ID}, \text{OP}, \text{LPAREN}, \text{RPAREN} \}, \quad L = \{w : w \text{ represents a valid arithmetic expression}\}$$

Using this extension, we could write expressions such as

ID OP LPAREN ID OP ID RPAREN


to represent the tokenization of an input such as $a - (b * c)$. The following DFA recognizes expressions that never nest parentheses.



For example, expressions such as $a + b$, $(a + b) * c$ and $(a + b) * (c - d)$ will be recognized (in tokenized form) but expressions like $a - (b * (c - d))$ will not because there is a pair of parentheses nested inside another pair.

Question 5.2.

Could we extend the DFA to allow two levels of nested parentheses? What about three levels?

Solution: The DFA can be extended to allow for any finite level of nesting by simply repeating what we did to add the first level of parentheses. 

Comment 5.5

In the following section, we discuss context-free languages, which have the expressive power to represent, among other things, structures with arbitrary nesting.

5.4 Context-Free Languages

Definition 5.7: Grammar

A **grammar** is the language of languages (Matt Might). A grammar helps us describe what we are allowed and not allowed to say.

Definition 5.8: Context Free Grammar

A **Context-Free Grammar** (CFG) is a 4-tuple (N, T, P, S) where

- N is an alphabet whose elements are called **nonterminal symbols**.
- T is an alphabet whose elements are called **terminal symbols**.
- P is a finite set of **production rules**, each of the form $A \rightarrow \beta$, where $A \in N$ and $\beta \in (N \cup T)^*$.
- $S \in N$ is a **start symbol**.

Comment 5.6

We write $V = N \cup T$; this is called the **vocabulary**, the set of all symbols used by the grammar (nonterminal or terminal).

Definition 5.9: Nonterminal Symbols

The **nonterminal symbols** are metasymbols used in the grammar to enforce a certain structure, but do not appear in the language described by the grammar.

Definition 5.10: Terminal Symbols

The symbols that actually appear in strings of the language are the **terminal symbols**.

Definition 5.11: Production Rules

The **production rules** are the “string rewriting rules”. The left-hand side always consists of a single nonterminal because these “meta-symbols” are the only ones that can be rewritten. The right-hand side of the rule (the string that the left-hand side can be rewritten to) is defined as an element of $(N \cup T)^*$.

Definition 5.12: Start Symbol

The **start symbol** serves as a starting point for the string rewriting process.

Example 5.5

The following CFG represents valid arithmetic expressions with arbitrary balanced parentheses:

$$N = \{ \text{expr} \}$$

$$T = \{ \text{ID}, \text{OP}, \text{LPAREN}, \text{RPAREN} \}$$

Productions :

$$\text{expr} \rightarrow \text{ID}$$

$$\text{expr} \rightarrow \text{expr OP expr}$$

$$\text{expr} \rightarrow \text{LPAREN expr RPAREN}$$

$$S = \text{expr}$$

5.4.1 Conventions

- Lower-case letters from the start of the alphabet, i.e., a, b, c, \dots , are elements of T (terminals).
- Lower-case letters from the end of the alphabet, i.e., w, x, y, z , are elements of T^* (strings).
- Upper-case letters from the start of the alphabet, i.e., A, B, C, \dots , are elements of N (nonterminals).
- Greek letters, i.e., $\alpha, \beta, \gamma, \dots$, are elements of V^* (recall this is $(N \cup T)^*$, strings of terminals and nonterminals).
- When not specified formally, the nonterminal on the left-hand side (LHS) of the first production is the start symbol.

5.4.2 Derivations

Example 5.6

For the grammar of expressions with balanced parentheses, give a derivation for the string ID OP LPAREN ID OP ID RPAREN, i.e., show $\text{expr} \Rightarrow^* \text{ID OP LPAREN ID OP ID RPAREN}$. The derivation begins with the start symbol, which is *expr* for our grammar.

Solution:

| | |
|---|---|
| $\text{expr} \Rightarrow \text{expr OP expr}$ | (applying $\text{expr} \rightarrow \text{expr OP expr}$) |
| $\Rightarrow \text{ID OP expr}$ | (applying $\text{expr} \rightarrow \text{ID}$) |
| $\Rightarrow \text{ID OP LPAREN expr RPAREN}$ | (applying $\text{expr} \rightarrow \text{LPAREN expr RPAREN}$) |
| $\Rightarrow \text{ID OP LPAREN expr OP expr RPAREN}$ | (applying $\text{expr} \rightarrow \text{expr OP expr}$) |
| $\Rightarrow \text{ID OP LPAREN ID OP expr RPAREN}$ | (applying $\text{expr} \rightarrow \text{ID}$) |
| $\Rightarrow \text{ID OP LPAREN ID OP ID RPAREN}$ | (applying $\text{expr} \rightarrow \text{ID}$) |



Definition 5.13: Directly Derive

Over a CFG (N, T, P, S) , we say that A **directly derives** γ , and write $A \Rightarrow \gamma$, if and only if there is a rule $A \rightarrow \gamma$ in P .

Definition 5.14: Derive

Over a CFG (N, T, P, S) , we say that α **derives** β , and write $\alpha \Rightarrow^* \beta$, if either $\alpha = \beta$, or if there exists γ such that $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow^* \beta$.

Comment 5.7

In other words, we derive the sequence β from α through zero or more applications of production rules. For the mathematically inclined, this is the reflexive transitive closure of the “directly derives” relation.

Definition 5.15:

Over a CFG (N, T, P, S) , a derivation of a string of terminals x is a sequence $\alpha_0 \alpha_1 \dots \alpha_n$ such that $\alpha_0 = S$ and $\alpha_n = x$ and $\alpha_i \Rightarrow \alpha_{i+1}$ for $0 \leq i < n$.

Example 5.7

For the example above, we have $\alpha_0 = \text{expr}$, $\alpha_1 = \text{expr OP expr}$, $\alpha_2 = \text{ID OP expr}$, and so on, ending at $\alpha_n = \text{ID OP LPAREN ID OP ID RPAREN}$.

5.4.3 The Language of a Grammar

Definition 5.16: Language of CFG

The language of a CFG (N, T, P, S) is $L(G) = \{w \in T^* : S \Rightarrow^* w\}$.

Definition 5.17: Context-free

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

Theorem 5.2

Every regular language is context free.


Proof. It suffices to show that for each of these six cases, it is possible to create a corresponding context-free grammar (and therefore a context-free language) that represents the regular language. We leave this as an exercise for the interested reader. \square

Example 5.8

Let $T = \{a, b\}$. Write a CFG for $\{a^n b^n : n \in \mathbb{N}\}$. Once you create the grammar, find a derivation in your grammar for the string $aaabbb$.

Solution: We take

$$\begin{aligned} N &= \{S\} \\ T &= \{a, b\} \\ \text{Productions: } S &\rightarrow \varepsilon \\ S &\rightarrow aSb \end{aligned}$$

The following is a derivation for the string $aaabbb$: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$. 

Question 5.3.

Write a CFG for palindromes over $\{a, b, c\}$.

Solution: We take

$$\begin{aligned} N &= \{S, M\} \\ T &= \{a, b, c\} \\ \text{Productions: } S &\rightarrow aSa \mid bSb \mid cSc \mid M \\ M &\rightarrow a \mid b \mid c \mid \varepsilon \end{aligned}$$

**Question 5.4.**

Write a CFG for the regular language defined by the regular expression $a(a|b)^*b$.

Solution: We take

$$\begin{aligned} N &= \{S, M\} \\ T &= \{a, b\} \\ \text{Productions: } S &\rightarrow aMb \\ M &\rightarrow aM \mid bM \mid \varepsilon \end{aligned}$$



5.4.4 Derivations and Parse Trees

Definition 5.18: Parse Tree

The root of the parse tree is the start symbol. Each non-leaf node is a nonterminal, and its immediate descendants are the right-hand side of the rule that was used for the nonterminal in the derivation.

Result 5.3

Based on how parse trees are created, we have the following properties:

- A derivation uniquely defines a parse tree.
- An input string can have more than one parse tree.

Question 5.5.

Can you come up with a CFG and an input string such that the string has two different parse trees with respect to the CFG?

Solution: See next section.



Definition 5.19: Leftmost Derivation

In a **leftmost derivation**, we always expand the leftmost nonterminal first. Formally, each step has the form:

$$xA\gamma \Rightarrow x\alpha\gamma$$

Analogously, in a rightmost derivation, we always expand the rightmost nonterminal first. Formally, each step has the form:

$$\beta Ax \Rightarrow \beta\alpha x$$

5.4.5 Ambiguous Grammars

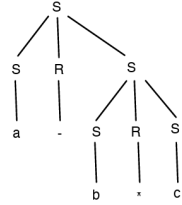
Consider the following CFG for expressions with arithmetic operations:

$$\begin{aligned} S &\rightarrow a \mid b \mid c \mid SRS \\ R &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

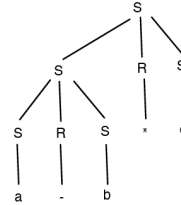
Discovery 5.1

Let's try to obtain a derivation for the input string $a - b * c$:

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



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



Definition 5.20: Ambiguous

A grammar is **ambiguous** if there is a string in the language which has more than one distinct leftmost derivation, or more than one distinct rightmost derivation, or equivalently, more than one parse tree.

Comment 5.8

The grammar for arithmetic expressions discussed above is ambiguous: we obtained two distinct derivations of our input string even though we *chose the same derivation style*.

Code 5.2

One way to avoid the ambiguity is to use precedence heuristics to guide the derivation process.

Example 5.9

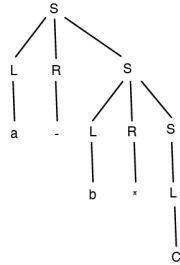
For example, we could force the language syntax to require parentheses. The following grammar is unambiguous since it only accepts strings such as $(a - (b * c))$ or $((a - b) * c)$.

$$S \rightarrow a \mid b \mid c \mid (SRS) \\ R \rightarrow + \mid - \mid * \mid /$$

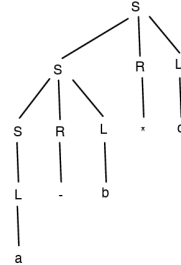
Left associativity or right associativity refers to whether operators with the same precedence are evaluated from left to right or from right to left. For addition, $(a + b) + c$ and $a + (b + c)$ are equivalent so it doesn't matter. But for subtraction, $(a - b) - c$ and $a - (b - c)$ are different, and we usually expect $a - b - c$ without parentheses to be evaluated left-to-right, that is, in a left-associative way.

We can enforce left or right associativity in our grammar by insisting on how the recursion works.

Right Associative Operations

$$\begin{aligned} S &\rightarrow LRS \mid L \\ L &\rightarrow a \mid b \mid c \\ R &\rightarrow + \mid - \mid * \mid / \end{aligned}$$


Left Associative Operations

$$\begin{aligned} S &\rightarrow SRL \mid L \\ L &\rightarrow a \mid b \mid c \\ R &\rightarrow + \mid - \mid * \mid / \end{aligned}$$


Comment 5.9

In addition to using this technique to change the associativity of expressions produced by a grammar, we can create a grammar that follows BEDMAS rules more closely by making $*$ and $/$ appear further down in the tree.

Discovery 5.2

Recall, with our depth-first post-order traversal, the deeper parts of the tree will be evaluated first, and therefore get a higher precedence.

Code 5.3

We add multiple “levels” to the grammar corresponding to levels of precedence.

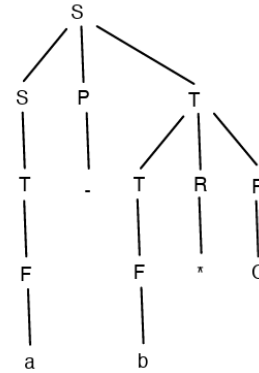
$$\begin{aligned} S &\rightarrow SPT \mid T \\ T &\rightarrow TRF \mid F \\ F &\rightarrow a \mid b \mid c \mid (S) \\ P &\rightarrow + \mid - \\ R &\rightarrow * \mid / \end{aligned}$$

The top level expressions generated by S can only use the $+$ and $-$ operators, which have lower precedence than the expressions generated by T that can only use $*$ and $/$.

Question 5.6.

Using the above grammar, find a leftmost derivation for $a - b * c$ and draw the corresponding parse tree.

Solution: We have

$$\begin{aligned}
S &\Rightarrow SPT \Rightarrow TPT \Rightarrow FPT \Rightarrow aPT \\
&\Rightarrow a - T \Rightarrow a - TRF \Rightarrow a - FRF \\
&\Rightarrow a - bRF \Rightarrow a - b * F \Rightarrow a - b * c
\end{aligned}$$


Question 5.7.

Consider the bitwise NOT operator \sim which is a unary operator (takes one argument instead of two). This operator should have higher precedence than all the binary operators. How can we add this operator to the above grammar in a way that ensures it has the correct precedence?

Solution: A simple way is to add the rule $F \rightarrow \sim F$. This ensures the NOT operator will appear deeper in the tree than the other binary operators and therefore will have higher precedence.



5.4.6 Parsing Algorithm

We are interested in obtaining the derivation, since the derivation uniquely specifies the parse tree that represents the program's structure (indeed, ultimately, we're more interested in this parse tree than the derivation per se).

Definition 5.21: Parsing

The problem of finding the derivation is called **parsing**.

Comment 5.10

The next two modules describe two different approaches to writing parsing algorithms.

6 LL(1) Top-Down Parsing

Comment 6.1

We will choose leftmost derivations, i.e., whenever we are at some α_i which contains multiple nonterminals, we will choose a rule for the leftmost nonterminal.

6.1 Augmented Grammars

To simplify the specification of parsing algorithms, it helps if the start symbol has only one associated production rule. If a grammar does not satisfy this condition, we can augment the grammar by creating $G' = (N', T', P', S')$ from our original CFG $G = (N, T, P, S)$ where

$$\begin{aligned} N' &= N \cup \{S'\} \\ T' &= T \cup \{\vdash, \dashv\} \\ P' &= P \cup \{S' \rightarrow \vdash S \dashv\} \end{aligned}$$

Definition 6.1: \vdash and \dashv

The symbols, \vdash and \dashv symbolize the beginning and end of the file respectively.

Discovery 6.1

The end of the file marker is useful since it is a symbol that is not considered part of the input.

6.2 Informal Top-Down Parsing Algorithm

Comment 6.2

Given that we choose to work with augmented grammars, there is a guarantee that there will be only one rule.

Algorithm 6.1

We apply the rule to get our first α , i.e., a step in our derivation;

$$\alpha_1 \text{ is } \vdash S \dashv$$

This was our setup stage. At this point, we apply the following loop:

From left to right, look at the symbols in the current α_i . If the symbol is a terminal, match it to the terminal at the same position in the input string s . If it does not match, report a parsing error and terminate. Otherwise, continue matching terminals until we hit the first nonterminal in α_i . This is the leftmost nonterminal. Choose an appropriate production rule for this nonterminal and apply it,

i.e., replace the nonterminal with the right-hand side of the chosen rule. This gives α_{i+1} . Repeat the algorithm until we either generate a parse error or no more nonterminals are left.

Example 6.1

Consider the following augmented grammar: (we just list the production rules; upper case letters are nonterminals and lower-case letters are terminals)

Suppose our input string s is $\vdash abywx \dashv$. At the setup stage, we get α_1 as $\vdash S \dashv$.

$S' \rightarrow \vdash S \dashv$
 $S \rightarrow AyB$
 $A \rightarrow ab$
 $A \rightarrow cd$
 $B \rightarrow z$
 $B \rightarrow wx$

- We begin looking at symbols in α_1 left to right. The first symbol is \vdash which matches the first symbol in the string s . We move to the next symbol and see that it is the nonterminal S . We apply the rule $S \rightarrow AyB$ to obtain $\alpha_2, \vdash AyB \dashv$.
- We begin the process again; \vdash from α_2 matches the first symbol in the string s and then we encounter the leftmost nonterminal, A . The algorithm chooses the appropriate rule by sneaking a peek at what the **next unmatched symbol** in the input is (it is a). The algorithm chooses to apply $A \rightarrow ab$. This gives us $\alpha_3, \vdash abyB \dashv$.
- The process repeats: the algorithm successively matches \vdash , then a , then b , then y with the input and then hits the leftmost nonterminal B . By sneaking a peek at the next unmatched symbol (it is w) the algorithm chooses the appropriate rule $B \rightarrow wx$ to produce $\alpha_4, \vdash abywx \dashv$.
- The process repeats once again: the algorithm matches \vdash , then a , then b , then y , then w , then x and then \dashv . At this point, there are no symbols left in α_4 and there are no unmatched symbols in the input string. The parse was successful.

The derivation obtained:

$$S' \Rightarrow \underbrace{\vdash S \dashv}_{\alpha_1} \Rightarrow \underbrace{\vdash AyB \dashv}_{\alpha_2} \Rightarrow \underbrace{\vdash abyB \dashv}_{\alpha_3} \Rightarrow \underbrace{\vdash abywx \dashv}_{\alpha_4} = s$$

Discovery 6.2: What is an “Appropriate” Rule?

“Sneaking a peek” has a more formal term; it is called using a **lookahead**. We will soon discuss how we can create a table $\text{Predict}[A][a]$ that, given a nonterminal A and a lookahead terminal a , will predict which rule to use for A .

Discovery 6.3: This is Inefficient

An obvious optimization is to stop tracking the prefix of the input (and α_i) that has already been matched.

Comment 6.3

Going back to the example we traced, since we always begin with α_1 as $\vdash S \dashv$, we can begin with the stack contents such that \vdash is on the top of the stack and \dashv at the bottom.

6.3 Formal Top-Down Parsing Algorithm

Algorithm 6.2

Start by pushing the start symbol of the grammar on the stack.

While the TOS is a terminal, pop it and match it against input. If it does not match, report a parse error. Otherwise continue with the next symbol. If TOS is a nonterminal A , pop it and query $\text{Predict}[A][a]$ where a is the first unmatched symbol from the input. If $\text{Predict}[A][a]$ has no rule for us, reject with a parse error. Otherwise, if $\text{Predict}[A][a]$ returns a rule $A \rightarrow \gamma$, apply the rule by pushing the symbols in γ **in reverse**. We push the symbols in reverse so that the leftmost symbol ends up on TOS.

Repeat until we either get a parse error, or the stack is empty. If the stack is empty, we accept.

Definition 6.2: TOS

TOS stands for top of the stack.

Code 6.1

Algorithm 6: Top-Down Parsing Algorithm

```

1 push  $S'$  ;
2 for each ' $a$ ' in  $\vdash$  input  $\dashv$  do
3   while top of stack is  $A \in N$  do
4     pop  $A$  ;
5     if  $\text{Predict}[A][a]$  gives  $A \rightarrow \gamma$  then
6       push the symbols in  $\gamma$  (right to left)
7     else reject
      // TOS is a terminal
8   if TOS is not ' $a$ ' then
9     Reject
10  else pop ' $a$ '
11 Accept // stack is necessarily empty

```

Example 6.2

We use the same example as above: The numbers in some cells of the Predict table specify which rule to apply given a nonterminal and lookahead, while an empty cell means there is no appropriate rule.

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow AyB$
- (3) $A \rightarrow ab$
- (4) $A \rightarrow cd$
- (5) $B \rightarrow z$
- (6) $B \rightarrow wx$

| | \vdash | a | b | c | d | w | x | y | z | \dashv |
|------|----------|---|---|---|---|---|---|---|---|----------|
| S' | 1 | | | | | | | | | |
| S | | 2 | | 2 | | | | | | |
| A | | 3 | | 4 | | | | | | |
| B | | | | | | 6 | | | 5 | |

The input string s is $\vdash a b y w x \dashv$:

| Read | Unread | Stack | Action |
|---------------------------|---------------------------|-------------------|---|
| ϵ | $\vdash a b y w x \dashv$ | S' | pop S' , $Predict[S'][\vdash]$ gives rule 1, push \dashv , S , \vdash |
| ϵ | $\vdash a b y w x \dashv$ | $\vdash S \dashv$ | Match \vdash |
| \vdash | $a b y w x \dashv$ | $S \dashv$ | pop S , $Predict[S][a]$ gives rule 2, push B , y , A |
| \vdash | $a b y w x \dashv$ | $A y B \dashv$ | pop A , $Predict[A][a]$ gives rule 3, push b , a |
| $\vdash a$ | $b y w x \dashv$ | $a b y B \dashv$ | match a |
| $\vdash a b$ | $y w x \dashv$ | $b y B \dashv$ | match b |
| $\vdash a b y$ | $w x \dashv$ | $y B \dashv$ | match y |
| $\vdash a b y$ | $w x \dashv$ | $B \dashv$ | pop B , $Predict[B][w]$ gives rule 6, push x , w |
| $\vdash a b y w$ | $x \dashv$ | $w x \dashv$ | match w |
| $\vdash a b y w x$ | \dashv | $x \dashv$ | match x |
| $\vdash a b y w x \dashv$ | ϵ | \dashv | match \dashv |
| $\vdash a b y w x \dashv$ | ϵ | ϵ | Accept |

Discovery 6.4

An interesting thing to note is that at any time during the algorithm, we can obtain α_i by concatenating the contents in the **Read** and **Stack** columns.

