CS 241 Spring 2018

Foundations of Sequential Programs

Kevin Lanctot

Much of this material comes from, or is based on, lecture notes by Brad Lushman and lectures slides by Troy Vasiga.

Table Of Contents

Topic 1 – Representing Data

Key Ideas

- Understand Binary, Decimal, Two's Complement and Hexadecimal representations of integers
- Converting between binary and decimal numbers
- Adding and subtracting binary numbers
- Data representation: bit, nibble, byte and word
- Representing Characters: ASCII, Unicode

References

- CO&D sections 2.4 and 2.9
- https://www.student.cs.uwaterloo.ca/~cs241/ConversionChart.pdf

Number Systems

The Decimal Number System

- *Humans* often represent numbers using combinations of 10 different symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
- Called *base 10*, *radix 10* or the *decimal system.*

The Binary Number System: Signed and Unsigned Integers

- *Computers* represent numbers using combinations of 2 different symbols {0, 1}.
- Called *base 2*, *radix 2* or the *binary system.*

The Hexadecimal Number System

- *Compromise* easier to use than binary but harder than decimal
- Represent numbers using combinations of 16 different symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}.

Binary Number System

Why Do Computers Use Binary?

- Originally used base 10.
- Led to complicated designs in the age of vacuum tubes.
- Have to be able to distinguish between 10 different states.
- Konrad Zuse’s mechanical computer Z1 (developed 1935 – 1938) was the first to use a binary representation.
- It led to a much *simpler design.*
- Bonus: it is also a *more reliable* way to ...
  - store information over time, e.g. hard drive
  - transmit information over distance, e.g. network

Unsigned Integers

Decimal Representation

$$50{,}320_{10} = 5 \cdot 10^4 + 0 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 0 \cdot 10^0$$

$$50{,}320_{10} = 5 \cdot 10000 + 0 \cdot 1000 + 3 \cdot 100 + 2 \cdot 10 + 0 \cdot 1$$

5	0	3	2	0
place holder for 10,000's	place holder for 1,000's	place holder for 100's	place holder for 10's	place holder for 1's

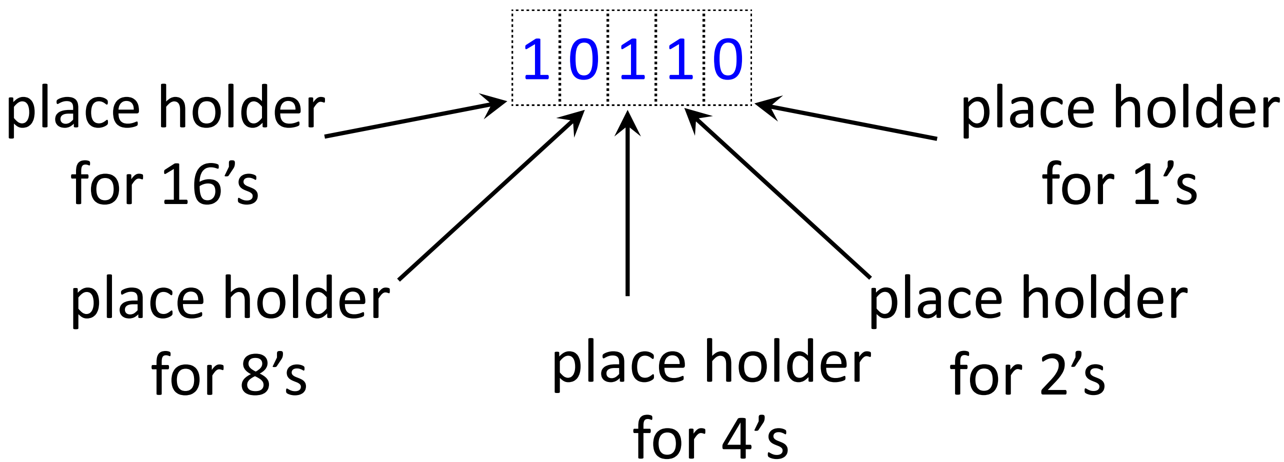
- *key idea:* each time you move over one digit from right to left, multiply the placeholder *by 10*

Unsigned Integers

Binary Representation

$$1\,0110_{\mathbf{2}} \;=\; 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 \;=\; 22_{\mathbf{10}}$$

$$1\,0110_{\mathbf{2}} \;=\; 1\cdot 16 + 0\cdot 8 + 1\cdot 4 + 1\cdot 2 + 0\cdot 1 \;=\; 22_{\mathbf{10}}$$



- *key idea:* each time you move over one digit from right to left, multiply the placeholder *by 2*
- write 2 or 10 as a subscript to distinguish the representations

Unsigned Integers

Converting Binary → Decimal Representation

key idea: explicitly write the value of each placeholder

E.g. 1010_2

$1010_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

$1010_2 = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$

$1010_2 = 10_{10}$

E.g. 10110_2

$10110_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

$10110_2 = 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$

$10110_2 = 22_{10}$

Unsigned Integers

Converting Decimal $\rightarrow$ Binary Representation

- repeatedly divide by target base (i.e. 2)
- keep track of the quotient and the remainders
- remainders generate bits from *right to left*...

Example

- Convert 22_{10} to binary format

  22 / 2 = 11 remainder 0

  11 / 2 = 5 remainder 1

  5 / 2 = 2 remainder 1

  2 / 2 = 1 remainder 0

  1 / 2 = 0 remainder 1
- therefore $22_{10} = 10110_2$

Convert from One Radix to Another

Why Does this Algorithm Work?

- try converting decimal to decimal to see how it works
- repeatedly divide by target base (i.e. 10)
- remainders generate digits from *right to left*...

Example

- Convert 50320_{10} to decimal format

50320 / 10 =	5032	remainder	0
5032 / 10 =	503	remainder	2
503 / 10 =	50	remainder	3
50 / 10 =	5	remainder	0
5 / 10 =	0	remainder	5

- therefore $50320_{10} = 50320_{10}$

Binary Addition

- similar to addition of decimals
- add digits from right to left and include carry
- with these basic rules… you can calculate any sum

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \quad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ +\,1 \\ \hline 10 \end{array}$$

$$\begin{array}{r} {\scriptstyle 1\,1\;\;\;\;} \\ \mathtt{00001}_2 \\ +\;\mathtt{01011}_2 \\ \hline \mathtt{01100}_2 \end{array} \qquad \begin{array}{r} \\ \mathtt{1}_{10} \\ +\mathtt{11}_{10} \\ \hline \mathtt{15}_{10} \end{array}$$

Two Issues

1. Fixed width (*i.e. n*-bit representation) means the possibility of *overflow*: the answer may take more than *n* bits to represent. We'll ignore this issue, but CS251 doesn't.
2. How do we represent negative numbers?

Signed Integers: Attempt 1

Issues with Sign Extension

First some vocabulary…

- fixed width *n*-bit representation
  - *most significant bit (MSB)*: left-most bit (highest value)
  - *least significant bit (LSB)*: right-most bit (lowest value)
- Attempt 1: *sign extension*
  - i.e. treat the MSB as the sign
  - 0 means positive, 1 means negative
  - e.g. 0001_2 is $+1_{10}$, 1001_2 is -1_{10} (in four bit case)
- Problem

  two ways to represent zero: 0000 and 1000

Signed Integers: Attempt 2

4-bit Two's Complement

- *goal:* get rid of this pesky two 0's issue
- to represent a negative number: *invert the bits and add 1*

		invert		add 1		
0_{10}:	0000	$\rightarrow$	1111	$\rightarrow$	0000	0_{10}
1_{10}:	0001	$\rightarrow$	1110	$\rightarrow$	1111	-1_{10}
4_{10}:	0100	$\rightarrow$	1011	$\rightarrow$	1100	-4_{10}
7_{10}:	0111	$\rightarrow$	1000	$\rightarrow$	1001	-7_{10}

- now have a single zero: 0000
- bonus: easier to implement in hardware
- *note*: because you invert bits, you *must always specify the word size*

  -1 in 8-bit two's complement is 1111 1111

  -1 in 16-bit two's complement is 1111 1111 1111 1111

Two's Complement

4-bit 2's Comp

7_{10}	0111
6_{10}	0110
5_{10}	0101
4_{10}	0100
3_{10}	0011
2_{10}	0010
1	0001
0	0000
-1	1111
-2_{10}	1110
-3_{10}	1101
-4_{10}	1100
-5_{10}	1011
-6_{10}	1010
-7_{10}	1001
-8_{10}	1000

Why Does Two's Complement Work?

- *Key Idea:* The MSB represents $-(2^{n-1})$, the rest represent positive powers of two.
- This change makes no difference for positive numbers, just for negative ones.

$0\cdot(-2^3) + 0\cdot 2^2 + 1\cdot 2^1 + 2\cdot 2^0 = 2+1 = 3$

$1\cdot(-2^3) + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+4+2+1 = -1$

$1\cdot(-2^3) + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = -8+3 = -5$

$1\cdot(-2^3) + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 = -8$

Two's Complement

Why Does Two's Complement Work?

- *Key Idea:* Ask what binary pattern would be added to *x* in order to get 0. That is the pattern for -*x*. E.g. let *x* = 1 in 8-bit 2's comp.

```
 1 1 1 1 1 1 1 1
   0 0 0 0 0 0 0 1    (1)      note we ignore the last carry bit
 + 1 1 1 1 1 1 1 1   (-1)      more on that later
   ---------------
   0 0 0 0 0 0 0 0
```

- It does not matter if the 0's or 1's occurs in the bottom or top row. E.g. let *x* = 10 1101 (45_{10}) in 8-bit 2's complement.

```
 1 1 1 1 1 1 1 1
   0 0 1 0 1 1 0 1    (45)     we need two 1's to
 + 1 1 0 1 0 0 1 1   (-45)     get the first carry bit
   ---------------
   0 0 0 0 0 0 0 0
```

and the rest is the complement

Two's Complement

4-bit 2's Comp

7_{10}	0111
6_{10}	0110
5_{10}	0101
4_{10}	0100
3_{10}	0011
2_{10}	0010
1	0001
0	0000
-1	1111
-2_{10}	1110
-3_{10}	1101
-4_{10}	1100
-5_{10}	1011
-6_{10}	1010
-7_{10}	1001
-8_{10}	1000

Two's Complement Shortcut

Algorithm: Working from right (LSB) to left (MSB)

a) copy the bits up to and including the first 1

b) for the rest, put the complement

2: 0010
-2: 1110

3: 0011
-3: 1101

4: 0100
-4: 1100

5: 0101
-5: 1011

6: 0110
-6: 1010

7: 0111
-7: 1001

Two's Complement

Why Does Two's Complement Work?

- it is *modular arithmetic* but wraps around after 7 rather than after 15
- e.g. -1 ≡ 15 mod 16
  comp(0001) + 1 = 1110 + 1 = 1111 = 15_{10}
- e.g. -4 ≡ 12 mod 16
  comp(0100) + 1 = 1011 + 1 = 1100 = 12_{10}
- e.g. -7 ≡ 9 mod 16
  comp(0111) + 1 = 1000 + 1 = 1001 = 9_{10}
- In two's complement, *the most significant bit of a negative number always 1*

	Signed	Unsigned
0111	7	7
0110	6	6
0101	5	5
0100	4	4
0011	3	3
0010	2	2
0001	1	1
0000	0	0
1111	-1	15
1110	-2	14
1101	-3	13
1100	-4	12
1011	-5	11
1010	-6	10
1001	-7	9
1000	-8	8

Subtraction

How to subtract

To subtract, just add the two's complement of the second value (the subtrahend)

Example 1: 6-5

```
0101      5
1011      -5 in 4-bit 2's comp

1 1 1 0
 0110       6
+1011    +(-5)
10001       1
```

ignore last carry bit

Example 2: 6-7

```
0111      7
1001      -7 in 4-bit 2's comp

0 0 0 0
 0110       6
+1001    +(-7)
01111      -1
```

ignore last carry bit

Two's Complement: Overflow

Example 3: 5 + 3

5 + 3 = overflow error in 4-bit two's complement

```
~~0~~ 1 1 1
  0101          5
+0011          +3
-----          --
  1000         -8
```

- If two positive integers are added together and the result is negative, this change in sign indicates an *overflow error*.
- When adding 5 + 3, there is overflow in Example 3.
- You can also have overflow when you add two negative numbers and get a positive one.

Hexadecimal Numbers

The Problem with Humans using Binary Numbers

- *problem:* binary digits are hard to read or remember and it is easy to make a mistake reading or typing them
- *convention:* typically binary numbers are written with a space after every four bits (starting from the right)
  - incorrect: 10110100011000010010111000111111
  - correct: 1011 0100 0110 0001 0010 1110 0011 1111
- *simplification:* after grouping them, convert each group of four bits to a decimal value:

1011	0100	0110	0001	0010	1110	0011	1111
11	4	6	1	2	14	3	15

Hexadecimal Numbers

The Problem with Humans using Binary Numbers

- *key idea:* introduce six new symbols {a, b, c, d, e, f} to represent the two-digit values 10, 11, 12, 13, 14, and 15
- 1011 0100 0110 0001 0010 1110 0011 1111 is represented as

  b 4 6 1 2 e 3 f
- There are a variety of ways to represent a number in hexadecimal: e.g. it can be written as ...

  bad0124 or BAD0124 or 0xbad0124 or 0xBAD0124
- i.e. you may use *capital or small letters, often with a leading 0x…*

Hexadecimal Numbers

Table to Convert between Binary and Hexadecimal

$0000_{bin} = 0_{hex}$	$1000_{bin} = 8_{hex}$
$0001_{bin} = 1_{hex}$	$1001_{bin} = 9_{hex}$
$0010_{bin} = 2_{hex}$	$1010_{bin} = a_{hex}$
$0011_{bin} = 3_{hex}$	$1011_{bin} = b_{hex}$
$0100_{bin} = 4_{hex}$	$1100_{bin} = c_{hex}$
$0101_{bin} = 5_{hex}$	$1101_{bin} = d_{hex}$
$0110_{bin} = 6_{hex}$	$1110_{bin} = e_{hex}$
$0111_{bin} = 7_{hex}$	$1111_{bin} = f_{hex}$

Who Uses What

Where are they used

- *Humans* use and represent numbers in decimal.
- *Computers* use and represent numbers in binary.
- People! Computers! Why can't we all just get along?
- Compromise position
  - When looking at the *low level workings* of a computer, programmers often use hexadecimal.
  - When talking about *memory locations* (pointers, references) programmers often use hexadecimal.
  - *Why: It is easy to convert* between hexadecimal and binary representation.

Data Representation

How to Interpret Data

- *Interpretation is in the eye of the beholder.*
- What does the following bit pattern represent?

  0111 1100 0110 0001 0010 1110 0011 1111
- It could be an unsigned 32-bit int, a signed 32-bit int, two unsigned 16-bit ints, 4 English chars, 1 char from a foreign language, a machine instruction, part of an audio clip, a picture, a video, etc.
- Storage devices (typically) represent data as 0’s and 1’s.
- Digital circuits just process 0’s and 1’s.
- We must (somehow) keep track of what the data means, i.e. *context*.

Data Representation

Bit

- a single 1 or 0 (voltage level, magnetic orientation)

Nibble

- 1 nibble = 1 hexadecimal digit = 4 bits

Byte

- 1 byte = 2 hexadecimal digits = 8 bits
- useful range to represent an English character

Data Representation

Word

- It depends on the processor:
  - for 32-bit *architecture*: 1 word = 4 bytes = 32 bits,
  - for 64-bit architecture: 1 word = 8 bytes = 64 bits.
- For CS 241, we'll used a 32-bit architecture
  - i.e. the processor can transfer 32 bits in parallel (at the same time).
- As more transistors can fit on a chip, it increases the circuit capacity.
- Individual bytes are still accessible from memory.

Representing Data: ASCII

American Standard Code for Information Interchange (ASCII)

ASCII to Hex conversion: e.g. A is hex 41, C is hex 43, S is hex 53

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	“	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Representing Data: ASCII

Another Way of Representing the ASCII Table

bin	dec	hex	char
0	0	0	NUL
1	1	1	STX
10	2	2	SOT
11	3	3	ETX
100	4	4	EOT
101	5	5	ENQ
110	6	6	ACK
111	7	7	BEL
1000	8	8	BS
1001	9	9	HT
1010	10	A	LF

bin	dec	hex	char
101011	43	2B	+
101100	44	2C	,
101101	45	2D	-
101110	46	2E	.
101111	47	2F	/
110000	48	30	0
110001	49	31	1
110010	50	32	2
110011	51	33	3
110100	52	34	4
110101	53	35	5

Representing Data: ASCII

ASCII Cautions

- *ASCII inherited much from Baudot (meant for teletypes)* including control characters such as SOH (start of header) STX (start of text) ETX (end of text), EOT (end of transmission), LF (line feed), CR (carriage return)
- the first 32 symbols are control characters
- Different OS's interpret some of them differently
- To end a line in ...
  - Linux / UNIX: "\n"
  - MS Windows text editors: "\r\n"
  - Macs up to OS-9 "\r"
- in Linux use dos2unix to convert Windows text files to Linux text files (i.e. remove the \r's).

Representing Data: Multilingual Codes

Unicode

- originally different countries had different codes
- hard to mix different languages in the same document
- *goal: create a standard for most written languages*
- Unicode = *Uni*fication *Code*
- currently ~110,000 characters from ~100 scripts
  - English, French, Spanish, Italian, etc., use a Roman script.
  - Russian, Ukrainian, Serbian, etc., use a Cyrillic script
  - Arabic, Persian, Pashto, Kurdish, etc., use an Arabic script.
- programming languages that have multilingual support use Unicode rather than ASCII to represent text (e.g. Python, Java).

Topic 2 – MIPS Assembly Language

Key Ideas

- High Level Language vs. Assembly Language vs. Machine Code
- opcodes (operation codes) and operands
- the CS241 subset of the MIPS32 instruction set

References

- CO&D Chapter 2 *Instructions: Language of the Computer*
- https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsref.pdf

Overview

High Level Language - HLL
- e.g. C, C++, Racket, Python

↓

Assembly Language - AL
- e.g. MIPS, x86-64, ARMv8

↓

Machine Code - MC
- sequence of 0's and 1's associated with a particular processor

```
a += 1;
```

↓

```
lis $1
.word 0x1
add $2, $2, $1
```

↓

0001 0011 1000 0000
0010 1010 0101 0100
0100 0100 0010 0000
0100 0010 0011 1010
0010 0110 0100 0001…

For binary numbers, *put a space every 4th bit* to make it easier to read.

Overview

High Level Language (HLL)

- meant to be read and *understood by humans* (smart ones anyways ;-)
- meant to be as *convenient as possible for computer programmers*
- processor independent
  - e.g. can use C++ for many difference processors
- a single statement in a HLL may be translated into several statements in Assembly Language
- most programmers program in a HLL

Overview

Machine Code (MC)

- meant to be *executed by processors*
- meant to be convenient for computer hardware *so that computer processors can execute it quickly*, e.g. use a binary encoding, 2's complement etc.
- e.g. Jellybean challenge
- processor dependent: machine code that works for an Intel Core i7 won't work on an ARMv8 processor
- no sane person today (except as a brief learning experience) programs in machine code
- also called *Machine Language*

Overview

Assembly Language (AL)

- meant to be a *compromise between a HLL and MC*
- it is MC with simple modifications so that humans can understand it easier (e.g. written in mnemonics, assembler directives, labels).
- for the most part, a single statement in AL is translated to a single statement in machine code
- you can take the AL for one processor and run it on another (that's what we'll be doing in CS241) using a simulator
- only a small minority of programmers program in AL
- an *Assembler* translates a program from assembly language to machine code
- you will be building a MIPS assembler in this course

MIPS Architecture

What is MIPS

- MIPS is one particular family of processors
- popular, simple and *easiest to learn*
- If you look up MIPS on the web note that
  - multiple revisions exist, e.g. MIPS I, MIPS II, MIPS III, ...
  - it has evolved over time ⇒ it is not just a single standard
  - the version we will be looking at, MIPS32, is a 32-bit architecture, ignore the rest
- recall that a 32-bit architecture means the pathways from one component to the next transfer 32 bits in parallel
- for MIPS, each instruction also takes exactly 32 bits
  - other processors, such as x86-64, have variable length instructions

C++ vs. MIPS Assembly Language

C++ code:

```
a = 10;
b = 15;
c = a + b;
```

Equivalent MIPS Assembly Language:

```
lis $5            ; load the next word into register 5
.word 0xa         ; a is hexadecimal for 10
lis $7            ; load the next word into register 7
.word 0xf         ; f is hexadecimal for 15
add $3, $5, $7    ; register 3 = register 5 + register 7
jr $31            ; jump to the address stored in $31
                  ; i.e. terminate the program
```

High Level vs. Assembly Language

Assembly Language

- one instruction per line
- uses mnemonics for instructions, e.g. *lis* for load immediate and skip, *jr* for jump (to address stored in) register
- *big difference: assembly language uses registers rather than variables* to hold and manipulate data (e.g. *$3, $5, $7*)
- can have a large number of variables in a HLL but there are only a limited number of general purpose registers in AL
- for MIPS32
  - there are 32 registers, called $0 .. $31
  - each register holds 32 bits
- typical range for the number of general purpose registers in many current processors is 15–32 (e.g. x86-64 and ARMv8)

High Level vs. Assembly Language

Registers

- registers are a small amount of very fast memory (e.g. 128 bytes) *where the processor stores data temporarily so it can manipulate it* (e.g. add, sub etc.)
- we will use the numerical names $0-$31
- you may also see names like a0, a1, v0, v1, fp, sp, ra, etc. for registers which indicate how they are typically used
- just like we sometimes use variables *x, y* and *z* to represent three numbers, we will sometimes use $s, $t and $d as generic names for three registers where *s*, *t* and *d* can be anyone of the 32 registers

High Level vs. Assembly Language

Arithmetic Operators and Registers

- In a *High Level Language,* you typically manipulate data in terms of variables, arithmetic operators and functions, e.g.

  total = subtotal + GST;

  root1 = (-b + sqrt((b**2) – (4*a*c))) / (2*a);

- In *Assembly Language*
  - use words (mnemonics): *add*, *sub*, *mult*, *div* rather than symbols +, -, *, /
  - specify registers, e.g. $2, rather than variables
  - some registers have a specific purpose
    - in MIPS, we reserve $29 for the frame pointer (fp), $30 for stack pointer (sp), $31 for a return address (ra) and $0 always contains zero (more about these terms later)

Machine Code

What is Machine Code (MC)

- binary code – comprised of 0s and 1s
- directly executed by the processor
- the program (a sequence of bits) is split into instructions with the following format:
  - operation code (*opcode*) + *operands*
  - instructions specify what operations the processor should execute and the location of the data
    - *opcode* designates the *operation*, say add or sub
    - *operands* designate the *data sources and destination*, which are either registers or (sometimes) memory locations in RAM
- e.g. in AL add $d, $s, $t means set the value in $d to be equal to the value in $s plus the value in $t (i.e. $d = $s + $t)
- same order you would write it in C / C++ / Java / Python etc.

Machine Code

Example: add

in AL: add $d, $s, $t

in MC: 0000 00ss ssst tttt dddd d000 0010 0000

- opcode
  - in AL: add
  - in MC: 0000 00 ____ ____ ____ 000 0010 0000
- operands
  - in MC: *sssss*, *ttttt*, and *ddddd* are binary numbers between 00000 and 11111 that specify which registers ($0 to $31) to obtain (the source) and store (the destination) the data
  - $2^5 = 32$, so it takes 5 bits to specify 32 registers

Machine Code

Example: add

- format for add \$d, \$s, \$t
  in MC: 0000 00ss ssst tttt dddd d000 0010 0000
- e.g. add \$1, \$3, \$7
  in MC: 0000 0000 0110 0111 0000 1000 0010 0000
- e.g. add \$3, \$7, \$15
  in MC: 0000 0000 1110 1111 0001 1000 0010 0000
- e.g. add \$7, \$15, \$31
  in MC: 0000 0001 1111 1111 0011 1000 0010 0000
- recall $1_{10}=00001_2$ $3_{10}=00011_2$ $7_{10}=00111_2$
  $15_{10}=01111_2$ $31_{10}=11111_2$

Machine Code

Example: add vs. sub

- add $d, $s, $t in AL is the following in MC
  0000 00ss ssst tttt dddd d000 0010 0000 and
- sub $d, $s, $t in AL is the following in MC
  0000 00ss ssst tttt dddd d000 0010 0010
- the *opcode* is a bit pattern that turns on and off various components of the processor so that whatever flows to the Arithmetic Logic Unit (ALU) will be added (if the 2nd last bit is 0) or subtracted (if the 2nd last bit is 1)
- the operands $s and $t signal which register values should flow into the ALU to be added or subtracted
- the operand $d specifies where the result should be stored

Instruction Set

Varieties of Instruction Sets

- An *instruction set* is the repertoire of *instructions understood by a processor.*
  - e.g. *add*, *sub*, *lis* (load immediate and skip) and *jr* (jump register) that we saw in the samples of MIPS assembly language
- Different processors have different instruction sets but they have many commonalities.
- We will use a subset of the MIPS instruction set listed here: https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsref.pdf
- In order to keep our assignments simple, we will restrict ourselves to these 20 instructions.

Some Basic MIPS AL Instructions

Trivial C Program:

```
void main(){
   return;
}
```

Equivalent MIPS Program

```
jr $31
```

- When the OS starts a program, it allocates some resources (such as memory) to the program and it puts a return address in $31.
- *To end a program jump to the address stored in $31*, i.e. jump back to the OS, which will free up the resources.
- In CS241 your programs should always end with `jr $31`.
- It gracefully terminates your program and the simulator (instead of the OS) will print out some useful information and then exit.

Some Basic MIPS AL Instructions

Addition and Subtraction

add $d, $s, $t

- i.e. $d = $s + $t
- add (the contents of) registers $s and $t
- place result in register $d

sub $d, $s, $t

- i.e. $d = $s - $t
- subtract (the contents of) register $t from (the contents of) register $s
- place the result in register $d

Some Basic MIPS AL Instructions

Assembly Language Instructions: add, sub

- always have two sources (of data) and one destination (for the result)

  C++: r1 = r2 + r3;

  MIPS : add $1, $2, $3

- the destination can be the same as one of the sources

  C++: r1 += r2;

  C++: r1 = r1 + r2;

  MIPS: add $1, $1, $2

- could even have

  MIPS: add $1, $1, $1

Some Basic MIPS AL Instructions

Arithmetic Operations, e.g. add

- complex expressions must be broken up into a sequence of simpler expressions that each have two source operands/registers and one destination

C++: r1 = r2 + r3 + r4 + r5

means r1 = (((r2 + r3) + r4) + r5)

MIPS :

```
add $1, $2, $3
add $1, $1, $4
add $1, $1, $5
```

Some Basic MIPS AL Instructions

Jumping

jr $s

- meaning: jump (to the address stored in) register $s and start executing code at this new location
- *used to implement returning from a function call or a program*
  - load my current address into $s
  - then call the function, i.e. go to a different address
  - when the function is done, I need to return to the address (or location) where I came from so I execute jr $s
- E.g. there could be many places in C++ code where I call sqrt(). Each time I call it, I first need a store my current location so that when sqrt() is done, it knows where to return to.
- *Convention:* for a function, register $31 holds the address you return to after the function (or program) is done

Some Basic MIPS AL Instructions

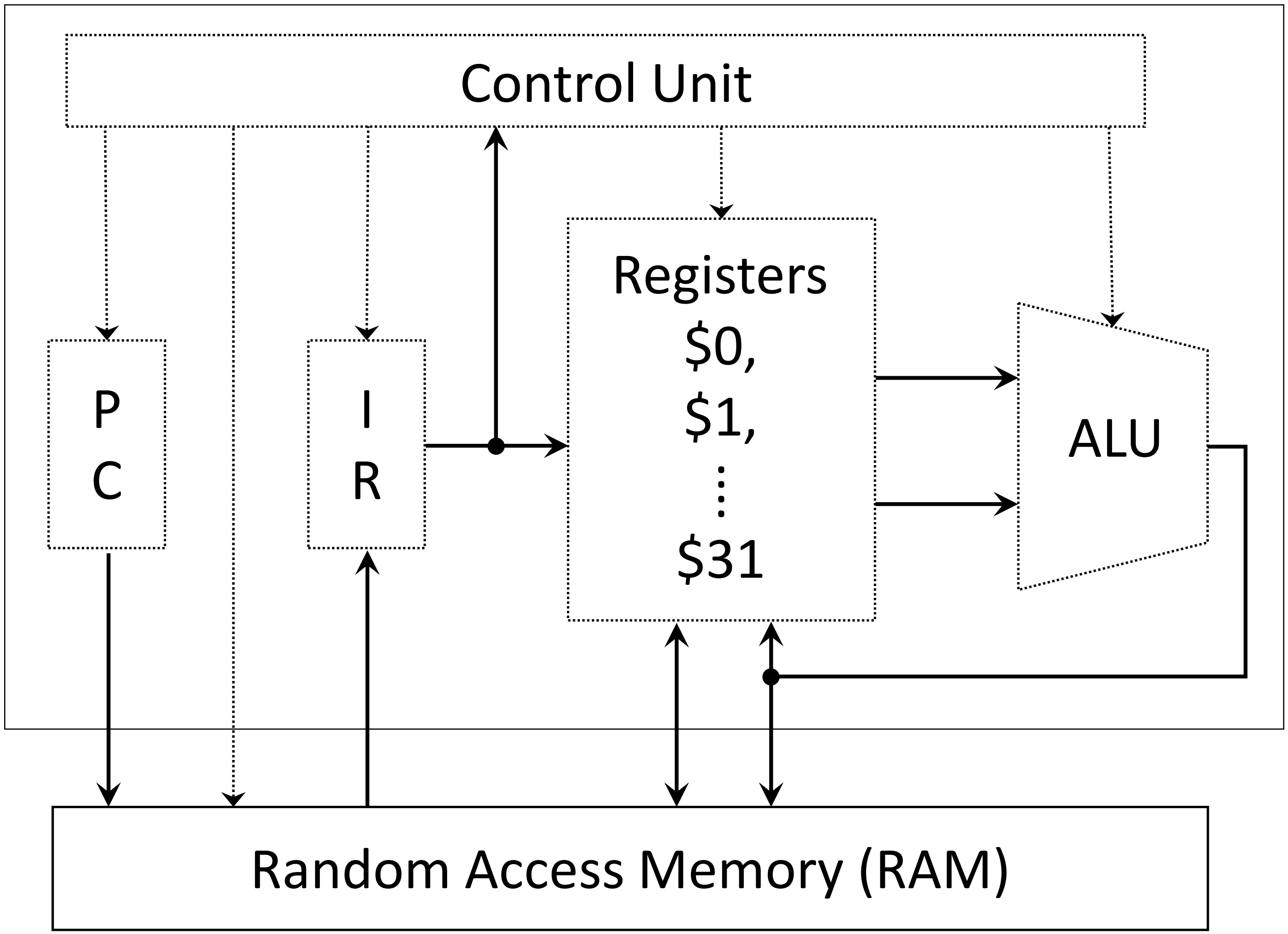
Constants

- to load the constant *i* into the register $d use lis and .word

lis $d
.word i

- lis means *load immediate and skip*
  - load the next value (in this case *i*) into $d and then skip over (i.e. don't try and execute) the next word
  - i.e. interpret *i* as data rather than as an instruction
- .word means store the value *i* right after the lis $d instruction
- It is called an *assembler directive* which is an instruction for the assembler (as compared to a MIPS instruction, such as jr $31, which gets translated into machine code).