Theorem 6.1

The algorithm can produce an error in one of two ways:

1. The TOS is a terminal, but it does not match the next input symbol.
2. The algorithm queries $Predict[A][a]$ for some A and a and finds either no rule, or more than one rule.

Comment 6.4

The algorithm we just discussed is called LL(1) parsing. The first L represents the Left-to-Right scan of input, the second L indicates that the algorithm produces Leftmost derivations, and the 1 indicates that we looked ahead at 1 symbol.

Definition 6.3: LL(1) Grammar

A grammar is **LL(1)** if and only if each cell of the Predict table contains at most one rule.

6.4 Constructing the Predict Table

We have talked about the Predict table as a two-dimensional lookup table. But really, it is a function that produces a set of rules that can apply when $A \in N'$ (a nonterminal) is on the TOS and $a \in T'$ (a terminal) is the next input symbol. Formally:

$$First(\beta) = \{a \in T' : \beta \Rightarrow^* a\gamma, \text{ for some } \gamma \in V^*\}$$

i.e., the set of terminals that can be the first symbol in a string derived from $\beta \in V^*$, and

$$\text{Predict}[A][a] = \{A \rightarrow \beta : a \in First(\beta)\}$$

Comment 6.5

The Predict function as described above misses some rules.

Example 6.3

Consider the example:

- (1) $S' \rightarrow XY$
- (2) $X \rightarrow \varepsilon$
- (3) $Y \rightarrow z$

| | |
|------|-----|
| | z |
| S' | 1 |
| X | |
| Y | 3 |

Notice that $S' \Rightarrow XY \Rightarrow Y \Rightarrow z$ is a derivation for the string z . However, when the algorithm queries $\text{Predict}[X][z]$, there is no rule for this. We haven't made a mistake in this table entry since for the only rule for X , $X \rightarrow \varepsilon$, $First(\varepsilon) = \{\}$ and therefore $z \notin First(\varepsilon)$.

Code 6.2

What is missing from our definition of **Predict** is accounting for rules where the nonterminal is nullable, i.e., it derives ε .

We define the following:

$$Nullable(\beta) = \begin{cases} true & \text{if } \beta \Rightarrow^* \varepsilon \\ false & \text{otherwise} \end{cases}$$

and

$$Follow(A) = \{b \in T' : S' \Rightarrow^* \alpha Ab\beta \text{ for some } \alpha, \beta \in V^*\}$$

i.e., $Follow(A)$ is a set of terminals that can come immediately after A in a derivation starting at the start symbol S' .

Based on this, we can update our definition for **Predict** to this correct definition:

$$\text{Predict}[A][a] = \{A \rightarrow \beta : a \in \text{First}(\beta)\} \cup \{A \rightarrow \beta : \beta \text{ is nullable and } a \in \text{Follow}(A)\}$$

6.4.1 Computing Nullable

Discovery 6.5

- $\text{Nullable}(\varepsilon) = \text{true}$ by definition
- $\text{Nullable}(\beta) = \text{false}$ whenever β contains a terminal symbol.
- $\text{Nullable}(AB) = \text{Nullable}(A) \wedge \text{Nullable}(B)$

Based on these observations, it suffices to compute $\text{Nullable}(A)$ for all $A \in N'$.

Code 6.3

Algorithm 7: $\text{Nullable}(A)$ for all $A \in N'$.

```

1 Initialize  $\text{Nullable}(A) = \text{false}$  for all  $A \in N'$  ;
2 repeat until nothing changes ;
3   for each production rule in  $P$  do
4     if  $(P \text{ is } A \rightarrow \varepsilon) \text{ or } (P \text{ is } A \rightarrow B_1 \cdots B_k \text{ and } \bigwedge_{i=1}^k \text{Nullable}(B_i) = \text{true})$  then
5        $\text{Nullable}(A) = \text{true}$ 

```

Example 6.4

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow bSd$
- (3) $S \rightarrow pSq$
- (4) $S \rightarrow C$
- (5) $C \rightarrow lC$
- (6) $C \rightarrow \varepsilon$

| Iteration | 0 | 1 | 2 | 3 |
|-----------|-------|-------|-------|-------|
| S' | false | false | false | false |
| S | false | false | true | true |
| C | false | true | true | true |

When the algorithm starts (iteration 0), all nonterminals are marked as non-nullable. On the first iteration, C is marked nullable due to Rule 6. On the second iteration, S is marked nullable due to Rule 4 and the fact that C is nullable. On the third iteration, no new nonterminals are marked as nullable, so we are done.

6.4.2 Computing First

We begin by presenting an algorithm for computing $\text{First}(A)$ where A is a nonterminal.

Code 6.4

Algorithm 8: $First(A)$ for all $A \in N'$

```

1 Initialize  $First(A) = \{\}$  for all  $A \in N'$ ; repeat until nothing changes ;
2   for each rule  $A \rightarrow B_1B_2 \cdots B_k$  in  $P$  do
3     for  $i \in \{1, \dots, k\}$  do
4       if  $B_i \in T'$  then
5          $First(A) = First(A) \cup \{B_i\}$ ; break;
6       else
7          $First(A) = First(A) \cup First(B_i)$  ;
8         if  $Nullable(B_i) == False$  then break;

```

Comment 6.6

The algorithm inspects each rule of the form $A \rightarrow \beta$. Let β be $B_1B_2 \cdots B_k$. The algorithm begins to sequentially assess each B_i beginning at B_1 . If B_i is a terminal, the terminal is added to $First(A)$ and the rest of β is not processed since it can no longer contribute to $First(A)$ (as a terminal has already been encountered). If B_i happens to be a nonterminal, then $First(A)$ should contain $First(B_i)$. Additionally, we only look at B_{i+1} if B_i is nullable (as otherwise B_{i+1} cannot contribute to $First(A)$).

It is also useful to have the ability to compute the First set of an arbitrary β , i.e., not necessarily the right-hand side of a rule.

Algorithm 9: $First(\beta)$ where $\beta = B_1B_2 \cdots B_n \in V^*$

```

1 result =  $\emptyset$  ;
2 for  $i \in \{1, \dots, n\}$  do
3   if  $B_i \in T'$  then
4     result = result  $\cup \{B_i\}$ ; break;
5   else
6     result = result  $\cup First(B_i)$  ;
7     if  $Nullable(B_i) == False$  then break;

```

Example 6.5

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow bSd$
- (3) $S \rightarrow pSq$
- (4) $S \rightarrow C$
- (5) $C \rightarrow lC$
- (6) $C \rightarrow \varepsilon$

| Iteration | 0 | 1 | 2 | 3 |
|-----------|--------|--------------|---------------|---------------|
| S' | $\{\}$ | $\{\vdash\}$ | $\{\vdash\}$ | $\{\vdash\}$ |
| S | $\{\}$ | $\{b, p\}$ | $\{b, p, l\}$ | $\{b, p, l\}$ |
| C | $\{\}$ | $\{l\}$ | $\{l\}$ | $\{l\}$ |

6.4.3 Computing Follow

Code 6.5

Algorithm 10: Follow(A) for all $A \in N$ (Recall $N = N' \setminus \{S'\}$)

```

1 Initialize  $Follow(A) = \{\}$  for all  $A \in N$  ;
2 repeat until nothing changes ;
3   for each rule  $A \rightarrow B_1 B_2 \cdots B_k$  in  $P'$  do
4     for  $i \in \{1, \dots, k\}$  do
5       if  $B_i \in N$  then
6          $Follow(B_i) = Follow(B_i) \cup First(B_{i+1} \cdots B_k)$  ;
7         if  $\bigwedge_{m=i+1}^k Nullable(B_m) == True$  or  $i == k$  then
8            $Follow(B_i) = Follow(B_i) \cup Follow(A)$ 

```

Example 6.6

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow bSd$
- (3) $S \rightarrow pSq$
- (4) $S \rightarrow C$
- (5) $C \rightarrow lC$
- (6) $C \rightarrow \varepsilon$

| Iteration | 0 | 1 | 2 |
|-----------|--------|--------------------|--------------------|
| S | $\{\}$ | $\{\vdash, d, q\}$ | $\{\vdash, d, q\}$ |
| C | $\{\}$ | $\{\vdash, d, q\}$ | $\{\vdash, d, q\}$ |

6.4.4 Computing Predict Table

We now look at the algorithm that computes the Predict table leveraging these previously described algorithms.

Code 6.6

Algorithm 11: Computing Predict

```

1 Initialize  $Predict[A][a] = \{\}$  for all  $A \in N'$  and  $a \in T'$  ;
2 for each production  $A \rightarrow \beta$  in  $P$  do
3   for each  $a \in First(\beta)$  do
4     Add  $A \rightarrow \beta$  to  $Predict[A][a]$ 
5   if  $Nullable(\beta) == True$  then
6     for each  $a \in Follow(A)$  do
7       Add  $A \rightarrow \beta$  to  $Predict[A][a]$ 

```

Example 6.7

- (1) $S' \rightarrow \vdash S \dashv$
 (2) $S \rightarrow bSd$
 (3) $S \rightarrow pSq$
 (4) $S \rightarrow C$
 (5) $C \rightarrow lC$
 (6) $C \rightarrow \varepsilon$

Summary Table

| | Nullable | First | Follow |
|------|----------|---------------|--------------------|
| S' | false | $\{\vdash\}$ | $\{\}$ |
| S | true | $\{b, p, l\}$ | $\{\dashv, d, q\}$ |
| C | true | $\{l\}$ | $\{\dashv, d, q\}$ |

Predict Table

| | \vdash | \dashv | b | d | p | q | l |
|------|----------|----------|-----|-----|-----|-----|-----|
| S' | 1 | | | | | | |
| S | | 4 | 2 | 4 | 3 | 4 | 4 |
| C | | 6 | | 6 | | 6 | 5 |

Discovery 6.6

Recall our definition of LL(1). Looking at the Predict table, we see that each entry in the Predict table has at most one rule. This grammar is LL(1).

Question 6.1.

Trace the LL(1) parsing algorithm using the input string $\vdash b l b d \dashv$ using the grammar and predict table just constructed.

Solution: We have

| Read | Unread | Stack | Action |
|--------------|-------------------------|-------------------|---|
| ϵ | $\vdash b l b d \dashv$ | S' | pop S' , $Predict[S'][\vdash]$ gives rule 1, push \dashv , S , \vdash |
| \vdash | $b l b d \dashv$ | $\vdash S \dashv$ | Match \vdash |
| \vdash | $b l b d \dashv$ | $S \dashv$ | pop S , $Predict[S][b]$ gives rule 2, push d , S , b |
| \vdash | $b l b d \dashv$ | $b S d \dashv$ | match b |
| $\vdash b$ | $l b d \dashv$ | $S d \dashv$ | pop S , $Predict[S][l]$ gives rule 4, push C |
| $\vdash b$ | $l b d \dashv$ | $C d \dashv$ | pop C , $Predict[C][l]$ gives rule 5, push C , l |
| $\vdash b$ | $l b d \dashv$ | $l C d \dashv$ | match l |
| $\vdash b l$ | $b d \dashv$ | $C d \dashv$ | pop C , $Predict[C][d]$ has no rule! |

This input generates a parse error; the input string cannot be derived by the grammar!



Question 6.2.

Using the algorithms discussed above, create tables showing the iterations in computing Nullable and the First and Follow sets for the following grammar:

- (0) $S' \rightarrow \vdash S \dashv$
 (1) $S \rightarrow c$
 (2) $S \rightarrow QRS$
 (3) $Q \rightarrow R$
 (4) $Q \rightarrow d$
 (5) $R \rightarrow \varepsilon$
 (6) $R \rightarrow b$

Solution: We have

Nullable 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 |

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} |

Follow Table:

| Iter | 0 | 1 | 2 |
|------|----|------------|------------|
| S | {} | { \neg } | { \neg } |
| Q | {} | {b,c,d} | {b,c,d} |
| R | {} | {b,c,d} | {b,c,d} |



Question 6.3.

For the same grammar as above, compute the Predict Table.

Solution: We have

| | \vdash | b | c | d | \neg |
|------|----------|--------|--------|--------|--------|
| S' | {0} | | | | |
| S | | {2} | {1, 2} | {2} | |
| Q | | {3} | {3} | {3, 4} | |
| R | | {5, 6} | {5} | {5} | |

Comment 6.7

Note that this tells us that the grammar is NOT LL(1).



Question 6.4.

Construct the four tables (Nullable, First, Follow and Predict) for the following grammar:

- (0) $S' \rightarrow \vdash S \neg$
- (1) $S \rightarrow Bb$
- (2) $S \rightarrow Cd$
- (3) $B \rightarrow aB$
- (4) $B \rightarrow \varepsilon$
- (5) $C \rightarrow cC$
- (6) $C \rightarrow \varepsilon$

Solution: We have

| | Nullable | First | Follow |
|------|----------|--------------|------------|
| S' | False | { \vdash } | {} |
| S | False | {a,b,c,d} | { \neg } |
| Q | True | {a} | {b} |
| R | True | {c} | {d} |

Predict Table:

| | \vdash | a | b | c | d | \neg |
|------|----------|---|---|---|---|--------|
| S' | 0 | | | | | |
| S | | 1 | 1 | 2 | 2 | |
| Q | | 3 | 4 | | | |
| R | | | | 5 | 6 | |



6.5 Parse Tree

Consider the following grammar and predict table:

- (0) $S' \rightarrow \vdash S \dashv$
- (1) $S \rightarrow T Z$
- (2) $Z \rightarrow + T Z$
- (3) $Z \rightarrow \varepsilon$
- (4) $T \rightarrow F T'$
- (5) $T' \rightarrow * F T'$
- (6) $T' \rightarrow \varepsilon$
- (7) $F \rightarrow a$
- (8) $F \rightarrow b$
- (9) $F \rightarrow c$

| | \vdash | a | b | c | + | * | \dashv |
|------|----------|---|---|---|---|---|----------|
| S' | 0 | | | | | | |
| S | | 1 | 1 | 1 | | | |
| Z | | | | | 2 | | 3 |
| T | | 4 | 4 | 4 | | | |
| T' | | | | | 6 | 5 | 6 |
| F | | 7 | 8 | 9 | | | |

Comment 6.8

We can prove (with a lot of work) that the string $\vdash a * b + c \dashv$ is in the language.

Discovery 6.7

The derivation is the sequence of rules that were applied in that order:

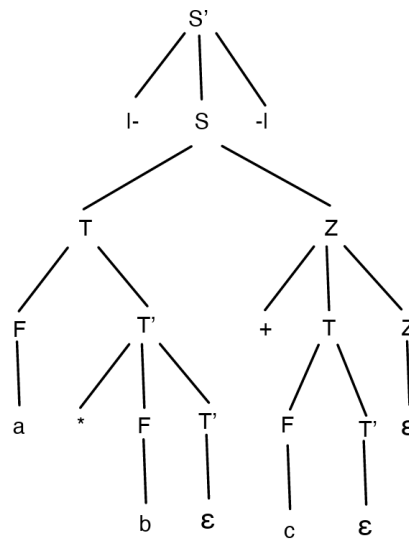
Rules 0, 1, 4, 7, 5, 8, 6, 2, 4, 9, 6 and 3

Code 6.7

A parser can use the derivation to generate the parse tree.

Example 6.8

We show the derivation and the corresponding parse tree below:

$$\begin{aligned}
 S' &\Rightarrow \vdash S \dashv && (\text{rule 0}) \\
 &\Rightarrow \vdash T Z \dashv && (\text{rule 1}) \\
 &\Rightarrow \vdash F T' Z \dashv && (\text{rule 4}) \\
 &\Rightarrow \vdash a T' Z \dashv && (\text{rule 7}) \\
 &\Rightarrow \vdash a * F T' Z \dashv && (\text{rule 5}) \\
 &\Rightarrow \vdash a * b T' Z \dashv && (\text{rule 8}) \\
 &\Rightarrow \vdash a * b Z \dashv && (\text{rule 6}) \\
 &\Rightarrow \vdash a * b + T Z \dashv && (\text{rule 2}) \\
 &\Rightarrow \vdash a * b + F T' Z \dashv && (\text{rule 4}) \\
 &\Rightarrow \vdash a * b + c T' Z \dashv && (\text{rule 9}) \\
 &\Rightarrow \vdash a * b + c Z \dashv && (\text{rule 6}) \\
 &\Rightarrow \vdash a * b + c \dashv && (\text{rule 3})
 \end{aligned}$$


6.6 Limitations of LL(1) Grammars

Theorem 6.2

Reasoning based on the definitions of Nullable, First and Follow, we can conclude that a grammar is LL(1) if and only if:

- No two distinct rules with the same LHS can generate the same first terminal.
- No Nullable symbol A has the same terminal a in both its First and Follow sets.
- There is only one way to derive ε from a Nullable symbol.

Question 6.5.

The following grammar is not LL(1).
Give reasoning to prove this claim.

- (1) $S \rightarrow S + T$
- (2) $S \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5, 6, 7) $F \rightarrow a \mid b \mid c$

Comment 6.9

This was the grammar we constructed in the previous module to respect the BEDMAS order of operations (giving higher precedence to multiplication over addition by making it deeper in the tree).

Solution: We have

$$\{a\} \subseteq First(F) \subseteq First(T)$$



Theorem 6.3

Left recursive grammars are **never** LL(1).

Proof. Exercise?



Definition 6.4: Left Recursive Grammar

To recall, a **left recursive grammar**, as the name indicates, is a grammar where the recursion on a nonterminal happens on the left within the right-hand side.

6.6.1 Writing Right Recursive Grammars

We can simply switch the recursion to the right. This would change the parse tree, and thus the meaning, of any input string, so it's not a good solution, but let's at least explore whether it works:

- (1) $S \rightarrow T + S$
- (2) $S \rightarrow T$
- (3) $T \rightarrow F * T$
- (4) $T \rightarrow F$
- (5, 6, 7) $F \rightarrow a \mid b \mid c$

Question 6.6.

Is the grammar presented above LL(1)?

Solution: The grammar is still not LL(1). $\text{Predict}[S][a]$ contains both rules for S .



Discovery 6.8

The problem here is that the left-hand side of the two rules for S have a common prefix (T).

Theorem 6.4

A grammar with two or more rules for the same nonterminal with a common left prefix of length k , cannot be LL(k).

Comment 6.10

As discussed, one solution is to increase the lookahead. The grammar above is LL(2). Note that we don't provide an algorithm for generating the predict table for an LL(2) parser, because LL(2) parsing is very rarely used in practice and generating such predict tables is much more complex.

Code 6.8

If we want to continue with the LL(1) algorithm, we can try *left factoring*.

Definition 6.5: Left Factoring

Suppose a grammar has rules $A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n$, all with a common prefix of $\alpha \neq \epsilon$ on the right-hand side. Then, we can change these to the following equivalent grammar by left factoring:

$$\begin{aligned} A &\rightarrow \alpha B \\ B &\rightarrow \beta_1 \mid \dots \mid \beta_n \end{aligned}$$

Example 6.9

Applying this technique to the previous example:

| Left recursive: | Right recursive: | Right recursive factored: |
|---|---|---|
| (1) $S \rightarrow S + T$ | (1) $S \rightarrow T + S$ | (1) $S \rightarrow T X$ |
| (2) $S \rightarrow T$ | (2) $S \rightarrow T$ | (2, 3) $X \rightarrow + S \mid \varepsilon$ |
| (3) $T \rightarrow T * F$ | (3) $T \rightarrow F + T$ | (4) $T \rightarrow F Y$ |
| (4) $T \rightarrow F$ | (4) $T \rightarrow F$ | (5, 6) $Y \rightarrow * F \mid \varepsilon$ |
| (5, 6, 7) $F \rightarrow a \mid b \mid c$ | (5, 6, 7) $F \rightarrow a \mid b \mid c$ | (7, 8, 9) $F \rightarrow a \mid b \mid c$ |

Code 6.9

An alternate approach to converting a left-recursive grammar to be right recursive is to apply a different transformation: replace each pair of rules $A \rightarrow A\alpha$ and $A \rightarrow \beta$ where β does not begin with the nonterminal A with the rules:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example 6.10

Using this on our grammar we get:

| Left recursive: | Right recursive: |
|---|---|
| (1) $S \rightarrow S + T$ | (1) $S \rightarrow T Z'$ |
| (2) $S \rightarrow T$ | (2, 3) $Z' \rightarrow + T Z' \mid \varepsilon$ |
| (3) $T \rightarrow T * F$ | (4) $T \rightarrow F T'$ |
| (4) $T \rightarrow F$ | (5, 6) $T' \rightarrow * F T' \mid \varepsilon$ |
| (5, 6, 7) $F \rightarrow a \mid b \mid c$ | (7, 8, 9) $F \rightarrow a \mid b \mid c$ |

Discovery 6.9

However, there is a problem with both of our right recursive LL(1) grammars: they are right associative.

Result 6.1

The Top-Down parsing algorithm we have studied, LL(1), is not compatible with left-recursive grammars. We discuss an alternate, even more powerful, parsing algorithm in the next module.

7 Bottom-Up Parsing

Briefly, in bottom-up parsing, we start at the input string (which is a sequence of terminals in the grammar) and make our way towards the start symbol for the grammar.

7.1 Bottom-Up Parsing, Informally

Since we start at the input string, the algorithm goes as follows: begin reading input symbols one character at a time, left to right. If we recognize the right-hand side of a rule, replace it with its left-hand side.

Definition 7.1: Shift

A **shift** consumes the next input symbol and pushes it on the stack.

Definition 7.2: Reduce

A **reduce** pops the right-hand side of a rule off the stack and pushes its left-hand side. (This includes rules where the right-hand side is empty, e.g. $A \rightarrow \varepsilon$, in which case the end result is that the left-hand side nonterminal is pushed).

7.2 Bottom-Up Parsing, Less Informally

Comment 7.1

The vagueness of our algorithm comes from the fact that we have not formalized how to determine when the top n symbols of the stack represent the right-hand side of a rule. There is a beautiful theorem dictating when to reduce.

We begin by developing an intuition before discussing the theory.

Example 7.1

Consider the following grammar:

- (1) $S' \rightarrow \vdash E \dashv$
- (2) $E \rightarrow E + T$
- (3) $E \rightarrow T$
- (4) $T \rightarrow \text{ID}$

Definition 7.3: Item

An **item** is a rule with a bookmark (represented as a \bullet) somewhere on the right-hand side.

The item $E \rightarrow \bullet E + T$ is called a *fresh item*, indicating that none of the right-hand side is on the stack. If the algorithm was to push an E on the stack, we would update the fresh item to get the new item, $E \rightarrow E \bullet + T$. This tells us that E is on the stack. If we then push $+$ and T , we get

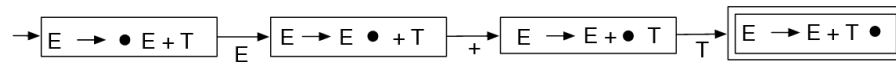
the updated item $E \rightarrow E + T\bullet$. This last item called *reducible*, since the bookmark indicates that the entire right-hand side of the rule is on the stack, meaning we can reduce.

Comment 7.2

If the above was hard to follow, let's reword the idea. Initially, none of the right-hand side for this rule is on the stack. Once we push E , we *transition to a state* where the bookmark had been moved past E . Similarly, after pushing $+$ and T , we *transition* the bookmark forward until we reach a state where the bookmark is at the end of the right-hand side. At this point, we are willing to *accept* that the right-hand side is all on the stack.

Discovery 7.1

We are treating the positions of the bookmark as states of a DFA, and are transitioning between states on the symbol that gets pushed on the stack. The diagram below captures this:



Algorithm 7.1

The following steps can be used to produce the DFA directly, without first constructing an NFA:

1. Create a start state with a fresh item for the rule with the start symbol on the left-hand side. (We are assuming the grammar is augmented, so that there is a unique such rule.)
2. Select a state q_i that contains at least one non-reducible item. For each non-reducible item in q_i , create a transition to a new state q_j on the symbol X that follows the bookmark. Take all items from q_i where the bookmark is followed by an X , update the bookmark to be right after X , and add these updated items as items in q_j .
 - For each newly created state, and each item in the state, if the symbol following the updated bookmark is a nonterminal, say A , add fresh items for all rules with A on the left-hand side to the new state. If this creates fresh items where a bookmark is followed by a nonterminal, say B , add fresh items for all rules with B on the left-hand side. Repeat until no more items are created.
 - If the above process results in a state that **already exists** (that is, there is already another state with the exact same set of items in it), discard the new state, and instead connect q_i to the existing state. This avoids creating duplicate states with the same set of items (although creating duplicate states does not make the DFA incorrect, it is just wasteful).
3. Repeat step 2 until no new states are discovered.
4. Mark states containing reducible items as accept states.

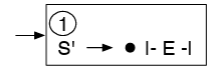
Comment 7.3

The L stands for Left-to-right scan of the input, the R for Rightmost derivation, and the 0 is the number of symbols we look ahead in the input to make parsing decisions (whether to shift or reduce, and which rule to reduce by).

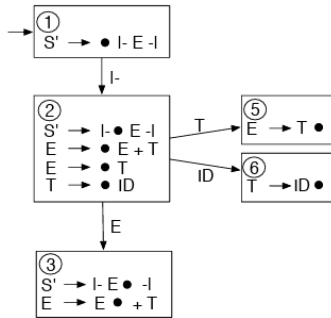
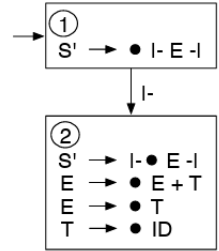
We show this DFA construction algorithm through a step-by-step construction below. We number each state so that the description can refer to the numbered states. Let's use the grammar discussed above (grammar reproduced for convenience):

- (1) $S' \rightarrow \mid E \mid$
- (2) $E \rightarrow E + T$
- (3) $E \rightarrow T$
- (4) $T \rightarrow ID$

We begin by creating a fresh item for rule 1.
This creates state 1 shown on the right.

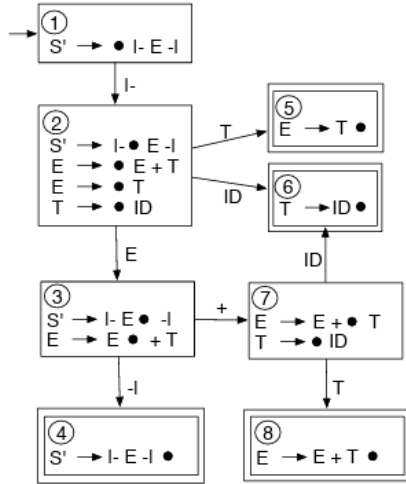
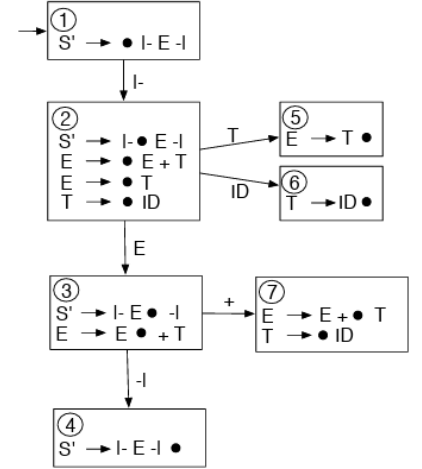


In step 2, we select state 1 since it has a non-reducible item. We create state 2. We have updated the item from State 1 to: $S' \rightarrow \bullet E \mid$. Notice how, since the bookmark now precedes a nonterminal E in state 2, we added two fresh items: $E \rightarrow \bullet E + T$, $E \rightarrow \bullet T$. Since this introduced a new fresh item where the bookmark is followed by a nonterminal T , we added the fresh item for the rule for T , i.e., $T \rightarrow \bullet ID$.



Still in step 2, we select state 2 since it has multiple non-reducible items. There are two items that have the bookmark right before the nonterminal E . We create a transition to state 3 on symbol E . In state 3, these two items have been updated by transitioning the bookmark past E . Also, since state 2 has a non-reducible item where the bookmark is followed by T , we create state 5 with the reducible item: $E \rightarrow T \bullet$. Similarly, state 2 also had the non-reducible item: $T \rightarrow ID$. which leads us to create state 6, with the reducible item: $T \rightarrow ID \bullet$.

Still in step 2, the only state still to be processed is state 3 (since state 5 and 6 do not have non-reducible items). In processing state 3, we create state 4 that transitions the item: $S' \rightarrow E \bullet$ – to the reducible item: $S' \rightarrow E - \bullet$. Also, from state 3 we transition to state 7 which updates the item: $E \rightarrow E + T \bullet$ to $E \rightarrow E + \bullet T$. Since this has created an item where the bookmark is followed by a nonterminal, we add fresh items for the rules for that nonterminal. In this case, the fresh item: $T \rightarrow \bullet ID$. was added.



Still in step 2, this time we only have state 7 to process. We create state 8 by transitioning the T symbol to produce the reducible item: $E \rightarrow E + T \bullet$ in state 8. For the other non-reducible item: $T \rightarrow ID$ in state 7, we need a state with the item: $T \rightarrow ID \bullet$. We can reuse the existing state 6 for this. Alternatively, we could have created a new state 9 with this reducible item, but there is no need for this duplicate state. There are no more new states that contain non-reducible items, so step 2 and 3 are complete. Step 4 requires adding accept states to those states that have a reducible item. For our example, these are states 4, 5, 6, and 8.

Definition 7.4: LR(0) Parsing DFA

The DFA created is called the **LR(0) Parsing DFA**.

Definition 7.5: LR(0) Parsing Algorithm

The algorithm is known as **LR(0) Parsing Algorithm**.

7.3 ATrace Through The LR(0) Algorithm

Let's look at the first few steps of the algorithm in action. We will use the same grammar and LR(0) DFA for this grammar that we just constructed and parse the input string $\vdash ID + ID \neg$. The action column is determined by running the current contents of the stack through the LR(0) DFA and choosing the action based on the state we end up in:

| Read | Unread | Stack | Action |
|----------------------|---------------------------------------|---------------------|--|
| ϵ | $\vdash \text{ID} + \text{ID} \dashv$ | | Running stack through DFA results in state 1. Shift \vdash |
| \vdash | $\text{ID} + \text{ID} \dashv$ | \vdash | Running stack through DFA results in state 2. Shift ID |
| $\vdash \text{ID}$ | $+ \text{ID} \dashv$ | $\vdash \text{ID}$ | Running stack through DFA results in state 6. Reduce rule 4. |
| $\vdash \text{ID}$ | $+ \text{ID} \dashv$ | $\vdash \text{T}$ | Running stack through DFA results in state 5. Reduce rule 3. |
| $\vdash \text{ID}$ | $+ \text{ID} \dashv$ | $\vdash \text{E}$ | Running stack through DFA results in state 3. Shift $+$. |
| $\vdash \text{ID} +$ | $\text{ID} \dashv$ | $\vdash \text{E} +$ | and so on. |

Discovery 7.2

We have the algorithm formalized below. In the pseudocode, **Reduce[state]** is a function that, given a state, tells us whether this is a reduce state and, if yes, it gives the rule that is applicable.

Code 7.1

Algorithm 12: LR(0) algorithm, input LR(0) DFA $(\Sigma, Q, q_0, \delta, A)$

```

1 stateStack.push  $q_0$ ;
2 for each symbol  $a$  in  $\vdash x \dashv$  from left to right do
3   while Reduce[stateStack.top] is some production  $B \rightarrow \gamma$  do
4     symStack.pop symbols in  $\gamma$ ;
5     stateStack.pop  $|\gamma|$  states;
6     symStack.push  $B$ ;
7     stateStack.push  $\delta[\text{stateStack.top}, B]$ ;
8   symStack.push  $a$ ;
9   reject if  $\delta[\text{stateStack.top}, a]$  is undefined;
10  stateStack.push  $\delta[\text{stateStack.top}, a]$ ;
11 accept;

```

Comment 7.4


Notice line 10. If the top of the state stack tells us that we cannot reduce, we should try to shift, but this can cause a parse error if the LR(0) DFA does not define a transition from the current state on the next input symbol (a in the algorithm).

Question 7.1.

For the grammar and LR(0) DFA discussed above, show that $\vdash + \dashv$ will cause a parser error.

Solution:

| Read | Unread | Symbol Stack | State Stack | Action |
|------------|-------------------|--------------|-------------|--|
| ϵ | $\vdash + \dashv$ | | 1 | State 1 is a shift state. Shift \vdash |
| \vdash | $+ \dashv$ | \vdash | 1 2 | State 2 is a shift state. Shift $+$ |

While we can push $+$ on to the Symbol stack there is no transition defined from the current state 2 on the symbol we just pushed, $+$. The parser reports a parse error. 

7.3.1 Optional: LR(0) Formalism

Above, we discussed how we can use the LR(0) DFA to guide when to reduce and which rule to reduce with. This stems from a theorem by Knuth (On the Translation of Languages from Left to Right, 1965) which we discuss very briefly for the interested reader.

Comment 7.5

Those who wish for a little more detail are recommended to take CS 444. Those wishing for a lot more detail can look at CS462.

Definition 7.6: Sentential Form

Given a grammar $G = (N, T, P, S)$, γ is a **sentential form** if $S \Rightarrow^* \gamma$.

Discovery 7.3

Note that γ is not necessarily a string of terminals; it is a form that can be derived by repeated application of rules starting at the start symbol. It may include both terminals and nonterminals.

Definition 7.7: Viable Prefix

We say α is a **viable prefix** if it is the prefix of a sentential form and the remainder of the sentential form after α is terminals, i.e., there exists some string of terminals y such that $S \Rightarrow^* \alpha y$.

Discovery 7.4

In simpler terms, a viable prefix, when concatenated with some string of terminals, forms a sentential form, i.e., it can be derived from the start symbol.

Theorem 7.1: Knuth

The crux of Knuth's theorem is that the set of viable prefixes, configurations that have the viability to lead to a successful derivation, is itself a regular language.

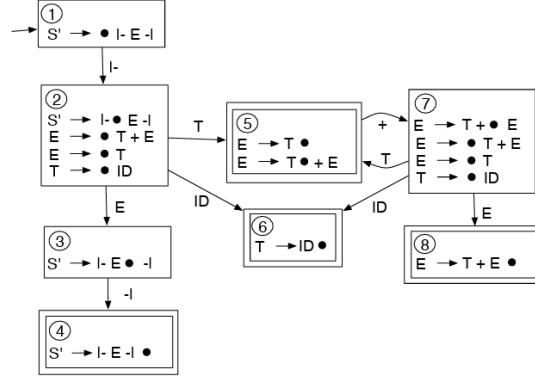
7.4 Action Conflicts

While many grammars can be parsed using the LR(0) parsing algorithm, there exist grammars that are not LR(0) (meaning they cannot be parsed by the LR(0) algorithm). We discuss one such example and then define *parsing conflicts*.

Example 7.2

Consider the following grammar and the associated LR(0) DFA. Notice that the grammar is similar to the grammar above except that it is right-recursive in rule 2.

- (1) $S' \rightarrow \mid E \mid$
- (2) $E \rightarrow T + E$
- (3) $E \rightarrow T$
- (4) $T \rightarrow ID$



Consider state 5. State 5 has two items, one of which, $E \rightarrow T \bullet$, is reducible. According to our algorithm, if we have a reducible item at a state, we should reduce using the rule for that item. But state 5 also has a non-reducible item, $E \rightarrow T \bullet + E$, which suggests that shifting the next symbol is also a feasible choice.

Result 7.1

The algorithm we have discussed will not be able to decide which of the two actions is the right decision, since it does not look at the next input symbol.

Definition 7.8: Shift-Reduce Conflict

The conflict in this DFA is called a **shift-reduce conflict**.

Discovery 7.5

A **shift-reduce conflict** occurs when a state in the parsing DFA has two items of the form $A \rightarrow \alpha \bullet a \beta$ and $B \rightarrow \gamma \bullet$ where, as always, $a \in T'$ is a terminal and $\alpha, \beta, \gamma \in V^*$ are strings over the vocabulary (strings that may mix terminals and nonterminals).

Definition 7.9: Reduce-Reduce Conflict

A **reduce-reduce conflict** occurs when a state in the parsing DFA has two items of the form $A \rightarrow \alpha \bullet$ and $B \rightarrow \beta \bullet$ where $\alpha, \beta \in V^*$.

Definition 7.10: LR(0)

We say that a grammar is **LR(0)** if and only if the LR(0) automaton for the grammar does not have any shift-reduce or reduce-reduce conflicts.

Comment 7.6

Based on this definition, the right recursive grammar discussed above is not LR(0). In the next section, we discuss how we can increase the power of bottom-up parsing algorithms by looking ahead at the next input symbol.

7.5 Using Lookaheads

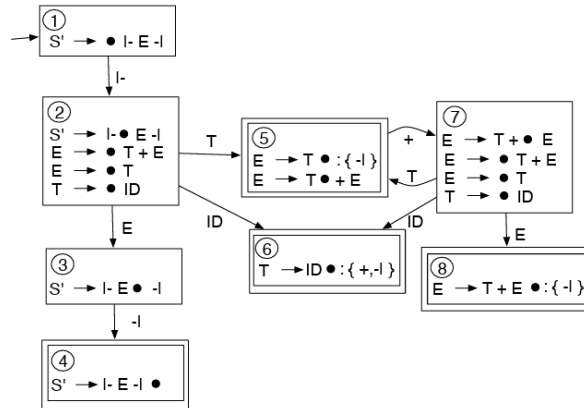
Definition 7.11: Simple LR(1)

Simple LR(1), or **SLR(1)** for short, uses one symbol of lookahead and makes a decision on whether to reduce based on whether the lookahead symbol is in the *Follow* set for the left-hand side nonterminal.

In SLR(1), we extend the LR(0) DFA by extending our definition of *items*: for each reducible item, add the *Follow* set for the left-hand side nonterminal to the item as the *lookahead tag*.

Comment 7.7

In our example, this creates the items $E \rightarrow T + E \bullet : \{-\}$, $E \rightarrow T \bullet : \{-\}$ and $T \rightarrow ID \bullet : \{+, -\}$. This isn't needed for non-reducible items (shift items), as the lookahead is evident from the item: it's the terminal that comes after the bookmark.



Question 7.2.

Create the SLR(1) DFA for this grammar. Are there any Reduce-Reduce conflicts? Are there any Shift-Reduce conflicts? Is the grammar SLR(1)?

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow Aa$
- (3) $S \rightarrow bAc$
- (4) $S \rightarrow dc$
- (5) $S \rightarrow bda$
- (6) $A \rightarrow d$

Solution: There is reduce-reduce conflict but no shift-reduce conflict. This is by definition not SLR(1). 🎵

7.5.1 Optional: Formally defining LR(1) and SLR(1)

In our previous optional section, we formalized LR(0) parsing and defined the **Reduce** set as,

$$\text{Reduce}(\alpha) = \{A \rightarrow \gamma : \alpha = \beta\gamma \text{ and } \beta A \text{ is a viable prefix}\}$$

that can be used to reduce α , the current contents of the stack.

How does LR(1) work? In LR(1) we strengthen our definition of **Reduce** to be:

$$\text{Reduce}(\alpha, a) = \{A \rightarrow \gamma : \alpha = \beta\gamma \text{ and } \beta Aa \text{ is a viable prefix}\}$$

The a is the look ahead symbol. In other words, now the Reduce set contains the rules we should use to reduce given that we know that a is the next input symbol and with the condition that after the reduction βAa continues to be a viable prefix.