Simplified View of a Processor and RAM



Simplified View of a Computer

Random Access Memory (RAM)

- *stores data (while the power is on)*
- also called primary storage or main memory
- the processor can *directly access* literally billions of memory locations with instructions like load word (lw) and store word (sw)

Processor

- *manipulates data*
- consists of two main parts
  1. *control unit:* controls the flow of data throughout the processor
  2. *data path:* stores, manipulates (or processes) the data

Simplified View of a Processor

Data Path

Major components include

- *Program Counter (PC):* holds the address of the current (or next) instruction
- *Instruction Register (IR):* holds the instruction that is being (or is about to be) executed
- *Arithmetic Logic Unit (ALU)*: performs arithmetic and logic operations (add, sub, mult, div, and, or, not)
- *general purpose registers:* a small amount of temporary (and very fast) storage within the data path

Simplified View of a Computer

Missing from diagram …

Secondary Storage

- *stores data (even when power is off)*
- typically a hard disk drive (HDD), a solid state drive (SSD), or some combination of both
- not considered at this point

Input / Output Devices

- varies, but typically includes devices such as a keyboard, mouse, display, speakers, USB ports
- not considered at this point

Conditional Execution

C++ vs. MIPS

- In general, programming languages we need the ability to alter the path the computation takes depending on input or on intermediate results
- in *C++* we have control structures like…
  - if … else
  - while loops
  - for loops
- in *MIPS* we have
  - branch if equal (beq)
  - branch if not equal (bne)
  - set if less than, for signed integers (slt)
  - set if less than, for unsigned integers (sltu)

Conditional Execution

Branching

beq $s, $t, i

- branch if equal
- compare the contents of registers *$s* and *$t*
- *if equal*, skip *i* instructions
- *i* can be positive (to go forward) or negative (to go backwards)

bne $s, $t, i

- branch if not equal
- compare the contents of registers *$s* and *$t*
- *if not equal*, skip *i* instructions
- *i* can be positive or negative

Simplified View of a Computer

Fetch-Execute Cycle

- The following code is stored in RAM starting at location 0x1000 and the PC=0x1000

RAM Address	RAM Contents	Disassembled
0x1000	0x00a71820	add $3, $5, $7
0x1004	0x01234822	sub $9, $9, $3
0x1008	…	

- *Fetch:* The first instruction would be fetched from RAM location 0x1000 and stored in the Instruction Register (IR).
- Execute: The instruction would be decoded and add $3, $5, $7 would be executed, i.e. the contents of $5 and $7 would flow to the ALU where they would be added and the result stored in $3. Simultaneously the PC is incremented by 4, i.e. PC=0x1004.

Simplified View of a Computer

Fetch-Execute Cycle

- Now PC=1004

RAM Address	RAM Contents	Disassembled
0x1000	0x00a71820	add $3, $5, $7
0x1004	0x01234822	sub $9, $9, $3
0x1008	…	

- *Fetch:* The next instruction would be fetched from RAM location 0x1004 and stored in the Instruction Register (IR).
- *Execute:* The instruction would be decoded and sub $9, $9, $3 would be executed, i.e. the contents of $9 and $3 would flow to the ALU where they would be subtracted and the result stored in $9. The PC would be incremented by 4 to 0x1008.
- This process is called the *Fetch-Execute Cycle*.

Conditional Branches *beq* and *bne*

The Program Counter (PC)

- note: the PC stores an address, i.e. the memory location of the instruction you are currently (or about to) execute
- i.e. it keeps track of where you are in the program
- incrementing the PC happens automatically after each instruction is loaded into the Instruction Register (IR)
- for MIPS, each instruction is 4 bytes long, so calculating the address of the next instruction (generally) means incrementing the PC by 4.
- *key point:* the value of the PC determines which instruction will be fetched and executed next so …

Conditional Branches *beq* and *bne*

The Program Counter (PC)

- to *skip over* some code (say skipping over one of the branches in an *if … else* statement) *add a multiple of 4* to the PC
- to *go backward* in the code (say to go back to the beginning of a *while* loop) *subtract off some multiple of 4* from the PC
- to start executing a specific subroutine, set the PC to the address where that subroutine starts
- *key point:* changing the value of the PC by a multiple of 4 changes which instruction will be executed next

Conditional Branches *beq* and *bne*

Calculating how far to branch

- reference sheet definition
  bne $s, $t, i
  if ($s != $t) PC += i × 4
- i.e. if the contents of $s is not equal to the contents of $t then increment the program counter by 4*i*
- since the size of each instruction is 4 bytes, therefore PC += i×4 skips over *i* instructions
- *key point:* this change is in addition to the default incrementing of the PC by 4 that happens each time an instruction gets executed
- this instruction *branches to* L_b+4+4*i*, where L_b is the location of the bne instruction
- *representation: i* is represented in 16-bit two's complement

Conditional Branches *beq* and *bne*

Calculating how far to branch

Addr	*Instruction*		
0x0ff8	sub \$4, \$4, \$1	⟵	to go here *i* = -3
0x0ffc	sub \$4, \$4, \$2	⟵	to go here *i* = -2
0x1000	beq \$4, \$5, i	⟵	*i* = -1 causes an infinite loop
0x1004	add \$4, \$4, \$3	⟵	happens anyway
0x1008	add \$4, \$4, \$4	⟵	to go here *i* = 1
0x100c	add \$4, \$4, \$5	⟵	to go here *i* = 2
0x1010	add \$4, \$4, \$6	⟵	to go here *i* = 3

E.g. for beq \$4, \$5, 3 (i.e. $i = 3$) PC = 0x1000 + 4 + (4×3) = 0x1010.
Recall that 16 in decimal is 0x10 (in hexadecimal).

Conditional Setting

Set if Less Than (slt)

- Useful if you don't want to test for equality but want to *test if the contents of one register is less than another*
- here *set* means make equal to 1 (or *True*)
- side note: *reset* means make equal to 0 (or False)
- details

  slt $d, $s, $t

  compare register $s and $t

  if $s < $t then set $d (i.e. $d = 1)

  if $s ≥ $t then reset $d (i.e. $d = 0)
- often it is used before beq and bne

Conditional Setting

Set if Less Than (slt)

- by reversing the order of the registers \$s and \$t in the slt instruction, i.e.

  slt \$d, \$s, \$t vs. slt \$d, \$t, \$s

  and combining with either bne or beq we get 4 combinations

slt \$d, \$s, \$t bne \$d, \$0, i	slt \$d, \$s, \$t beq \$d, \$0, i
slt \$d, \$t, \$s bne \$d, \$0, i	slt \$d, \$t, \$s beq \$d, \$0, i

- with these 4 combinations you can branch when:
  \$s < \$t, \$s ≤ \$t, \$s > \$t, or \$s ≥ \$t

Conditional Setting

Set if Less Than Unsigned (sltu)

- *many instructions which have integers as arguments come in two varieties: signed* and *unsigned*
- unsigned in another way of saying “natural numbers” where here natural numbers include 0
  - typically used for addresses
- signed is another way of saying “integers”
  - negative integers are represented using two’s complement
- with 32-bit architecture
  - unsigned ints have a range from 0 to $(2^{32} - 1)$
  - signed ints have a range -2^{31} to $(2^{31} - 1)$

Memory Model

Memory Access

- the maximum possible size of memory: 2^{32} bytes = 4 GB
- think of it as one big array, *Mem*[ ]
- two different approaches to accessing memory
  - *byte addressing*:
    can access any of the 2^{32} bytes directly
  - *word aligned addressing*:
    - can only access any of the 2^{30} words directly
    - addresses must be divisible by 4,
    - in hexadecimal, valid addresses always end in 0, 4, 8 or c
    - 0, 4, 8, 0xc, 0x10, 0x14,0x18, 0x1c, ... are all valid addresses
    - 1, 2, 3, 5, 6, 7, 9, 0xa, 0xb, 0xd, ... are all invalid addresses
    - recall: for MIPS32 there are 4 bytes in a word
- *MIPS uses word aligned addressing*

Base Plus Offset Addressing Mode

Memory Access

The sum $s+i is the RAM address where the data comes from (source) or goes to (destination).

lw $t, i($s)

- *load word* from Mem[$s+i] into register $t
- the sum $s+i must be word-aligned (divisible by 4)

sw $t, i($s)

- *store word* from register $t into Mem[$s+i]
- the sum $s+i must be word-aligned (divisible by 4)

When specifying an address as a sum, e.g. $s+i, the register $s is called the *base register* and the parameter i is called the *offset.*

What is the purpose of the offset?

Base Plus Offset Addressing Mode

Accessing Elements of a Structure

- We have an offset *i* because often many related items are stored in sequence in memory.
- *The offset allows access to each of the items* in relation to a single base address.
- One use of the addressing mode is for accessing local variables and arguments in a function call.
- e.g. for the following function

```
convert_date (int month, int day){
   int i = 0;
   …
}
```

Base Plus Offset Addressing Mode

Accessing Elements of a Structure

```
convert_date (int month, int day){
   int i = 0;
   …
```

- Assume the arguments and local variables are stored starting at the address stored in $29. To access the...
  - month: `lw $t, 0($29)`
  - day: `lw $t, 4($29)`
  - *i*: `lw $t, 8($29)`
- What you are really saying is to access the ...
  - day, add 4 to the base address stored in register $29
  - *i*, add 8 to the base address stored in register $29
- More on this topic later when we discuss *stack frames.*

More Arithmetic Operations in MIPS

Multiplication and Division

- these operations use two special registers *hi*, *lo*

mult $s, $t

- multiply the contents of registers $s and $t
- result may be too big to fit in one register
- place the most significant 32 bits in *hi*
- place the least significant 32 bits in *lo*
- for the purposes of this course: assume the answer is always 32 bits or less, so you only need to consider the *lo* register

div $s, $t

- divide the contents of register $s by the contents of register $t and place the quotient in *lo*, and the remainder in *hi*

More Arithmetic Operations in MIPS

Multiplication and Division

- *recall: there are two versions of integers*
  - *unsigned:* positive integers and 0 only
  - *signed:* positive and negative integers, i.e. two's complement

multu $s, $t

- same as mult but treat the numbers in $s and $t as unsigned integers

divu $s, $t

- same as div but treat the numbers in $s and $t as unsigned integers

More Arithmetic Operations in MIPS

Accessing Results

- you gain access to the values stored in the special registers *hi* and *lo* using the mfhi and mflo commands

mfhi $d

- copy contents of the hi register to $d

mflo $d

- copy contents of the lo register to $d

Comments

- a comment begins with a semicolon and continues to the end of that line

; this is a comment

Conditional Branches

Example: If Statement

- *Task*: Compute the absolute value of $1, store the result in $1, then return.
- Temp values: $2 will store true if $1 is negative.

C++

```
if (r1 < 0) {r1 = 0 - r1; } return;
```

MIPS assembly language

Instructions/Data *Comments*

```
slt $2,$1,$0          ; is $1 < 0 ?
beq $2,$0,1           ; if false, skip 1 line
sub $1,$0,$1          ; else negate $1
jr $31;               ; return
```

Conditional Branches

In MIPS Assembly Language

Addr	Contents	Comments
`0x0`	`slt $2,$1,$0`	`; is $1 < 0 ?`
`0x4`	`beq $2,$0,1`	`; if false, go to end`
`0x8`	`sub $1,$0,$1`	`; else negate $1`
`0xc`	`jr $31;`	`; return`

- **`beq $2,$0,1`** means **if** (\$2 == 0) **then** skip forward 1 instruction
- the actual calculation is as follows PC = $L_b + 4 + 4i$
- PC = 0x4 + 4 + 4×1 = 0xc (or in decimal: 4 + 4 + 4 = 12)

0x4	L_b, i.e. the location of the **`beq`** instruction
4	amount the PC is incremented automatically
4×1	the amount to adjust the PC by in bytes, i.e. how far to branch because of the **`beq`** instruction

Branch Labels

Calculating Offsets

- labels make assembly language easier: leave the computation of branch offsets to the assembler
- *create a label*
  - a single word followed by colon
  - first character must be a letter
  - rest of the label can be a combination of letters and numbers
- *assembler program computes the actual offset*
- if you add more statements inside the loop, the assembler automatically recalculates the offset
- for assembly languages with variable length instructions, this is even more helpful

Branch Labels

Without Labels

Contents	Comments
`slt $2,$1,$0`	`; is $1 < 0 ?`
`beq $2,$0,1`	`; if false, go to end`
`sub $1,$0,$1`	`; else negate $1`
`jr $31;`	`; return`

Branch Labels

With Labels

Labels	Contents	Comments
	`slt $2,$1,$0`	`; is $1 < 0 ?`
	`beq $2,$0,end`	`; if false, go to end`
	`sub $1,$0,$1`	`; else negate $1`
`end:`	`jr $31;`	`; return`

`end:` is the label definition

- it is placed in first column and it always ends with a colon
- it refers to a specific location
- when it is used elsewhere (i.e. the **`beq`** instruction on the 2nd line) it refers to the location where it is defined (i.e. the last line)
- it is defined once, but may be used many times

Label Naming

Labels and Scope

- make *labels* readable, descriptive and intuitive, just like variable and function names
- *label* definitions must be unique within scope
- assume they only need to be unique within a single source file for now (i.e. you can use same *label* in different files)
- later on you will learn how to deal with *labels* that must be understood by other files (i.e. externally/globally)
- *labels* may be generated manually (i.e. when a human creates an assembly language program) vs. automatically (when a compiler generates them)

Conditional Branches

Example: Implementing if ... else ...

In C++

```
if (r1 == 0)
    r2 = r2 + r3; // thenPart
else
    r2 = r2 + r4; // elsePart
```

In MIPS

```
           beq $1, $0, thenPart  ;if r1==0
           add $2, $2, $4        ;else part
           beq $0, $0, cont      ;always branch
thenPart:  add $2, $2, $3        ;then part
cont:      …                     ;continue with
           …                     ;rest of program
```

Assembly File

What does an Assembly File Contain?

Typically organized as three columns. Each line can contain

1. Label declarations (0 or more)
2. MIPS Instruction xor Data definition (0 or 1)
3. Comments (0 or 1) – start with a semicolon

I.e. there can be

- blank lines,
- lines with only a label on it,
- lines with only an instruction on it
- lines with only a comment on it, etc

There is no choice in the order: labels first, instruction xor data definition next, comment last.

Assembly File

Format

Numbers can be: hexadecimal, positive or negative decimal

- *hexadecimal:* use 0x prefix, e.g. 0x20 (32 in decimal)
- *positive decimal:* don't use 0x prefix, e.g. 32
- *negative decimal:* don't use 0x prefix, but do use a negative sign e.g. -32

```
Labels      Instructions/Data      Comments
start:      lis $1
            .word 0x20             ; $1=32 in decimal
            lis $2
            .word 32               ; $2=32
            lis $3
            .word -32              ; $2=-32
end:        jr $31                 ; end program
```

Arrays

Indexing into an Array

- I'll call A[0] the 0^{th} element, A[1] the 1^{st} element etc.
- You have an array, A, where
  - the indices start at 0, i.e. A[0], A[1], A[2], …
  - the size of each element in the array is *4* bytes.
- If the address of A[0] is in register \$1, then
  - the address of A[1] is \$1+4,
  - the address of A[2] is \$1+8,

  ⋮

  - the address of A[*i*] is \$1+4*i*
- The address of the 0^{th} element is called the *base address.*
- *The address of the i^{th} element is*

  base address + (i × size of an element)

Arrays

Example: Accessing the element 5 of an array

```
;; Input:    $1 base address of array
;; Output:   $3 5th element of the array, i.e. A[5]
;; $4 the size of each element
;; $5 temp storage

        lis $5              ; index into array
        .word 5
        lis $4              ; size of each element
        .word 4             ;
        mult $5,$4          ; offset to 5th element
        mflo $5             ;
        add $5,$1,$5        ; address of 5th element
        lw $3,0($5)         ; $3 gets A[5]
        jr $31              ; return
```

Input and Output

Memory Mapped I/O

- For CS 241, input /output from devices (such as a keyboard or a screen) are treated as reading from and writing to memory.
- I.e. use the MIPS instructions **`lw`** and **`sw`**, with specific memory locations.
- The data will be encoded as a single ASCII value per word (with the most significant 3 bytes being 0).
- To *output a char to the stdout,* store the ASCII value of that character in memory location 0xFFFF000C.
- To *read a char from the stdin,* load the value stored at memory location 0xFFFF0004.

Input and Output

Memory Mapped I/O Example

```
;; Print "CS\n" on stdout
    lis $1                ; address of output buffer
   .word 0xFFFF000C
    lis $2
    .word 67              ; ASCII C
    sw $2,0($1)           ; write to stdout
    lis $2
    .word 83              ; ASCII S
    sw $2,0($1)           ; write to stdout
    lis $2
    .word 10              ; ASCII newline
    sw $2,0($1)           ; write to stdout
    jr $31                ; return
```

Control Structures

Example: Sum Integers in C

- Task: Sum the integers 1 to 13, store sum in r3, then return.

C++

```
int r1 = 1;           // constant 1
int r2 = 13;          // integers to be summed
int r3 = 0;           // answer

while (r2 != 0) {
   r3 = r3 + r2;      // r3 = 13 + 12 + 11 + …
   r2 = r2 - r1;      // r2 = 13, 12, 11, …
}
return;
```

Control Structures

Example: Sum Integers in MIPS Assembly Language

Labels *Instructions/Data* *Comments*

```
;;      $1 constant 1
;;      $2 integers to be summed
;;      $3 answer
        lis $1                 ; $1 = 1
        .word 1
        lis $2                 ; $2 = 13
        .word 13
        add $3,$0,$0           ; $3 = 0
loop:   add $3,$3,$2           ; $3 = $3 + $2
        sub $2,$2,$1           ; $2 = $2 - 1
        bne $2,$0,loop         ; loop until $2==0
        jr $31                 ; return
```

Subroutines

Key Challenges in Implementing Subroutines

In order to implement functions we need to answer four questions.

1. How do we ensure that data stored in registers (that we want to use again) is not overwritten by the subroutine we call?
2. How do we call and return from a subroutine?
3. How do we pass arguments to the subroutine?
4. How do we return values from a subroutine?

Subroutines

Subroutines vs. Functions

- *subroutines*: assembly language's version of functions
- programmers must do more work, essentially implement a function using: labels, PC, **`lw`**, **`sw`**
- *function name* ⇒ go to this label / memory location and start executing the instructions you find there
- *arguments and return values* ⇒ agree to place certain values in certain registers or memory locations

  \- *gone*: no concept of type checking
- *local scope, variables* ⇒ *gone*: can access any register and most memory locations (more on that later)

Subroutines

Storing Essential Data

- A subroutine can call another subroutine (or itself)
- What about registers that are in use?
- For example, say we have
  - important data stored in registers 1 to 4
  - want to call subroutine *sum* which uses registers 2 and 3 as "local variables" / temporary values
  - registers ≠ local variables, i.e. subroutine *sum* will overwrite these important values
- must save the *current execution context* (set of register values) before executing the body of *sum* and restore the context once *sum* has finished
- *Key Question:* save where?

The Call Stack

Solution: Use a stack

- *solution:* store data (which you will need later) on the *call stack* (a.k.a. the *run-time stack*)
- use part of main memory (i.e. RAM) as a stack
  - last-in first-out queue
- *convention:* stack grows downward in memory
  - i.e. from a high address down to a lower address
  - i.e. you would subtract from the current top of the stack to make room for new items
- *convention:* the address of the top of the stack (the top item on the stack) is stored in the stack pointer (SP) register
- convention: typically register $29 is the SP in MIPS
- *exception:* in our MIPS simulator we use $30

The Call Stack

Saving Context on the Stack

- *save* (a.k.a.) *push onto the stack*
- two step process
  1. store the register values on the stack
  2. decrement stack pointer (SP) to reflect the change

Restoring Context from the Stack

- *restore* (a.k.a.) *pop off the stack*
- two step process
  1. increment stack pointer (SP) to reflect the change
  2. load values back into the registers (in this case $2 and $3)
- For both: each item is 4 bytes in size

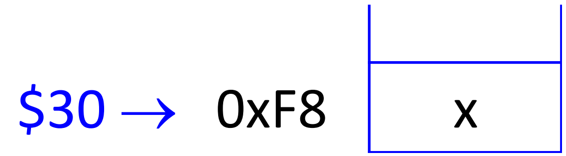
Example: store and then restore the values in $2 and $3 on the stack and the initial value of the SP ($30) is 0xF8…

The Call Stack

Stack

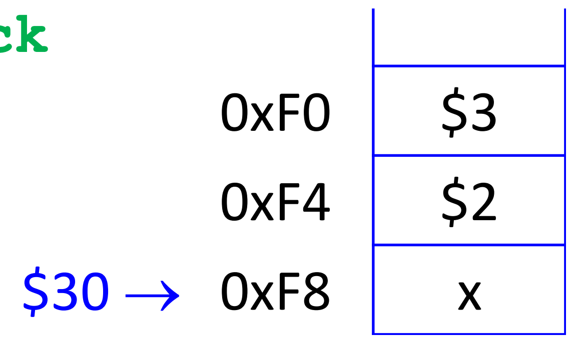
Saving $2 and $3 on the Stack

```
;; 0. Initially
```



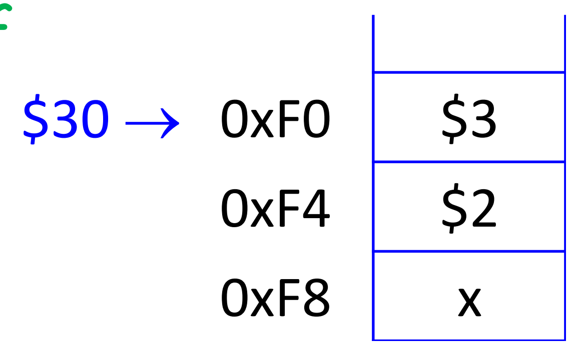
```
;; 1. Store $2 and $3 on the stack

sw $2,-4($30)
sw $3,-8($30)
```



```
;; 2. Decrement the stack pointer

lis $3
.word 8
sub $30,$30,$3
```



The Call Stack

Restoring $2 and $3 from the Stack

```
;; 0. Initially
```

$30 →	0xF0	$3
	0xF4	$2
	0xF8	x

```
;; 1. Increment the stack pointer
   lis $3
   .word 8
   add $30,$30,$3

;; 2. Copy values back into
;;    registers $2 and $3
   lw $3,-8($30)
   lw $2,-4($30)
```

	0xF0	$3
	0xF4	$2
$30 →	0xF8	x

Calling and Returning from a Subroutine

Calling a Subroutine: Attempt #1

- to call a subroutine *jump to the memory location where the routine is located* and starting executing the code there, e.g.

```
0x00      lis $5          ; store addr of
0x04      .word sum       ; label sum in $5
0x08      jr $5           ; jump to sum
0x0C      ...             ; return HERE
   ⋮
sum:
   ⋮
```

- *Problem:* how do we know where to return to when the subroutine `sum` is finished?

Subroutines

```
if {amount_requested > account_balance)
   printf("Request a lower amount")
else {
   printf("Collect money from dispenser")
   dispense(amount_requested)
}
```

Challenges of Using Subroutines

- call/return – how to redirect execution?
  - call is *static* ⇒ always go to same location
    e.g. the beginning of the **printf** function
  - return is *dynamic* ⇒ must track where to return to
    e.g. which line of C called the **printf** function
- complications: nested call/return, recursion

Subroutines

Two Instructions

`jalr $s`

- meaning: *jump and link register*
- copy the address of next instruction (PC) to $31
- set PC to the address stored in `$s`
- start executing code at this new location
- typically used to *call a subroutine*

`jr $s`

- meaning: *jump (to the address in) register $s*
- set PC to $s
- start executing code at this new location
- convention: register $31 holds return address
- typically used to *return from a subroutine call*

Calling and Returning from a Subroutine

Calling a Subroutine: Attempt #2

- *need to store current location of the PC using* **`jalr`** which stores the address of the next statement (0x0C) in $31

```
0x00        lis $5          ; store addr of
0x04        .word sum       ; label sum in $5
0x08        jalr $5         ; jump to sum
0x0C        ...             ; return HERE
   ⋮
sum:
   ⋮
```

- $31 now contains the address 0x0C.
- Problem: what if $31 previously had a valid return address
  - e.g. this subroutine was called by another or the subroutine is recursive

Calling and Returning from a Subroutine

Calling a Subroutine

Solution: save the contents of $31 on the stack

Save $31 on the stack before calling the subroutine **`sum`**

1. push $31 onto the stack and update the stack pointer
   note: once $31 is saved on the stack the register can be used as a temp register to help update the stack pointer.
2. jump to subroutine **`sum`** using **`jalr`**

⋮

Restore $31 after returning from the subroutine **`sum`**

1. update stack pointer
2. pop value from stack and store in $31

Calling and Returning from a Subroutine

Calling a Subroutine

```
                                ; calling sum
main:   sw $31,-4($30)          ; 1. push $31 onto
        lis $31                 ;    the stack and
        .word 4                 ;    update SP($30)
        sub $30,$30,$31         ;
        lis $5                  ; 2. load addr of
        .word sum               ;    subroutine sum
        jalr $5                 ;    and jump to it

                                ; returning from sum
        lis $31                 ; 1. update SP($30)
        .word 4                 ;    by adding 4
        add $30,$30,$31         ;
        lw $31,-4($30)          ; 2. pop top of stack
        jr $31                  ;    into $31 & return
```

Subroutines: arguments and results

Passing Arguments and Returning Results

- *Problem: need to pass arguments and return result(s)*
- can use registers, stack, or both
- need to agree between caller and callee
  - for now (A2) we'll use registers
  - later on (A9-A10) when we must handle an arbitrary number of arguments, we'll use the call stack (a.k.a. run-time stack)
- there are other standards (e.g. CS 350)
- your use of registers must be documented
- Example:
  - Create a function that will sum the first *n* natural numbers (i.e. answer = 1 + 2 + … + *n*).
  - The input, *n*, is in \$2; return the answer in \$3.

The Subroutine

Passing Arguments and Returning Results

1. Document your use of registers in function header

```
; sum - adds the integers 1..N
; Registers:
; $1 – i: which will range from 1 to N
; $2 – N: the argument
; $3 – answer: the return value
```

Subroutines

Passing Arguments and Returning Results

2. *Save the current contents of any registers you are changing on the stack* (except $3 where you will place the result). In this case save the contents of $1 and $2.

```
sum:
      sw $1,-4($30)      ; push $1 onto stack
      sw $2,-8($30)      ; push $2 onto stack
      lis $1             ; update SP, reuse $1
      .word 8
      sub $30,$30,$1
```

- In the last 3 lines, the value stored in $1 has just been saved on the stack so $1 is now available to store the temporary value 8.

Subroutines

Passing Arguments and Returning Results

3. *Initialize the answer ($3), create the constant 1 (in $1), then calculate the sum by repeatedly decrementing i ($2)*

```
      add $3,$0,$0      ; initialize answer = 0
      lis $1            ; initialize i = 1
      .word 1
  top:                  ; while loop
      add $3,$3,$2      ; answer = answer + i
      sub $2,$2,$1      ; i = i - 1;
      bne $2,$0,top     ; loop while i ≠ 0
```

Subroutines

Passing Arguments and Returning Results

4. *Restore the previous contents of any registers you used from the stack and then return*

```
lis $1              ; update stack pointer $30
.word 8             ; reusing $1
add $30,$30,$1      ;
lw $2,-8($30)       ; restore register $2
lw $1,-4($30)       ; restore register $1
jr $31              ; return
```

Recursive Subroutines

Creating a Recursive Subroutine

- Same as calling a subroutine except now you are calling yourself.
- Two cases:
  1. *if base case:* detect base case and return correct result.
  2. *else recursive case:*
     Do not look ahead.
     Combine current value with the result from the recursive call.
- *Hint:* code routine up in your favourite high level language (or in pseudocode) and then translate it directly into MIPS Assembly Language.
- See Example 7 in the resource section of the course web page for an example of a recursive version of the sum 1 to *n* problem.

Examples Provided on CS241 Homepage

See "Material for Assignment 2 (and beyond) on homepage

- Example 0: add $5 and $7, store result in $3
- Example 1: add 42 and 52, storing result in $3
- Example 2: find the absolute value of $1
- Example 3: read element 5 of an array into $3
- Example 4: calculating 13+12+...+2+1
- Example 5: outputting characters
- Example 6: calling a subroutine
  a) calling code
  b) subroutine code
- Example 7: calling a recursive subroutine
  a) calling code
  b) recursive subroutine

Covered
in Lecture

Low Level Errors

Common Errors

- illegal instruction
  - plus $1, $2, $3 ; no such opcode
- assignment to read-only register
  - add $0, $1, $2 ; $0 is read only
- division by 0
  - div $1, $0
- alignment violation
  - lw $1, 3($0) ; address must be a multiple of 4
- bad opcode: trying to interpret data as an instruction
- and possibly others...
- usually result in exception and termination

Low Level Errors

Debugging Errors

- debugging assembly language programs is difficult
  - *terminate the program (jr $31) at various places and study the values in the registers,* especially the PC, $30 (SP), $31 (RA)
  - or if you are using functions (where $31 gets overwritten), copy $31 into an unused register (say $26) and do **`jr $26`** to terminate the program
  - could also use output to screen
- general techniques
  - analyze log output
  - controlled step-by-step execution

    ⇒ need some kind of virtual environment
  - verify assertions

Other Instructions

For the sake of completeness I'll mention that there are other instructions

- *immediate*
  - replace register operand with 16-bit constant
- *logical*
  - AND, OR, XOR, NOT, etc.
- *floats*
  - floating point arithmetic
- *bit operations*
  - shift left and shift right
- *jump*
  - long-range unconditional branch

Topic 3 – Implementing an Assembler

Key Ideas

- the purpose of an assembler
- binary files vs. ASCII representations of binary files
- An assembler’s two passes: 1. Analysis and 2. Synthesis
- syntactic and semantic errors
- scanning, tokens and intermediate representation
- the symbol table
- calculating addresses of instructions and dealing with labels
- bitwise operations: and, or shift left, shift right

The Assembler

Overview

- *An assembler converts an assembly language program* (i.e. what you created in Assignment 2) *into its corresponding machine code* (i.e. what you created Assignment 1).
- In Assignment 1: *you were* the assembler.
- In Assignment 2: *you used* the assembler cs241.binasm.
- In Assignments 3 and 4: *you will create* (most of) a small assembler.

`jr $31`

Assembler ↓

0x03e00008

or

0000 0011 1110 0000
0000 0000 0000 1000

The Assembler

Overview

- The input to an assembler is a text file containing a sequence of assembly language instructions, e.g. `jr $31`
- The *input is an ASCII text file,* i.e. something that can be edited with a text editor.
- The *output is a binary file* which encodes MIPS instructions, i.e. something which typically cannot be edited with a text editor.
- A file containing *n* MIPS instructions would be 4*n* bytes long.
- You can view with xxd.
- *The binary file is different from an ASCII text file* containing a sequence of 1's and 0's that represent the `jr $31` instruction, which would be 32 bytes long (since each 0 or 1 is an ASCII character).

The Assembler: the Steps

Steps in the Process

- We take two passes through the code: *Analysis* and *Synthesis*
- *Pass 1: Analysis*

  Read in the text file containing MIPS assembly language instructions and

  - Scan each line, breaking it into components
  - Create an intermediate representation
  - Parse components, checking for errors.
- *Pass 2: Synthesis*
  - (Possibly check for more errors)
  - Construct the equivalent binary MIPS machine code.
  - Output the binary MIPS machine code.

The Assembler: Analysis

Pass 1 Analysis

- *The input* is an ASCII text file containing a sequence of assembly language instructions, e.g.

```
total: beq $1, $2, end        ; $1 total cost
```

- *Purpose: to recognize components of the instructions*
- How: break down each line of assembly language into *tokens.*
- In English grammar you can break down a sentence into words and classify them as verb, noun, adjective, etc. to describe the role each word performs.
- For assemble language, you break up an assembly language instruction into components and classifying these components.

The Assembler: Analysis

Pass 1 Analysis and Tokens

For MIPS assembly language there are 11 kinds of tokens

- REGISTER: the 32 registers, i.e. $0, $1, $2, … $31
- INT: positive and negative integers, e.g. 1, 41, -312, 4000
- HEXINT: integers in hexadecimal format, e.g. 0x1, 0x20, 0x345
- LABEL: declaration of a label, e.g. total:, end:, main:, …
- ID: an opcode (e.g. **`add`**, **`sub`**, **`jr`**, …) or the use of a label without a colon (e.g. **`end`** in the **`beq`** instruction above)
- DOTWORD: e.g. the **`.word`** directive
- LPAREN , RPAREN, COMMA, WHITESPACE
- ERR (i.e. bad or invalid token)

The input is broken down into a series of tokens so that each component is classified as one of these 11 kinds of tokens.

The Assembler: Analysis

Pass 1 Analysis and Tokens

- We will provide code (in C++ and Racket) called a *scanner* that reads in the assembly language file and breaks down each line into *a series of tokens* for you, e.g.

```
main: lis $1
      .word 0xa
```

Token: LABEL {main:}
Token: ID {lis}
Token: REGISTER {$1} 1
Token: DOTWORD {.word}
Token: HEXINT {0xa} 10

This means, of course, you can only do the rest of the assignments in one of these languages.

The Assembler: Analysis

Pass 1 Analysis and Tokens

- For the assembly language instruction

  ```
  end:  jr $31
  ```

  the tokens are

  Token: LABEL {end:}
  Token: ID {jr}
  Token: REGISTER {$31} 31

- The part *in all caps* (e.g. LABEL, ID, REGISTER) is called the *kind* (of token).
- The part *in braces* (e.g. end:, jr, $31) is the string representation of the token that was found in the source file, called a *lexeme*.
- *For some* tokens, such as REGISTER, INT and HEXINT, our scanner also provides the integer corresponding to the lexeme.

The Assembler: Analysis

Another Example

- For the assembly language instruction

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

  the tokens are

  Token: ID {lw}
  Token: REGISTER {$3} 3
  Token: COMMA {,}
  Token: INT {-4} -4
  Token: LPAREN {(}
  Token: REGISTER {$30} 30
  Token: RPAREN {)}

- Note: *each token always has a kind and a lexeme* but not all tokens have a corresponding integer.

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- This pass also checks for *syntax errors*, i.e. *improper form or structure*.
- e.g. in English the sentence "Look at the barking brown big two dogs." does not have proper syntax.
- e.g. in MIPS assembly language
  - error: lw $1
  - error: lw $3 0($4)
  - error: lw $3, 0($4
  - error: lw lw $3, 0($4)
  - error: lw $3, $4, $5
  - error: lw $3, 9999999999($4)

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- This pass also checks for *semantic errors,* i.e. *what does it mean?*
- The sentence "Colorless green ideas sleep furiously." (N. Chomsky) is grammatically correct but meaningless.
- In MIPS assembly language a semantic error would be defining the same label twice. If that label is used in a `beq` instruction you would not know which of the two locations to branch to.
- I.e. semantic analysis asks: What does this label mean here?
- The version of MIPS that we use is documented here: https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsasm.html
- In future assignments you learn how to formally describe a computer language.

The Assembler: Analysis

Pass 1 Analysis: Error Checking

- *Big hint:* just recognize the proper form and call everything else an error.
- There is no need to identify the type of error, but you may find it helpful to do so.

The output is

1. an *intermediate representation*
   which is a form of the input that is easy to work with
   e.g. a list (or vector) of lines where each line is a list (or vector) of tokens
2. the *Symbol Table*
   which maps labels (such as `total`) to addresses (such as 0x0000 001c)

The Symbol Table

Pass 1 Analysis: Input

```
main:   lis $2
        .word main
        add $3,$0,$0
top:    add $3,$3,$2
        lis $1
        .word 1
        sub $2,$2,$1
        bne $2,$0,next
        bne $0,$0,top
next:   mult $3,$4
        mflo $4
        slt $6,$5,$4
```

Output: Symbol Table

- maps labels to addresses e.g.

Label	Address
main	0x0000
top	0x000C
next	0x0024

Intermediate Representation

Pass 1 Analysis: Intermediate Representation

At the very least, intermediate representation

- removes comments
- creates tokens
- keeps your program as ASCII / Unicode characters

More elaborate versions of intermediate representation

- take a bigger step towards representing elements of the program as machine code rather than ASCII

CS241's version of the intermediate representation depends on the language, it is either

- C++: a vector of vectors of tokens or
- Racket: a list of lists of tokens

The Assembler: Synthesis

Pass 2 Synthesis

- *The input* is the intermediate representation and the symbol table (i.e. the output from the analysis pass).
- *The purpose* is to translate
  - the labels into addresses.
  - the intermediate representation into machine code.
- *The output* is machine code for a particular processor.

The Assembler

Why Two Passes?

- *A label can be used before it is defined* (especially in the equivalent of an if … else statement)

```
        bne $1, $0, next          ; if r1==0
        add $2, $2, $4            ; then r2 += r4
next:   sw $2 0($3)
```

- Two labels can refer to each other

```
prev:   bne $1, $0, next
          ⋮
next:   beq $1, $0, prev
```

- So in the first pass, you may encounter a label before it is defined.

Assembler Implementation

General Strategy

- *test every detail of the MIPS Assembly Language Spec*
  - e.g. you could print it out and check off items as they are implemented
- must know the language better than a programmer
- error reporting can be unsophisticated
  - report ERROR in cerr/stderr, meaningful details are optional
- don't try to think about all possible errors just *be very specific about what you are expecting*, e.g.
  - the opcode **`jr`** should be followed by: a register,
  - the opcode **`mult`** should be followed by: a register, a comma, a register

Assembler Implementation

Recall: Format of Input

- Each line of assembly language is of the format

```
label(s)     instruction      comment
main:        lis $1           ; $1 = 1
             .word 0x1
```

- Each of these three components are optional
  - A line may have 0 (i.e. blank), 1, 2 or all 3 of them.
- They *must occur in this order*: label(s), instruction, comment if they are present.
- There can be many labels on a line but at most 1 instruction and 1 comment per line.
- Lines without an *instruction* are called *null lines* and do not specify an instruction word.

Assembler Implementation

Calculating the Locations for Instructions

- ignore all labels (comments and blank lines will be removed)
- count the number of preceding instructions to calculate the address an instruction
- each instruction is exactly 4 bytes long

Location	Input
0x00	; my prog
0x00	
0x00	start:
0x00	add $1, $2, $3
0x04	middle: centre: ; important
0x04	lw $2, 0($1)
0x08	add $2, $2, $4
0x0C	end: jr $31

Implementing Pass 1

Pseudocode for Pass 1: Analysis

```
PC= 0                                          // program counter
for each line of input {
    scan line                                  // create tokens
    create intermediate representation

    for each LABEL token {                     // process labels
        if already in symbol table
            report ERROR and exit              // DO NOT continue
        add (label, PC) pair to symbol table
    }

    if token is an OPCODE {                    // process instruction
        if remaining tokens are not what is expected
            report ERROR and exit              // DO NOT continue
        PC += 4
    }
}
```

Implementing Pass 1

Pseudocode for Pass 1: Analysis

```
PC= 0                                      // program counter
for each line of input {
    scan line                              // ⇐ we'll help here
    create intermediate representation     // ⇐ and here
```

- Use the starter code provided for the various languages: C++14 or Racket.
- In future assignments, you will learn how to identify tokens yourself.
- Typically you use another program (such as lex or flex) to help you with this task.

Implementing a Symbol Table

Input

```
a:      lis $1
        .word 0x1
        beq $0,$0,b
a:      add $1,$0,$0
        bne $2,$0,b
        ...
        beq $2,$0,a
        ...
b:      sub $2,$2,$1
```

ERROR: label **a** is defined multiple times.

Resolving Labels

- Problem: which location does the label **a** refer to?
- Labels can
  - be *defined only once*
  - but *used many times* as a operand
- Your assembler needs the ability to *add* and *find* (string, number) pairs in a data structure called the symbol table

Implementing a Symbol Table

In C++

- *could use a map*

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

map<string, int> st;

st["foo"] = 42;
```

Implementing a Symbol Table

In C++

- an *incorrect* way of accessing elements:

```
x = st["foo"]; // x gets 42
y = st["bar"]; // y gets 0, and (bar, 0)
               // gets added to st.
```

- a *correct* way of accessing elements:

```
if (st.find("biff") == st.end()) {
    ... not found ...
}
```

Assembler Implementation

Pseudocode for Pass 2: Synthesis

for each OPCODE in the *intermediate representation*
  translate to MIPS machine code
    look up any labels in the *symbol table*
  output the instruction as 4 binary bytes

Caution
For each instruction, the output is
- 32 bits (i.e. 4 bytes)
- *not 32 ASCII characters* (i.e. 32 bytes)
- most methods of outputting data such as “printf” or “cout <<” will automatically convert the data to ASCII representation
- this is what you did for A2P6 when you took a number as input and printed out a series of ASCII characters

Assembler Implementation

Translating Instructions

- *Use the MIPS reference sheet as your guide*
- e.g. for the command **`lis $2`** the format is

  0000 0000 0000 0000 dddd d000 0001 0100

  where ddddd is 00010 (binary for 2)
- this step is very similar to Assignment 1
- but you must *encode this data in four bytes* which involves dealing with, and shifting around, bits
- we'll look at **`bne $2,$0,top`** in detail …

Sample Input

PC Labels Instructions

```
00  main:   lis $2
04          .word 0xd
08          add $3,$0,$0
0C  top:    add $3,$3,$2
10          lis $1
14          .word 1
18          sub $2,$2,$1
1C          bne $2,$0,top   ←
20          jr $31
24  beyond:
```

Symbol Table

Label	Address
main	0x00
top	0x0C
beyond	0x24

Implementing an Assembler

Building up a Instruction

- for `bne $2,$0,top`
- look up `top` in the symbol table, its is address 0x0C
- but *we need a number of instructions to jump* back or forward not an address
- $(L_l - L_b - 4) / 4 = (0x0C - 0x1C - 4) / 4 = (12 - 28 - 4) / 4 = -5$
  where L_l is the location of the label to branch to
  where L_b is the location of the branch instruction
- so now the instruction becomes `bne $2,$0,-5`
- the format the `bne` instructions is

  `0001 01ss ssst tttt iiii iiii iiii iiii`

  so we must build up each component of this instruction…

Assembler Implementation

Bitwise Operations

- typically the smallest unit of data that can be assigned directly is a single byte (i.e. a char)
- to manipulate anything smaller, we must use *bitwise operations* (operations that act on a single bit).
- *bitwise and*, a & b, performs the *and* operation on *individual bits*, e.g. for 8-bit values, it would be …

a =	0	1	0	0	1	0	1	1
b =	1	1	0	0	0	1	0	1
a & b =	0	1	0	0	0	0	0	1

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Assembler Implementation

Bitwise Operations

- Bitwise *and* is used to *mask off* or turn off bits (i.e. change a portion of the bits to 0's), e.g. for an 8-bit value

a	= 1 1 0 1 0 1 0 1
bit-mask (0x0F)	= 0 0 0 0 1 1 1 1
a & bit-mask	= 0 0 0 0 0 1 0 1

- Here the most significant nibble (half byte) of *a* has been masked off (reset to 0).
- If *a* is a 32-bit number, 0xffff would mask off the most significant 2 bytes, e.g.

a	= 1101 0011 1010 1000 1101 1010 1101 1111
0xffff	= 0000 0000 0000 0000 1111 1111 1111 1111
a & 0xffff	= 0000 0000 0000 0000 1101 1010 1101 1111

Assembler Implementation

Bitwise Operations

- *bitwise or*, a | b, performs the *or* operation on *individual bits*, e.g. for 8-bit values it would be

a =	0	1	0	0	1	0	1	1
b =	1	1	0	0	0	1	0	1
a | *b* =	1	1	0	0	1	1	1	1

a	*b*	*a* | *b*
0	0	0
0	1	1
1	0	1
1	1	1

- the *shift left operator*, <<, shifts bits left, introducing 0's on the right hand side, e.g. for 8-bit values it would be …

a =	0	1	1	0	1	0	0	1
a << 1 =	1	1	0	1	0	0	1	0
a << 2 =	1	0	1	0	0	1	0	0
a << 3 =	0	1	0	0	1	0	0	0
a << 4 =	1	0	0	1	0	0	0	0
a << 5 =	0	0	1	0	0	0	0	0
a << 6 =	0	1	0	0	0	0	0	0
a << 7 =	1	0	0	0	0	0	0	0

Assembler Implementation

Translating Instructions

- recall that the format of the `bne $2,$3,-5` instructions is

```
0001 01ss ssst tttt iiii iiii iiii iiii
↑       ↑      ↑      ↑                  ↑
32      26     21     16                 1
```

where the opcode 000101_2 = 5 shifted left 26 bits

5	0000 0000 0000 0000 0000 0000 0000 0101
5 << 26	0001 0100 0000 0000 0000 0000 0000 0000

s is 2 = 00010_2 shifted 21 bits left

2	0000 0000 0000 0000 0000 0000 0000 0010
2 << 21	0000 0000 0100 0000 0000 0000 0000 0000

t is 3 = 00011_2 shifted 16 bits left

3	0000 0000 0000 0000 0000 0000 0000 0011
3 << 16	0000 0000 0000 0011 0000 0000 0000 0000

Assembler Implementation

Translating Instructions

i is -5 in 16-bit two's complement notation

convert from 32-bit 2's comp by masking off the upper 16 bits

-5	1111 1111 1111 1111 1111 1111 1111 1011
0xffff	0000 0000 0000 0000 1111 1111 1111 1111
-5 & 0xffff	0000 0000 0000 0000 1111 1111 1111 1011

or'ing these parts all together we have

instr = (5 << 26) | (2 << 21) | (3 << 16) | (-5 & 0xffff)

(5 << 26) |	0001 0100 0000 0000 0000 0000 0000 0000
(2 << 21) |	0000 0000 0100 0000 0000 0000 0000 0000
(3 << 16) |	0000 0000 0000 0011 0000 0000 0000 0000
(-5 & 0xffff)	0000 0000 0000 0000 1111 1111 1111 1011
= instr	0001 0100 0100 0011 1111 1111 1111 1011

Assembler Implementation

Translating Instructions

- In C++ the instruction `bne $2,$3,-5` becomes
  ```
  unsigned int instr;
  instr = (5 << 26) | (2 << 21) | (3 << 16) | (-5 & 0xffff);
  ```
- However if you try `cout << instr;` you will get it represented as an integer in decimal format, e.g. 340000763 which is not what we want.
- The output operator (<<) will convert `instr` to the decimal representation and print it out as 9 bytes of ASCII: 0x33 (which is ASCII for 3), 0x34 (which is ASCII for 4), 0, 0, 0, 0, 0x37 (ASCII for 7),… just like you did for A2P6 where you printed out a number in decimal format using ASCII
- So *we must write out each byte as a char,* i.e. …

Assembler Implementation

Translating Instructions

- write out each byte as a char and *do not add newlines*

```
cout << char(instr >> 24) << char(instr >> 16)
     << char(instr >> 8)  << char(instr);
```

- `char()` only considers the least significant byte, the rest is ignored, e.g. `char(0x12345678) = char(0x345678)`

```
                   = char(0x5678)
                   = char(0x78)
                   = 'x'
```

- When we output the most significant byte of the word first, e.g. `(instr >> 24)` first, it is called *big endian* format.
- Other processors use *little endian* format, in which case we would write out the least significant byte of the word first.

Cautions

Caution # 1: Bitwise *or* and Negative Numbers

- for all x we have the following : -1 | x = -1
- -1 in 32-bit two's complement (hexadecimal) is 0xffffffff
- bitwise *or* anything with all 1's will give you back all 1's
- *caution:* any time a parameter may be a negative number always mask it to the appropriate size (using bitwise *and*) before using bitwise *or*

Caution 2: Arithmetic Shift vs. Logical Shift

- there are two types of shift operations
- they give the same results for
  - shift left
  - shift right when the MSB (most significant bit) is 0
- they give *different results for shift right when the MSB is 1*

Cautions

Caution 2: Arithmetic Shift vs. Logical Shift

- *Logical Shift*

```
unsigned int ui = 0x87654321 // C++ uses
ui >>  8 = 00876543          // logical shift
ui >> 16 = 00008765          // for unsigned ints
ui >> 24 = 00000087
```

- *Arithmetic Shift*

```
int si = 0x87654321          // C++ behaviour is
si >>  8 = ff876543          // implementation
si >> 16 = ffff8765          // dependent for
si >> 24 = ffffff87          // negative signed ints
```

- For shift right, logical shift adds 0'S on the left hand side, while *arithmetic shift duplicates the MSB*.
- It shouldn't be a issue on A2 where you are never printing out the bits introduced by the right shift.

Assembler Implementation

Hint for Translating Instructions

- CS 241's subset of MIPs assembly language instructions only come in *a few different formats*
  1. add, sub, slt, sltu
  2. mult, div, multu, divu
  3. mfhi, mflo, lis
  4. lw, sw
  5. beq, bne
  6. jr, jalr
  7. .word

  Hint: you might consider a function for each format rather than one function for each instruction.

Racket

Racket's Bitwise Operations

bitwise and	(bitwise-and)
bitwise inclusive or	(bitwise-ior)
shift integer i to the left n bits	(arithmetic-shift i n)
shift integer i to the right n bits	(arithmetic-shift i -n)
output a byte	(write-byte)

- E.g. (5 << 26) | (2 << 21) | (0 << 16) | (-5 & 0xffff) in Racket would be:

  (bitwise-ior (arithmetic-shift 5 26) (arithmetic-shift 2 21) (arithmetic-shift 5 26) (arithmetic-shift 0 16) (bitwise-and -5 #x7fff))

Topic 4 – Regular Languages

Key Ideas

- compiler
- scanner, lexical analyzer, lexer
- formal languages: alphabet, words, language
- Regular Languages
- operations: union, concatenation, Kleene star

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.
- available (for free, legally) on the web

Creating a Program

Overview

- We now understand enough about assembly language and machine code to be able to convert an assembly language program into its equivalent program in MIPS machine code.
- *Key question: how to translate a high level language, such as C++, into machine code?*
- *Compiler* translates a high level language (such as C++) into an assembly language program (such as MIPS assembly language).
  - You can view the assembly language it generates using the -S option in gcc/g++
- *Assembler* translates an assembly language program into machine code in an *object file* (e.g MERL or ELF).

The Compiler

What a Compiler Does

- *defining task: a compiler translates a program*
  - *from a source language*
  - *to a target language*
- typically from a high-level language (e.g. C++) to low-level language (e.g. MIPS assembly)
  - i.e. from a complex (feature rich) language to a simpler one
- typically followed automatically by an assembler
  - to generate machine code
- compiling has some similarities with assembling ...

The Compiler

Basic Compilation Steps

The *steps in compiling* a program from a high level language to an assembly language program are:

1. *scanning*: create a sequence of tokens (we provided this step for you in Assignments 3).
2. *syntax analysis*: create a *parse tree* (new)
3. *semantic analysis*: create a symbol table (similar an assembler) and *type checking* (new)
4. *code generation*: similar, but more complicated for a compiler (as compared to an assembler)

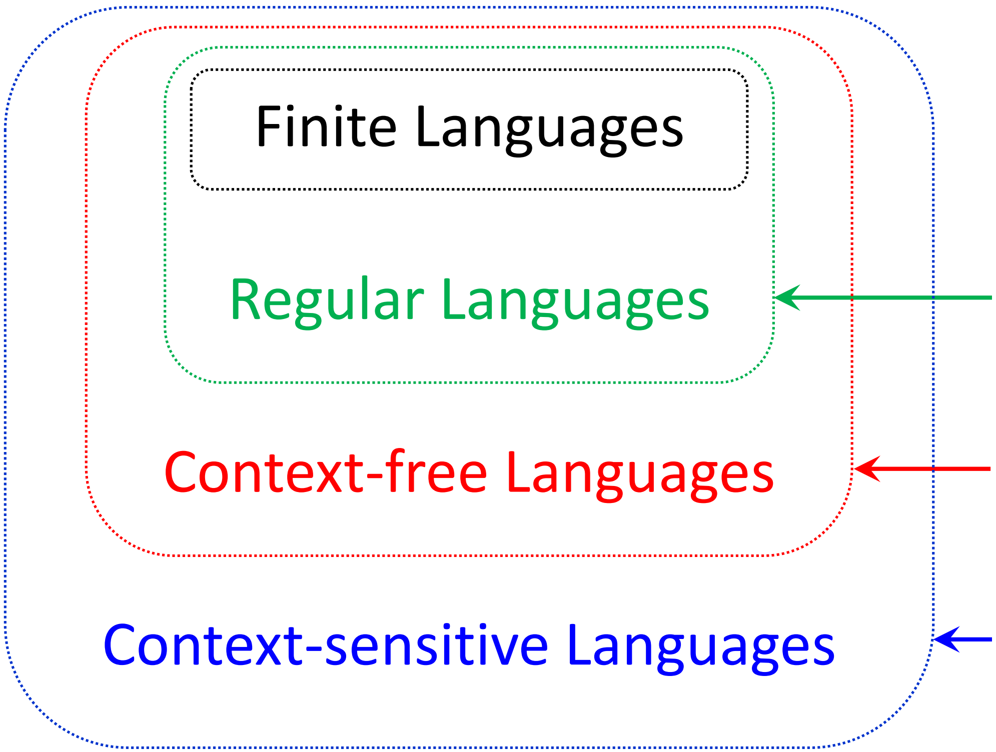
The Compiler

Basic Compilation Steps

- The goal of each of these steps is to *find increasingly more sophisticated errors* in a program.
- And if the program does have an error, then identify
  - the likely source of the error
  - how to fix it
- General approach: define an *increasingly more sophisticated set of languages* that can catch increasing more sophisticated types of errors.
- Caution: no compiler can find all errors.

The Compiler

Compilation Steps



Steps in Compiling

1. Lexical Analysis: identify each token
2. Syntactic Analysis
3. Semantic Analysis
4. Code Generation

Do not worry about steps 2–4 for now.

Step 1: Lexical Analysis

What is Lexical Analysis?

- A *scanner* or *lexer* performs *scanning* or *lexical analysis,* i.e. it breaks the input (a program) into a sequence of tokens, i.e. (kind, lexeme) pairs
- It answers the questions: What are the keywords, operators, constants, delimiters, IDs, etc. in the code?
- We need more kinds of tokens for a high level language than for assembly language, e.g.
  - *keyword*: int float if for while return ...
  - *operator*: + - * / = < <= > >= == != ...
  - *constant*: 0, 1, 2, ...
  - *delimiter*: ( ) { } [ ] , ; ...
  - *identifiers (IDs)*: maxEntry anArray numRows i answer ...

Step 1: Lexical Analysis

Scanner Input:

```
int maxEntry (int *anArray, int numRows) {
    // return the maximum entry in anArray
    etc.
```

Scanner Output:

- (INT, “int”)
- (ID, “maxEntry”)
- (LPAREN, “(“)
- (INT, “int”)
- (STAR, “*”)
- (ID, “anArray”)
- (COMMA, “,”)
- (INT, “int”)
- (ID, “numRows”)

  etc.

Step 1: Lexical Analysis

Some Kinds of Tokens

- *keywords*
  - easy to recognize
  - there are a fixed number of them, roughly 10 in WLP4 (CS241's Waterloo Language Plus Pointers Plus Procedures)
  - there is *never any ambiguity* about them
  - you cannot have a variable named *while* in C++
- *delimiters and operators*
  - easy to recognize
  - there are a fixed number of them
  - *some ambiguity*: does "*" represent multiplication or dereferencing a pointer

Step 1: Lexical Analysis

Some Kinds of Tokens

- *constants and names*
  - harder to recognize: variable length
  - need some sort of pattern matching
  - must determine when this token ends and the next one begins
  - there are an infinite number of possible names and constants in a typical programming language

Challenges

- *Challenge 1:* how to *specify* all the elements in the infinite set of valid tokens for CS241's WLP4, C++, Racket, etc.

Scanning Background

Challenges

- *Challenge 2:* clearly and unambiguously *recognize* all the tokens in a computer language, say WLP4.

Complications

- names and constants have variable length
- some tokens, such as “*”, mean different things in different contexts
- there are many types of identifiers: function names, function arguments, local variables

  - have to be able to recognize these different types
- *Approach:* We will use formal languages.

Formal Languages

Why Formal Languages?

Goal: give a *precise specification* of a language

- describe (specify) a computer language, such as C++
- in such a way that it is possible to tell if the input (i.e. a program) meets the specification
- in an automated fashion (i.e. a computer program).

Why do we need a formal (i.e. mathematical) way?

- as a means of communication
- to determine (i.e. prove mathematically) the expressive power and limitations of the language
- to guide how to make the software

Formal Languages

Approach

- For a language with a *finite size* it is easy to recognize if something is part of the language, just list all the valid words in the language. E.g. for English we have dictionaries.
- Problem: There are an *infinite number* of valid C++ identifiers or MIPS assembly language labels, so we need a method for dealing with infinite set.
- We will use *Regular Languages* to describe components of a computer language such as the set of all valid MIPS assembly language labels.
- Specifically we will use Regular Languages to describe the various kinds of tokens in a computer language.

Formal Languages

Building up a Formal Language

- *Alphabet* $\Sigma = \{ a, b \}$
  is a *finite set* of characters (a.k.a. symbols)
  i.e. there are only two characters in this alphabet

- *Strings* (a.k.a. words or sentences) are *finite sequences* of characters from the alphabet

  e.g. a, b, ba, abba, bababa

- A *language* is a *set of strings* over some alphabet

  e.g. $\mathcal{L} = \{a, b, ba, abba, bababa \}$

- Languages can be finite or infinite

  e.g. $|\mathcal{L}| = 5$ means the language $\mathcal{L}$ has five strings in it.

Regular Languages: Constants

Constants (a.k.a. the letters in our Alphabet)

- similar to the empty set, Ø, which has no elements, we have the empty string, ε, which has no characters in it.
- literal character: *a* in Σ, where Σ is our alphabet.
  - all the individual characters in the alphabet
  - the alphabet is always finite but the language may be infinite
  - e.g. there are 10 symbols that make up the natural numbers {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} but there are an infinite number of natural numbers
- This defines the single elements, but *how do we combine them to make words (a.k.a. strings)?*

Regular Languages: Basic Operations

Three Operations for Building Regular Languages

1. Union (a.k.a. *Alternation*)

R ∪ S is the union of set R and S,

- if R = {bne, beq} and S = {lw, sw}, then R ∪ S = {bne, beq, lw, sw}
- if R and S are regular languages, then so is R ∪ S
- regular languages are closed under union

2. Concatenation

R·S = { αβ : α in R and β in S}

- take a word from R and join it with a word from S
- if R = {grey, blue} and S = {jay, whale}, then R·S = {greyjay, greywhale, bluejay, bluewhale}

Regular Languages: Basic Operations

Three Operations for Building Regular Languages

2. Concatenation (continued…)

- concatenation with the empty string, ε, does nothing,
  - i.e. $\alpha\varepsilon = \varepsilon\alpha = \alpha$
- ε is the identity element under concatenation,
  - like 0 is for integer addition, i.e. $0 + x = x$,
  - and 1 is for integer multiplication, i.e. $1x = x$.
- if R = {dog, cat} and S = {fish, ε}, then R·S = {dog, cat, dogfish, catfish}
- if R and S are regular languages, then so is R·S.
- regular languages are closed under concatenation

Regular Languages: Basic Operations

Three Operations for Building Regular Languages

3. Repetition (a.k.a. *Kleene star*)

R* = smallest superset of R containing ε and closed under concatenation

- all possible combinations of the elements in R
- if R = {*a*} then R* = { ε, *a*, *aa*, *aaa*, *aaaa*, *aaaaa*, ... }
  i.e. any finite sequence of a's including no a's
- if R = {0, 1} then R* = { ε, 0, 1, 00, 01, 10, 11, 000, 001, ... }
  i.e. any finite sequence of 0's and 1's including ε
- in both these cases the size of the language R, i.e. |R|, is infinite.

Regular Languages: Basic Operations

Three Operations for Building Regular Languages

3. Repetition (a.k.a. *Kleene star*)

- if R is a regular language, then so is R*
- regular languages are closed under repetition
- use a superscript to denote R concatenated with itself, e.g.
  - e.g. if R = {*a*, *b*} then

    $R^0 = \{\varepsilon\}$ $\quad$ $R^2 = \{aa, ab, ba, bb\}$

    $R^1 = \{a, b\}$ $\quad$ $R^3 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\}$

- $R^i = R \cdot R^{i-1}$, i.e. R^i is the union R concatenated to itself *i*-1 times for each *i*.
- $R^* = \bigcup_{i=0}^{\infty} R^i$ i.e. R* is the union of R concatenated with itself any finite number of times.

Regular Languages: Examples

Some Finite Regular Languages

- the empty set Ø or { }
- {ε} is the language that consists of the empty string
- {*a*} is the singleton set consisting of the word *a*
- {*ab*} is the singleton set consisting of the word *ab*
- {*a*, *ab*, *aba*} is the set consisting of the three words *a*, *ab*, and *aba*
- *key idea:* use these three operations to specify more complicated regular languages
- {*a*}∪{*b*} is the set {*a*, *b*}
- ({*h*}∪{*c*})·{*at*} is the set {*hat*, *cat*}
- ({*a*}∪{*b*}) · ({*c*}∪{*d*}) is the set {*ac*, *ad*, *bc*, *bd*}

Regular Languages: Examples

Some Infinite Regular Languages over the Alphabet Σ = {*a*, *b*}

- {a}* = { ε, a, aa, aaa, ... }
  any finite sequence of a's including no a's
- {a}*·{b} = { b, ab, aab, aaab, ... }
  any finite sequence of a's including no a's followed by a b
- ({a}∪{b})* = { ε, a, b, aa, ab, ba, bb, aaa, aab ... }
  any finite sequence of a's and b's including the empty string
- {a}·({a}∪{b})* = { a, aa, ab, aaa, aab, aba, abb, aaaa, ... }
  the set of stings over {a, b} that begin with a
- Later on a more convenient way of specifying regular languages well be introduced, regular expressions.

Recognizing A Regular Language

Task

- to be able to clearly and unambiguously recognize all the tokens in a computer language

Approach

- once we've *specified* the tokens in our programming language using regular languages
- we need to *recognize* it with a Deterministic Finite Automata...

Topic 5 – Deterministic Finite Automata

Key Ideas

- deterministic finite automata (DFA)
- states, start state, accepting states, transitions
- formal definition of a DFA
- implementing a DFA

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

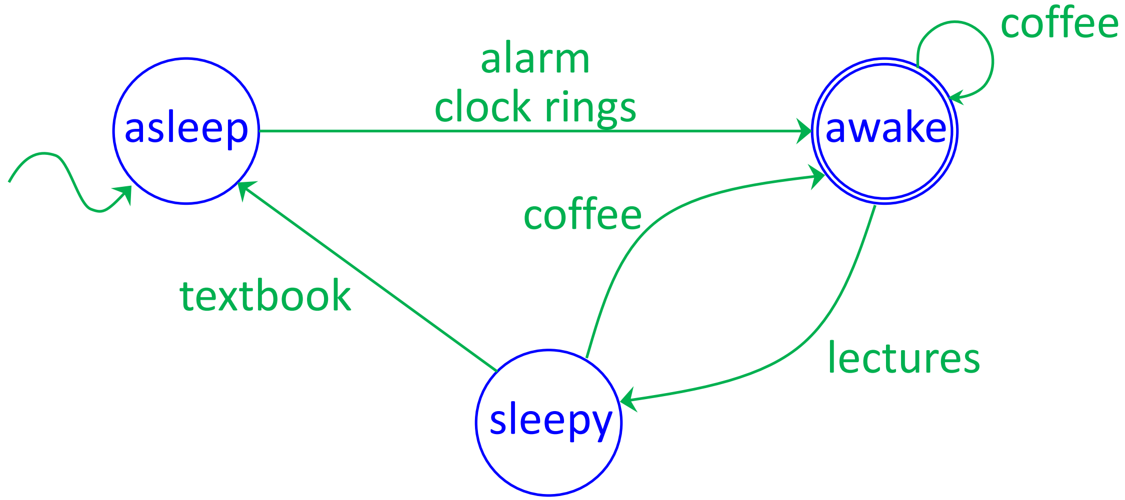
Deterministic Finite Automata (DFA)

- Also known as a deterministic *finite state machine* (FSM)
- Goal: to be able to clearly and unambiguously *recognize all the tokens in a computer language*
- The components of a DFA are
  - A finite *set of states* (represented by circles) including
    - one *start state* and
    - (possibly many) *accepting states*
  - A finite *set of input symbols* known as the alphabet
  - A finite *set of transitions* (represented by edges) from one state to another determined by the input
- The DFA determines if the input is accepted (is a word in the language) or rejected (is not a word in the language)
- In our case: is the input a valid token (and if so, which one)

DFA Diagram

Example

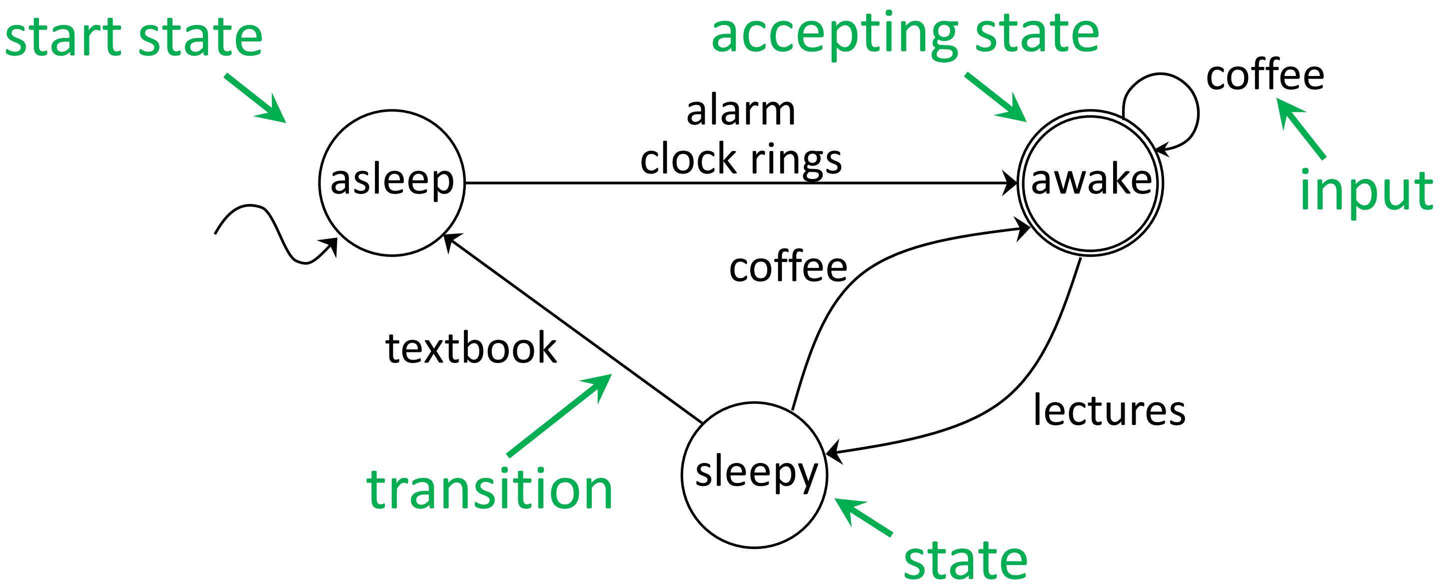
- Start state: asleep (has a curvy arrow pointing to it)
- Accepting state (a.k.a. end state): awake (has a double circle)
- Transitions: change states when input occurs: e.g. if you are in a sleepy state and drink coffee, go to the awake state.



DFA Diagram

Example

- Start state: asleep (has a curvy arrow pointing to it)
- Accepting state (a.k.a. end state): awake (has a double circle)
- Transitions: change states when input occurs: e.g. if you are in a sleepy state and drink coffee, go to the awake state.



Parts of a DFA

Comparison to Programming Languages

Similar to what you would see in a program

- a unique place to start
- transitions to various states
- one (or possibly many) places to end.

Start State → `int main () {`

Transitions → `if (input == 'a')` … `else if (input == 'b')`

Error if no transition → `else return error`

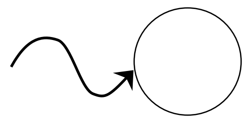
Accepting State → `return 0;`

```
int main () {
    ...
    if (input == 'a')
        ...
    else if (input == 'b')
        ...
    else
        return error
    return 0;
}
```

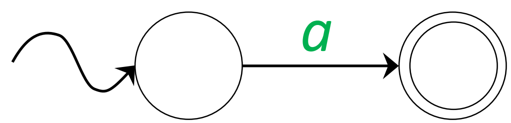
Deterministic Finite Automata (DFA)

Examples of DFAs

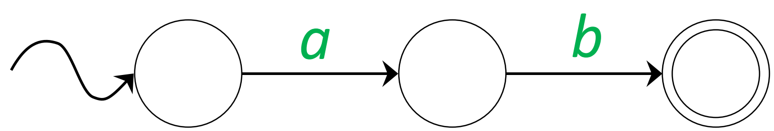
- Accepts nothing:



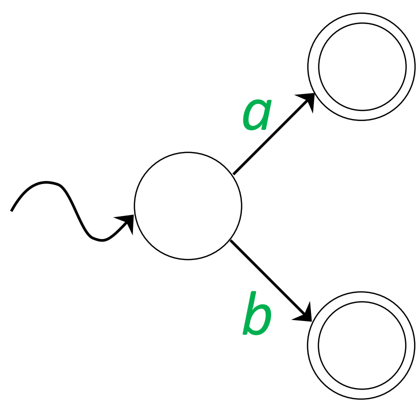
- Accepts {*a*} :



- Accepts {*ab*}:
  (concatenation)



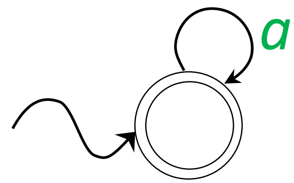
- Accepts {*a*, *b*}:
  (union)



Deterministic Finite Automata (DFA)

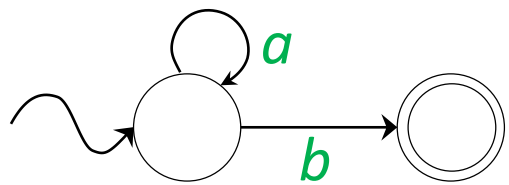
Examples of DFAs

- Accepts *a**: (repetition)



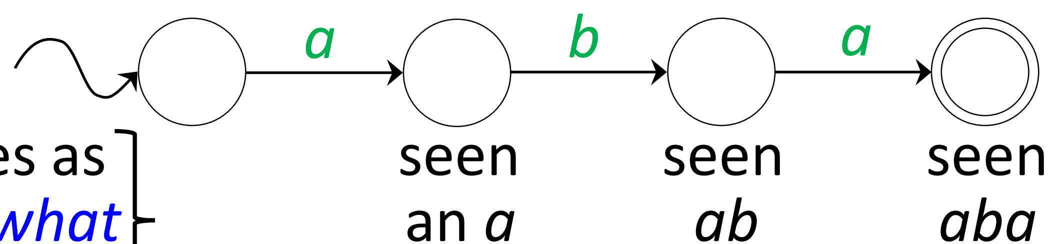
0 or more *a*’s

- Accepts *a*b*:



0 or more *a*’s followed by a *b*

- Accepts *aba*:



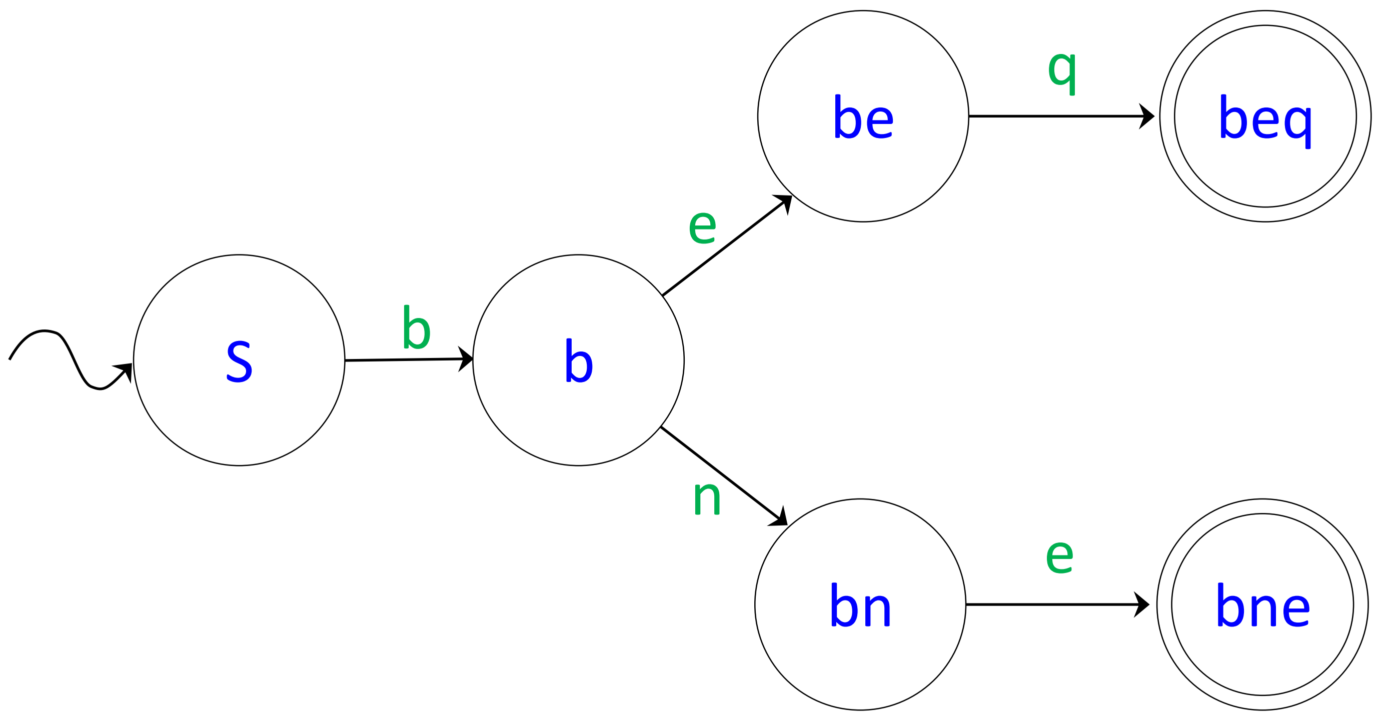
- Think of the states as *keeping track of what has been seen so far.*

- Combine these basic patterns to make more complicated DFA’s that recognize various tokens.

Deterministic Finite Automata (DFA)

Example of a DFA that Accepts a Finite Language

- Create a DFA that recognizes the two MIPS branch instructions, i.e Σ ={b,e,n,q} and $\mathcal{L}$ = {bne, beq}



Deterministic Finite Automata (DFA)

Features of a DFA

- Easy to trace where you are in the computation
- it is *deterministic*, i.e. for each state, the transitions out of that state are uniquely labelled (no pair of transitions with the same label)
- *there are no explicit error states*
  - If you are in a state, and the DFA gets an input, say x, such that there is no edge out of that state with that label on it, it is an error and the word is not in the language accepted by the DFA.
- The language accepted by the DFA M is called $\mathcal{L}(M)$
  - for the previous slide $\mathcal{L}(M)$ = {bne, beq}.

Deterministic Finite Automata (DFA)

Examples of DFAs

Let Σ ={a,b,c}

- Exercise 1: Create a DFA that accepts the language of strings that contain exactly one *a*, one *b*, and no *c*'s.

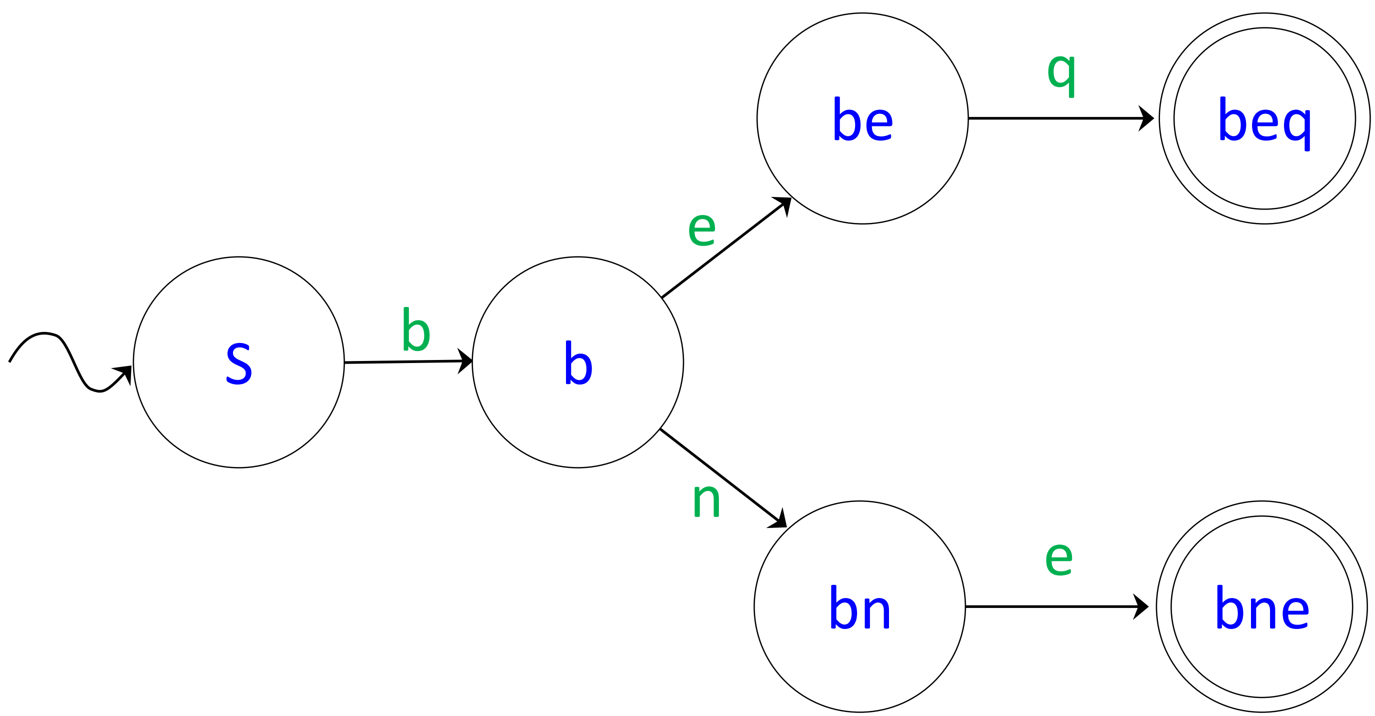
- Exercise 2: Create a DFA that accepts the language of strings that contain at least one *a*.

- Exercise 3: Create a DFA that accepts the language of strings that contain an even number of *a*'s (including 0 *a*'s).

Deterministic Finite Automata (DFA)

Recall this Example of a DFA

- This DFA recognizes the MIPS branch instructions, i.e.
  Σ ={b,e,n,q} and $\mathcal{L}$ = {bne, beq}



Deterministic Finite Automata (DFA)

Formal Definition

A DFA is a 5-tuple (Σ , Q, q_0, A, δ) where

- Σ is a finite alphabet, e.g. Σ ={b,e,n,q}
- Q is a finite set of states, e.g. Q={S, b, be, bn, beq, bne}
- q_0 is start state, e.g. q_0 = {S}
- A is the set of accepting states, e.g. A= { beq, bne }
- δ: Q x $\Sigma \rightarrow$ Q is a transition function that maps from the set of (state, symbol) pairs to a state, e.g. δ(S, b) = b; δ(b, e) = be; δ(b, n) = bn; δ(be, q) = beq; δ(bn, e) = bne.

  E.g. δ(b, e) = be means if the DFA is in state b and the input is e, then go to state be.

Deterministic Finite Automata (DFA)

Implementing a DFA

- Input, a sequence of characters from Σ: $c_{1,}$ c_2, … c_n

state $\leftarrow q_0$	*// start in the start state*
for i = 1 to n **do**:	*// for each character in the input*
state $\leftarrow \delta$ (state, c_i)	*// change state based on the input*
return (state $\in$ A)	*// did it end in an accepting state*

- Output TRUE (i.e. state $\in$ A) means $c_1c_2\cdots c_n$ is a word in the language recognized by the DFA, output FALSE otherwise.
- Typically implement δ (state, c_i) as a table…

Deterministic Finite Automata (DFA)

Implementing a DFA

- Implement δ as a table where
  - each row corresponds to a different state,
  - each column corresponds to a letter in the alphabet, Σ,
  - 🚫 means error.

Input

States

δ	b	e	n	q
S	b	🚫	🚫	🚫
b	🚫	be	bn	🚫
bn	🚫	bne	🚫	🚫
bne	🚫	🚫	🚫	🚫
be	🚫	🚫	🚫	beq
beq	🚫	🚫	🚫	🚫

Topic 6 – Finite Automata

Key Ideas

- Non-deterministic Finite Automata (NFA)
- ε-Non-deterministic Finite Automata (ε-NFA)
- transducers
- implementing a NFA

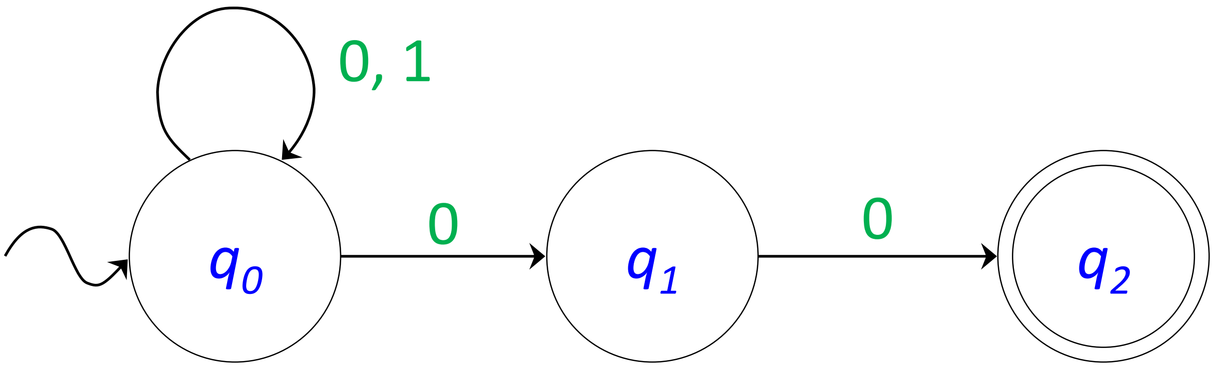
References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Non-deterministic Finite Automata (NFA)

How a NFA Differs

- Key Difference: In a NFA, *two or more transition leaving the same state can have the same label yet lead to different states.*
- The next state in non-deterministic, i.e. it is a set of possible states rather than a single state.
- In state q_0 with input 0, the NFA can stay in q_0 *and* go to state q_1 i.e. its next state is the set $\{q_0, q_1\}$.

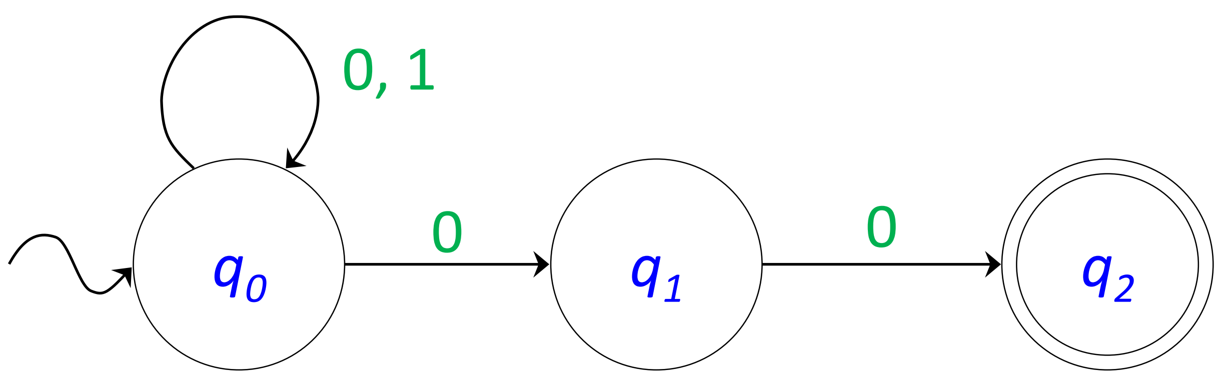


Non-deterministic Finite Automata (NFA)

Comparison with DFA

- A string is accepted if *at least one path* leads to an accepting state.
- A string is rejected if *no paths* lead to an accepting state.
- The NFA accepts $\{0,1\}^* \cdot \{00\}$, i.e. the language of strings over the alphabet $\{0, 1\}$ that end with 00.

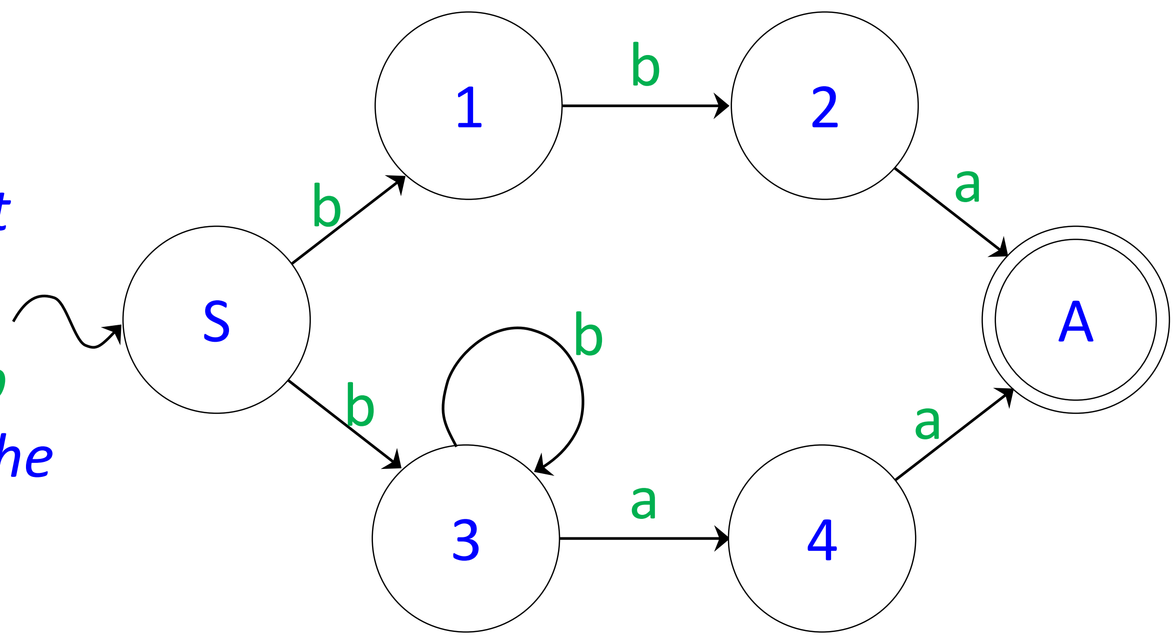
- It is often easier to design an NFA rather than an equivalent—but more complex—DFA (e.g. to tokenize input).
- Algorithms exist to convert an NFA to an equivalent DFA.



Non-deterministic Finite Automata (NFA)

Comparison with DFA

- Let Σ ={a, b} and let $\mathcal{L}$ = {bba, bb*aa}, i.e. $\mathcal{L}$ is: 2 *b*'s followed by an *a* or at least one *b* followed by two *a*'s.
- First try this as a DFA.
- Next consider the NFA:
- If we are in state *S* and we get input *b* we move to *the set of states {1, 3}*.
- If we get another *b* we then move to *the set of states {2, 3}*.



Non-deterministic Finite Automata (NFA)

Comparison with DFA

- An NFA is a FA that *allows you to be in multiple states at the same time,* i.e. a set of states.
- Terminology: 2^Q is the *power set* of Q, i.e. all the possible subsets of Q.
- E.g. if Q = {a, b, c} then 2^Q is

  { { }, {a}, {b}, {c}, {a, b}, {a, c} {b, c}, {a, b, c} }
- We use the notation 2^Q because $| 2^Q | = 2^{|Q|}$
- For a NFA the *transition relation maps onto a set of states rather than a single state*, T: $Q \times \Sigma \rightarrow 2^Q$
- If in state q with input c, if there is no transition from that state with that input then T(q, c) = { }, the empty set.

Non-deterministic Finite Automata (NFA)

Implementing a NFA

- The input is a sequence of characters from Σ, i.e. $c_1 c_2 \cdots c_n$