What does SLR(1) do? SLR(1) is a compromise between LR(0) and LR(1). Instead of checking that βAa is a viable prefix, we define the SLR(1) **Reduce** set to be:

$$\text{Reduce}(\alpha, a) = \{A \rightarrow \gamma : \alpha = \beta\gamma \text{ and } \beta A \text{ is a viable prefix and } a \in \text{Follow}(A)\}$$

In other words, we now add the condition that we will only consider reducing using a rule if the condition for LR(0) holds as well as that the next input symbol is in the Follow set of the nonterminal whose rule we are considering.

Discovery 7.6

Note that this is a heuristic. If we know that βAa is a viable prefix (LR(1) requirement), this implies that βA is a viable prefix and $a \in \text{Follow}(A)$.

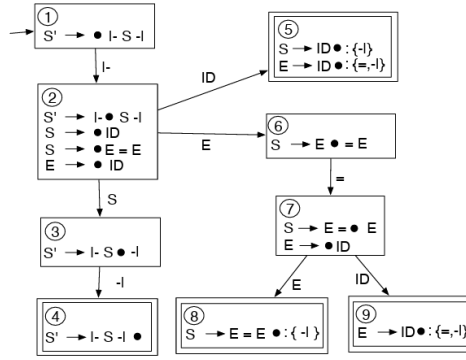
Comment 7.8

The reverse is not necessarily true.

7.6 SLR(1), LALR(1) and LR(1)

Example 7.3: Limitations of SLR(1)

$S' \rightarrow \vdash S \dashv$
 $S \rightarrow ID$
 $S \rightarrow E = E$
 $E \rightarrow ID$
 $\text{Follow}(S) = \{ \dashv \}$
 $\text{Follow}(E) = \{ =, \dashv \}$



Consider state 5, notice that the intersection of these two Follow sets is not empty, which poses a problem.

Comment 7.9

This example shows a reduce-reduce conflict, but note that shift-reduce conflicts in SLR(1) are also possible.

Discovery 7.7

This example highlights the limitation of SLR(1): just because a symbol is in the follow set of a nonterminal does not mean that there is a viable derivation if we rely on just that information.

Definition 7.12: LR(1) DFA

The **LR(1) DFA** is created by only adding a subset of the Follow set to each reducible item, corresponding to the lookaheads for which a particular rule is valid to reduce by.

Comment 7.10

While LR(1) is very powerful, it is hardly ever used in practice. The primary reason is that the number of states in the resulting LR(1) DFA can be exponentially larger than the LR(0) DFA, depending on the grammar.

Discovery 7.8

By contrast, the SLR(1) DFA uses the same number of states as LR(0).

Result 7.2

An even better compromise than SLR(1) is to use a construction called LALR(1)^a, short for Lookahead LR(1).

^aWe do not discuss LALR(1) in this course, but the basic idea is to merge states of the LR(1) DFA that only differ in terms of their lookaheads, and hope that this does not lead to conflicts.

7.7 But Where is my Parse Tree

Algorithm 7.2

Recall that the symbol stack we have used in our parsing algorithms contained symbols (terminals, nonterminals). We can change this stack to be a stack of tree nodes! In other words, instead of pushing and popping symbols, we will be pushing and popping leaf nodes, subtrees and eventually the full parse tree.

Question 7.3.

- (1) $S' \rightarrow \vdash S \dashv$
- (2) $S \rightarrow AyB$
- (3) $A \rightarrow ab$
- (4) $A \rightarrow cd$
- (5) $B \rightarrow z$
- (6) $B \rightarrow wx$

Create the LR(0) DFA.

Given the input string: $\vdash a b y w x \dashv$, give a step by step construction of the parse tree showing Read, Unread, Tree Stack, State Stack and Action columns like the examples in this module.

7.8 Summary

Not all rules in a programming language can be enforced through a context-free grammar. Things like checking that a variable used has been previously defined, and a value assigned to a variable has the right type, requires **context-sensitive analysis**, where context does matter. In the next module, we will talk about such analyses.

8 Context-Sensitive Analysis

In this module, we discuss the *context-sensitive analysis* requirements for a small programming language. Context-sensitive analysis is also known as *semantic analysis* because it involves detecting errors in the semantics (meaning) of the program, as opposed to the syntactic (structural) errors detected during parsing.

8.1 Intro to WLP4

In this section, we give an introduction to a small language named WLP4 and then discuss the context-sensitive analyses needed for this language.

Code 8.1

WLP4 programs are a sequence of C++ functions that could be executed if an appropriate main function was provided. Instead of a main function, valid WLP4 programs must have a special `wain` procedure. This is enforced by the context-free grammar, i.e., a WLP4 program will not pass through the parsing stage unless an appropriate `wain` function has been defined.

A procedure in WLP4 (except for `wain`) can have an arbitrary, but fixed, number of parameters. Parameters, and other variables, can have one of two types: `int` or `int*`, i.e., an integer type or a pointer to integer type. The `wain` procedure must always have two parameters, the second of which must be an `int`. This restriction corresponds to the input requirements for our MIPS emulators `mips.twoints` and `mips.array`, both of which provide two arguments to the program in `$1` and `$2`, and both of which have a number as the second argument. Once we compile a WLP4 program to MIPS, we can run it with one of these two emulators.

The language supports `while` loops, an `if` statement (an `else` clause, even if empty, is always required), and input and output through the `getchar` built-in, `println` and `putchar`. A specialized `return` statement is also supported which must appear only once in a procedure as the last statement. This requirement is also enforced by the context-free grammar, i.e., an attempt to write a procedure with more than one `return` statement in the procedure or without a `return` statement as the last statement in the procedure will lead to a *parser error*. The language also supports using heap memory through a `new` expression and a `delete` statement.

8.2 Context-Sensitive Analyses in WLP4

Comment 8.1

As discussed earlier, not all language rules can be enforced by a context-free grammar (and therefore a parser).

Example 8.1: Rules regards to types

Perhaps the most important of these are rules regarding types; in a statically typed language, we must ensure that type rules are not violated.

Comment 8.2

Type checking is a complicated process, so we will defer our discussion of it until later in the module.

Example 8.2: Rules regards to variables

Another set of rules that must be enforced are those surrounding variables. Two rules that are common in many languages (including WLP4) are:

1. We cannot declare more than one variable with the same name in the same scope.
2. A variable cannot be used before it has been declared.

Theorem 8.1

In theory, we could use context-sensitive languages (a language more powerful than context-free languages) and the corresponding context-sensitive grammars to formally define such rules.

Comment 8.3

However, a much easier and more practical option is to use a code-based solution to traverse the parse tree, gather information, and enforce rules that require context-sensitive information. This means that it is worth taking some time to set up infrastructure to enable easy traversal of the parse tree.

8.2.1 Tree Traverse Examples

The following is a sketch of what tree traverse could look like in C++.

Code 8.2

```
1  class Node {
2      public:
3          string rule; // e.g. expr expr PLUS term
4          vector<Node*> children;
5  };
6
7  An example of a traversal routine that visits every node would look like:
8
9  void doSomething(const Node *tree){
10     // do something before visiting children
11
12     for(Node *child: tree->children){
13
14         // visit the children
15         doSomething(child);
```

```

16     }
17
18     // do something after visiting children
19 }

```

There are other ways to traverse the tree. For example, rather than visiting every node, you may want to search the tree for specific kinds of nodes, and then process those nodes in a different way than just continuing the traversal.

Code 8.3

For example, the following function `findDeclarations` recursively searches for all nodes with the rule `dcl → type ID` (corresponding to a variable declaration). Once such a node is found, the node is passed to another helper function called `processDeclaration` (not defined here).

```

1  void findDeclarations(const Node *tree) {
2      if (tree->rule == "dcl type ID") {
3          processDeclaration(tree);
4      } else {
5          for (Node *child: tree->children) {
6              findDeclarations(child);
7          }
8      }
9  }

```

Discovery 8.1

Note the use of if/else to ensure that after finding a declaration node, the `findDeclarations` function does not further explore the children of that node (which would be useless and inefficient, because a declaration subtree cannot contain another declaration subtree as a descendant).

Definition 8.1: Concrete Syntax Tree

The parse tree created by a parser is often called a **concrete syntax tree**.

Often, this parse tree is passed through a tree transformation stage before applying context-sensitive analyses.² During this stage, the tree is pruned by removing useless nodes such as those needed to ensure that the grammar is unambiguous and to satisfy the requirements of a specific parsing algorithm.

Definition 8.2: Abstract Syntax Tree

The transformed tree is called an **abstract syntax tree**, or AST.

8.3 Catching Identifier Errors

There are two types of identifiers in WLP4 programs: variable and procedure names. The rules surrounding both types of identifiers are the same. We talk about variables first and then discuss procedures.

We first consider the following example: Should the WLP4 compiler generate an error for the code shown below on the left?

Code 8.4

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

The answer to the above question is no. The compiler should not produce an error since this code is legal. In particular, even though variable `x` is defined twice in this program, it is defined only once within each procedure. The rule regarding variables says that we cannot have duplicates within a procedure. This means that we will need to create a separate symbol table for each procedure since, while MIPS labels were global, variables in WLP4 are local to the scope they are declared in.

Result 8.1

The context-sensitive analysis (our traversal) can compute this as a single data structure by mapping each procedure name (the key) to its symbol table (the value), e.g.,

```
map<string, map<string, string>> tables
```

where the key for the inner table is the variable name and the value the type defined for this variable stored as a string.

Comment 8.4

However, it is more ideal to use a better datatype than string for storing variable type information.

8.3.1 More Examples in WLP4

Example 8.3

Let's now look at some other examples. Is the following program a valid WLP4 program?

Code 8.5

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

No, this program is not a valid program and should cause an error; the variable `x` is **not** defined in procedure `wain`. This means that while detecting duplicates we will need to ensure that we are looking for variables using the symbol table for that procedure (things would be a little complicated if WLP4 allowed global variables).

Example 8.4

Should this code compile?

Code 8.6

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

While the variables are fine (only one *x* is defined in each scope), there is a different problem. There are two procedures with the same name, *f*. The program to the left is not a valid WLP4 program since it has a duplicate function. WLP4 requires that all procedure names be unique. This means that the analysis needs to check that all functions in the program have unique names.

Example 8.5

Is the following a valid WLP4 program?

Code 8.7

```
int f() {
    int f = 1;
    return f + 1;
}
int g(int g) {
    return g - 1;
}
int wain(int a, int b) {
    return a;
}
```

This is indeed a valid WLP4 program. The interesting thing to note is that it is legal to have variables that have *the same name as any procedure*. In fact, the WLP4 specification clearly addresses this and states that if a variable *x* is declared in a procedure *p*, all occurrences of *x* within *p* refer to variable *x*, even if a procedure named *x* has been declared. The code on the left is the extreme version, where the procedure *f* has a local variable also named *f* and procedure *g* has a parameter named *g*. The specification talks about this special case and states that all occurrences of the identifier refer to the variable and not the procedure.

Example 8.6

The above suggests this interesting scenario:

Code 8.8

```
int p(int p) { return p(p); }
```

As per the WLP4 specification, all occurrences of the identifier *p* are treated as a variable and not a procedure even though a procedure with that name exists. This means that the code above should **not** compile.

Having looked at these examples, let's now discuss how we can enforce the context-sensitive rules for WLP4 using tree traversals. We begin with a traversal that starts at the root of the parse tree. We must determine when we have encountered a tree node that represents a procedure.

Discovery 8.2

Given our rudimentary structure (a single `Tree` class) this means we must determine when we are at a tree node whose rule field matches a rule for a procedure. Referring to the [context-free grammar for WLP4](#), a tree node that has one of the following rules represents a new procedure:

```
procedure → INT ID LPAREN...
main → INT WAIN...
```

At this point, we can check for duplicate procedures; if our master map of all symbol tables (we called it `tables` above) already contains a mapping with the same name as the name of the procedure we just discovered, we know we have encountered a duplicate procedure and can generate an appropriate error. If this procedure is not a duplicate, we need to create a new symbol table for this procedure and store it in our master map. We can then continue with the traversal inside the procedure.

Algorithm 8.1

It is a good idea to make a note of the currently active symbol table, i.e., the symbol table for the procedure we are traversing. Ideally, the compiler would contain some extra infrastructure to house these symbol tables. Perhaps a class that aggregates all the symbol tables and also maintains a note of the currently active symbol table.

Comment 8.5

However, since WLP4 is simple enough, a global variable storing the currently active symbol table would also suffice.

Within procedures, variables are either declared in the parameter list of a procedure or at the start of the procedure's body. Referring to the context-free grammar for WLP4, we can observe that we only need to look for parse tree nodes where the rule is `decl → type ID`. Populating the symbol table for each such declaration is simply a matter of retrieving the lexeme for the `ID` token (child 2) and the actual type from the `type` child (child 1).

8.3.2 Checking Variable Use

Theorem 8.2

In WLP4, the context-free grammar enforces that all variables must be declared at the top of the procedure, before any statements for the procedure.

Result 8.2

This means that two passes are not needed. By the time our first traversal reaches the **statements** child of the following two rules, the symbol table for this procedure is already complete, since the **params** and **dcls** children of the tree node have already been visited.

```
procedure → INT ID LPAREN params RPAREN
          LBRACE dcls statements RETURN expr SEMI RBRACE
main → INT WAIN LPAREN dcl COMMA dcl
      RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE
```

This means that we can check our “declaration before use” rule by simply traversing the statements subtree and the return expression looking for rules that use variables (we will talk about function calls later).

Discovery 8.3

The two rules of interest are **factor** → ID and **lvalue** → ID. The actual check is straightforward; when we encounter a node that represents one of these rules, retrieve the lexeme for the ID token and check that an entry with this name exists in the symbol table for the current procedure.

8.3.3 Checking Procedure Calls

Like variables, procedures must also be declared before use. This means that it is in fact incorrect to first collect a list of all procedures that were declared in a program and then check that each procedure call matches the name of a declared procedure; **WLP4** explicitly disallows calling a procedure that is declared later in the code.

Result 8.3

This means that the analysis must validate the names of procedures being called **during** the creation of the top-level symbol table (that we have been calling **tables**).

Discovery 8.4

For procedure calls, the rules of interest are **factor** → ID LPAREN RPAREN (call without arguments) and **factor** → ID LPAREN arglist RPAREN (call with arguments).

During the traversal, when a parse tree node with one of these rules is encountered, there are two steps:

1. First, check the symbol table for the current procedure to ensure the ID does not refer to a *variable*. As discussed earlier, if a local variable has the same name as a procedure, the variable takes priority. And you cannot “call” a variable in **WLP4**, so an error should be produced in this case.

2. If the above check passes, check the top-level symbol table to see if the ID refers to a previously declared procedure. If it does not, this is an error.

Calling a procedure also requires passing the correct number of arguments to the procedure. Additionally, the types of the arguments must match the parameter types.

Definition 8.3: Signature

When a new procedure declaration is discovered during the traversal we should store the procedure's **signature**, which is the information needed to (correctly) call it.

Discovery 8.5

Note that all procedures in WLP4 return an integer value, i.e., procedures cannot return a pointer and there are no void types in WLP4. Therefore, the signature is simply a sequence representing the types for the parameters of that procedure. Continuing our simplification, we'll store the types as strings, but again, it would be more practical to use something better suited.

We can store signature information in a couple of different ways.

Code 8.9

1. One way could be to create a different map which maps the procedure name to a vector of strings where the strings represent the types for the parameters; the size of the vector will tell us how many parameters this procedure expects.
2. An alternate way is to update the definition of tables, the data structure which we had defined to map each procedure to its symbol table. We can now map the procedure name to a pair that contains a vector of strings as the first element, i.e., the signature for the procedure and the symbol table for the procedure as the second value. In C++, the definition would look like:

```
map<string, pair< vector<string>, map<string, string> >> tables;
```

signature *symbol table*

Comment 8.6

At this point though, the data structure has become overly cumbersome to work with. Rather than literally using this data structure, a better alternative would be to create a class called Procedure that stores the signature and local symbol table, with properly named accessor methods like getSignature and getVariableType, and have a map of these classes.

Code 8.10: Obtaining Signature Information

Obtaining the signature information is straightforward. The relevant subtree must either have the rule `params` $\rightarrow \varepsilon$, which indicates that this procedure has no parameters, or the rule `params` \rightarrow `paramlist`. If it is the latter, we must traverse the children which will have the rules `paramlist` \rightarrow `dcl` and `paramlist` \rightarrow `dcl` COMMA `paramlist` to retrieve the declarations of the parameters.

A simple loop or recursive helper function can be used to extract the type information from the `dcl` subtrees.

Example 8.7

As a concrete example, consider the following WLP4 program:

```
1  int f() {  
2      int *a = NULL;  
3      return 9;  
4  }  
5  int wain(int a, int b) {  
6      int x = 10;  
7      return x + a + b;  
8  }
```

The symbol table for `f` would contain a mapping of `a` \rightarrow `int*`. Similarly, the symbol table for the `wain` procedure would contain the following: `a` \rightarrow `int`, `b` \rightarrow `int` and `x` \rightarrow `int`. The signature for `f` is the empty vector, written as `[]` whereas the signature for `wain` is `[int, int]`.

8.4 Catching Type Errors

Given a sequence of bits, it is not possible to say what these bits represent. The same sequence of bits could represent an *unsigned integer*, a *signed integer in two's complement* representation, or an *ASCII character*.

Example 8.8

For example, consider the following WLP4/C/C++ snippet:

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

The type system prevents the assignment of the integer 7 to the variable `a`, since `a` is supposed to store addresses. Not preventing this would lead to disaster, e.g., we might later try to read from address 7.

Code 8.11

WLP4 only has two types: `int` and `int*`.

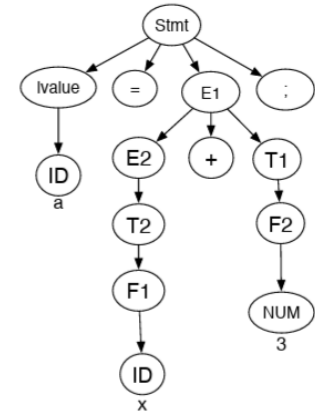
Comment 8.7

In CS 444, students implement a type system for a rather large subset of Java.

Recall that we have already collected type information for variables in each procedure. The question now is, how does one catch a type error? The diagram on the right shows the parse tree that would be generated by a WLP4 parser for the statement

`a = x + 3 ;`

The type system requires that the type for the left-hand side be the same as the type for the right-hand side. We must, therefore, determine the types for the left and right expressions.



Algorithm 8.2

The following is an admittedly hard-to-follow textual description of how we can determine the types for the left and right expressions.

The type of the left-hand side can be determined by traversing the left most child of `Stmt`, i.e., by determining the type for the `lvalue`. Since `lvalue` has one child, `ID`, the type for `lvalue` will be the type for `ID`. We would have already stored the type for this `ID`, the variable `a`, in the symbol table. Therefore, we can infer that the left-hand side has the type that we obtain for variable `a` from the symbol table.

We must also compute the type of the right-hand side. This type can be determined by traversing the third child, written as `E1` in the diagram. To compute this type, we must compute the type of the `Expr` labelled `E2`, the `Term` labelled `T1` and then apply the type system rule for addition. The type of `E2` is the type of its one child, the `Term` labelled `T2`. The type of `T2` is the type of its child, the `Factor` labelled `F1`. The type of `F1` is the type of its child, an `ID`. The type of this `ID`, the variable `x`, should already be in the symbol table. We have inferred that the type of `E2` is the type for the variable `x`. We must also similarly infer the type of `T1`. Using a similar approach, we determine that the type of `T1` is the type of `F2`, which is the type of `NUM` that, by definition, is `int`.

We now know what the types of the left and right expressions for addition are. At this point, we would use the type system rule for addition to determine the resulting type for adding variable `x` to an `int`. Notice that this would produce `int` if `x` is an `int` and `int*` if `x` is a pointer, i.e., pointer addition. This would give us the type for `E1`. At this point, we would be in a position to confirm that the left-hand side of the assignment has the same type as the right-hand side. If that is not the case, a type error would be generated.