```
states ← {q0}                    // start in the start state
for ci in input do:              // for each char in the input
    s' ← { }                     // initialize s' to the empty set
    for s in states do:          // for each state you are in,
        s' ← s' U T(s, ci)       // find all possible next states
    states ← s'
return (states ∩ A ≠ { } )       // is the NFA in an accepting state
```

- Output TRUE if one of the states you end up in is an accepting state (i.e. in the set A)
- Recall $T(s, c_i)$ is the set of states that the NFA will go to when it is in state s and processes input c_i.

Non-deterministic Finite Automata (NFA)

Implementing a NFA

- Similar to C++ where sum is initialized to 0, you iterate through states and sum accumulates the sum of all the elements in states.

```
int states[] = {1, 2, 3};
int sum = 0;                    // identity element for addition
for (auto &s : states)          // sum = 0 + 1 + 2 + 3
   sum = sum + s;
```

- Here s' is initialized to the empty set, you iterate through the states and s' accumulates the union of all the states that the NFA can go to from states s with input c_i.

states = $\{q_1, q_2, q_3\}$

s' $\leftarrow$ { } // identity element for union

for s in states **do:** // s' = { } U $T(q_1, c_i)$ U $T(q_2, c_i)$ U $T(q_3, c_i)$

s' $\leftarrow$ s' U $T(s, c_i)$

Non-deterministic Finite Automata (NFA)

Example 1

- Input: c_1c_2 = 00 A = $\{q_2\}$
- $T(q_0, 0) = \{q_0, q_1\}$ $T(q_0, 1) = \{q_0\}$ $T(q_1, 0) = \{q_2\}$

Code	Value of Various Variables
1. states $\leftarrow \{q_0\}$	states = $\{q_0\}$
2. **for** c_i in input **do**:	$c_1 = 0$
3. $\quad$ s' $\leftarrow$ { }	s' = { }
4. $\quad$ **for** s in states **do:**	s = q_0
5. $\quad\quad$ s' $\leftarrow$ s' U T(s, c_i)	s' = { } U $T(q_0, 0) = \{q_0, q_1\}$
6. $\quad$ states $\leftarrow$ s'	states = $\{q_0, q_1\}$

Now repeat the **for** loop (lines 2-6) one more time…

Non-deterministic Finite Automata (NFA)

Example 1

- Input: $c_1c_2 = 00$ $\qquad A = \{q_2\}$
- $T(q_0, 0) = \{q_0, q_1\}$ $\qquad T(q_0, 1) = \{q_0\}$ $\qquad T(q_1, 0) = \{q_2\}$
- from previous slide, currently states = $\{q_0, q_1\}$

2. **for** c_i in input **do**:	$c_2 = 0$
3. $\quad s' \leftarrow \{\ \}$	$s' = \{\ \}$
4. $\quad$ **for** s in states **do:**	s in $\{q_0, q_1\}$
5. $\qquad s' \leftarrow s' \cup T(s, c_i)$	$s' = \{\ \} \cup T(q_0, 0) = \{q_0, q_1\}$
	$s' = \{q_0, q_1\} \cup T(q_1, 0) = \{q_0, q_1, q_2\}$
6. $\quad$ states $\leftarrow s'$	states $= \{q_0, q_1, q_2\}$
7. **return** (states $\cap A \neq \{\ \}$)	$\{q_0, q_1, q_2\} \cap \{q_2\} = \{q_2\}$
	$\{q_2\} \neq \{\ \}$ so return TRUE

Non-deterministic Finite Automata (NFA)

Example 2

- Input: $c_1c_2c_3$ = 001 A = $\{q_2\}$
- $T(q_0, 0) = \{q_0, q_1\}$ $T(q_0, 1) = \{q_0\}$ $T(q_1, 0) = \{q_2\}$
- first two iterations through the loop are the same as before so currently states = $\{q_0, q_1, q_2\}$

2. **for** c_i in input **do**:	c_3= 1
3. $s' \leftarrow \{\ \}$	$s' = \{\ \}$
4. **for** s in states **do:**	s in $\{q_0, q_1, q_2\}$
5. $s' \leftarrow s' \cup T(s, c_i)$	$s' = \{\ \} \cup T(q_0,1) \cup T(q_1,1) \cup T(q_2,1)$
	$s' = \{\ \} \cup \{q_0\} \cup \{\ \} \cup \{\ \} = \{q_0\}$
6. states $\leftarrow s'$	states = $\{q_0\}$
7. **return** (states $\cap$ A $\neq \{\ \}$)	$\{q_0\} \cap \{q_2\} = \{\ \}$
	$\{\ \} \neq \{\ \}$ is FALSE

Non-deterministic Finite Automata (NFA)

Comparison with DFA

- Let Σ ={a, b, c} and let $\mathcal{L}$ be the language such at each string in $\mathcal{L}$ contains at most two different letters in it. E.g. *ab*, *bbcc* and *aaaccc* are in $\mathcal{L}$ but *abc* is not.
- NFA version

Non-deterministic Finite Automata (NFA)

Comparison with DFA

- Let Σ ={a, b, c} and let $\mathcal{L}$ be the language such at each string in $\mathcal{L}$ contains at most two different letters in it. E.g. *ab*, *bbcc* and *aaaccc* are in $\mathcal{L}$ but *abc* is not.
- DFA version

Working with DFAs vs. NFAs

DFAs

- *easier:* to implement

NFAs

- *simpler:* tend to have less states than a corresponding DFA that accepts the same language
- *slower:* require a set data type

Expressive Power

- The two types have the *same expressive power.*
- I.e. languages that can be recognized with one, can be recognized with the other.

Deterministic Finite Automata (DFA)

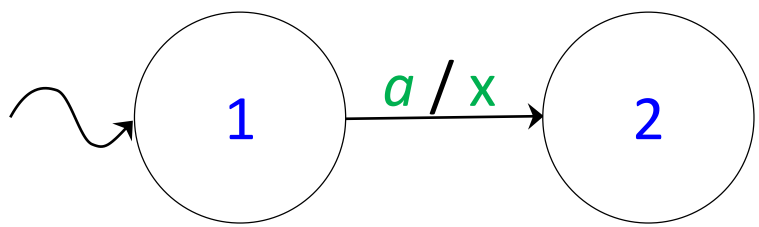
Where are DFA's used?

- lexer / scanner / translating (that's us!)
- transforming input (transducers)
- searching in text
- a computer processor is a highly complex DFA where
  - the states are the values of all the registers and the stack
  - the input is the next instruction (fetched from RAM)
- Alan Turing imagined a computer as a combination of a finite state machine + memory
  - in his case a memory = tape
  - now we use RAM

Extensions

Transducers

- *extension*: for each transition, provide the ability to output a single character
- e.g. if the FA is in state 1, and the next input character is an *a*, then output an *x* and go to state 2.

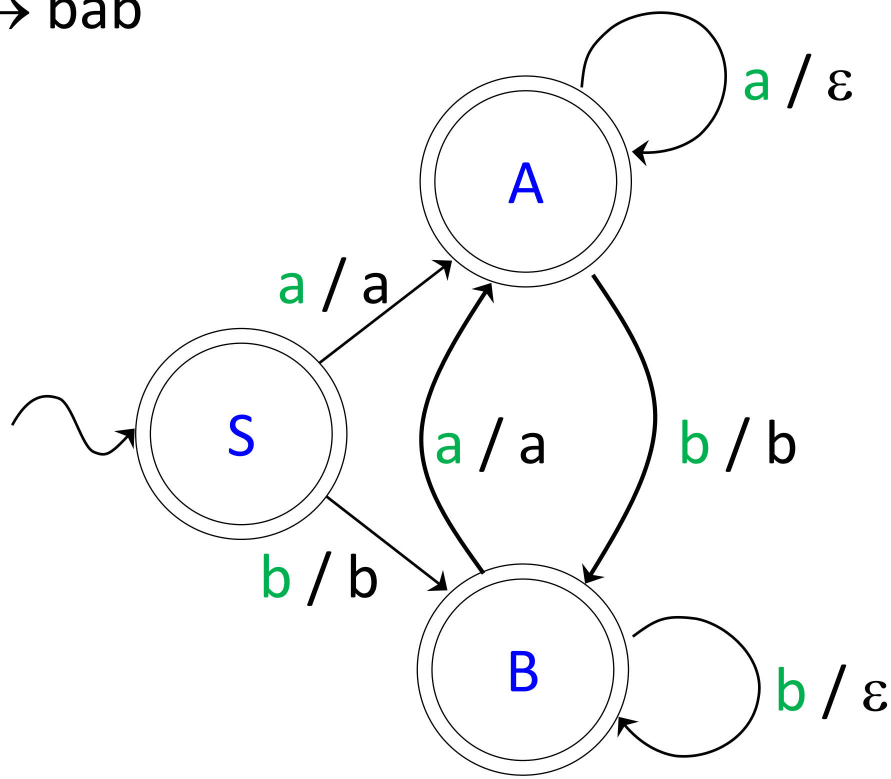


- for a lexer / scanner the output will be a token

Extensions

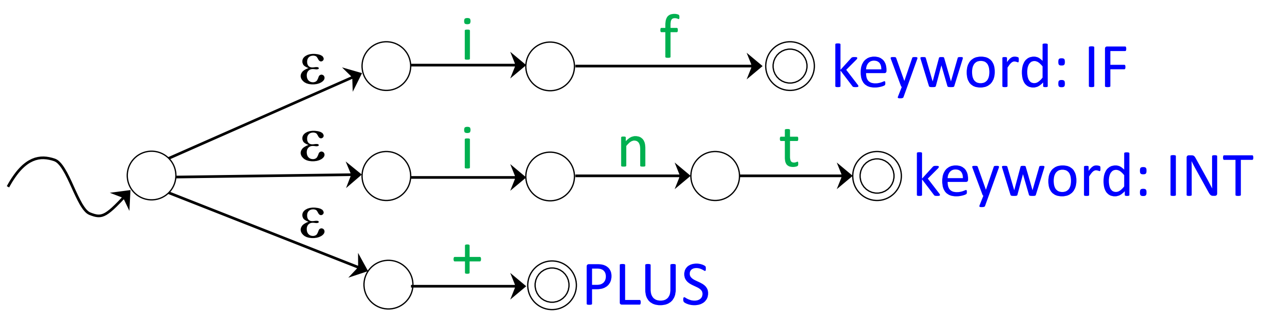
Transducers

- This transducer removes stutters (the same character more than once in a row) from the input stream, i.e. aaabbaa → aba baaaaaabbb → bab



ε-Non-deterministic Finite Automata (ε-NFA)

- An *ε-NFA* allows the use of *ε-transitions*, i.e. a transition that occurs without consuming (or requiring) any input.
- ε-NFAs are useful when you want to join together several DFAs that each recognize different tokens
- e.g. an ε-NFA



- an ε-NFA can be converted to an NFA (more on this topic later).

Topic 7 – Regular Expressions

Key Ideas

- Regular Expressions
- Regular Expressions and Regular Languages
- Precedence Rules
- RegExs in Linux
- Extensions to Regular Expressions

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Scanning Background

Approach

- Use *regular expressions* to specify the tokens in our language
- then use a *lexer generator*
  - to convert our specification into an efficient program for recognizing tokens (i.e. a lexer or scanner)
  - examples of lexer generators are: lex, flex, ANTLR
- Lexers use deterministic finite automata to recognize tokens.
- But first, what is a *regular expression*?
- Answer: a *precise way of describing a language* (i.e. a set of strings) in particular a regular language…

Regular Expressions

Recursive Definition

Regular expressions are a way of *specifying* regular languages.

The elements (base cases) of a regular expression are

- $\varnothing$ i.e. $\mathcal{L} = \{\ \}$, i.e. the empty set,
- ε i.e. $\mathcal{L} = \{ \varepsilon \}$, i.e. the language consisting of ε,
- a where $a \in \Sigma$ i.e. $\mathcal{L} = \{ a \}$ the language consisting of a single symbol.

The expressions are built up via three operations

- *concatenation:* E_1E_2 where $E_1$and E_2 are regular expressions,
- *union:* $E_1|E_2$ where $E_1$and E_2 are regular expressions,
- *repetition:* E* where E is a regular expression.

Note that $\varnothing$ concatenated with anything yields $\varnothing$.

Regular Expressions

Regular Expressions and Regular Sets

For the alphabet Σ = { *a, b*}, the regular expression …

- *a* specifies the language {*a*}
- *ab* specifies the language {*ab*}
- *a*|*b* specifies the language {*a*, *b*}
- *aa*|*ab*|*bb* specifies the language {*aa*, *ab*, *bb*}
- *a** specifies the language { ε, *a*, *aa*, *aaa*, *aaaa*, ... }
- *a***b* specifies the language { *b*, *ab*, *aab*, *aaab*, *aaaab*, ... }
- (*a*|*b*)* specifies the language { ε, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, ... }

Regular Expressions: Issues

Precedence Rules

- *conflicting rules*: need precedence rules
  - does a|ab* mean (a|(ab))* or a|(a(b*)).
    1. Kleene star has the highest precedence
    2. concatenation
    3. union has the lowest precedence
  - use parenthesis to clarify

Regular Expressions

Examples

Create a Regular Expression for each language.

Σ ={a, b, c, r}, $\mathcal{L}_1$ = {cab, car, carb}

Σ ={a}, $\mathcal{L}_2$ = {w: w contains an even # of a's}

Σ ={a, b}, $\mathcal{L}_3$ = {w: w contains an even # of a's}

Regular Expressions

Examples

Create a DFA and a Regular Expression for each language.

Σ ={a, b}, $\mathcal{L}_1$ = {w: w contains either aa or bb}

Σ ={a, b}, $\mathcal{L}_2$ = {w: w contains no occurrence of aa or bb}

Regular Expressions

Regular Expressions (RegEx) and Linux

- For those of you who use Linux, you use regular expression all the time e.g. `ls A2*.asm` means list all the files that start with "A2" and end with ".asm"

Several Linux tools use regular expressions

- grep / egrep: search regular expressions in text files
- sed: stream editor for transforming text files
- awk: pattern scanning and processing language
- make: software building utility
- *You don't have to know about any of these tools.*

Regular Expressions

Extensions

- may see the use of the following to help simplify regular expressions, especially in Linux
- *square brackets* (with ranges)
  - [a-z] means $a|b|c|\ldots|z$
  - i.e. match one of the letters in the range a-z
  - [a-z] will match a lowercase letter in the English alphabet
  - [A-Z,a-z] will match a letter (uppercase or lower case) in the English alphabet
  - [A-Z,a-z,0-9] will match an alphanumeric character

Regular Expressions

Extensions

- *plus sign*: one or more
  - like star but excluding ε
  - [0-9]+ means [0-9][0-9]*
  - matches non-negative integers (possibly with leading 0's).
- *question mark:* matches 0 or 1 occurrence
  - [1-9]?[0-9] means ([1-9] | ε)[0-9]
  - matches one digit numbers or two digit numbers without a leading 0.
- *dot* matches any single character
  - .at matches hat, cat, fat, mat, bat, 7at, Aat, etc.
- there are many other extensions to regular expressions

Topic 8 – Scanners

Key Ideas

- scanning
- simplified maximal munch
- scanners and ε-NFAs
- scanners and DFAs

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Scanning

Quick Review

- Recall what we are trying to do: translate from a high level language to assembly language
- introduced regular expression and finite automata as a way to *specify* and *identify* words in the language
- Question: how does that work in practice?

Scanning

Scanner

- *Input:* some string *w* and a language *L*
  - in assembly language: “mult $1, $2”
  - in C++ “i = 1;”
- *Output:* a sequence of tokens
  - (ID, “mult”) (REG, “$1”) (COMMA, “,”) (REG, “$2”)
  - (ID, “i”) (BECOMES, “=”) (NUM, “1”) (SEMI, “;”)
- *Challenge:* may be more than one possible answer:

  0x12ab vs 0 x 12 ab

  HEXINT vs INT ID INT ID

  Answer: take the longest possible correct run of chars

Simplified Maximal Munch Scanning

Input

- Input consists of k characters: $c_0c_1c_2c_3 \cdots c_k$ is 12 + ⋯
- Basic Idea: *keep going until you reach an error state* (i.e. you have gone one character too far) *then go back to the previous character*
  - here 1 and 2 are part of an integer but ‘ ’ is not, so with ‘ ’ you have gone one character too far.
- *Step 1:* look at next character and check the next state
- *Step 2:* **if** the next_state == ERROR (i.e. you've gone too far)
  then look at the current state
  - *Step 2a:* **if** it was not an accepting state, **then** report a *fatal error*
  - *Step 2b:* **if** it was whitespace, **then** ignore
  - *Step 2c:* **if** it was an accepting state, **then** output the token
  - *Step 2d:* go to start state q_0, i.e. begin looking for the next token

Simplified Maximal Munch Scanning

```
i = 0                                          // start at first char and
state = q0                                     // start state of the DFA
loop:
  if ( i < k ):                                // 1: if not at end of input
    next_state = δ(state, c_i)                 //      calculate next state
  else:                                        // else end of input so
    next_state = ERROR                         //     no valid next state
  if (next_state == ERROR):                    // if next_state is too far
    if (state ∉ accepting_states):             // 2a: not a valid token
      report a fatal error and exit            //    error in input
    if (state ≠ White_space):                  // 2b: skip white space
      output token                             //    2c: output token
    state = q0                                 // 2d: go to start state
    if (i == k):                               // halt if no more input
      exit
  else:                                        // no error so
    state = next_state                         //     update state and
    i = i + 1                                  //     consider next char
```

Scanning

Two Subtleties with the Code

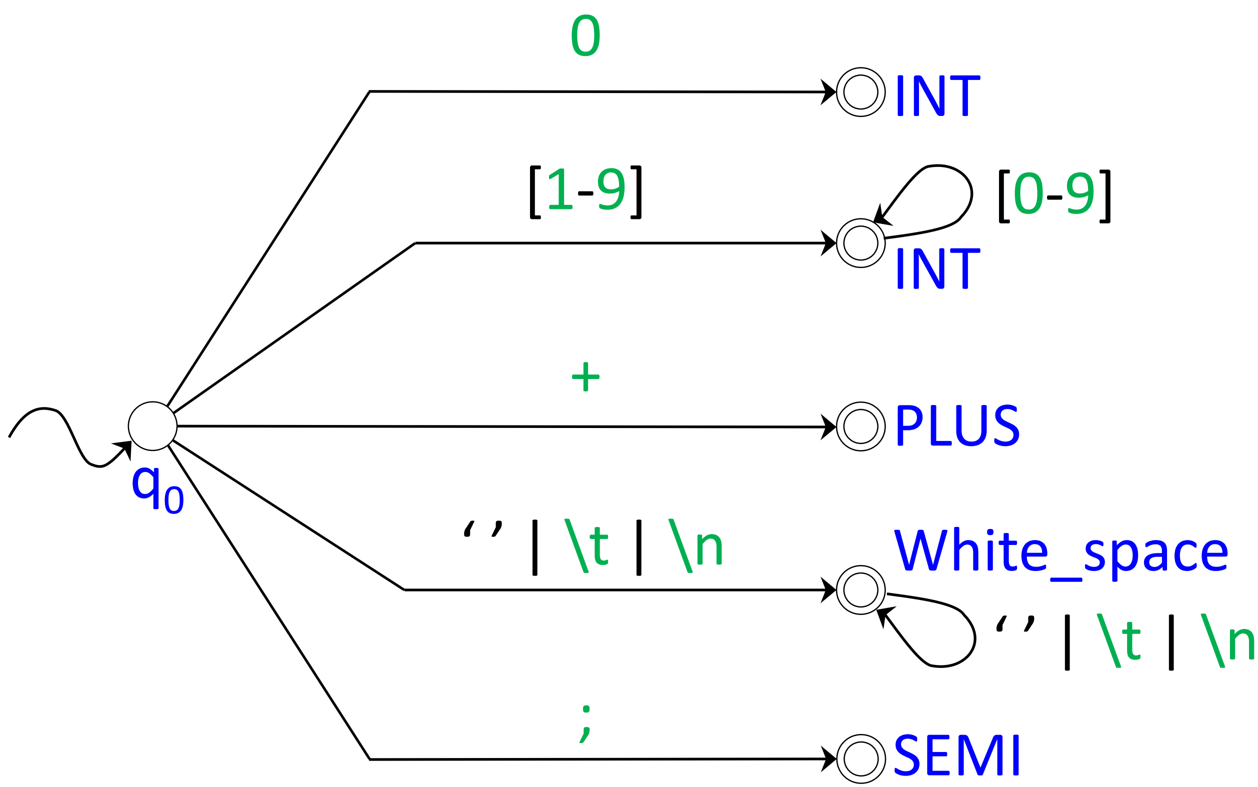
- If next_state == ERROR (lines 9-15):
  If you get an ERROR (line 8) the char counter *i* is not incremented, but the DFA does go to the start state (line 13) and *you reconsider the i^{th} character as the start of the next token*.

- When i == k (as a result of line 17-18) this is one char beyond the end of the input:

  The next_state is not updated using δ(state, c_i) (line 5) but is set to ERROR (line 7) and so if state is an accepting state (skip line 10) and the token is not White_space (line 11) then *output the token* (line 12) and exit the program (line 14-15).

Scanners and DFAs

An DFA that Recognizes a Subset of WLP4 tokens



Simplified Maximal Munch Scanning

Simplified Maximal Munch Example

Input: $c_0c_1c_2c_3c_4c_5$ is 12 +3; and the input size k = 6.

- *Goal:* want to output a single token (INT, “12”), not two tokens (INT, “1”), (INT, “2”).
- *Approach*: continue until something other than INT is seen
- i = 0, c_0 = 1 state = q_0, next_state = INT
- i = 1, c_1 = 2 state = INT, next_state = INT
- i = 2, c_2 = ‘ ’ state = INT, next_state = ERROR

  - output token (INT, “12”), line 12
  - go to q_0 the start state, line 13
  - check if at end of input, line 14-15
  - do not increment i, that is skip over lines 17-18
  - now process c_2 = ‘ ’ in state q_0 rather than in state INT

Simplified Maximal Munch Scanning

Simplified Maximal Munch Example

Input: $c_0c_1c_2c_3c_4c_5$ is 12 +3; and the input size k = 6.

- i = 2, c_2 = ‘ ’ state = q_0, next_state = White_space
- i = 3, c_3 = + state = White_space, next_state = ERROR
  - since state = White_space, do not output a token (lines 11-12) but go to start state (line 13) and process + again
- i = 3, c_3 = + state = q_0, next_state = PLUS
- i = 4, c_4 = 3 state = PLUS, next_state = ERROR
  - output token (PLUS, “+”), line 12
  - go to q_0 (start state), line 13
  - do not increment i, that is, skip over lines 17-18
  - now process c_4 = 3 in state q_0 rather than in state PLUS

Simplified Maximal Munch Scanning

Simplified Maximal Munch Example

Input: $c_0c_1c_2c_3c_4c_5$ is 12 +3; and the input size k = 6.

- i = 4, c_4 = 3 state = q_0, next_state = INT
- i = 5, c_5 = ; state = INT, next_state = ERROR
  - output (INT, “3”) and go to start state, lines 8-13
- i = 5, c_5 = ; state = q_0, next_state = SEMI
- i = 6, the test i < k on line 4 is false so next_state = ERROR, lines 6-7
- since next_state = ERROR, since state ϵ accepting_states (lines 8-9) and state ≠ White_space (line 11) then output (SEMI, “;”) and exit (lines 14-15).

Scanners and FAs

Differences between a Scanner and a Finite Automata

- A scanner *splits the input up into tokens*.
- An FA *checks if the input is a string of a language*

Using a DFA to Implement a Scanner.

- describe each of the set of tokens by a regular expression (we'll do a small subset).
  - *keywords*: if int
  - *ID*: [a-z,A-Z][a-z,A-Z,0-9]*

  operators: { + - * / % }
  delimiters: { ( ) { } , : }

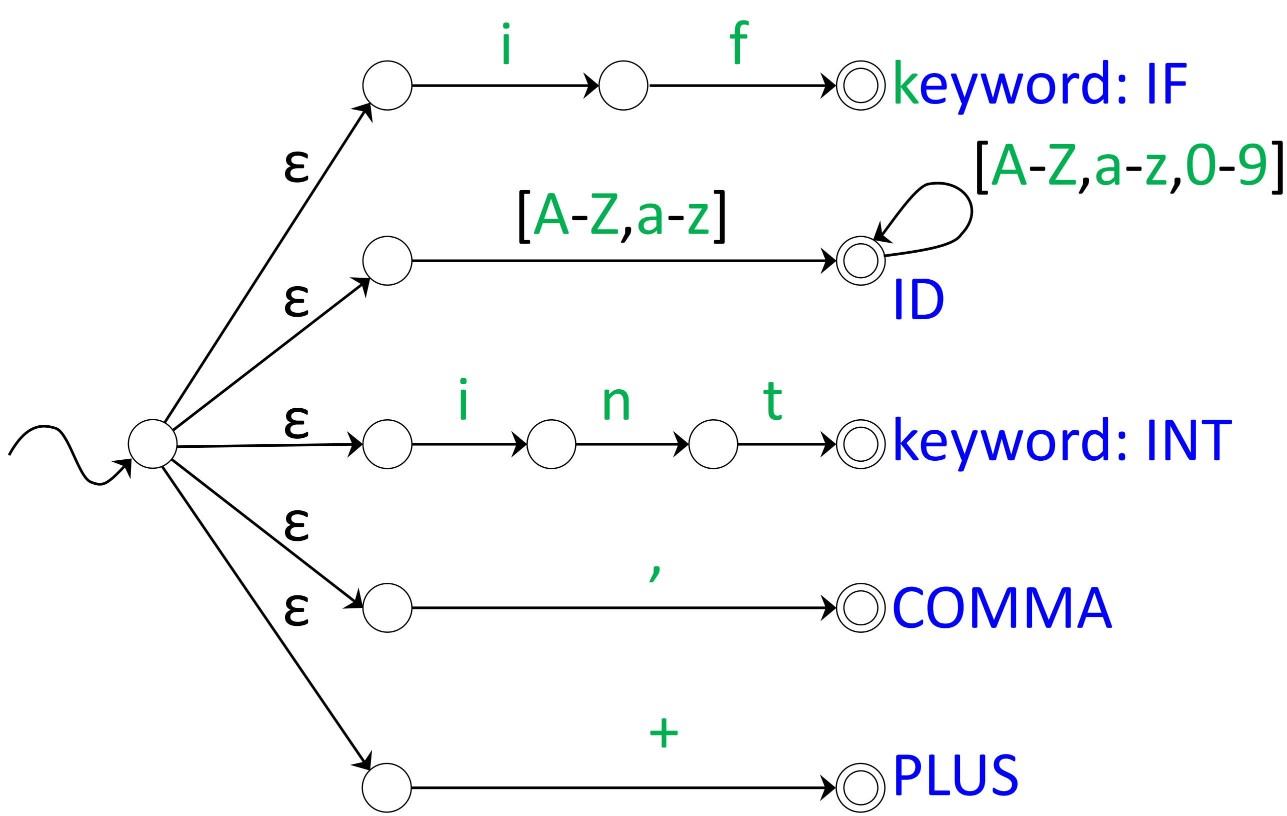
Scanners and NFAs

Using an ε-NFA to make a Scanner

- create an NFA for each regular expression
- mark the accepting states by the type of token they accept
- combine all the individual NFAs into a single large one (using ε transitions)
  - sometimes called λ (lambda) transitions
- convert from an ε-NFA to an NFA and then to a DFA
- *To keep the diagram simple*:
  - I'm using a subset of WLP4

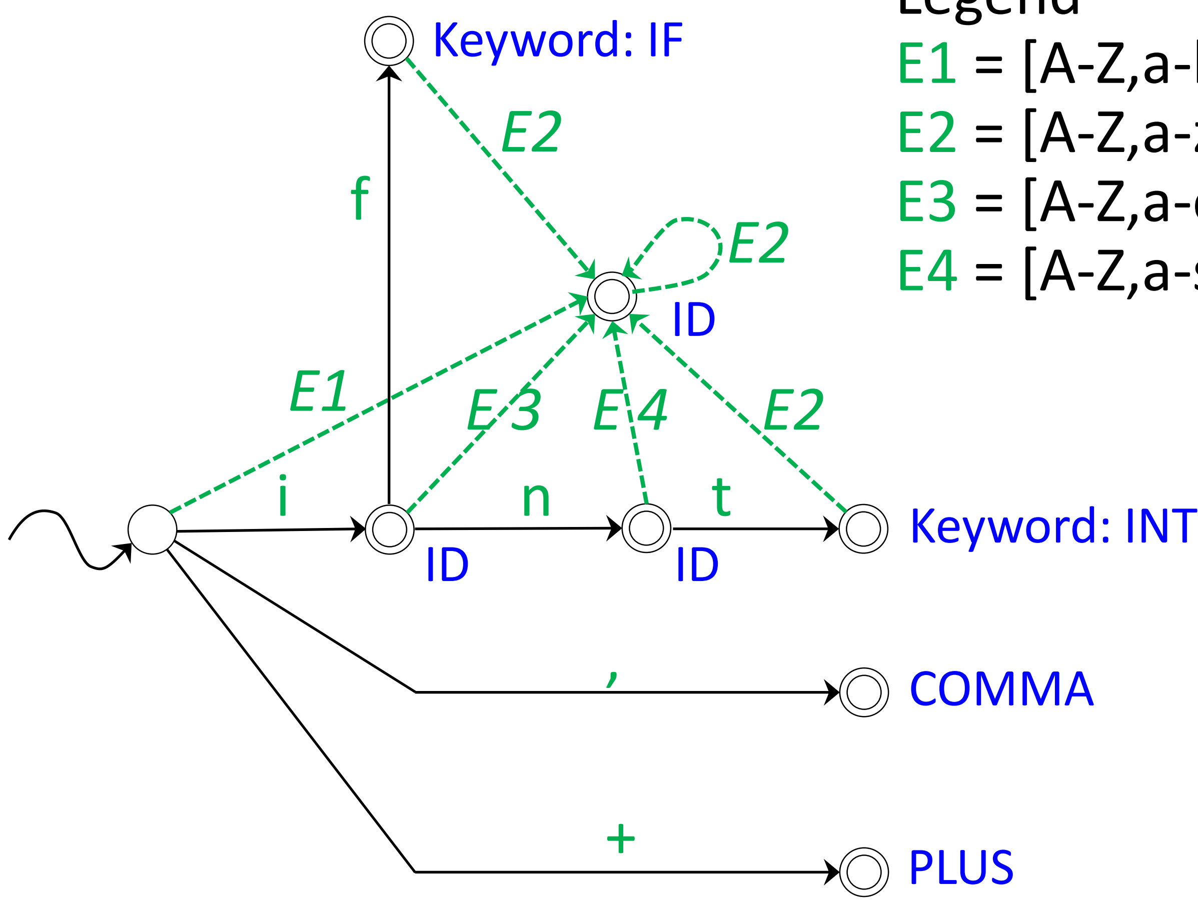
Scanners and NFAs

An ε-NFA that Recognizes a Subset of WLP4 tokens



Scanners and DFAs

The Corresponding DFA that Recognizes our Tokens

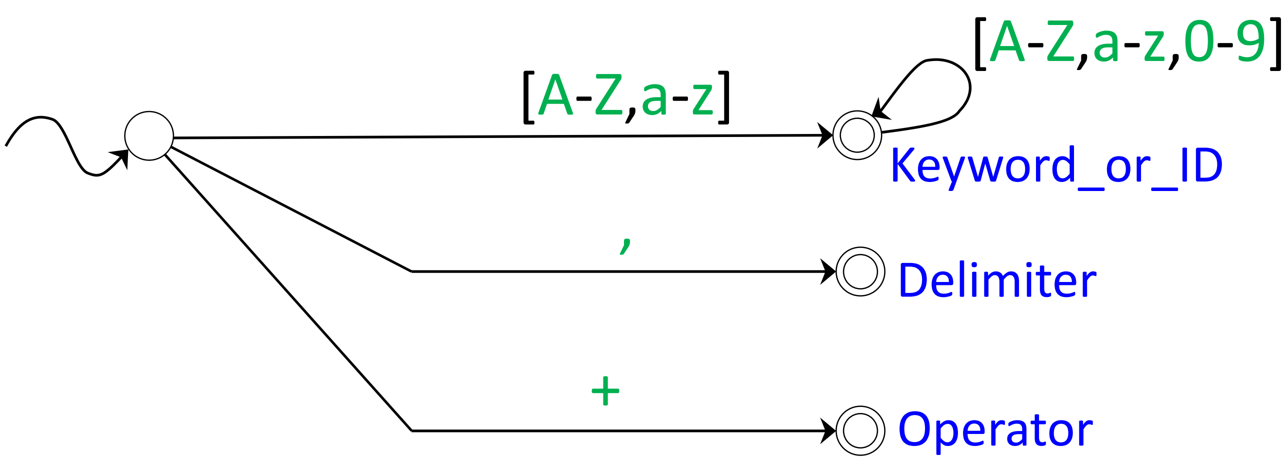


Legend
E1 = [A-Z,a-h,j-z]
E2 = [A-Z,a-z,0-9]
E3 = [A-Z,a-e,g-m,o-z,0-9]
E4 = [A-Z,a-s,u-z,0-9]

Scanners and DFAs

The Corresponding DFA that Recognizes our Tokens

- Generally it is easier to use a *DFA for only part of the task of recognizing tokens.*
  1. Combine IDs and all the Keywords into one token (Keyword_or_ID) and check if it is a particular keyword afterwards using a dictionary data structure (like a C++ set).
  2. Recognize if the input is an integer constant with the DFA and then check if it is in the valid range using C++ or Racket.



Topic 9 – Regular Languages II

Key Ideas

- convert a RE to an ε-NFA
- convert an ε-NFA to an NFA
- convert an NFA to a DFA
- equivalence of Regular Expressions (RE), DFA's, NFA's and ε-NFA's

References

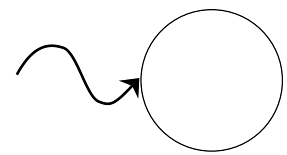
- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Regular Expressions (RE) to ε-NFAs

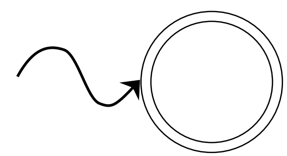
Convert an RE to an ε-NFA

Basic Idea: build up the ε-NFA recursively from the elements of a regular expression (i.e. structural induction). First the base cases.

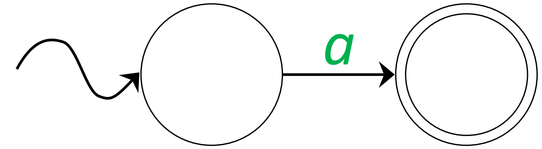
- If the RE is ∅ then the ε-NFA is:
  - no accepting state



- If the RE is ε then the ε-NFA is:
  - it accepts the empty string and nothing else



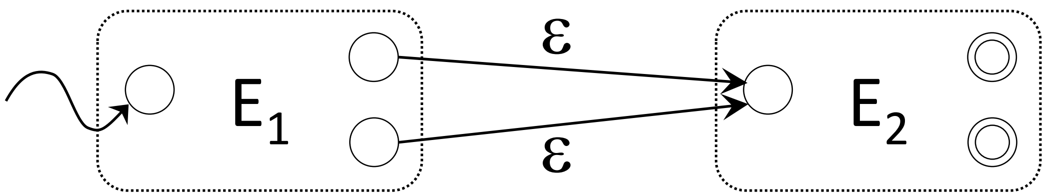
- If the RE is *a* then the ε-NFA is:



Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form E_1E_2 (i.e. *concatenation*) then convert the states of the ε-NFA that recognizes E_1 into non-accepting states and link them to the start state of the ε-NFA that recognizes E_2 via ε-transitions.

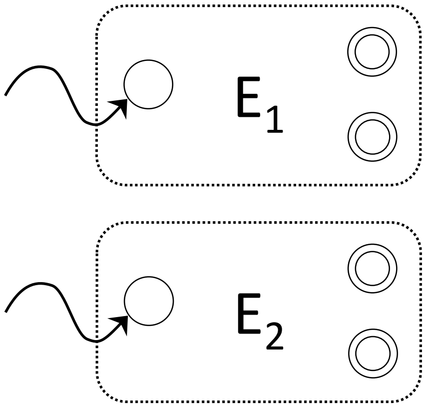


- Note: expressions and automata occur *in sequence*

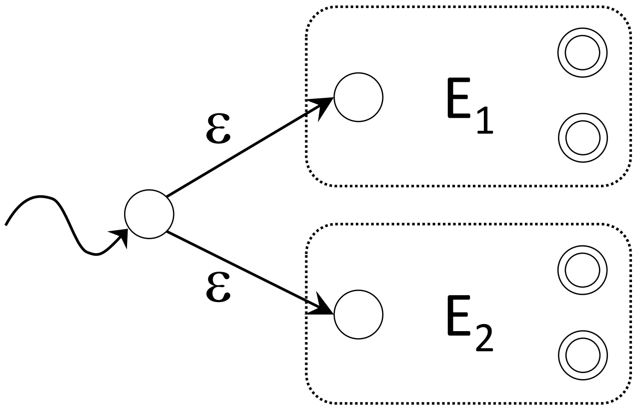
Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form $E_1|E_2$ (i.e. *union*): create a new start state and link it, via ε-transitions, to the start states of the ε-NFAs that recognizes E_1 and E_2.



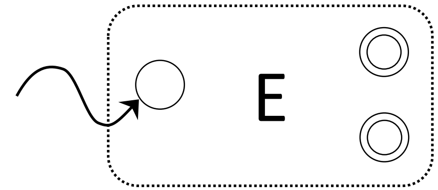
- Note: expressions and automata occur *in parallel*



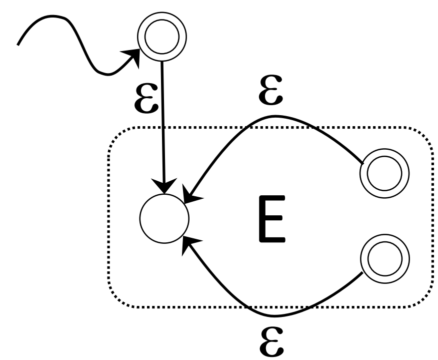
Regular Expressions (RE) to ε-NFAs

Convert an RE to an ε-NFA

If the RE is of the form E* (i.e. *repetition*):



- connect all the accepting states of the ε-NFA that recognizes E to the start state using ε-transitions
- if the start state is not an accepting state then create a new start state that makes an ε-transition to the old one (so that ε is now accepted)

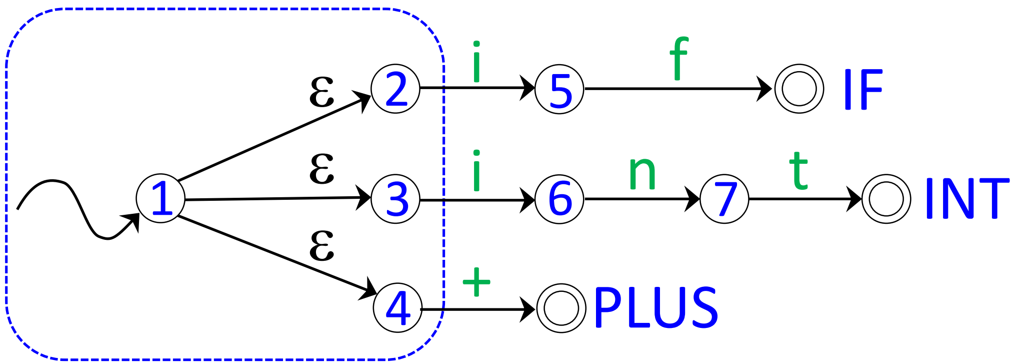


- Note: expressions and automata occur *in a cycle.*

Converting ε-NFA to NFA

ε-closure

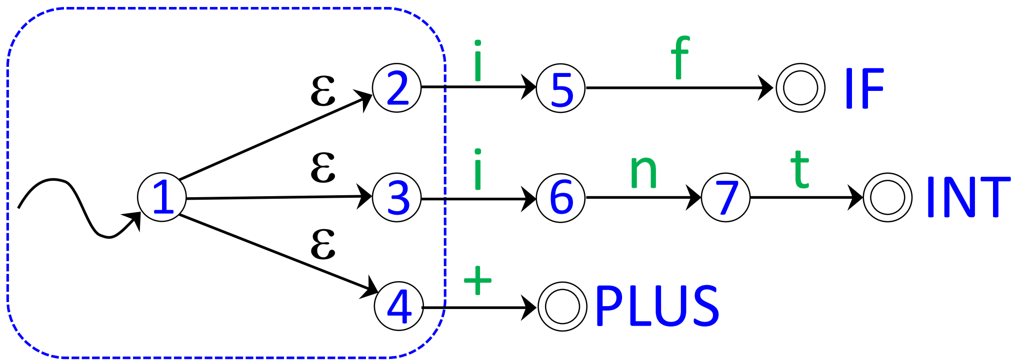
- The *ε-closure* of a state (or set of states) is the set of states that can be reached from that state (or set of states) by ε-transitions.



- The ε-closure (also denoted as ε*) of 1 is the set {1, 2, 3, 4}.
- To replace the ε-transitions from a state *q*, for each input symbol look at (i) the ε-closure of *q* (ii) followed by the transitions due to that input symbol (iii) followed by the ε-closure of the results from step (ii). Repeat this for each state.

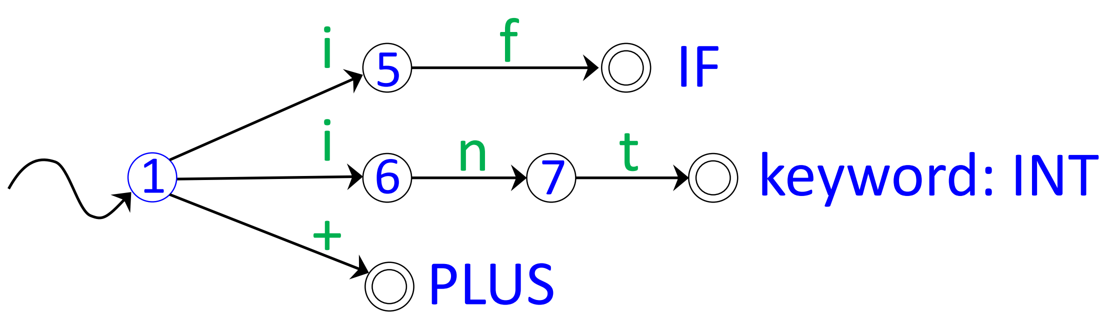
ε-Non-deterministic Finite Automata (ε-NFA)

Converting an ε-NFA to a NFA



E.g. ε-closure({1}) = {1,2,3,4}

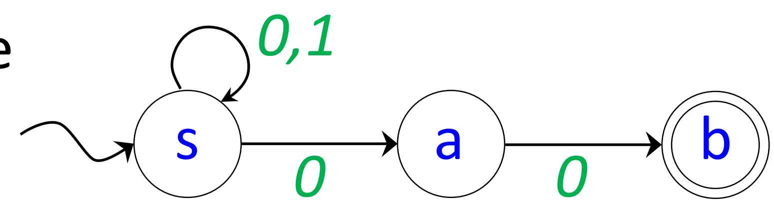
- input i: go from {1,2,3,4} to {5,6} and ε-closure({5,6}) = {5,6}.
- input +: go from {1,2,3,4} to {PLUS} and ε-closure({PLUS}) = {PLUS}.



Converting NFA's to DFA's

Subset Construction: Example 1

Basic Idea: identify a single state in the DFA with a set of states in the NFA.



Starting with the start state

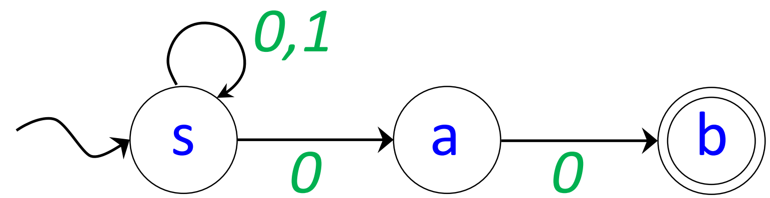
1. From State: for each possible input, track the set of Next States that can be reached.
2. If the Next State is new set of states, add it to the table and repeat step 1 for that new set of states.
3. Continue until any set that appears in the Next State column also appears in State column.

State	Input	Next State
{s}	0	
	1	

Converting NFA's to DFA's

Subset Construction: Example 1

- Starting with the start state {s} consider all possible inputs.
- state {s}
  0: stay in s or move to a, i.e. {s, a}
  1: stay in s, i.e. {s}
- The union of all these possibilities {s, a} U {s} is a new state {s, a}, so add {s, a} to State column.
- Consider all possible transitions from this new state {s, a}.

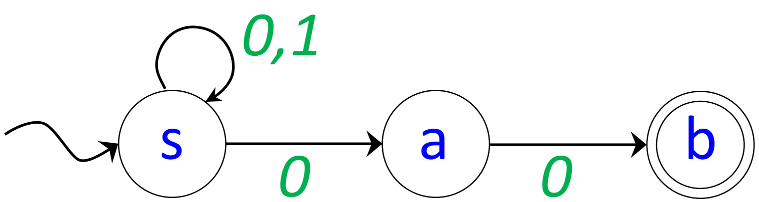


State	Input	Next State
{s}	0	{s, a}
	1	{s}
{s, a}	0	
	1	

Converting NFA's to DFA's

Subset Construction: Example 1

- From state {s, a}
  input 0
  - s: stay in s or move to a, i.e. {s, a}
  - a: move to b, i.e. {b},
  input 1
  - s: stay in {s}
  - a: drops out, i.e. { }
- The union of all these possibilities is {s, a} U {b} U {s} U { } = {s, a, b} so add {s, a, b} to the State column and consider all possible inputs when in this new state.



State	Input	Next State
{s}	0	{s, a}
	1	{s}
{s, a}	0	{s, a, b}
	1	{s}
{s, a, b}	0	
	1	

Converting NFA's to DFA's

Subset Construction: Example 1

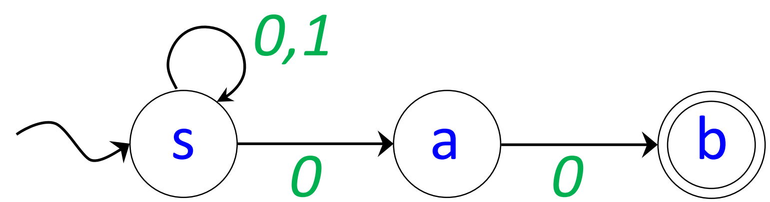
- From state {s, a, b}

  input 0

  - s: stay in s or move to a, i.e. {s, a}
  - a: move to b, i.e. {b}
  - b: no options, drops out, i.e. { }

  input 1

  - s: stay in s, i.e. {s}
  - a: no options, drops out, i.e. { }
  - b: no options, drops out, i.e. { }
- The union of all these possibilities is {s, a, b} which is already in the table.
- Create a DFA using this table.



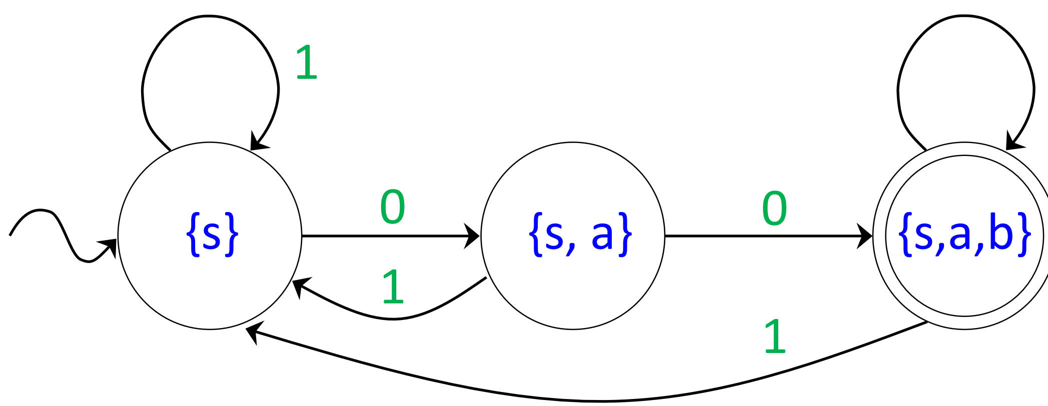
State	Input	Next State
{s}	0	{s, a}
	1	{s}
{s, a}	0	{s, a, b}
	1	{s}
{s, a, b}	0	{s, a, b}
	1	{s}

Converting NFA's to DFA's

Subset Construction: Example 1

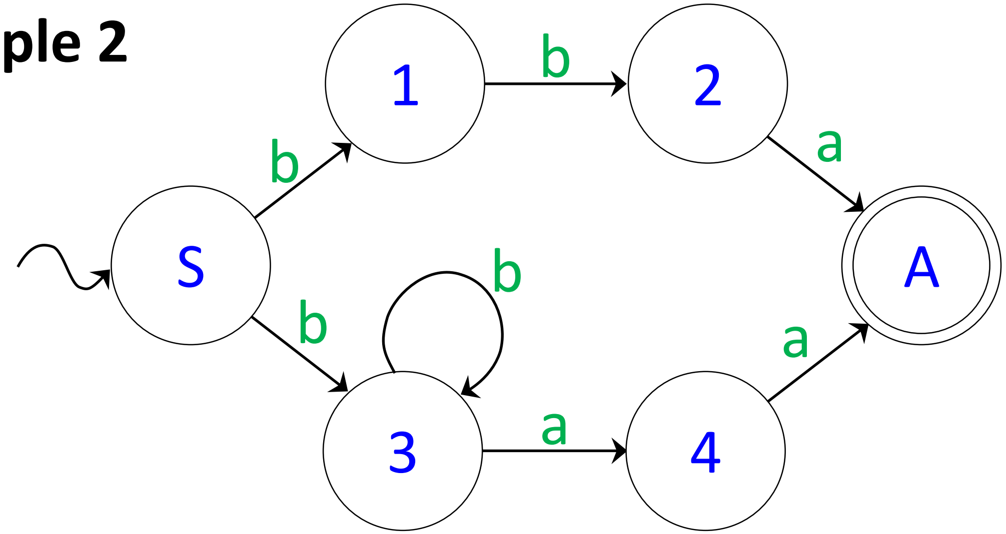
- Connect up the states with their corresponding transitions and inputs.
- The state that just contains the start state of the NFA, {s}, is also the start state of the DFA.
- Any DFA state that contains an accept state of the NFA (i.e. b) is also an accept state in the DFA.

State	Input	Next
{s}	0	{s, a}
	1	{s}
{s, a}	0	{s, a, b}
	1	{s}
{s, a, b}	0	{s, a, b}
	1	{s}



Converting NFA’s to DFA’s

Subset Construction: Example 2



- Recall the following NFA.
- in state {S}
  - input a: drops out
  - input b: move to {1, 3}
- for new state {1, 3}
  - input a: {4}
  - input b: move to {2, 3}
- for new state {4}
  - input a: {A}
  - input b: drops out
- for new state {2, 3}
  - input a: {A, 4}
  - input b: {3}
- Etc, see the table on the next slide for all seven new states

Converting NFA's to DFA's

Subset Construction: Example 2

State	Input	Next State
{S}	a	{ }
	b	{1,3}
{1,3}	a	{4}
	b	{2,3}
{4}	a	{A}
	b	{ }
{2,3}	a	{A, 4}
	b	{3}

State	Input	Next State
{A}	a	{ }
	b	{ }
{A,4}	a	{A}
	b	{ }
{3}	a	{4}
	b	{3}

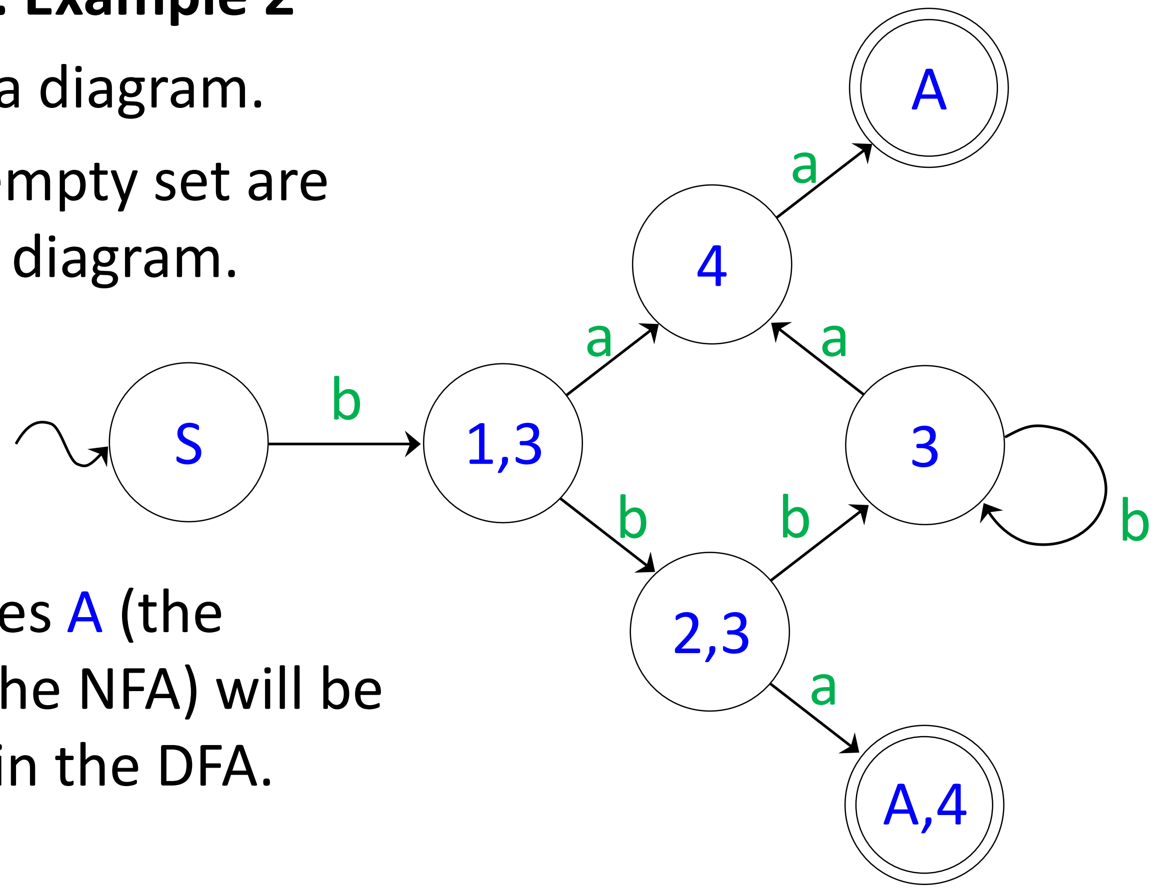
Now create a DFA with seven states using this table.

Converting NFA's to DFA's

Subset Construction: Example 2

Convert the table to a diagram.

- Transitions to the empty set are not included in the diagram.



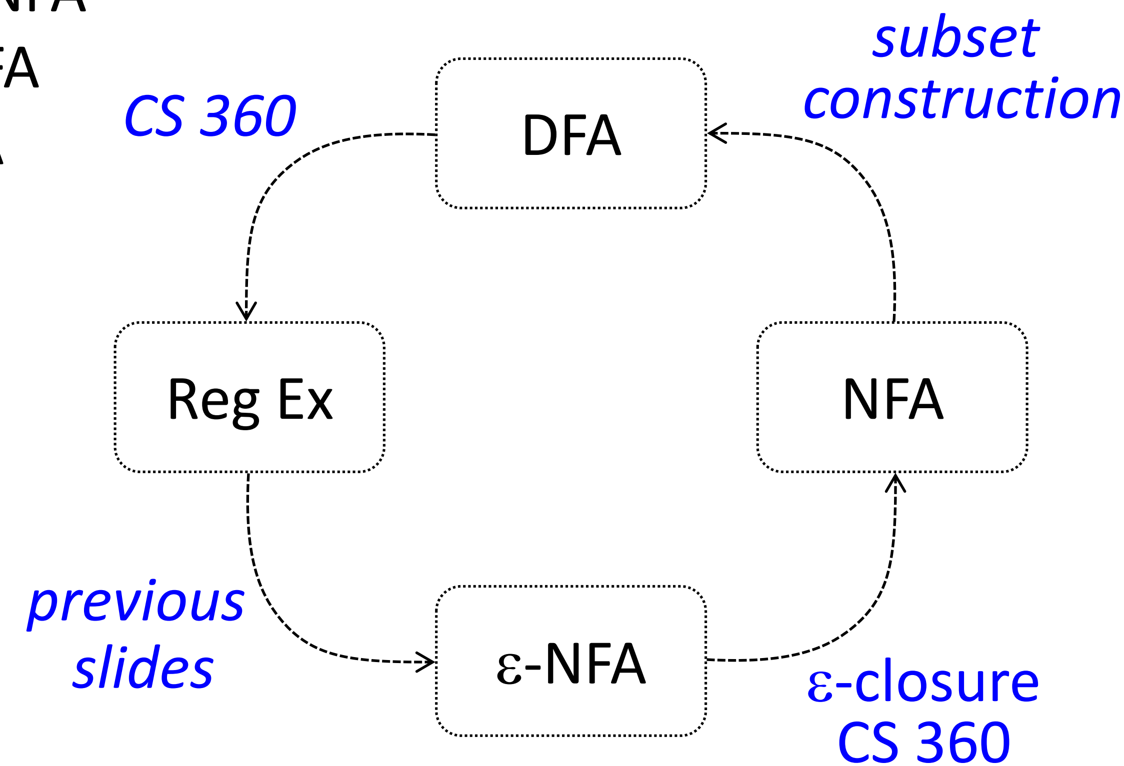
- Any sets the includes A (the accepting state in the NFA) will be an accepting state in the DFA.

Regular Languages

Equivalence

A *regular language* can be

- specified by a regular expression
- recognized by an ε-NFA
- recognized by an NFA
- recognized by a DFA



Topic 10 – Context-free Grammars I

Key Ideas

- limitations of Regular Languages
- Context-free Grammars (CFGs)
- terminals and non-terminals
- production rules and derivations
- formal definition of a context-free grammar
- left recursion and right recursion
- leftmost and rightmost derivations

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.1 to 3.4.

What is Next?

What is Missing from Regular Languages

- We now have the ability to recognize all the tokens in our programming language.
- Analogy: we can *recognize the individual words* (i.e. tokens), but we need to
  - *recognize valid sentences* (*i.e. sequences of tokens*): we'll call this step *parsing* or *syntactic analysis*
  - *recognize the meaning of sentences*: we'll do this later on

What is Next?

Recall: Basic Compilation Steps

The steps in translating a program from a high level language to an assembly language program are:

WLP4 text file

↓ 1. *scanning*: identify the tokens *Done*

WLP4 tokens

↓ 2. *syntactic analysis*: check order of tokens *Now*

parse tree

↓ 3. *semantic analysis*: create a symbol table and perform type checking *Later*

⋮

↓ 4. *code generation* *Later*

MIPS Assembly Language

What is Next?

Recall: Staging

- different stages check for different types of errors
- can improve error messages
- simplifies compiler code (more modular)
- *Syntax: verify the structure / format of the sequence of tokens*
  - Valid MIPS assembly language: add $1, $2, $3
  - Not valid MIPS assembly language: $1 add,, $2
  - Valid WLP4 / C++: int sum = 0;
  - Not valid WLP4 / C++: = sum ; 0 int
- *Semantics:* meaning
  - Does the function have the right number of arguments?
  - Does the function have the right type of arguments?
  - What is that variable's type?

Motivation for CFG's

Limitations of Regular Languages

- Goal: check if the syntax of a program is correct.
- *Key Problem: we need a more powerful tool than regular languages / DFAs / NFAs to check the syntax.*
- I.e. given $\Sigma = \{a, b\}$, it must have the ability to recognize the language $\mathcal{L} = \{w$: number of a's in w = the number of b's in w$\}$.
- E.g. in programming you must be able to recognize

balanced parentheses

(( ) (( )))

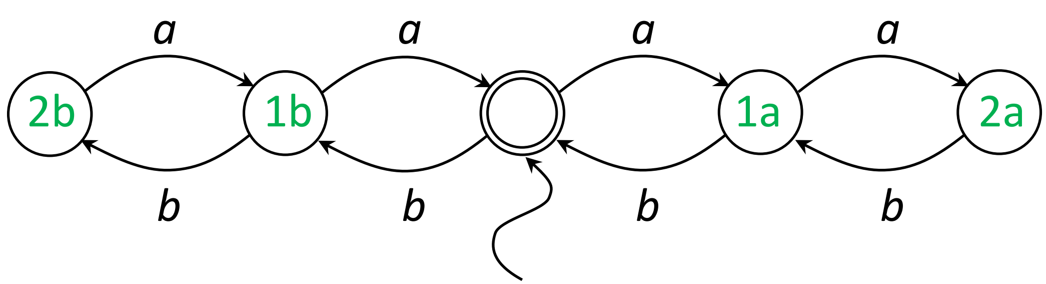
balanced braces