Discovery 8.6

The above should convince the reader that to determine type correctness, we need a tree traversal. At any given node, we can recursively compute the types of child nodes. The base case of this recursive process is when you encounter a leaf node storing something like an `ID` token (whose type you can look up in the current procedure's symbol table) or a `NUM` token (whose type is `int` by definition). You should add a **type** field to the parse tree class; When you compute the type of a node, store it in the field so the computed type can be accessed by other nodes.

Result 8.4

After recursively annotating the child subtrees with types, by looking at the grammar rule for the node we are currently at, we can determine which type rule from the type system is applicable. Using the types of the children (which we assume have been computed and stored in each child node) we can determine if the rule has been violated, compute the type of the current node if not, and store the type in the node.

```
void annotateTypes(Node *tree) {
    for(Node *child: tree->children) {
        //recursively compute type of each child subtree
        annotateTypes(c);
    }
    // refer to the relevant type system rule for this tree node
    // use the computed types of children to determine if the rule is violated
    // if it is not violated, store the computed type in the tree node
}
```

8.4.1 Type Inference Rules

Definition 8.4: Post System of notating deductive logic

Here, we use a standard notation that most would be familiar with from a course such as CS245; premises for a conclusion go above a horizontal bar, the conclusion below the bar, and empty premises means that the conclusion always holds. This is the **Post System of notating deductive logic**.

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

$$\frac{\langle \text{ID.name}, \tau \rangle \in \text{declarations}}{\text{ID.name} : \tau}$$

Numbers, by definition, have type `int`.

$$\frac{}{\text{NUM} : \text{int}}$$

NULL, by definition, has type `int*`.

$$\frac{}{\text{NULL} : \text{int}*}$$

Though it looks like a function call, `getchar` is implemented as a keyword. It always evaluates to an `int`.

$$\frac{}{\text{getchar}() : \text{int}}$$

Parentheses do not change the type.

$$\frac{E : \tau}{(E) : \tau}$$

Taking the address of an `int` produces an `int*`.

$$\frac{E : \text{int}}{\&E : \text{int}*}$$

Dereferencing an `int*` produces an `int`.

$$\frac{E : \text{int}*}{*E : \text{int}}$$

The expression `new int[E]` takes an `int` value and produces an `int*`.

$$\frac{E : \text{int}}{\text{new int}[E] : \text{int}*}$$

Multiplying two integer values produces an integer value.

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

Comment 8.8

We specify only what is allowed; what is not allowed is everything not specified.

Dividing an integer with another integer produces an integer.

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

The modulo operator can be applied to two integers and produces an integer.

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

The addition of two integers produces an integer.

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

The addition of a pointer to an integer or vice versa produces a pointer. Notice that the absence of a rule that adds two pointers implies the operation to be illegal.

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

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

The subtraction of two integers produces an integer.

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

Subtracting an integer from a pointer produces a pointer.

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

Subtracting two pointers produces an integer. This rule might seem counter-intuitive but is used to determine the number of elements between two addresses. Notice the absence of a rule that allows subtracting a pointer from an integer.

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

A procedure call must check that the arguments of the call match the types of the parameters for the procedure. Procedure calls always return integers.

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

8.4.2 Type Checking Statements

Comment 8.9

Since expressions produce values, these values have types. We discussed above the way to infer types for expressions. However, programs are comprised of statements which in turn contain expressions. While statements do not produce values (and therefore do not have an inferred type), we must still check the type correctness of statements.

Definition 8.5: Well-typed

We say a statement is **well-typed** if its components are well-typed.

An expression is well-typed if a type can be inferred.

$$\frac{E : \tau}{\text{well-typed}(E)}$$

The `println` statement is well-typed if and only if the expression to print has type `int`, i.e., the language does not support printing pointers.

$$\frac{E : \text{int}}{\text{well-typed}(\text{println}(E);)}$$

The `putchar` statement is well-typed if and only if the expression to print has type `int`. `putchar` interprets its operand as an ASCII value, so a pointer would make no sense here.

$$\frac{E : \text{int}}{\text{well-typed}(\text{putchar}(E);)}$$

Deallocation is well-typed if and only if the parameter has type `int*`, i.e., we can only deallocate pointers.

$$\frac{E : \text{int}^*}{\text{well-typed}(\text{delete}[]E;)}$$

An assignment is well-typed if and only if the types of the Left and Right side are of the same type.

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

While the above rule requires that the type of the left-hand side be the same as the type of the right-hand side, that, on its own, is not enough. Another requirement, when assigning a value, is that the left-hand side must represent a storage location, i.e., the left-hand side must be an `lvalue`. In other words, while `x = y` is legal, `3 = y` is not. In our language, `lvalues` include variables as well as the result of dereferencing an address.

Discovery 8.7

It turns out that the way the context-free grammar for WLP4 is designed, there is no need to check that the left-hand side is an `lvalue`, i.e., this requirement is already enforced by the grammar.

An empty sequence of statements is well-typed.

A sequence of statements is well-typed if and only if each statement in the sequence is well-typed.

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

Discovery 8.8

To type check the correctness of control flow statements, `if` and `while`, we must have a way to check the correctness of the conditional expression that is evaluated. Typically, these expressions would evaluate to a boolean. However, WLP4 does not have a `bool` type. Instead, we will use our notion of well-typed and say that a test is well-typed if the operands for the comparison are of the same type.

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

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

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

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

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

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

An **if** statement is well-typed if and only if the components of the statement are well-typed.

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

A **while** statement is well-typed if and only if the components of the statement are well-typed.

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

An empty sequence of declarations is well-typed.

A variable that is declared to be an integer is well-typed if it is initialized with an integer value.

$$\frac{\text{well-typed}(\text{dcls}) \quad \langle \text{ID.name}, \text{int} \rangle \in \text{declarations}}{\text{well-typed}(\text{dcls int ID = NUM;})}$$

A variable that is declared to be a pointer is well-typed if it is initialized with a NULL value.

$$\frac{\text{well-typed}(\text{dcls}) \quad \langle \text{ID.name}, \text{int*} \rangle \in \text{declarations}}{\text{well-typed}(\text{dcls int* ID = NULL;})}$$

We must also type check the declaration of procedures. A procedure is well-typed if the declarations and statements are well-typed and it returns an integer:

$$\frac{\text{well-typed}(\text{dcls}) \quad \text{well-typed}(S) \quad E : \text{int}}{\text{well-typed}(\text{int ID}(\text{dcl}_1, \dots, \text{dcl}_n) \{ \text{dcls } S \text{ return } E; \})}$$

We must add an extra restriction on the **wain** procedure. The procedure **wain** is well-typed if the second parameter is an **int**, the declarations and statements are well-typed and the procedure returns an integer:

$$\frac{\text{dcl}_2 : \text{int} \quad \text{well-typed}(\text{dcls}) \quad \text{well-typed}(S) \quad E : \text{int}}{\text{well-typed}(\text{int wain}(\text{dcl}_1, \text{dcl}_2) \{ \text{dcls } S \text{ return } E; \})}$$

8.5 Other Context-Sensitive Analyses

Theorem 8.3

High-level languages often have other requirements that must be satisfied.

Example 8.9

For example, in Java, a variable must be initialized before it is used. Consider the following Java program:

```

1 public class HelloWorld{
2     public static void main(String[] args){
3         int x;
4         System.out.println(x);
5     }
6 }
```

This program will be rejected by the Java compiler, with an error that the variable `x` has not been initialized.

Example 8.10

C and C++ do not have such a requirement (which leads to all sorts of bugs!).

Code 8.12

WLP4 enforces that all variables are initialized before they are used, but does so through the context-free grammar; a variable when declared must be initialized to a value.

Code 8.13

WLP4 procedures can only have one **return** statement, as the last statement of the procedure. There is a reason the language was designed this way. If multiple returns were allowed, an analysis would be needed to check that a value is returned on all paths through the program.

Comment 8.10

This analysis is non-trivial.

Example 8.11

Consider the following C/Java function:

```
1  int foo(int x) {  
2      if ( x < 0 ) return -1;  
3      else return 1;  
4  }
```

The analysis needs to keep track of all paths through the function and ensure that all paths return an appropriate value. This is not always possible. For example, in Java, the following code will not compile:


```
1  int foo(int x) {  
2      if ( x < 0 ) return -1;  
3      else if (x == 0 ) return 0;  
4      else if (x > 0 ) return 1;  
5  }
```

`int foo(int x) if (x < 0) return -1; else if (x == 0) return 0; else if (x > 0) return 1;`

Question 8.1.

What about in C/C++?

Solution: The above code does not generate an error (C/C++ have the philosophy of letting pro-

grammers shoot themselves!). However, in most compilers, it will generate a warning if the program is compiled with warnings enabled (e.g., `-Wall`). 

Example 8.12

Java imposes another requirement: there should be no *dead code*, i.e., code that is unreachable. Consider the following function:

```
1  int foo(int x) {  
2      return 1;  
3      int a = 0;  
4      return 2;  
5  }
```

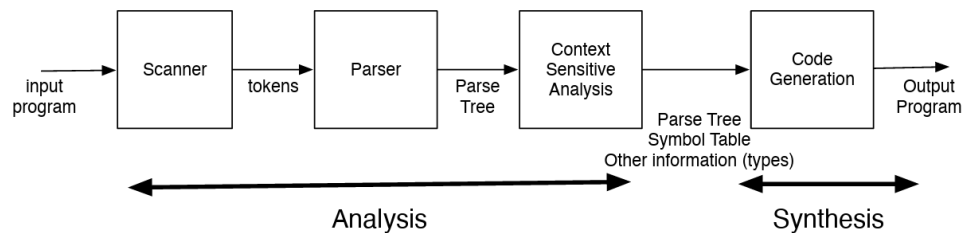
The programmer who wrote this should be fired! Java compilers are required to report an “unreachable statement error”. C/C++ do not have a similar requirement. WLP4, once again, avoids this since the only way to return from a function is through the last statement of the function.

9 Code Generation

Finally! We have completely covered the Analysis stage of the compiler: scanning, parsing, and context-sensitive analysis. Any input program that successfully passes through this stage is a syntactically and semantically correct program. We will now discuss the Synthesis stage.

Comment 9.1

In our simplistic view of a compiler, the synthesis stage comprises just of code generation. That is not entirely true of real compilers. Compilers will often perform optimizations both before and after the code generation stage.



Result 9.1

An important distinction to make is that the Synthesis stage is meant to generate a program as its output, not evaluate or execute the program.

9.1 Accessing Variables

Example 9.1

Let's begin by generating code for some simple WLP4 programs:

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

The following would be a valid equivalent MIPS program:

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


Now think about what if we want to return b instead. The two parse trees for each of the programs are almost identical.

Code 9.1

Hence to know which register to use, we need a way determine where each variable is stored based on its name (the lexeme of the ID token).

Solution: Recall from our context-sensitive analysis stage that we already have a symbol table for each procedure that maps variable names to their type. We could extend this symbol table with a location entry for each variable as follows:

| Name | Type | Location |
|------|------|----------|
| a | int | \$1 |
| b | int | \$2 |

Once we have this location entry for each variable, code generation should be simple: while traversing the parse tree, if we encounter a variable, we can refer to the location entry for that variable to see which register holds the value associated with this variable. 

Discovery 9.1

There is no upper bound to the number of variables in a WLP4 function, but the number of registers in MIPS is limited. Hence we can't just put them all in registers.

Proof. We will use the stack to store all variables, including the parameters of wain.

Comment 9.2

We could put some variables in registers and some on the stack, and in fact, a real-world code generator would definitely do this because registers are much faster to access. However, exclusively using the stack will make things more consistent and make our code generator easier to write.

□

Example 9.2

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

Would generate the following code:

```
; begin prologue
lis $4          ; new convention: $4 always contains 4, for convenience
.word 4
sw $1, -4($30)  ; store variable a at $30-4
sw $2, -8($30)  ; store variable b at $30-8
sub $30, $30, $4
sub $30, $30, $4
; end prologue
lw $3, 4($30)   ; load from stack location reserved for variable a
; begin epilogue
add $30, $30, $4 ; pop from the stack twice (since we pushed 2 variables)
add $30, $30, $4
jr $31
```

Code 9.2

In the code we generated, instead of keeping track of which register contains the variable, we now track which location on the stack stores the variable. We can do that by mapping each variable to an offset with respect to (w.r.t.) \$30.

Discovery 9.2

Do you see a problem?


1. First, the offset to variables will depend on the number of variables in the function.
2. Second, the stack is likely going to be used for holding other values. i.e., the stack pointer will likely be updated once we start using the stack to store additional information, e.g., making procedure calls.

Solution: The first issue is easy to resolve in WLP4 since all variables must be declared at the top of the function. The code generator can precompute the offsets for each variable in the program and set up the stack accordingly in the prologue.

For the second issue, a standard approach is to dedicate another register to always point to a fixed memory location within the stack while executing a procedure.

Definition 9.1: Frame Pointer

We call this a **frame pointer**.

By keeping the frame pointer unchanged within a procedure, we can use fixed offsets w.r.t. the frame pointer. We will use \$29 as the frame pointer. 

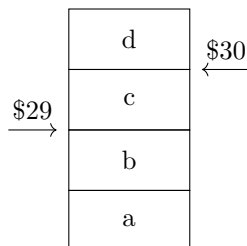
Theorem 9.1

The frame pointer \$29 divides parameter variables from non-parameter variables. Parameters are at positive offsets from the frame pointer, and non-parameters are at non-positive (zero or negative) offsets.

Example 9.3

Below, we have added an local variable *c* and *d* to the program to better illustrate this. Notice that the parameters *a* and *b* have positive offsets 8 and 4, while the non-parameter local variables *c* and *d* have non-positive offsets 0 and -4 .

```
1 int wain(int a, int b){  
2     int c = 0;  
3     int d = 241;  
4     return a;  
5 }
```



| Name | Type | Offset (from \$29) |
|------|------|--------------------|
| a | int | 8 |
| b | int | 4 |
| c | int | 0 |
| d | int | -4 |

Result 9.2

In general, if there are n parameters and the first parameter is “parameter 0”, then parameter i is at offset $4(n - i)$. If the first non-parameter is “non-parameter 0”, then non-parameter i is at offset $-4i$.

9.1.1 Initializing Variables

Compare the code for pushing the parameters a and b to the code for pushing c and d :

```
sw $1,-4($30)      ; push parameter variable a
sub $30, $30, $4    ; update stack pointer
sw $2,-4($30)      ; push parameter variable b
sub $30, $30, $4    ; update stack pointer
```

Above we are pushing a and b , and we simply place the values in $\$1$ and $\$2$ on the stack. However, for c and d , we need to push specific initial values for these variables, 0 and 241 respectively.

```
sw $0,-4($30)      ; push non-parameter variable c = 0
sub $30, $30, $4    ; update stack pointer
lis $3              ; load initial value for d
.word 241
sw $3,-4($30)      ; push non-parameter variable d = 241
sub $30, $30, $4    ; update stack pointer
```

For c , since the initial value is 0 and $\$0$ always contains 0, we can take a bit of a shortcut: we can just use a Store Word instruction with $\$0$. However, d 's initial value is 241 which we are unlikely to already have lying around in a register. We load 241 into $\$3$ and then store $\$3$ on the stack to initialize d .

Comment 9.3

You may recall the annoying rule that variable initializations in WLP4 cannot have complex expressions on the right hand side; you can only initialize variables to constants. This rule may be annoying when writing WLP4 programs, but when compiling them it makes your task easier.

9.1.2 NULL Pointers

With regards to initialization, there is one more interesting case to consider. Variable declarations in WLP4 can be of `int*` (pointer) type, in which case the only valid initial value is `NULL`¹:

```
int *ptr = NULL;
```

In this course, we are using a simplified MIPS emulator instead of a real operating system, and 0x00 is a valid address. In fact, unless you use a relocating loader, then 0x00 will be the address where your program code is loaded. This means when you deference 0x00, instead of getting an error, you'll probably

¹We usually think of `NULL` as representing address zero (0x00), and we expect that if we try to access this address, the program will crash with an error. This works in real-world systems, since the operating system usually enforces that 0x00 is part of an inaccessible segment of memory, and trying to access it will produce the segmentation fault error you are no doubt familiar with. You'll learn more about how and why the operating system enforces this in CS350.

just get the first word of your program code. This is undesirable behaviour; we actually want dereferencing NULL to cause an immediate crash because it indicates a programming error.

Code 9.3

One way to achieve the desired behaviour (crashing) in our CS241 version of MIPS is to associate NULL with an address that is not divisible by 4. For instance, we can choose 0x01. Recall that if a MIPS program tries to access memory at an unaligned address, that is, an address that is not a multiple of 4, it causes the program to crash. Since that is exactly the behaviour we want, we will make the convention that NULL is equal to 0x01.

Example 9.4

Adopting the convention that NULL is 1, this mean you could initialize a pointer variable to NULL as follows:

```
lis $3          ; load initial value for ptr
.word 1
sw $3, -4($30)   ; push ptr = NULL
sub $30, $30, $4  ; update stack pointer
```

Comment 9.4

Some aspects of code generation require different handling for pointers and integers, but this is one case where the two types are basically handled in the same way.

9.2 Expressions With Binary Operations

Let's now look at more complicated expressions that involve binary operations.

Definition 9.2: `code(expr)`

We will write `code(a)` to indicate the code that our code generator will generate to load the value for variable `a` into register `$3`.

Example 9.5

```
1  int wain(int a, int b){
2      return a - b;
3  }
```

Note that the following is obviously not going to work:

```
code(a) ; results in lw $3, 8($29), which we will write as $3 <- a
code(b) ; results in lw $3, 4($29), which we will write as $3 <- b
sub ....
```

Notice the obvious issue: loading the value for `b` in `$3` **clobbers**^a the previous value in that register (the value for variable `a`). We need temporary storage. In pseudocode this could look as follows:

```
code(a)          ; $3 <- a
add $5, $3, $0    ; $5 <- $3
code(b)          ; $3 <- b
sub $3, $5, $3    ; $3 <- a - b
```

^a“Clobber” is actually the term of science here. It is the standard term for one step overwriting the result of a previous step due to using the same storage, particularly a CPU register.

Discovery 9.3

Notice that the above program needs one temporary register to store the result of the expression. However, as our expression gets larger, e.g.

$$(((a + b) - c) + d) - e + \dots$$

we will need more temporary registers. Therefore,

Solution: We will store temporary values on the stack.



Definition 9.3: push(reg) & pop(reg)

We will write `push($3)` to represent a store of the value in the supplied register (in this case `$3`) on the stack and an appropriate update to the stack pointer. We will also write `pop($5)` to represent a load from the top of the stack into the supplied register (in this case `$5`) and an appropriate update to the stack pointer.

Example 9.6

`push($3)` generates:

`sw $3, -4($30)`

`sub $30, $30, $4` `pop($5)`

`pop($5)` generates:

`add $30, $30, $4`

`lw $5, -4($30)`

Result 9.3

The pseudocode for generating the code for `a-b` can be written as:

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

Comment 9.5

Note that these code snippets rely on the convention that \$4 always contains the value 4. When you write your code generator, you will likely create helpers function to generate code for a push or a pop. You must not call these helper functions until you have generate code that initializes \$4 to 4, or they will not work correctly.

Question 9.1.

Notice that this produces suboptimal code; why push \$3 when we are going to pop it soon after?

Solution: The intent here is that of simplicity, and perhaps more importantly, universality: this would work no matter how complicated `code(b)` is, so long as it puts its result in \$3 and is consistent with the stack. This is a universal code generation scheme that can be used for all expressions that involve binary operations, simply by changing the operation performed on the final line. 🎵

We will continue to create shorthand to represent our code generation strategies. For example, consider the grammar rule $\text{expr}_1 \rightarrow \text{expr}_2 \text{ PLUS term}$, we will write the following shorthand for generating the code for expr_1 :

```
code(expr1) = code(expr2)
              push($3)
              code(term)
              pop($5)
              add $3, $5, $3
```

Remark: Since other arithmetic operations on integers (subtraction, multiplication, division and modulo) work similarly to addition, we leave the code generation strategies for them as an exercise.

Comment 9.6

One concern is how to handle pointer arithmetic, where the operands might be a mix of pointers and integers, or might both be pointers. We discuss this shortly, after first discussing some other pointer operations.

9.3 Unary Operations, Assignment, & Lvalues

Definition 9.4: Unary Operation

A **unary operation** is an operation that only takes one operand.

Definition 9.5: Lvalue

An **lvalue** is an expression that represents a location where a value is stored.

Discovery 9.4

In WLP4, the only expressions which are considered lvalues are expressions consisting of single variables like `a`, and dereferenced pointer-type expressions like `*(array + index)`. Examples of expressions that are *not* lvalues are things like `3` or `a-b` or `241*c`; it would not make sense to assign a value to such an expression because it does not represent a storage location.

Discovery 9.5

There are only two contexts in which an lvalue can appear in WLP4; the left-hand side of an assignment statement, and as the operand of an address-of operator.

The WLP4 grammar serves as a formal definition of **lvalues** and enforces the contexts in which they can appear. These are the grammar rules which define **lvalues**:

```
1      lvalue → ID
2      lvalue → STAR factor
3      lvalue → LPAREN lvalue RPAREN
```

Code 9.4

In the first rule, `ID` refers to a variable. In the second rule, `STAR` refers to the dereference operator and `factor` is the expression being dereferenced. The third rule simply says that if you put parentheses around an **lvalue**, it is still an **lvalue**.

These are the grammar rules which permit an lvalue on the right-hand side:

```
1      statement → lvalue BECOMES expr SEMI
2      factor → AMP lvalue
```

Code 9.5

The first rule is the assignment statement rule. The second is the address-of rule (where `AMP` is short for ampersand, meaning the `&` character).

9.3.1 Pointer Dereference

There are actually two versions of the pointer dereference rule:

Notice that there are actually two versions of the pointer dereference rule:

```
1      factor → STAR factor
2      lvalue → STAR factor
```

The factor on the right-hand side is an expression that resolves to some memory address.

Question 9.2.

How are they different?

Solution: In the first case, $\text{factor}_1 \rightarrow \text{STAR factor}_2$, we want to return the value stored at the memory address. We generate code that computes the memory address, and then use a Load Word instruction:

```
code(factor1) = code(factor2)    ; $3 contains the memory address
lw $3, 0($3)                   ; $3 now contains the value at the address
```

For the second rule, once again, we want to compute the memory address, but what we do with this memory address depends on the context where the `lvalue` is used.

1. If the `lvalue` is used in `statement \rightarrow lvalue BECOMES expr SEMI` then we should store the `expr` on the right-hand side of the assignment statement in the memory address.
2. If the `lvalue` is used in `factor \rightarrow AMP lvalue` then we should simply return the memory address.



Result 9.4

So, there are two cases for implementing pointer dereference. In the case with `factor` on the lefthand side, we want the dereference to produce a *value*; in the case with `lvalue` on the left-hand side, we want it to just produce the *address* being pointed to, and this address can be used in different ways.

9.3.2 Address-of

We have seen how to implement `factor \rightarrow AMP lvalue` in the case where the `lvalue` is defined by the rule `lvalue \rightarrow STAR factor`. But what if the `lvalue` is a variable, i.e., `lvalue \rightarrow ID`? In this case, we want to return the memory address corresponding to the variable whose name is `ID`. Since we are storing everything on the stack using offsets from the frame pointer, it is not difficult to compute this address.

```
lis $3
.word <offset for ID>
add $3, $29, $3
```

To complete the implementation of assignment statements, we need to also handle the (`lvalue \rightarrow ID`) case. One approach is to compute the address of the `ID` as shown above, then generate code for the `expr` on the right-hand side of the assignment and store it in the address. One could also use a more direct approach of doing a Store Word directly at the correct offset:

```
code(expr)
sw $3, <offset for ID>($29)
```

9.3.3 Handling lvalue Summary

Result 9.5

The following table summarizes the handling of `lvalues` in WLP4.

| | | |
|--|--|--|
| | <code>statement</code> \rightarrow lvalue BECOMES <code>expr SEMI</code> | <code>factor</code> \rightarrow AMP lvalue |
| <code>lvalue</code> \rightarrow ID | Store the expr value at the address of ID | Return the address of ID |
| <code>lvalue</code> \rightarrow STAR <code>factor</code> | Evaluate the dereferenced factor to obtain a memory address and store the expr value at this address | Evaluate the dereferenced factor to obtain a memory address and return this address |

9.4 Pointer Arithmetic

Theorem 9.2

The basic idea is: When your code generator is generating code for addition or subtraction, examine the types of the subexpressions (using the information gathered during semantic analysis) and generate different code depending on the types.

Example 9.7

Consider the rule `expr1 \rightarrow expr2 PLUS term` where `type(expr2) == int*` and `type(term) == int`, that is, adding an integer value to an address. When we add an integer to an integer-pointer, we do not simply add the integer value, we add `sizeof(int)` times the integer value. Therefore, we must compute `expr2 + (4 \times term)`, since `sizeof(int)` is 4.

```
code(expr1) = code(expr2)
              push($3)
              code(term)
              mult $3, $4      ; $4 always has the value 4
              mflo $3
              pop($5)         ; $5 <- expr2
              add $3, $5, $3
```

Example 9.8

There is also the case of pointer-pointer subtraction: `expr1 \rightarrow expr2 MINUS term` where the types are `type(expr2) == int*` and `type(term) == int*`. We can compute this by subtracting the value of `term` from `expr2` (which gives the raw memory address difference) and then dividing by `sizeof(int)` to get the number of elements, i.e., `(expr2 - term) / 4`.

```
code(expr1) = code(expr2)
              push($3)
              code(term)
              pop($5)
              sub $3, $5, $3
              div $3, $4
              mflo $3
```

9.5 Input / Output

Comment 9.7

The `println` statement needs to be handled specially, so is delayed to a later section. Instead, this section will focus on `putchar` and `getchar`.

Discovery 9.6

`putchar` and `getchar` correspond to the following rules:

```
statement → PUTCHAR LPAREN expr RPAREN SEMI
factor → GETCHAR LPAREN RPAREN
```

Code 9.6

For `putchar`:

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

For `getchar`:

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

9.6 If Statements and While Loops

These correspond to the following rules:

```
statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE
statement → IF LPAREN test RPAREN LBRACE statements RBRACE ELSE LBRACE statements RBRACE
```

where the `test` nonterminals correspond to a Boolean condition.

Comment 9.8

WLP4 does not have a Boolean type or Boolean logical operations, so the form of conditions is quite restricted.

The test nonterminal can only be expanded as follows:

- `test` \rightarrow `expr EQ expr`
- `test` \rightarrow `expr LT expr`
- `test` \rightarrow `expr LE expr`
- `test` \rightarrow `expr NE expr`
- `test` \rightarrow `expr GT expr`
- `test` \rightarrow `expr GE expr`

Theorem 9.3

A comparison test can be implemented in the same way as binary operations like addition, subtraction, etc. but once the two expression values are determined, generate code that performs the comparison, and produces either 0 (true) or 1 (false) in \$3 depending on the result.

Code 9.7

For less than (LT) and greater than (GT), this can be done with a single `slt` (Set Less Than) instruction. The other comparison operators require slightly more work.

Question 9.3.

Suppose you are comparing two expressions with \geq (GE). The value of the left expression has been computed and stored in \$5, and the value of the right expression has been computed and stored in \$3. At the very start of the generated code, the constant 1 was loaded into \$11. Give a sequence of **two** MIPS instructions that stores 1 in \$3 if $\$5 \geq \3 and 0 otherwise.

Solution: We have

```
slt $3, $5, $3
sub $3, $11, $3
```

The trick here is that “greater than or equal” is the logical negation of “less than”.



Discovery 9.7

One small caveat is that WLP4 also allows you to compare a pointer to a pointer to determine which address is larger. Integers in WLP4 are *signed*, but pointers represent memory addresses, which cannot be negative. Thus, for pointers you should actually use the `sltu` instruction (Set Less Than Unsigned) as the basis of your comparisons.

Question 9.4.

There is one issue that is relatively trivial to deal with when writing conditionals and loops by hand, but a little tricky in a code generator: all label names used in a MIPS assembly program must be unique.

Solution: There is a simple solution that works in all situations. When you generate the unique label names for a while loop, increment the counter once, then generate both the “start” and “end” label names before processing the statements in the body.



9.7 Printing Integers & Calling External Procedures

WLP4 supports a `println` statement, which prints the decimal (base 10) value of an integer to standard output, followed by a newline. The rule is:

$$\text{statement} \rightarrow \text{PRINTLN LPAREN expr RPAREN SEMI}$$

The `expr` is the integer value to print.

Comment 9.9

Because numbers in computers are stored in binary, not base-10, printing out an integer in base-10 is non-trivial.

While it's technically possible to output all the code to perform this task every time `println` is used, it would be more practical to have a MIPS procedure, and have the code for `println` simply call that procedure.

Discovery 9.8

In addition to `println`, the `new` operator and `delete` statement also rely on calling complicated procedures. Outputting a copy of every single procedure you use will bloat your generated code and add to compilation time. A more standard approach is to precompile commonly used procedures and place them in the *runtime environment*.

Definition 9.6: Runtime environment

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 and so on.

Example 9.9

For example, the standard C library, providing functions such as `malloc`, `free`, `printf` and `memcpy`, is part of the execution environment of C programs. On Unix and Unix-like systems including Linux, it is provided in `libc.so`. Archaic, nonstandard operating systems have other conventions; for instance, on Windows, it is provided by `msvcrt.dll`.

Result 9.6

Real compilers rely on runtime environments since this allows the binary they produce to be smaller.

In CS241, we use our homegrown *MIPS Executable Relocatable Linkable (MERL)*² format for object files. The `.merl` format contains MIPS machine code (not assembly) along with additional information

²MERL is based very loosely on the so-called “a.out” format, which was the original format for object and executable files on Unix. That format actually has no name, and was simply called “a.out” retrospectively because this is the default name for the output of Unix C compilers.

needed by both the *linker* and the *loader*. We will be addressing these topics and the format in a lot more detail in future modules.

While generating code, if the compiler needs to use the `print` procedure that will be part of the runtime, it will generate an assembler directive to `import print`. Importing a procedure named `print` simply requires having the following appear **once** in the generated assembly (by convention at the top of the generated file):

```
.import print
```

Once the compiler is done generating output, the result is an assembly file which might now contain `import` statements. This means that we must now use an assembler that understands imports. We call this assembler the *Linking Assembler* (`cs241.linkasm`). It is an assembler that produces object files rather than just MIPS machine code. Suppose the output generated by the compiler is in a file named `output.asm`. Running `cs241.linkasm` produces a MERL file:

```
cs241.linkasm < output.asm > output.merl
```

The `output.merl` file is the object file for the WLP4 program the compiler was to compile. It contains MIPS machine code along with an announcement that this code expects to be linked to an appropriate `print` procedure. Imagine the existence of `print.merl`, an object file for the `print` procedure. We can link them using a linker that understands the MERL format:

```
cs241.linker output.merl print.merl > linked.merl
```

Notice how the linker takes a number of object files, links them, and produces a new object file which contains the combined machine code and “announcements”. To produce the pure MIPS machine code, we must strip out the MERL metadata from the object file. This can be done using the `cs241.merl` tool. We show how to use the tool below and will discuss this in more detail in a future module:

```
cs241.merl 0 < linked.merl > final.mips
```

Coming back to the topic at hand, we can generate code for the WLP4 `println` statement by relying on our knowledge of what the `print` procedure we will be linking with expects. This procedure expects the integer to be printed in register 1. Just like any other procedure, we will call `print` using the `jalr` instruction which overwrites register 31. Using the shorthand we had previously developed, we can write the following:

```
code(println(expr);) = push($1) ; if current value in $1 needs to be preserved
                      + code(expr)
                      + add $1, $3, $0
                      + push($31)
                      + lis $5 + .word print
                      + jalr $5 + pop($31)
                      + pop($1)
```

Comment 9.10

In the pseudo-code above, we chose to save the original value in register 1 before copying into it the value to be printed. This might be unnecessary depending on other decisions made during code generation.

9.8 Generating Code for WLP4 Procedures

Comment 9.11

We have just discussed how to call externally defined procedures. External procedures are used for the memory management functionality in WLP4 (`new` and `delete`). We will discuss implementation strategies for `new` and `delete` in detail in next Module.

9.8.1 Starting with `wain`

Discovery 9.9

Recall that in WLP4, the `wain` function is the entry point for the program, meaning the program must start by executing `wain`. However, `wain` is always the last function that appears in the WLP4 source code.

Question 9.5.

We have two options to ensure `wain` executes first.

Solution: The first is to insert instructions at the beginning of the generated program that perform any necessary initializations, and then branch to a label that represents the code produced for `wain`, e.g., `beq $0, $0, wain`, assuming that the code for `wain` is prefixed with this label.

An alternate approach is to organize our generated output such that the output for `wain` appears first. Any procedures that are defined in the WLP4 program would appear after the end of the code for `wain`.

Comment 9.12

To ensure that the code for `wain` (and other procedures) does not “fall through” to the code for the next procedure, we must ensure that each procedure ends with a `jr $31` instruction.



9.8.2 Calling Conventions

Definition 9.7: Calling Convention

Generally, one defines a calling convention that describes the low-level aspects of how procedures are handled.

Example 9.10

A calling convention may include things like:

- How parameters are passed to a procedure by the caller;
- How a procedure retrieves its parameters once it is called;
- How return values are passed back to the caller;
- Which registers are preserved when a procedure is called;
- How procedures are allowed to modify the stack;
- How setup and cleanup of the stack is divided between the caller and the procedure being called.

Result 9.7

Here is our suggested calling convention for implementing procedures in your WLP4 code generator.

- Parameters for a procedure are stored on the stack by the caller, directly before jumping to the procedure with `jalr`. The arguments are evaluated and pushed to the stack in left-to-right order, meaning the *rightmost argument is on top of the stack* when a procedure is called.
- After a procedure is called, it immediately *sets up a “local” frame pointer* in \$29, one memory slot above \$30. Since parameters are already on the stack, this means parameters can be accessed using *positive offsets* from the frame pointer, and non-parameters are accessed using *non-positive offsets* (zero or negative), just like our earlier convention for `wain`.
- Return values are stored in \$3.
- Procedures can freely modify the stack above \$30, but not at \$30 or below, and must preserve \$30 by popping everything they push.
- The caller is responsible for popping the arguments to a procedure, since the caller is the one that pushes the arguments.
- The caller is also responsible for preserving its own local frame pointer (\$29) and return address (\$31). Before pushing arguments, the caller should save \$29 and \$31 on the stack. After the call returns, and after popping arguments, the caller should restore \$29 and \$31.

Question 9.6.

Above, we were a little non-specific about which registers should be preserved. We said that the caller should preserve registers \$29 and \$31, and the procedure being called should preserve \$30 (by popping everything that it pushes). What about other registers?

Solution: The answer is that this aspect of the calling convention is up to you. It is possible to write a code generator for WLP4 where no other registers are preserved by a procedure call. However, this depends on the design decisions you make elsewhere in your code generator. A good rule of thumb is the following:

Suppose that in between the time where the code generator sets the value of a register, and the last time the code generator uses that value, it is possible that a procedure could be called. Then the calling convention should require procedures to preserve that register.



9.8.3 Calling Procedures

With our calling convention established, this is relatively straightforward. There are two procedure call rules:

```
factor → ID LPAREN RPAREN
factor → ID LPAREN arglist RPAREN
```

Code 9.8

The first rule is for a call with no arguments. For these calls, simply save \$29 and \$31, perform the call with `jalr` (using the ID token to determine which procedure to call) and then restore \$29 and \$31 when the call returns.

Code 9.9

The second rule has an `arglist` nonterminal representing a sequence of arguments. The rules related to `arglist` are:

```
arglist -> expr COMMA arglist
arglist -> expr
```

In other words, an `arglist` tree is essentially structured like a linked list of `expr` nodes. In other words, an `arglist` tree is essentially structured like a linked list of `expr` nodes.

Comment 9.13

To perform a procedure call with arguments, recall that our calling convention said you should push the arguments after saving \$29 and \$31, but before doing the `jalr`. Traverse the `arglist` tree to generate code for each `expr` and push it to the stack. Keep track of the *total number of arguments* (or look up this information in your procedure table from semantic analysis—it is the number of elements in the *signature*).

Once the call returns, our calling convention dictates that you must pop all the arguments *before* restoring \$29 and \$31. This is where knowing the total number of arguments comes in handy. If there are n arguments, you can simply generate n copies of the following instruction:

```
add $30, $30, $4 ; assuming $4 contains the constant 4
```

You do not need to store the arguments in any particular register as you pop them. The point is just to remove them from the stack (which we accomplish by incrementing the stack pointer to move it downwards).

9.8.4 Generating The Procedure Itself

Code 9.10

The rule for the `wain` procedure is (in more concise notation):

```
main → int wain(dcl, dcl) dcls statements RETURN expr;
```

The rule for a non-`wain` procedure is (in more concise notation):

```
procedure → int ID(params) dcls statements RETURN expr;
```

Discovery 9.10

The `wain` procedure requires you to do some initialization work at the start (e.g., setting up constants used for the entire program) since it is the first procedure that runs. Otherwise the high-level idea is the same for `wain` and other procedures:

1. Set up the frame pointer, compute the table of offsets, and push local variables on the stack as necessary.
2. Generate code for the statements in the body of the procedure.
3. Generate code for the return expression.
4. Clean up the stack and return.

Comment 9.14

Steps (2) and (3) are identical between `wain` and other procedures and just involving running your code generation process on the relevant subtree (`statements` or `expr`). Step (1) and Step (4) are where differences arise:

- For `wain`, there are exactly two parameters, which are provided to the program in \$1 and \$2. For other procedures, there can be any number of parameters and one must examine the `params` subtree to compute the offset table.
- For `wain`, the parameters must be pushed to the stack *by wain itself*. For other procedures, the parameters *must not* be pushed to the stack by the procedure itself, because our calling convention mandates that parameters are pushed by the caller.
- Because `wain` pushes its own parameters, the frame pointer \$29 in `wain` should be set up after pushing the parameters. For other procedures, it should be set up as soon as the procedure begins.
- Procedures other than `wain` might need to preserve registers on the stack, depending on the calling convention.

- When cleaning up the stack, **wain** must pop its parameters. Other procedures *must not* pop their parameters because it is the caller's responsibility to pop them. Both **wain** and other procedures should pop the non-parameter local variables they push.
- Technically, **wain** doesn't need to clean up the stack at all because stack cleanup for **wain** is the last thing that happens before the program ends. However, for other procedures, properly cleaning up the stack is *essential* or the assumptions that make the calling convention work will fall apart.

Example 9.11

Below, we provide a pseudocode outline how to generate code for procedures *other than wain*.

```
code(procedure) = procedureLabel:
    sub $29, $30, $4          ; set up the frame pointer
    code(dcls)                ; push non-parameter local variables
    push registers to save    ; if required by the calling convention
    code(statements)         ; statements
    code(expr)               ; return expression
    pop saved registers       ; if required by the calling convention
    pop local variables       ; only the non-parameters!
    jr $31
```

9.8.5 Label Names for Procedures

Recall that when discussing code generation for **if** statements and **while** loops, we talked about the need to generate unique label names. We suggested using counter variables that you increment whenever you generate a new label. This might lead to the code generator defining label names like **while1** or **else1**.

Discovery 9.11

There is nothing stopping a WLP4 program from defining procedures with the same name! An evil person who lives to create chaos might decide to name their **while1** or **else1**.

Solution: One extreme reaction could be to simply disallow the use of such procedure names in WLP4, like how **new** and **delete** are reserved keywords. However, this is indeed extreme. The recommended approach is to have the compiler be careful when generating its internal label names. For example, by simply attaching a prefix to all WLP4 procedures names, the problem can be eliminated. If the code generator always appends, say, **P** in front of labels corresponding to procedures, and never generates a label starting with an **P** anywhere else, we guarantee that duplicate labels will never be generated. For example, if an evil user defines a procedure called **else1**, the compiler would actually generate a label called **Pelse1** and use that to refer to the procedure. 🎵

10 Runtime Support: Loading & Linking

We'll now look at two distinct questions, connected by the idea of providing runtime support to programs.