```
{
   {
   }
}
```

Motivation for CFG's

Limitations of Regular Languages

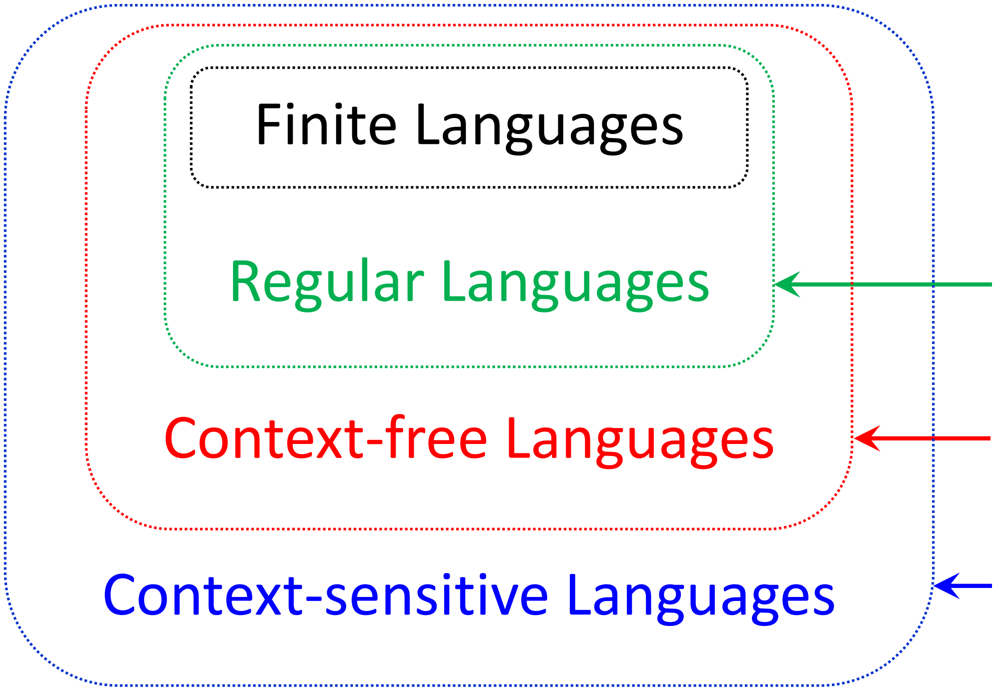
- Create a DFA that recognizes the language $\mathcal{L}$ = {*w*: number of *a*'s in *w* = the number of *b*'s in w} over alphabet Σ ={*a*, *b*}.
- Easy if the difference in the number of *a*'s and *b*'s is fixed, say 2.



- *Impossible if the potential difference is unbounded.*
- DFAs are good for tracking a finite number of things, e.g. strings with 3 *b*'s in a row.
- But the potential number of nested parentheses is unbounded.
- We need an unbounded stack to track if the number of left and right parentheses are equal.

The Compiler

Recall: Chomsky Hierarchy



Steps in Compiling

1. lexical analysis: find each token
2. syntactic analysis recognize with 1 stack
3. semantic analysis
4. code generation

- All Finite Languages are Regular Languages
- All Regular Languages are Context-free Languages

Example – Simple Sentence

Specifying a Valid Structure

English has rules that guide sentence structure

(1) <sentence> → <subj phrase> <verb>
(2) <subj phrase> → <article> <noun>
(3) <article> → the
(4) <noun> → dog
(5) <verb> → barks

These rules have two types of components

1. *terminals*:
   components that appear in the output e.g. the, dog, barks
2. *non-terminals / variables*:
   specify the format of the sentence
   components that do not appear in the output

Specification Components

Specifying a Valid Format

- production rules guide the expansion of a non-terminal into zero or more terminals, non-terminals, or both

Derivation of the sentence "The dog barks."

<sentence>

⇒ <subj phrase> <verb> (1)

⇒ <article> <noun> <verb> (2)

⇒ the <noun> <verb > (3)

⇒ the dog <verb> (4)

⇒ the dog barks (5)

- The derivation is similar to a formal proof in mathematics, i.e. justify each step with a rule.

Example CFG

Typical CS241 Example

G: (1) S $\rightarrow$ *aSb* // *aSb* is the *concatenation of a*, S, *b*
(2) S $\rightarrow$ D // 2 rules with S on the LHS is *union*
(3) D $\rightarrow$ *c*D // D on both sides of a rule is *recursion*
(4) D $\rightarrow$ ε

- Rules *always have* a *single non-terminal* on the left hand side.
- Rules *can have* a mixture of *terminals, non-terminals* or ε on the right hand side.
- The word *accb* is in the language generated by the grammar G, i.e. L (G), since we can *derive accb* from G.
- Notation: use '$\rightarrow$' for rules and '$\Rightarrow$' for derivations
- Derivation: $S \overset{}{\underset{1}{\Rightarrow}} aSb \underset{2}{\Rightarrow} aDb \underset{3}{\Rightarrow} acDb \underset{3}{\Rightarrow} accDb \underset{4}{\Rightarrow} accb$

Example CFG

Typical CS241 Example

G: (1) $S \rightarrow aSb$
(2) $S \rightarrow D$
(3) $D \rightarrow cD$
(4) $D \rightarrow \varepsilon$ Sometimes written as $D \rightarrow$

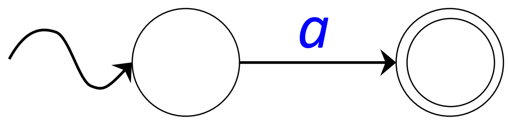
Derivation: $S \underset{1}{\Rightarrow} aSb \underset{2}{\Rightarrow} aDb \underset{3}{\Rightarrow} acDb \underset{3}{\Rightarrow} accDb \underset{4}{\Rightarrow} accb$

- Derivations apply a sequence of rules, i.e.
  - to get from $S \Rightarrow aSb$ replace S in LHS with aSb (using rule 1)
  - to get from $aSb \Rightarrow aDb$ replace S in LHS with D (using rule 2)
  - to get from $aDb \Rightarrow acDb$ replace D in LHS with cD (using rule 3)
  - to get from $acDb \Rightarrow accDb$ replace D in LHS with cD (using rule 3)
  - to get from $accDb \Rightarrow accb$ replace D in LHS with ε (using rule 4)

Example CFGs

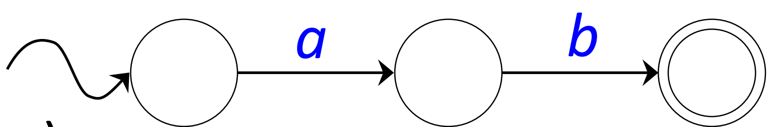
Regular Expressions vs. DFAs vs. Context-free Grammars

- *a*



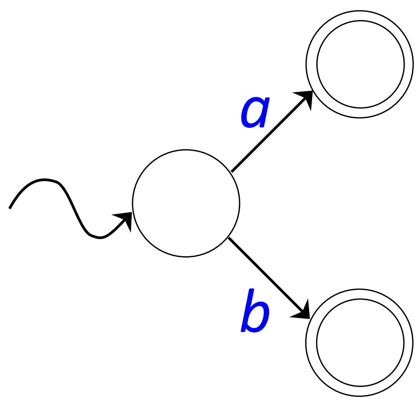
(1) $S \to a$

- *ab*
  (concatenation)



(1) $S \to ab$

- *a*|*b*
  (union)

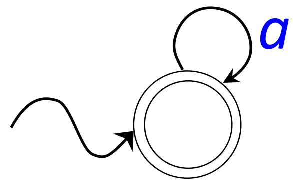


(1) $S \to a$
(2) $S \to b$
or as
(1) $S \to a \mid b$

Example CFGs

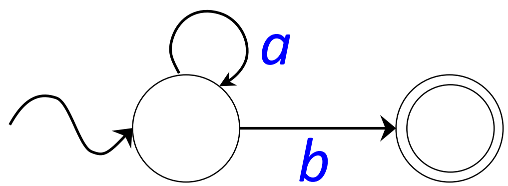
Regular Expressions, DFAs and Context-free Grammars

- a^*



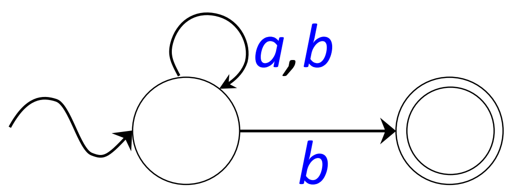
(1) $S \rightarrow Sa$
(2) $S \rightarrow \varepsilon$

- a^*b



(1) $S \rightarrow aS$
(2) $S \rightarrow b$

- $(a|b)^*b$



(1) $S \rightarrow aS$
(2) $S \rightarrow bS$
(3) $S \rightarrow b$

Derivation

How to Derive a String

- i.e. how to recognize if a string is part of the language
- apply production rules (one at a time) to generate a valid string
  - begin with the *start symbol*
  - repeatedly rewrite one *non-terminal* using one rule
  - continue until there are no more *non-terminals*
- the resulting sequence of *terminals* is a *syntactically correct* string

Informal Definition

- *language of a CFG*: the set of all valid strings (sequences of *terminals*) that can be derived from the *start symbol*

CFG Definitions

Informal Definitions

- G is a context-free grammar
- L (G) is the language (set of words) specified by G
- a *word*: a sequence of terminals that can be derived by applying the rules of the CFG
- a *derivation*: starting with the start symbol, applying a sequence of rules until there are no more non-terminals
- *Production Rules* (a.k.a. *Rewrite Rules*) capture
  - union
  - concatenation
  - recursion (which is strictly more powerful than repetition)

General Approach

Differences compared to Regular Languages

- *Context-free languages* are built from:
  - finite sets
  - concatenation
  - union
  - recursion

  (finite sets, concatenation, union) *same* as Regular Languages

  (recursion) *new* replaces repetition

- Recognizers for Regular Languages use
  1. a finite amount of memory

- Recognizers for Context-free Languages use
  1. a finite amount of memory
  2. one (unbounded) stack (you'll see where the stack gets used later on)

CFG Components

Informal Definition

Context-free grammars consist of a four-tuple {N, T, P, S}

- N is a finite set of non-terminals
  - they *never appear* at the end of the derivation
- T is a finite set of terminals
  - they *may appear* at the end of the derivation
- P is a finite set of production rules in the form $A \rightarrow \beta$ where
  - A is a non-terminal, i.e. $A \in N$
  - β is a repetition of terminals and non-terminals, i.e. $\beta \in (N \cup T)^*$
- S is the start symbol, $S \in N$
  - by convention it is on the LHS of the first rule.

CFG Components

Unpacking the Example

- N = {S, D}, i.e. the set of non-terminals
- T = {*a*, *b*, *c*} i.e. the set of terminals
- P = the set of production rules in the form A $\rightarrow$ β, e.g.
  - where the rules
    - have a single element of N on the LHS, i.e. A ϵ N
    - have elements of (N U T)* on the RHS, i.e. β ϵ (N U T)*

S	$\rightarrow$	*a*S*b*	where A is S and β is aSb
S	$\rightarrow$	D	A is S and β is D
D	$\rightarrow$	*c*D	A is D and β is *c*D
D	$\rightarrow$	ε	A is D and β is ε

- S is the start symbol, S ϵ N and by convention it is on the LHS of the first rule.

Example CFG

More Examples

G:	(1)	S	$\rightarrow$	*aSb*
	(2)	S	$\rightarrow$	D
	(3)	D	$\rightarrow$	*c*D
	(4)	D	$\rightarrow$	ε

Think of the non-terminal S as representing "generate *a*'s and *b*'s" and D as representing "generate *c*'s or disappear."

- derive: aaabbb

$$S \underset{1}{\Rightarrow} aSb \underset{1}{\Rightarrow} aaSbb \underset{1}{\Rightarrow} aaaSbbb \underset{2}{\Rightarrow} aaaDbbb \underset{4}{\Rightarrow} aaabbb$$

- derive: ccc

$$S \underset{2}{\Rightarrow} D \underset{3}{\Rightarrow} cD \underset{3}{\Rightarrow} ccD \underset{3}{\Rightarrow} cccD \underset{4}{\Rightarrow} ccc$$

Another Example CFG

Balanced Parentheses

- Task: Create a CFG that access accepts words with balanced parentheses
- Example words: ε, (), (()), ()(), (() ()), ...

(1) $S \rightarrow (S)$

(2) $S \rightarrow S\ S$

(3) $S \rightarrow \varepsilon$

Another Example CFG

Balanced Parentheses

- Derive (( )):

- Derive (( ) ( )):

Derivations

Grammar for Language on {*a*, *b*} that Contains at Least One *a*

- *Right-recursion*: a non-terminal is on both the LHS and the RHS of a rule and it is the *rightmost symbol* on the RHS.

G: (1) $S \rightarrow bS$
(2) $S \rightarrow aD$
(3) $D \rightarrow aD$
(4) $D \rightarrow bD$
(5) $D \rightarrow \varepsilon$

Think of the non-terminal S as representing "have not generated an *a* yet" and D as "have generated an *a*."

- derive *bbab* (hint: generate it from left to right)

$$S \underset{1}{\Rightarrow} bS \underset{1}{\Rightarrow} bbS \underset{2}{\Rightarrow} bbaD \underset{4}{\Rightarrow} bbabD \underset{5}{\Rightarrow} bbab$$

- derive *aaba* (hint: generate it from left to right)

$$S \underset{2}{\Rightarrow} aD \underset{3}{\Rightarrow} aaD \underset{4}{\Rightarrow} aabD \underset{3}{\Rightarrow} aabaD \underset{5}{\Rightarrow} aaba$$

Derivations

Grammar for Language on {*a*, *b*} that Contains at Least One *a*

- *Left-recursion*: a non-terminal is on both the LHS and the RHS of a rule and it is the *leftmost symbol* on the RHS.

G:	(1)	S	$\rightarrow$	S*b*
	(2)	S	$\rightarrow$	D*a*
	(3)	D	$\rightarrow$	D*a*
	(4)	D	$\rightarrow$	D*b*
	(5)	D	$\rightarrow$	ε

Think of the non-terminal S as representing "have not generated an *a* yet" and D as "have generated an *a*."

- derive *bbab* (hint: generate it from right to left)

$$S \underset{1}{\Rightarrow} Sb \underset{2}{\Rightarrow} Dab \underset{4}{\Rightarrow} Dbab \underset{4}{\Rightarrow} Dbbab \underset{5}{\Rightarrow} bbab$$

- derive *aaba* (hint: generate it from right to left)

$$S \underset{2}{\Rightarrow} Da \underset{4}{\Rightarrow} Dba \underset{3}{\Rightarrow} Daba \underset{3}{\Rightarrow} Daaba \underset{5}{\Rightarrow} aaba$$

Derivations

Grammar for Language on {*a*, *b*} that Contains an Even # of *a's*

G: (1) $S \rightarrow bS$
(2) $S \rightarrow Sb$
(3) $S \rightarrow aSa$
(4) $S \rightarrow \varepsilon$

The *a*'s are generated in pairs, from the centre outwards.

- derive *baa*: $S \Rightarrow bS \Rightarrow baSa \Rightarrow baa$
- derive *aab*: $S \Rightarrow Sb \Rightarrow aSab \Rightarrow aab$
- derive *babaaba*:
  hint: since *a*'s are generated in pairs start at the outside and work your way towards the middle of the *a*'s
  $S \Rightarrow bS \Rightarrow baSa \Rightarrow babSa \Rightarrow babSba \Rightarrow babaSaba \Rightarrow babaaba$

Derivations

Grammar for Language on {*a*, *b*} that Contains an Even # of *a's*

G: (1) $S \rightarrow bS$
(2) $S \rightarrow Sb$
(3) $S \rightarrow aSa$
(4) $S \rightarrow \varepsilon$

The *a*'s are generated in pairs, from the centre outwards.

- The string *aba* has *two different* derivations

1. $S \underset{3}{\Rightarrow} aSa \underset{1}{\Rightarrow} abSa \underset{4}{\Rightarrow} aba$

2. $S \underset{3}{\Rightarrow} aSa \underset{2}{\Rightarrow} aSba \underset{4}{\Rightarrow} aba$

- When a grammar has two different derivations for the same string the grammar is called *ambiguous.* More on this later.

Another Example CFG

Binary Numbers

- In this language, the words are binary numbers with no leading 0's (other than 0)

1. $B \to 0$
2. $B \to D$
3. $D \to 1$
4. $D \to D0$
5. $D \to D1$

Here

- the non-terminal B means generate a 0 or D
- the non-terminal D means generate a number with a leading 1

Note: the grammar is left-recursive (rules 4 and 5) so it will generate the bits from right to left.

Another Example CFG

Binary Numbers

- Derive: 0
- Derive: 1
- Derive: 10
- Derive: 101

1. $B \rightarrow 0$
2. $B \rightarrow D$
3. $D \rightarrow 1$
4. $D \rightarrow D0$
5. $D \rightarrow D1$

Another Example CFG

Binary Expressions

- In this language the words are binary numbers with no leading 0's (other than 0) and with + or - operators using infix notation (between numbers, not before them).

1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$

Here

- E means arithmetic expression
- B means generate a 0 or D
- D means generate a number with a leading 1

Another Example CFG

Binary Expressions

- Derive: 10+1 using a *leftmost derivation* (i.e. always expand the leftmost non-terminal first).

- $E \overset{1}{\Rightarrow} E + E \overset{3}{\Rightarrow} B + E \overset{5}{\Rightarrow} D + E \overset{7}{\Rightarrow} D0 + E \overset{6}{\Rightarrow} 10 + E \overset{3}{\Rightarrow}$
  $10 + B \overset{5}{\Rightarrow} 10 + D \overset{6}{\Rightarrow} 10 + 1$

- Derive: 10+1 using a *rightmost derivation* (i.e. always expand the rightmost non-terminal first).