Question 10.1.

- How do we load programs into memory and start executing them?
- How do we allow our programs to rely on a standard library of functions for things like input, output, and memory allocation?

10.1 Loader

Definition 10.1:

The loader is provided by the computer's operating system.

The operating system controls which programs run. To run a program, the operating system calls upon the loader to copy the program from disk storage into main memory (RAM) and prepare it for execution. The following shows pseudocode for a very simple operating system and loader:

| | |
|--|---|
| 1 operating system ver. 1.0 2 repeat: 3 P = next program to run 4 loader(P) 5 jalr \$0 6 beq \$0, \$0, repeat | loader ver. 1.0 for (i=0; i<codeLength; ++i) { MEM[4*i] = P[i] } |
|--|---|

Discovery 10.1

In the pseudocode above, once the program to run has been chosen, the loader loads it by copying the code from P into MEM. This simple loader loads the selected program starting at address 0x00. We use `codeLength` as a count of number of words in the program P. The i^{th} word of P, denoted $P[i]$, is loaded into the 4 bytes starting at $4*i$. Once the program has been loaded, the operating system jumps to address 0x00. Once the program returns, the operating system looks for the next program to run.

Question 10.2.

Why is loading all programs at address 0x00 not realistic?

Solution: First of all, the operating system itself is a program that is running in memory, so it needs to be loaded somewhere. Additionally, modern real-world operating systems support multiple programs running at the same time. It is not possible to have all these programs loaded at the start of RAM. 🎵

Code 10.1

Let's have the loader search for an appropriate amount of free RAM, load the program at beginning

of that memory, then return the starting address of that memory to the operating system:

| | | |
|---|---------------------------|----------------------------------|
| 1 | operating system ver. 2.0 | loader ver. 2.0 |
| 2 | repeat: | $\alpha = \text{findFreeRAM}(N)$ |
| 3 | P = choose program to run | for (i=0; i<codeLength; ++i) { |
| 4 | \$3 = loader(P) | MEM[$\alpha+4*i$] = P[i] |
| 5 | jalr \$3 | } |
| 6 | beq \$0, \$0, repeat | \$30 = α + N |
| 7 | | return α to OS |

In the updated pseudocode for the loader, we introduce a new variable N that represents the amount of memory the loader decides to allocate to the program. In our memory model for a program, the memory assigned to the program includes memory needed for the instructions in the program, and additional memory for the stack and heap. In other memory models, you would have more sections (such as memory for read-only data and global values).

Comment 10.1

The loader above would indeed load an assembled MIPS program into the chosen memory location. The problem is that there is a high chance that the MIPS program will not execute correctly.

Question 10.3.

What would cause an assembled MIPS program to fail to execute correctly when loaded at starting address α by the loader? Try to come up with an answer before reading on.

Solution: The solution is: *Labels*.



Example 10.1

The following example illustrates the problem.

| | | | |
|----|-----------------|---------|-------------|
| 1 | Assembly | Address | Instruction |
| 2 | ----- | | |
| 3 | lis \$3 | 0x00 | 0x00001814 |
| 4 | .word 0xabc | 0x04 | 0x00000abc |
| 5 | lis \$1 | 0x08 | 0x00000814 |
| 6 | .word A | 0x0c | 0x00000018 |
| 7 | jr \$1 | 0x10 | 0x00200008 |
| 8 | B: | | |
| 9 | jr \$31 | 0x14 | 0x03e00008 |
| 10 | A: | | |
| 11 | beq \$0, \$0, B | 0x18 | 0x1000fffe |
| 12 | .word B | 0x1c | 0x00000014 |

Labels used in branches are not affected if the loader loads the program at an address other than 0x00 because our equation computes the relative difference between the use and definition of a label.

The other use of labels is as part of `.word` directives. There, recall, the assembler simply replaces the `.word` with the memory address of where the label was defined. This memory address was computed by the assembler by assuming that the program was loaded at starting address `0x00`.

In the example above, `.word A` was assembled to `0x0000018` because the label `A` was defined at location `0x18` assuming that the first instruction was at address `0x00`. Similarly, `.word B` was assembled to `0x14` because the label `B` was defined at address `0x14` if we were to assume that the first instruction is at address `0x00`.

Theorem 10.1

Since the actual starting address only becomes apparent once the loader finds the chunk of memory it will use for the program, it is the loader, not the assembler, that must fix this program.

Code 10.2


The loader must **relocate** words that used to represent `.word label` in the original assembly program. The actual relocation is straightforward: add an offset to the appropriate words representing the change in starting address.

Discovery 10.2

We have decided that our loader will relocate. However, there is still a problem. The problem is that the content that the loader is reading, from a binary file, looks as the code to the right (using hexadecimal as shorthand):

0x00001814
0x00000abc
0x00000814
0x00000018
0x00200008
0x03e00008
0x1000fffe
0x00000014

The problem is that the assembled file that the loader is going to load is simply a stream of bits, i.e., the loader has no way of finding which words were produced from the directive `.word label`.

Solution: The solution to this is to have the assembler encode information of what locations contained instances of a word used with a label. Therefore, instead of generating just pure machine code, the assembler should generate *object code*. 

Definition 10.2: Object Code

Object code contains machine code for the assembly program, plus additional information needed by the loader and (later) the linker.

Comment 10.2

In CS241, we use a homegrown object code format named MERL: MIPS Executable Relocatable Linkable. In the next section, we introduce this file format.

10.1.1 The MERL Format

The MERL format contains three sections: a header, the assembled code, and a footer. The assembled code section of the output is (almost) exactly as before. The header is exactly three words, as described below:

1. Word 1 is the binary encoding of `beq $0, $0, 2`, i.e., an unconditional jump over the next two words.
2. Word 2 is the address just past the end of the MERL file, i.e., where the MERL footer ends.
3. Word 3 is the address just past the end of the assembled code, i.e., where the MERL footer begins.

Code 10.3

The MERL footer, the third and last section of the MERL file, contains relocation entries (and later, two other types of entries used for linking, but we only discuss relocation entries for now).

Definition 10.3: Relocation Entry

Each relocation entry in the MERL footer consists of two words. The first word is a “format code”, which always has the value 1, indicating that the entry is a relocation entry. We will see other kinds of entries later with different format codes. The second word in a relocation entry gives the address of a word that needs to be relocated. This is how the loader determines which words it needs to modify by adding an offset.

Example 10.2

The table below shows the same assembly program as before. Like before, we also show the nonrelocatable assembled code. However, this time we also show the relocatable MERL output and what this MERL output would look like in assembly (which makes it easier to read).

| Assembly | Address | ML Not Relocatable | Address | MERL Relocatable | MERL In Assembly |
|-----------------|---------|--------------------|---------|------------------|------------------|
| | | | 0x00 | 0x10000002 | beq \$0, \$0, 2 |
| | | | 0x04 | 0x0000003c | .word endModule |
| | | | 0x08 | 0x0000002c | .word endCode |
| lis \$3 | 0x00 | 0x00001814 | 0x0c | 0x00001814 | lis \$3 |
| .word 0xabc | 0x04 | 0x00000abc | 0x10 | 0x00000abc | .word 0xabc |
| lis \$1 | 0x08 | 0x00000814 | 0x14 | 0x00000814 | lis \$1 |
| .word A | 0x0c | 0x00000018 | 0x18 | 0x00000024 | reloc1: .word A |
| jr \$1 | 0x10 | 0x00200008 | 0x1c | 0x00200008 | jr \$1 |
| B: | | | | | B: |
| jr \$31 | 0x14 | 0x03e00008 | 0x20 | 0x03e00008 | jr \$31 |
| A: | | | | | A: |
| beq \$0, \$0, B | 0x18 | 0x1000fffe | 0x24 | 0x1000fffe | beq \$0, \$0, B |
| .word B | 0x1c | 0x00000014 | 0x28 | 0x00000020 | reloc2: .word B |
| | | | | | endCode: |
| | | | 0x2c | 0x00000001 | .word 1 |
| | | | 0x30 | 0x00000018 | .word reloc1 |
| | | | 0x34 | 0x00000001 | .word 1 |
| | | | 0x38 | 0x00000028 | .word reloc2 |
| | | | 0x3c | | endModule: |

Question 10.4.

For the following MIPS assembly program, how many relocation entries does the corresponding MERL file have? Which line does each relocation entry correspond to?

```
lis $2
.word A
A: jr $2
beq $0, $1, B
.word 0x1000
B: jr $31
```

Solution: There is only one relocation entry, corresponding to the line `.word A`.



10.1.2 Loader Relocation Algorithm

Let's now look at how the loading process actually works when relocation is included.

Code 10.4

```
1 read_word()           // skip the first word in the MERL file
2 endMod <- read_word() // second word is the address of end of MERL file
3 codeSize <- read_word() - 12 // Use the third word to compute size of the
  code section
4  $\alpha$  <- findFreeRAM(N) // find starting address  $\alpha$ 
5 for (int i = 0; i < codeSize; i += 4) // load the actual program (not
  the header and footer)
6   MEM[ $\alpha$  + i] <- read_word() // starting at  $\alpha$ 
7 i <- codeSize + 12 // start of relocation table
8 while (i < endMod) {
9   format <- read_word()
10  if (format == 1) { // indicates relocation entry
11    rel <- read_word() // address to be relocated: relative to start
      of header
12    MEM[ $\alpha$  + rel - 12] +=  $\alpha$  - 12 // go forward by  $\alpha$  but also back by 12
```

```

13                                     // since we did not load the header
14     } else { ERROR // unknown format type }
15     i += 8           // update to next entry in MERL footer
16 }

```

The algorithm first computes the size of the code, `codeSize`. Since the word we just read is the address where the code segment ended, the size of the code is computed by subtracting the size of the header (since it appears before the code segment); 3 words or 12 bytes.

The algorithm then determines the starting address α where the code segment will be loaded. It looks for a free block of memory of N bytes, where N is some value bigger than `codeSize` and represents the total amount of memory available to the program.

Comment 10.3

We do not discuss how this free block of size N is found.

The algorithm then loads the code segment by reading `codeSize` number of words and placing them in memory starting at address α . Notice that the header and footer are not copied into memory, only the code segment. Notice also that any words in the code segment that were originally `.word` label have not yet been relocated. That comes in the next loop. The next loop begins at the start of the relocation table; the address (within the MERL file) where this starts is computed by adding back the 12 bytes we had subtracted to obtain `codeSize`).

The loop goes through the entire MERL footer ($i < \text{endMod}$). We are only expecting relocation entries. A word is read from the footer and checked that it is the value 1; the MERL format uses the value 1 to indicate that the next word is a relocation address. (Later, we will discuss other entries that might exist in the MERL footer.)

Once the value 1 has been read, the algorithm retrieves the next `word`. This is the address that has to be relocated. But this address is computed including the 12-byte MERL header. In other words, this address is computed assuming the MIPS code starts at address `0x0c` or 12. The MIPS code has been moved from address 12 to address α , so the difference in address is $\alpha - 12$. This means the value at $\text{MEM}[\alpha + \text{rel} - 12]$, where `rel` is the value we read from the relocation entry, needs to be relocated.

Relocating involves adding an offset corresponding to the change in starting address. Again, since the MIPS code segment of a MERL file starts at address 12, and was moved to address α , the offset we add is actually $\alpha - 12$. This explains the meaning of the complicated line $\text{MEM}[\alpha + \text{rel} - 12] += \alpha - 12$.

The algorithm then moves to the next entry in the MERL footer (incrementing the counter `i` by 8 since two words from the footer were read). We repeat this until all relocation entries are processed.

10.2 Linking

Question 10.5.

How can we separate a MIPS program into multiple files?

Question 10.6.

While working with multiple files, one file might refer to code in a different file (using `labels`) and rely on the assembler to jump to that label. This poses the question: how would the assembler resolve a reference to a label when the label is defined in a different file?

Solution: One simplistic solution is to first combine the different assembly (`.asm`) files into a single file and then assemble it. This could be as simple as running a command such as:

```
1 cat one.asm two.asm three.asm | cs241.binas > combined.mips
```



Discovery 10.3

It does suffer from some limitations. First, it requires assembling all the code in one go, i.e., there is no ability to have pre-assembled versions of some files. Second, it assumes that all the source code is available. This is often not possible when using external libraries which might only be made available as pre-assembled files. Third, the different assembly files might have used the same labels. By concatenating these files, we might end up getting duplicate labels.

Solution: To avoid these limitations, one can consider assembling first and then concatenating. We can use our relocating assembler (`linkasm`) to produce MERL files for each assembly file, and then concatenate them.



Discovery 10.4

The problem with the above approach is that concatenating MERL files does not produce a valid MERL file. The combined.merl file would simply contain all the contents of `one.merl` followed by `two.merl` and then `three.merl`. But that is not valid MERL; it has multiple headers and footers interspersed through the code!

Result 10.1

We would need a smarter form of concatenation. This smarter concatenation algorithm is implemented as a program called the *linker*.

Comment 10.4

We will discuss this shortly after we discuss an even bigger problem:

Question 10.7.

What is our relocating assembler going to do when it encounters a label that is not defined in the same file?

The solution to handling externally defined references to labels is to update the assembler once again.

Solution: Whenever the assembler encounters a label that cannot be resolved, the assembler will create an *External Symbol Reference* (ESR) entry in the MERL footer. It will then become the job of the linker to ensure that this label is resolved when it links the different MERL files. 🎵

Question 10.8.

One question that arises is whether the assembler should create an ESR entry every time it cannot resolve a label.

Solution: Recall that an assembler directive is an instruction for the assembler to perform. We mentioned earlier that in addition to the `.word` directive, there are two other directives in our dialect of MIPS assembly. One of these is the `.import` directive, which tells a MERL assembler that a certain label is meant to be found in an external file. 🎵

Example 10.3

```
1 .import print
```

Given this `import` directive, now if the assembler encounters a `print` label, it will generate an ESR. If a label cannot be resolved, and there is no `import` directive for it, then this causes an assembler error like before. To illustrate the MERL format for ESRs, we will use the following short example:

```
1 ; File: one.asm
2 .import proc
3 lis $1
4 .word proc
5 jalr $1
```

Example 10.4

| Address | MERL represented as assembly | Actual MERL file (binary) |
|---------|---|---------------------------|
| 0x00 | beq \$0, \$0, 2 | 0x10000002 |
| 0x04 | .word endModule | 0x00000034 |
| 0x08 | .word endCode | 0x00000018 |
| 0x0c | lis \$1 | 0x00000814 |
| 0x10 | use1: .word 0 ; placeholder for proc | 0x00000000 |
| 0x14 | jalr \$1 | 0x00200009 |
| | endCode: | |
| 0x18 | .word 0x11 ; format code for ESR | 0x00000011 |
| 0x1c | .word use1 ; address where proc is used | 0x00000010 |
| 0x20 | .word 4 ; length of label | 0x00000004 |
| 0x24 | .word 112 ; ASCII for p | 0x00000070 |
| 0x28 | .word 114 ; ASCII for r | 0x00000072 |
| 0x2c | .word 111 ; ASCII for o | 0x0000006f |
| 0x30 | .word 99 ; ASCII for c | 0x00000063 |
| 0x34 | endModule: | |

The MERL file begins with the standard MERL header that has an instruction to jump over the next two words. The next two words are the addresses for where the module ends (0x34) and where the code segment ends (0x18). After the header is the code segment. The second word in this code segment is the use of `.word proc`. The assembler will of course discover that this label has not been declared in the assembly file. However, since the `import` directive is there, an ESR entry in the MERL footer will be created. The assembler must still produce some output for this word, as a placeholder for when the linker finds the actual address of `proc`. The assembler uses 0 as a placeholder. This zero word will be replaced by the linker once the actual address of `proc` is known. After the assembled code segment, there is the MERL footer containing the ESR entry for `proc`, beginning at address 0x18. The first word in the ESR entry is the format code for the ESR, the value 0x11. This is followed by the address of the location where the label was used. Since `.word proc` appears at address 0x10, that value is used. Next comes the length of the label, in this case 4 (p, r, o and c). Followed by the length, are the ASCII values for each character in the label. Let's now consider the other file; the file that is going to provide the label that will be linked (`proc` in our example above). One thing should be clear to the reader: the file providing the label would also need to be a MERL file, with some type of MERL entry that indicates where the provided label was in the assembly file. An assembled program does not have any labels, since labels are replaced with numeric values by the assemblerdo not exist in machine code.

Definition 10.4: External Symbol Definition


In the MERL format, entries for label definitions are called **External Symbol Definitions** (ESD).

Question 10.9.

Which labels should have ESD entries?

Comment 10.5

One could imagine creating an ESD for every label, but this would mean making labels visible that you never intended to (imagine every internal loop label being made available to link to).

Solution: A better approach is to expect the programmer to specify which labels they want to be linkable, i.e., the programmer specifically exports the labels. 

Theorem 10.2

For this reason, MIPS assembly has a `.export` directive to match our `.import` directive.

Example 10.5

For an example, we show the file `two.asm`, which exports the label `proc` that our previous file `one.asm` imported.

```
1 ; File: two.asm
2 .export proc
3 proc: jr $31
```

The assembled MERL file created for this file is shown below. Like before, we show the file in assembly for ease of reading, and we also show addresses for ease of referencing:

| Address | MERL represented as assembly | Actual MERL file (binary) |
|---------|--|---------------------------|
| 0x00 | beq \$0, \$0, 2 | 0x10000002 |
| 0x04 | .word endModule | 0x0000002c |
| 0x08 | .word endCode | 0x00000010 |
| | proc: | |
| 0x0c | jr \$31 | 0x03e00008 |
| | endCode: | |
| 0x10 | .word 0x05 ; format code for ESD | 0x00000005 |
| 0x14 | .word proc ; address where proc is defined | 0x0000000c |
| 0x18 | .word 4 ; length of label | 0x00000004 |
| 0x1c | .word 112 ; ASCII for p | 0x00000070 |
| 0x20 | .word 114 ; ASCII for r | 0x00000072 |
| 0x24 | .word 111 ; ASCII for o | 0x0000006f |
| 0x28 | .word 99 ; ASCII for c | 0x00000063 |
| 0x2c | endModule: | |

Result 10.2

Now that we have covered the complete MERL format, which includes ESR (imported label) and ESD (exported label) entries in addition to relocation entries, we can discuss the linking algorithm in detail.

10.2.1 Linking Algorithm

We give a detailed algorithm for linking two MERL files `m1` and `m2` in pseudocode. We first discuss the steps at a high level and then dive deeper into a concrete algorithm that the reader should have no difficulty converting into actual code.

Comment 10.6

In the following, we use `m1.code` and `m1.table` as shorthand to refer to just the code segment or table segment of the `m1` MERL file, respectively.

Algorithm 10.1

Task1: Check for duplicate exports. Confirm that the two MERL files do not both export the same symbol(s). There cannot be two ESDs for the same label.

Task2: Combine code segments. One important consideration is that the order in which the MERL files are provided to the linker matters. Linking files `m1` and `m2`, in this order, will result in `m1.code` appearing before `m2.code` in the linked file's code segment.

Comment 10.7

We also need to relocate `m2.code`; we discuss the reason in tasks 3 and 4. In task 2, we simply concatenate `m1.code` and `m2.code` without modifying either code segment; future tasks will be need to reach the final outcome shown in the diagram on the right.

Task3: Relocate `m2.table`. A MERL table contains REL (relocation), ESD (External Symbol Definition) and ESR (External Symbol Reference) entries. Since task 2 will move `m2`'s code segment to below `m1`'s code segment, we will need to update the addresses in REL, ESD and ESR entries in `m2`'s table. This will require determining a relocation offset which will be equal to where `m1`'s code segment ends minus the size of the header (12), since we will not be duplicating the header from `m2` in the linked file. In other words, every table entry in `m2` must now be updated by adding this relocation offset to the address that the entry contains.

Task4: Relocate `m2.code` using REL entries. Each REL entry specifies an address in the code which needs to be relocated: it represents a “.word label” occurrence where the label value is known. In task 2, we moved `m2.code` so that it appears after `m1.code`, which changed the starting address of `m2.code`. In task 3, we changed the entries in `m2.table` so they are relative to the new starting address. However, there is still one more step: the lines of code that REL entries refer to need to be adjusted to accord with the new starting address. We need to go through each REL entry in the modified `m2.table`, and update the corresponding lines of `m2.code` by adding the relocation offset computed in task 3 to each such line.

Comment 10.8

It might be hard to understand why this is necessary right now; we will see an example later.

Task5: Resolve imports for `m1`. For each import (ESR) in `m1.table`, check if `m2.table` has a corresponding export (ESD). If such an export is found, the import can be resolved. Resolving an import requires updating the location of where the ESR occurs in `m1`'s code with the address of where the exported label is defined. But that is not all! Recall that the ESR had been created because there was an unresolved label used with a .word. Since we have now resolved the label, we now have the use of a .word with a resolved label, i.e., we need a relocation entry. Therefore, we must change the ESR entry we just resolved into an REL entry since if the label's definition was to move (due to relocation), the address would need to be relocated.

Task6: Resolve imports for `m2`. Repeat task 5 but with the roles of `m1` and `m2` swapped.

Task7: Create the table for the linked MERL file. The table for the linked MERL file is the concatenation of all the ESD and REL entries along with any unresolved ESR entries from the tables from `m1` and `m2`. We are guaranteed there will be no duplicates. The order of the table entries does not matter.

Task8: Compute the header for the linked MERL file. Recall that the MERL header requires outputting the total size of the MERL file (`endModule`) and the location where the code segment ends (`endCode`). The value for `endCode` is simply the size of the header (12) plus the size of the

combined code segment concatenated in task 2. The value for endModule is endCode plus the size of the table that was created for the linked MERL file in task 7.

Task9: Output the linked MERL file. Output the header which requires outputting the first word (beq \$0, \$0, 2 or 0x10000002) followed by endModule and endCode. Then output the combined code that was produced in task 2, followed by the table that was produced in task 7.

We now give the algorithm discussed above as tasks in a more concrete pseudocode form below:

Code 10.5

```
1 // Step 1: Check for duplicate export errors
2 for each ESD in m1.table {
3     if there is an ESD with the same name in m2.table {
4         ERROR (duplicate exports)
5     }
6 }
7
8 // Step 2: Combine the code segments for the linked file
9 // The code for m2 must appear after the code for m1.
10 // We treat linked_code as an array of words containing just the
11 // concatenation of the code segments
12 linked_code = concatenate m1.code and m2.code
13
14 // Step 3: Relocate m2 table entries
15 reloc_offset = end of m1.code - 12
16 for each entry in m2.table {
17     add reloc_offset to the number stored in the entry
18 }
19
20 // Step 4: Relocate m2.code
21 // It is essential that this happen after Step 3
22 for each relocation entry in m2.table {
23     index = (address to relocate - 12) / word size
24     add relocation offset to linked_code[index]
25 }
26
27 // Step 5: Resolve imports for m1
28 for each ESR in m1.table {
29     if there is an ESD in m2.table with a matching name {
30         index = (address of ESR - 12) / word size
31         overwrite linked_code[index] with the exported label value
32         change the ESR to a REL
33     }
34 }
35
36 // Step 6: Resolve imports for m2
37 Repeat Step 5 for imports from m2 and exports from m1
```

```

38
39 // Step 7: Combine the tables for the linked file
40 linked_table = concatenate modified m1.table and modified m2.table
41
42 // Step 8: Compute the header information
43 endCode = 12 + linked_code size in bytes
44 endModule = endCode + linked_table size in bytes
45
46 // Step 9: Output the MERL file
47 output beq $0, $0, 2 (0x10000002)
48 output endModule
49 output endCode
50 output linked_code
51 output linked_table

```

10.2.2 Tracing Through the Linking Algorithm

We trace through the different steps of this algorithm using the following `m1.asm` and `m2.asm` files. Doing this manually is tedious and is going to take a while.

Comment 10.9

You can find the video explanation at [this link](#).

10.2.3 Practice

Using the linking algorithm just discussed, produce the resulting MERL file when the MERL files for `one.asm` and `two.asm` (shown earlier in the course module) are linked in that order.

| Address | MERL represented as assembly | Actual MERL file (binary) |
|---------|---|---------------------------|
| 0x00 | beq \$0, \$0, 2 | 0x10000002 |
| 0x04 | .word endModule | 0x00000034 |
| 0x08 | .word endCode | 0x00000018 |
| 0x0c | lis \$1 | 0x00000814 |
| 0x10 | use1: .word 0 ; placeholder for proc | 0x00000000 |
| 0x14 | jalr \$1 | 0x00200009 |
| | endCode: | |
| 0x18 | .word 0x11 ; format code for ESR | 0x00000011 |
| 0x1c | .word use1 ; address where proc is used | 0x00000010 |
| 0x20 | .word 4 ; length of label | 0x00000004 |
| 0x24 | .word 112 ; ASCII for p | 0x00000070 |
| 0x28 | .word 114 ; ASCII for r | 0x00000072 |
| 0x2c | .word 111 ; ASCII for o | 0x0000006f |
| 0x30 | .word 99 ; ASCII for c | 0x00000063 |
| 0x34 | endModule: | |

Listing 1: MERL file for `one.asm`

| | Address | MERL represented as assembly | Actual MERL file (binary) |
|----|---------|--|---------------------------|
| 1 | 0x00 | beq \$0, \$0, 2 | 0x10000002 |
| 2 | 0x04 | .word endModule | 0x0000002c |
| 3 | 0x08 | .word endCode | 0x00000010 |
| 4 | | proc: | |
| 5 | 0x0c | jr \$31 | 0x03e00008 |
| 6 | | endCode: | |
| 7 | 0x10 | .word 0x05 ; format code for ESD | 0x00000005 |
| 8 | 0x14 | .word proc ; address where proc is defined | 0x0000000c |
| 9 | 0x18 | .word 4 ; length of label | 0x00000004 |
| 10 | 0x1c | .word 112 ; ASCII for p | 0x00000070 |
| 11 | 0x20 | .word 114 ; ASCII for r | 0x00000072 |
| 12 | 0x24 | .word 111 ; ASCII for o | 0x0000006f |
| 13 | 0x28 | .word 99 ; ASCII for c | 0x00000063 |
| 14 | 0x2c | endModule: | |
| 15 | | | |

Listing 2: MERL file for two.asm

Solution: The result of linking is:

| | Address | MERL represented as assembly binary) | Actual MERL file (|
|----|---------|--|--------------------|
| 1 | 0x00 | beq \$0, \$0, 2 | 0x10000002 |
| 2 | 0x04 | .word endModule | 0x00000040 |
| 3 | 0x08 | .word endCode | 0x0000001c |
| 4 | 0x0c | lis \$1 | 0x00000814 |
| 5 | 0x10 | use1: .word 0x18 | 0x00000018 |
| 6 | 0x14 | jalr \$1 | 0x00200009 |
| 7 | | proc: | |
| 8 | 0x18 | jr \$31 | 0x03e00008 |
| 9 | | endCode: | |
| 10 | 0x1c | .word 0x05 ; format code for ESD | 0x00000005 |
| 11 | 0x20 | .word proc ; address where proc is defined | 0x00000018 |
| 12 | 0x24 | .word 4 ; length of label | 0x00000004 |
| 13 | 0x28 | .word 112 ; ASCII for p | 0x00000070 |
| 14 | 0x2c | .word 114 ; ASCII for r | 0x00000072 |
| 15 | 0x30 | .word 111 ; ASCII for o | 0x0000006f |
| 16 | 0x34 | .word 99 ; ASCII for c | 0x00000063 |
| 17 | 0x38 | .word 0x01 ; format code for REL | 0x00000001 |
| 18 | 0x3c | .word use1 ; location to relocate | 0x00000010 |
| 19 | 0x40 | endModule: | |
| 20 | | | |

Listing 3: Result of Linking



11 Memory Management

11.1 The Core Problem

Discovery 11.1

If you allocate an array and then index it by -1 , thus reading outside the memory you allocated, you won't (usually) cause a segmentation fault; you'll just be touching somebody else's memory.

Definition 11.1: Mutator

In descriptions of memory managers, the actual program is usually called the **mutator**.

Comment 11.1

From the perspective of the memory manager, you have a single, giant array, and some outside agent (i.e., the program) is going to make requests of you; from your perspective, the way to handle those requests is to select chunks of your single array, even if from the perspective of that mutator, what it looks like is different arrays.

11.2 The Free List Algorithm

The allocator begins with a linked list containing one node that represents the entire pool of free memory; the heap. As parts of the heap are allocated, the linked list is updated to keep track of which parts of the heap are free.

Question 11.1.

Where is this linked list data structure stored?

Solution: In the heap, of course!



Definition 11.2: Free List Algorithm

The **free list algorithm** is a technique for dynamic memory management where the system maintains a linked list of all unused memory blocks.

Algorithm 11.1

- Each node (block) in the list stores:
 - The **size** of the block
 - A **pointer** to the next free block
- When allocating memory:
 - Search the list for a block that is large enough.
 - Remove it (or split it) and return it to the user.

- When freeing memory (deallocation):
 - Return the block to the free list.
 - Optionally merge (coalesce) it with adjacent free blocks to avoid fragmentation.

11.2.1 Allocation Strategies

Theorem 11.1

1. **First-Fit:** Choose the first block that is large enough.
2. **Best-Fit:** Choose the smallest block that fits.
3. **Worst-Fit:** Choose the largest block available.

Code 11.1

```

1  struct Block {
2      int size;
3      Block* next;
4  };
5
6  Block* freeList;
7  void* allocate(int n) {
8      Block* prev = NULL;
9      Block* curr = freeList;
10     while (curr != NULL) {
11         if (curr->size >= n) {
12             if (prev == NULL) freeList = curr->next;
13             else prev->next = curr->next;
14             return (void*) curr;
15         }
16         prev = curr;
17         curr = curr->next;
18     }
19     return NULL; // no suitable block
20 }

```

11.2.2 Deallocation and Coalescing

Theorem 11.2

When a block of memory is no longer needed, it must be returned to the free list. This involves:

- Inserting the block in the appropriate position (to keep the list sorted by address, if needed).
- **Coalescing:** If adjacent blocks in memory are also free, they are merged into a single larger

block to reduce fragmentation.

- This step helps to maintain larger contiguous blocks in the future.

Code 11.2

```
1 void free(void* ptr, int size) {
2     Block* newBlock = (Block*) ptr;
3     newBlock->size = size;
4
5     Block* curr = freeList;
6     Block* prev = NULL;
7
8     // Find position to insert into sorted list
9     while (curr != NULL && curr < newBlock) {
10         prev = curr;
11         curr = curr->next;
12     }
13     // Insert newBlock
14     newBlock->next = curr;
15     if (prev == NULL) freeList = newBlock;
16     else prev->next = newBlock;
17     // Coalesce with next
18     if (newBlock->next &&
19         (char*)newBlock + newBlock->size == (char*)newBlock->next) {
20         newBlock->size += newBlock->next->size;
21         newBlock->next = newBlock->next->next;
22     }
23     // Coalesce with previous
24     if (prev &&
25         (char*)prev + prev->size == (char*)newBlock) {
26         prev->size += newBlock->size;
27         prev->next = newBlock->next;
28     }
29 }
```

Comment 11.2

An interesting edge case that arises is the following situation: Suppose there is a free block of 32 bytes, and the user requests 24 bytes. You need four bytes for bookkeeping, so you need to allocate 28 bytes. The free block gets split into two blocks: one with 28 bytes and one with 4 bytes (32-28). But this creates a free block that is only 4 bytes, which is not big enough to store both the size and the address of the next free block. This block cannot be added to the free list. One simple solution is to simply identify this edge case and allocate the entire block of 32 instead of splitting it. This is acceptable since while the program requested 24 bytes, it does not mean we cannot allocate more.

11.3 Garbage Collection

Many languages will automatically deallocate memory that was allocated by a program once that memory is no longer needed. This implicit memory management is carried out through a process called garbage collection³.

Definition 11.3: Garbage Collection

In **garbage collection**, allocated memory that is no longer needed is the garbage.

Theorem 11.3

Java and Racket are two examples of languages that perform automatic garbage collection.

Example 11.1

Consider the following Java example:

```
1 void foo() {  
2     MyClass obj = new MyClass();  
3     ...  
4 } //obj no longer accessible
```

The method above heap allocates an object `obj`. Once the method returns, that object is no longer accessible. The garbage collector will automatically collect the memory associated with `obj`, i.e., there is no need for the programmer to explicitly deallocate the object.

Example 11.2

Consider the following example:

```
1 int f() {  
2     MyClass obj2 = null;  
3     if (x == y) {  
4         MyClass obj1 = new MyClass();  
5         obj2 = obj1;  
6     } // obj1 goes out of scope;  
7     ...  
8 } // obj2 no longer accessible;
```

The above example is meant to illustrate that garbage collection algorithms need to be quite sophisticated. The garbage collector must take the most conservative approach, i.e., it must not deem any memory to be garbage until it is absolutely certain that the program will never refer to it.

11.3.1 Reference Counting

³Garbage Collection is a huge field and in fact there are courses dedicated to just covering the different garbage collection algorithms. We give an extremely superficial intro to some of the popular algorithms.

Algorithm 11.2

The Reference Counting algorithm keeps track of the number of pointers that point to each block. Deallocating memory that is no longer needed is straightforward; whenever the reference count of a block reaches 0, that block can be reclaimed.

Comment 11.3

Readers familiar with `std::shared_ptr` are likely already familiar with the above description.

Discovery 11.2

The algorithm does also suffer from one other limitation: circular references, a block of memory that refers to another block of memory which refers back to the first block. Together these two blocks might not be accessible from anywhere else in the program but, since their reference counts are not zero, they would never be deallocated.

Result 11.1

Because of this limitation, reference counting is generally not considered as a complete or workable algorithm for garbage collection.

11.3.2 Mark and Sweep

Algorithm 11.3

This algorithm begins with a *Mark* phase where it discovers parts of the heap that are reachable from the stack and global variables. The entire stack (and global variables) is scanned for pointers leading into the heap. Each such heap block is marked as “reachable”.

If the marked heap blocks contain pointers, the algorithm follows any such pointers to discover new parts of the heap that are also reachable, and mark those as well. This is repeated again and again as long as more reachable blocks are discovered.

Once no more reachable blocks are found, meaning the entire reachable part of the heap has been marked, the algorithm conducts the *Sweep* phase; any block that was not marked is deallocated, since it was unreachable.

Theorem 11.4

The Mark and Sweep algorithm belongs to a class of garbage collection algorithms which can be referred to as “Stop the World” algorithms.

Result 11.2

While the algorithm can accurately collect garbage, it does suffer from the disadvantage of having to

stop the program from executing while the collector runs.

11.3.3 Copying Collector

Copying Collectors involve copying live blocks. The classic and original copying collector is *Cheney's Algorithm*, which splits the heap into two halves named **from** and **to**. Memory is only allocated from the **from** part of the heap. When this half fills up, the garbage collector runs and copies the reachable parts from **from** to **to**. Then the roles of **from** and **to** are reversed.

Discovery 11.3

One advantage of using this approach is that of automatic compaction; since memory is copied from one half of the heap to another, it can be laid out contiguously thereby avoiding any fragmentation.

Result 11.3

The algorithm is also a “Stop the World” algorithm and is not truly suitable for applications that expect real-time response guarantees. Additionally, the algorithm halves the amount of heap memory available to the program.

11.3.4 Generational Garbage Collection

In particular, copying collectors tend to work well when few objects survive collection (because there are fewer live blocks to copy over when the heap fills up) and mark-and-sweep works better when most objects survive collection (because there are fewer things to deallocate in the sweep phase).

Discovery 11.4

It has been observed that most objects die young, so many garbage collectors use this intuition to fuse multiple techniques.

Algorithm 11.4

The idea is that heap objects are split into generations where new objects are allocated in the youngest generation and collected through copying collection. Objects that survive these collections are moved to an older generation which uses mark-and-sweep or similar algorithms. The frequency of collection also varies, with the younger generations collected more frequently than older generations.

12 Closing Thoughts

You learned how a program is broken down into tokens by a scanner. These tokens are processed and built up into a tree structure by a parser, which is then analyzed in the semantic analysis phase and finally traversed to generate code.

You also learned how the generated assembly language code can be assembled, then linked together to produce a executable machine code program. This program can be placed in memory by a loader and finally run by the machine.

If you have fully grasped the course material and completed all the projects, making basic extensions to WLP4 should feel accessible. For example, if you understand the principles behind while loops, it is not too hard to implement other kinds of loops like `for`. A more ambitious extension could be to add a `bool` type to WLP4, and add support for `true` and `false` constants, as well as logical operators like `&&` or `||`, and allow for more complex tests in conditional statements and loops. Perhaps you could remove some of the annoying restrictions of WLP4; for example, what if you were allowed to write negative numbers in expressions without doing something like `0 - 1`? What changes would need to be made to the tokens, the context-free grammar, and the type system? How would you implement these changes in your compiler?

Some language features you may use regularly as a programmer, like the classes and methods of Object-Oriented Programming, may still seem incredibly daunting to implement. The upper year compilers course, CS 444, discusses the process of compilation at a deeper level and asks you to implement more advanced features.

The compiler you wrote, if you followed the recommendations in these notes, does not produce particularly efficient code. Real-world compilers apply a wealth of complicated optimization techniques in an attempt to produce the fastest code they can. Some of these optimization techniques are discussed in the optional Module 9b.

You may not ever write a compiler or debug an assembly language program ever again after this course, but the understanding of the process of compiling and executing programs will hopefully stick with you. Computers and programs are not magic. They are extremely complex things built on layers and layers of abstraction. Whether you consider yourself a “computer scientist”, a “software engineer”, or simply a “programmer”, the ability to peel away these layers and get to the heart of a problem is incredibly valuable.

Index

- ϵ -transitions, 50
- \vdash and \neg , 61
- `code(expr)`, 106
- `push(reg) & pop(reg)`, 107

- Abstract Syntax Tree, 88
- Alphabet, 18, 48
- Ambiguous, 58
- Analysis, 26
- Assembly Language, 17

- Bit, 6
- Bus, 14
- Byte, 6

- Calling Convention, 116
- Compiler, 17
- Computer Program, 13
- Concatenation, 20
- Concrete Syntax Tree, 88
- Context Free Grammar, 53
- Context-free, 56
- CPU, 13

- Derive, 55
- DFA, 23
- Directly Derive, 55

- Empty String, 48
- External Symbol Definition , 129

- Frame Pointer, 104
- Free List Algorithm, 135

- Garbage Collection, 138
- Grammar, 53

- Hexadecimal Notation, 6
- hi and lo, 14

- Input in MIPS, 38
- Instructions, 13
- Item, 75
- Kleene Star, 20

- Label definition, 34
- Language, 19, 48
- Language of CFG, 55
- Left Factoring, 73
- Left Recursive Grammar, 72
- Leftmost Derivation, 57
- Length, 19
- LL(1) Grammar, 65
- Lookahead, 62
- LR(0), 81
- LR(0) Parsing Algorithm, 78
- LR(0) Parsing DFA, 78
- LR(1) DFA, 84
- LSB, 9
- Lvalue, 108

- MSB, 9
- Mutator, 135

- Nibble, 6
- Nondeterministic Finite Automata, 49
- Nonterminal Symbols, 53

- Object Code, 123
- Output in MIPS, 38

- Parse Tree, 57
- Parsing, 26, 60
- Post System of notating deductive logic, 96
- Procedure, 45
- Production Rules, 54

- Recognition Problem, 21
- Reduce, 75
- Reduce-Reduce Conflict, 81
- Registers, 14
- Regular, 20
- Relocation Entry, 124
- Runtime environment, 114

- Scanner, 18
- Semantic Analysis, 38
- Semantics Analysis, 26
- Setential Form, 80

Shift, 75
Shift-Reduce Conflict, 81
Sign-magnitude Representation, 7
Signature, 93
Simple LR(1), 82
Simplified Maximal Munch, 25
Snanning, 26
Stack Pointer, 43
Start Symbol, 54
Static Allocation, 41
String, 18, 48
Terminal Symbols, 53
Tokenizer, 18
TOS, 63
Two's Complement Representation, 7
Unary Operation, 108
Union, 20
Viable Prefix, 80
Well-typed, 98
Word, 6