- $E \overset{1}{\Rightarrow} E + E \overset{3}{\Rightarrow} E + B \overset{5}{\Rightarrow} E + D \overset{6}{\Rightarrow} E + 1 \overset{3}{\Rightarrow}$
  $B + 1 \overset{5}{\Rightarrow} D + 1 \overset{7}{\Rightarrow} D0 + 1 \overset{6}{\Rightarrow} 10 + 1$

Topic 11 – Context-free Grammars II

Key Ideas

- parse trees
- ambiguous grammars
- left recursion and right recursion
- implementing associativity and precedence
- formal definitions of derives and directly derives

References

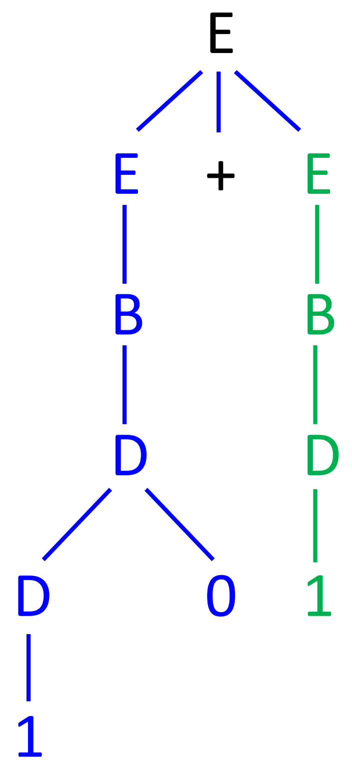
- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.1 to 3.4.

Parse Trees

$E \Rightarrow^* 10 + 1$ Parse Tree

The derivation

$E \Rightarrow E + E$	using rule $E \rightarrow E + E$
$\Rightarrow B + E$	using rule $E \rightarrow B$
$\Rightarrow D + E$	using rule $B \rightarrow D$
$\Rightarrow D0 + E$	using rule $D \rightarrow D0$
$\Rightarrow 10 + E$	using rule $D \rightarrow 1$
$\Rightarrow 10 + B$	using rule $E \rightarrow B$
$\Rightarrow 10 + D$	using rule $B \rightarrow D$
$\Rightarrow 10 + 1$	using rule $D \rightarrow 1$



can be represented

as a *parse tree.*

Parse Trees

Creating a Parse Tree

- also called derivation trees
- visualize the entire derivation at once
- the *root of the tree* is the start symbol: E
- *internal nodes* are the non-terminals: E, B, D
- the *children* of each internal node are given by a production rule
- the *leaf nodes* are the terminals
- the terminals occur in the tree in the same order as they occur in the input, i.e. 1, 0, +, 1
- parse trees (among other things) help visualize *ambiguous grammars*…

Ambiguous Grammars

Grammars

- Statements in English can be *ambiguous*.
- E.g. Chris was given a book by J. K. Rowlings.
  - Does *by* refer to *a book*?
    - i.e. The book was by J. K. Rowlings.
  - Does *by* refer to *was given*?
    - i.e. The book was given by J. K. Rowlings.
- Grammars for computer languages are at risk of being ambiguous: e.g. 1 - 10 + 11
- Does the grammar interpret the statement as (1 – 10) + 11 or 1 – (10 + 11) or both?

Ambiguous Grammars

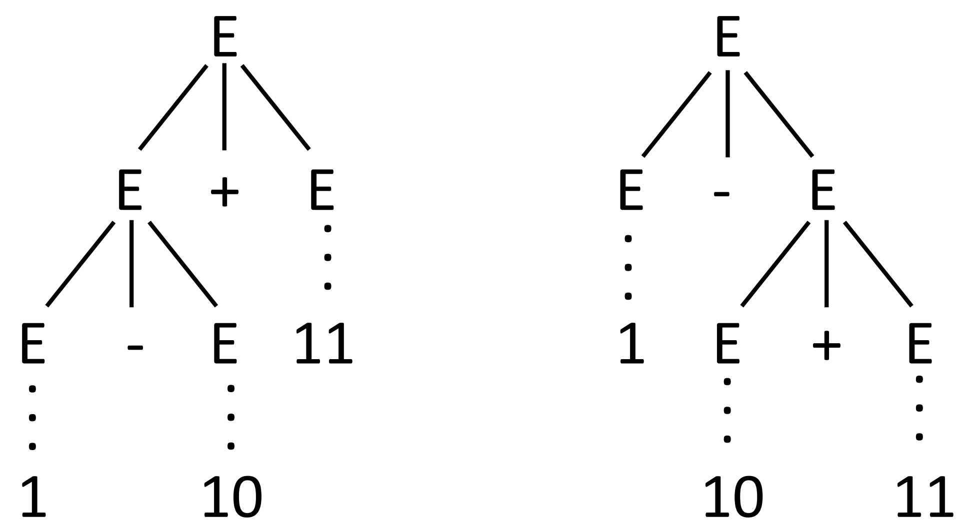
Parse Trees for $E \Rightarrow^* 1 - 10 + 11$

- The same string can have two different parse trees.
- If a grammar can generate at least one string that has two different parse trees, then the grammar is *ambiguous.*

R1 $E \rightarrow E + E$

R2 $E \rightarrow E - E$

- You can use
  a) R1 then R2 or
  b) R2 then R1
  to generate
  $E - E + E$
  which derives
  $1 - 10 + 11$



a) R1 then R2 b) R2 then R1

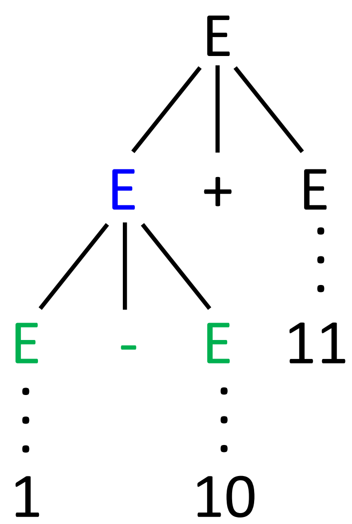
Ambiguous Grammars

Parse Trees for E ⇒* 1 - 10 + 11

- You may also have *two or more leftmost derivations* (or rightmost derivations) for the same string

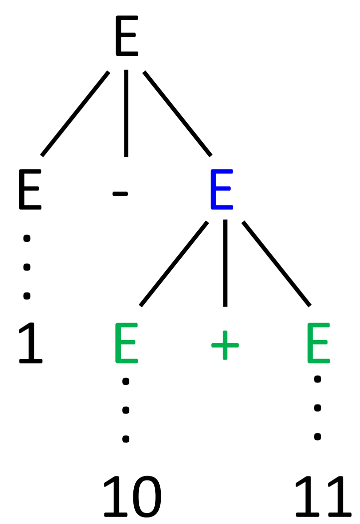
E ⇒ E+E ⇒ E-E+E ⇒ B-E+E ⇒ D-E+E ⇒ 1-E+E ⇒ …

yields this parse tree



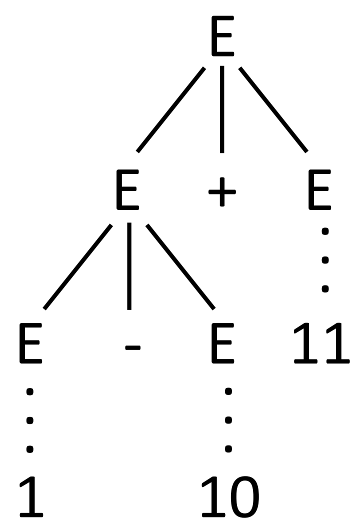
E ⇒ E-E ⇒ B-E ⇒ D-E ⇒ 1-E ⇒ 1-E+E ⇒ 1-B+E ⇒ 1-D+E …

yields this parse tree



Processing Order in a Parse Tree

Implications of Ambiguity



In order to understand how different parse trees relate to ambiguity (and other issues such as associativity and precedence) you must understand *how parse trees are processed for arithmetic expressions.*

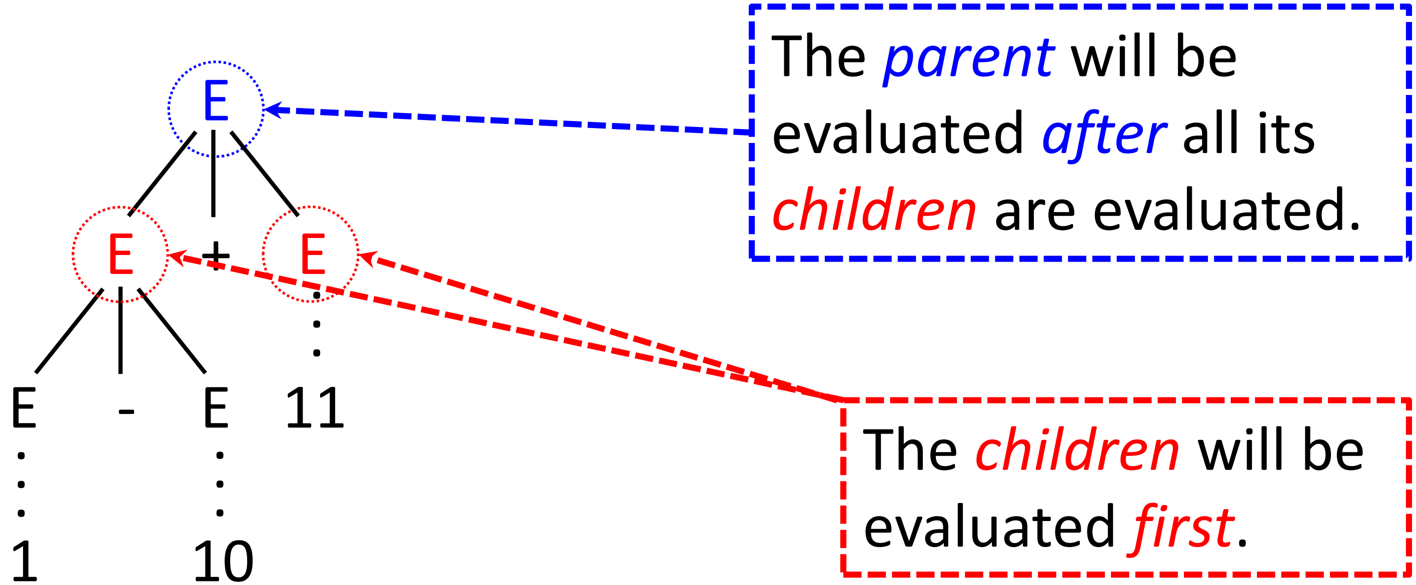
Parse trees are processed using a *post-order depth first* traversal for arithmetic expressions.

depth first – visit your first child and all its descendants before visiting your second child.

post-order – a type of depth first traversal where you process all your children before processing yourself.

Processing Order in a Parse Tree

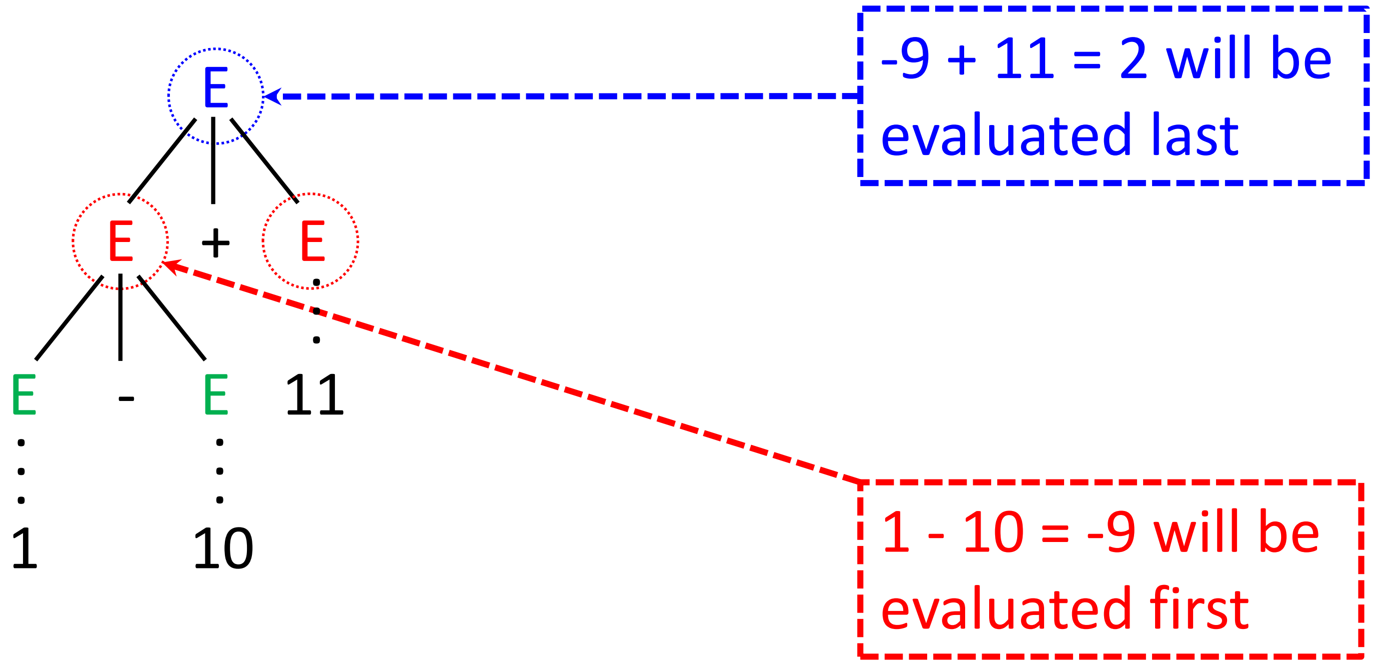
Properties of a Post-Order Traversal



- Post-Order Traversal: children will be evaluated before self.

Processing Order in a Parse Tree

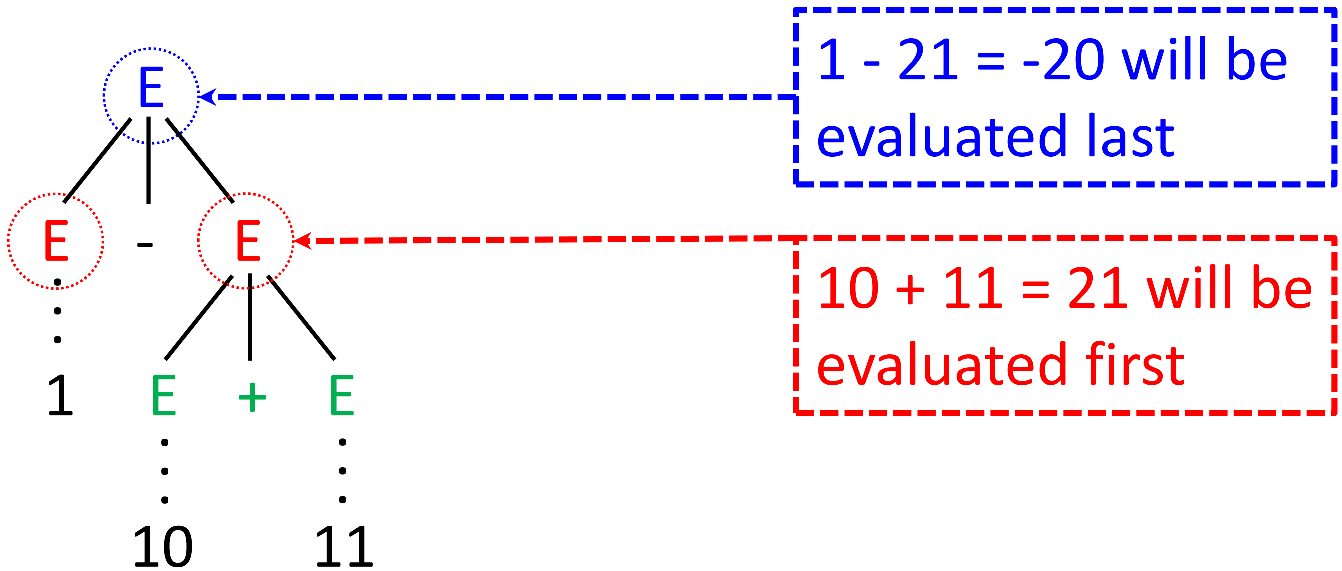
Properties of a Post-Order Traversal



- E ⇒ E + E ⇒ E - E + E
- Post-Order Traversal: children will be evaluated before self.
- For this tree "1 - 10 + 11" is evaluated as (1 - 10) + 11 = 2.

Processing Order in a Parse Tree

Properties of a Post-Order Traversal



- E ⇒ E - E ⇒ E - E + E
- Post-Order Traversal: children will be evaluated before self.
- For this tree “1 - 10 + 11” is evaluated as 1 - (10 + 11) = -20.

Ambiguous Grammars

Formal Definition

- *A string* *w* in a grammar is *ambiguous* if there is more than one parse tree for *w*.
- E.g. in our current grammar the string "1 - 10 + 11" is ambiguous.
- *A context-free grammar* G is *ambiguous* if there exists at least one string *w* such that $w \in \mathcal{L}(G)$ and *w* is ambiguous.
- E.g. the grammar that generated the string "1 - 10 + 11" is ambiguous.
- Because the string "1 - 10 + 11"is ambiguous in this grammar, it may be evaluated as
  a) (1 - 10) + 11 = 2
  b) 1 - (10 + 11) = -20

Ambiguous Grammars

Ambiguity

- *An ambiguous grammar means there is no unique derivation and hence no unique meaning* (for at least one string).
- When is a CFG ambiguous?
  - it is undecidable (like the Halting Problem)
  - certain ambiguities can be spotted
  - e.g. the same non-terminals in the RHS of a rule, as seen is rules 1 and 2 below:

    1. $E \rightarrow E + E$
    2. $E \rightarrow E - E$

- i.e. either the operator '+' or '-' can be generated first
- mixing left recursion and right recursion can cause ambiguity

Unambiguous Grammars

Binary Expressions

- Change the first two productions

1. $E \rightarrow \cancel{E + E}\ \ B + E$
2. $E \rightarrow \cancel{E - E}\ \ B - E$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$

- This change *makes addition and subtract operations right recursive* and forces the leftmost non-terminal to derive a binary number rather than another expression.
- The grammar generates the same words as the previous grammar but the parse tree for each derivation is unique.

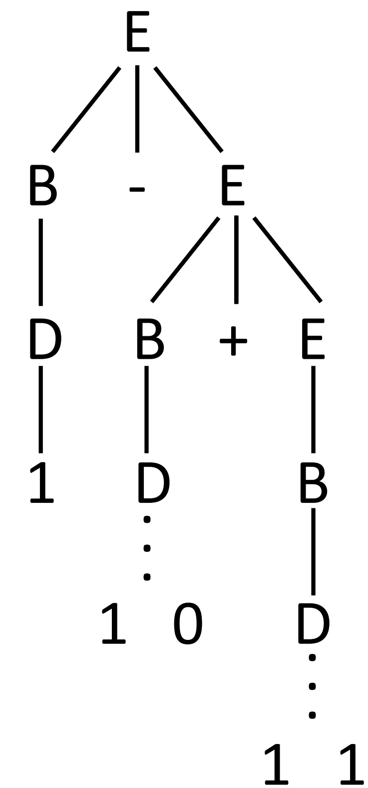
Unambiguous Grammars

Binary Expressions

- Change the first two productions

1. $E \rightarrow B + E$	5. $B \rightarrow D$
2. $E \rightarrow B - E$	6. $D \rightarrow 1$
3. $E \rightarrow B$	7. $D \rightarrow D0$
4. $B \rightarrow 0$	8. $D \rightarrow D1$

- The expression grows by adding more expressions (i.e. operators and digits) on the right hand side.
- *Since addition and subtraction right recursive,* the right side of the expression will be a child of the root and will be evaluated before the parent.



Associativity and Precedence

Dealing with Associativity and Precedence

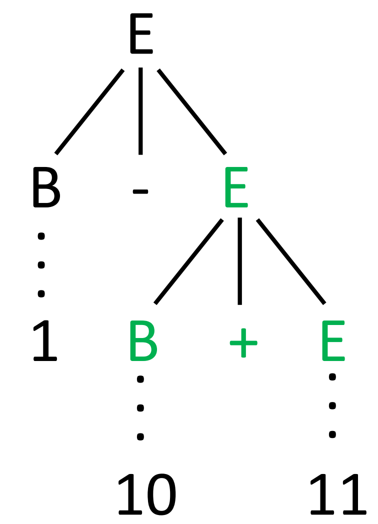
- CFGs can generate *balanced parentheses and implicit order of evaluating expressions* in the absence of parentheses.
- *associativity*: grouping equivalent operations
  - example: 6 - 3 + 4
  - is it read as (6 - 3) + 4 or 6 - (3 + 4)?
  - we want left associativity, i.e. evaluate from left to right (i.e. *have the left side farther from the root*)
- *precedence*: grouping non-equivalent symbols
  - example: 6 + 3 * 4
  - is it read as (6 + 3) * 4 or 6 + (3 * 4)?
  - we want multiplication to have precedence over addition (i.e. *have multiplication occur further from the root than addition*)

Associativity

Associativity of Expressions

- Recall this grammar.

1. $E \rightarrow B + E$	5. $B \rightarrow D$
2. $E \rightarrow B - E$	6. $D \rightarrow 1$
3. $E \rightarrow B$	7. $D \rightarrow D0$
4. $B \rightarrow 0$	8. $D \rightarrow D1$



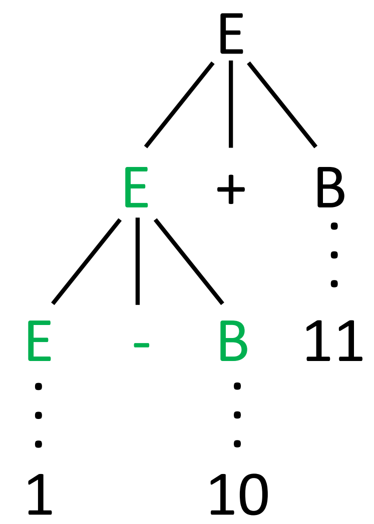
- Consider the tree corresponding to $E \Rightarrow B - E \Rightarrow B - B + E$
- The expression gets longer by adding more operators and digits (i.e. expressions) on the *right* hand side.
- Since *the children get evaluated before the parent*, 10 + 11 will be evaluated before 1 - ( )
- These rules enforce associativity from the *right*, i.e. 1 - (10 + 11)

Associativity

Associativity of Expressions

- Swap the order of E and B on the RHS of 1, 2.

1. $E \rightarrow E + B$
2. $E \rightarrow E - B$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$



- Consider the tree corresponding to $E \Rightarrow E + B \Rightarrow E - B + B$
- The expression gets longer by adding more operators and digits (i.e. expressions) on the *left* hand side.
- Since *the children get evaluated before the parent,* 1 - 10 will be evaluated before ( ) + 11
- These rules enforce associativity from the *left*, i.e. (1 - 10) + 11

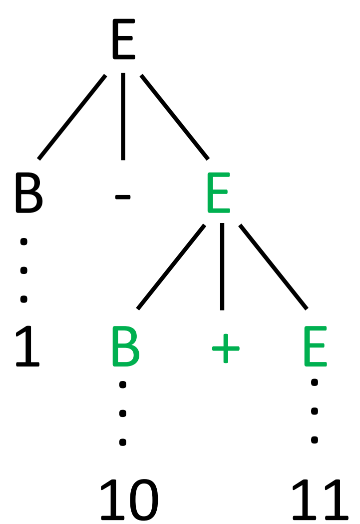
Associativity

When our grammar is *right recursive,* i.e.

1. $E \rightarrow B + E$
2. $E \rightarrow B - E$

our grammar becomes *right associative*, i.e.

$E \Rightarrow B - E \Rightarrow B - (B + E)$

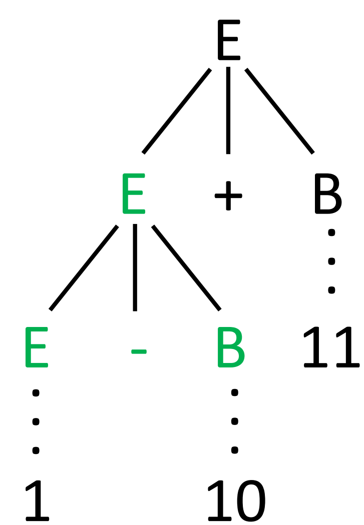


When our grammar is *left recursive,* i.e.

1. $E \rightarrow E + B$
2. $E \rightarrow E - B$

our grammar becomes *left associative,* i.e.

$E \Rightarrow E + B \Rightarrow (E - B) + B$

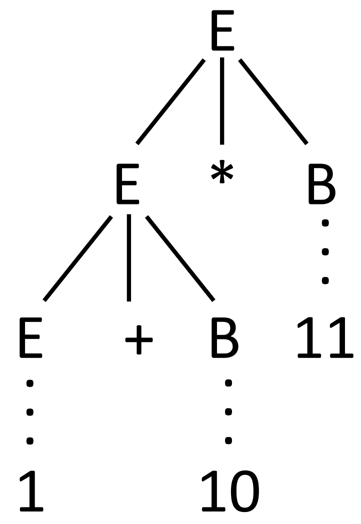


Precedence

Binary Expressions

- Now include multiplication and division.

1. $E \rightarrow E + B$	6. $B \rightarrow 0$
2. $E \rightarrow E - B$	7. $B \rightarrow D$
3. $E \rightarrow E * B$	8. $D \rightarrow 1$
4. $E \rightarrow E / B$	9. $D \rightarrow D0$
5. $E \rightarrow B$	10. $D \rightarrow D1$



- Consider the derivation $E \Rightarrow E * B \Rightarrow E + B * B$
- This grammar will evaluate the expression 1+10*11 as (1+10)*11 which *ignores the standard rules of precedence.*
- *Idea: have multiplication occur with children of E* (rather than with E itself) by creating a new non-terminal *T*.

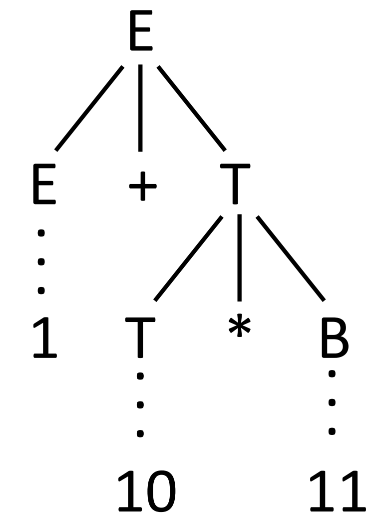
Precedence

Binary Expressions

- Introduce a new non-terminal T

1. $E \rightarrow E + T$
2. $E \rightarrow E - T$
3. $T \rightarrow T * B$
4. $T \rightarrow T / B$
5. $E \rightarrow T$
6. $T \rightarrow B$

6. $B \rightarrow 0$
7. $B \rightarrow D$
8. $D \rightarrow 1$
9. $D \rightarrow D0$
10. $D \rightarrow D1$



- Consider the derivation $E \Rightarrow E + T \Rightarrow E + T * B$
- This grammar will evaluate the expression 1 + 10 * 11 as 1 + (10 * 11)
- *Whenever the non-terminal T occurs, it will always be a child of E and will be evaluated before its parent.*

CFGs and Derivations

Formal Definitions

- Recall this simple grammar

1. $E \rightarrow E + E$	3. $E \rightarrow B$	5. $B \rightarrow D$	7. $D \rightarrow D0$
2. $E \rightarrow E - E$	4. $B \rightarrow 0$	6. $D \rightarrow 1$	8. $D \rightarrow D1$

- So far we've described *specific steps* in a derivation, such as $E - B + B \Rightarrow E - D + B$ using the rule $B \rightarrow D$.
- Now we want to refer to a *general step* in an arbitrary derivation, such as $\alpha A \beta \Rightarrow \alpha\gamma\beta$ using the rule $A \rightarrow \gamma$.
- So we introduce symbols α and β to refer to the symbols before and after the A (and γ) as a way of saying these parts do not change when the A gets rewritten as γ.
- These Greek letters can refer to ε, terminals (such as '+') non-terminals (such as 'E') or some combination (such as 'E-B+').

CFGs and Derivations

Formal Definition: Directly Derives

- $\alpha A\beta$ *directly derives* $\alpha\gamma\beta$ (written as $\alpha A\beta \Rightarrow \alpha\gamma\beta$) if there *is a production rule* $A \to \gamma$ where
  - $A \in N$ (i.e. *A* is a non-terminal) and
  - $\alpha, \beta, \gamma \in (N \cup T)^*$ (i.e. non-terminals, terminals, empty string)
- e.g. E-B+B $\Rightarrow$ E-D+B using the rule $B \to D$ because if we set 'E-'=α, 'B'=A, '+B'=β, and 'D'=γ then that step is in the format $\alpha A\beta \Rightarrow \alpha\gamma\beta$ using the rule $A \to \gamma$
- i.e. it doesn't matter what α and β are, as long as there is a production rule $A \to \gamma$, then $\alpha A\beta$ directly derives $\alpha\gamma\beta$
- *Informally*, *directly derives* means it takes one derivation step or one application of a production rule.

CFGs and Derivations

Formal Definition: Derives

- $\alpha A\beta$ *derives* $\alpha\gamma\beta$ (written as $\alpha A\beta \Rightarrow^* \alpha\gamma\beta$) if there *is a finite sequence of productions* $\alpha A\beta \Rightarrow \alpha\Theta_1\beta \Rightarrow \alpha\Theta_2\beta \Rightarrow \ldots \Rightarrow \alpha\gamma\beta$
  - again $A \in N$ and $\alpha, \beta, \gamma, \Theta_i \in (N \cup T)^*$

- e.g. with $E \underset{(1)}{\Rightarrow} E + E \underset{(3)}{\Rightarrow} B + E \underset{(5)}{\Rightarrow} D + E \underset{(7)}{\Rightarrow} D0 + E \underset{(6)}{\Rightarrow}$

  $$10 + E \underset{(3)}{\Rightarrow} 10 + B \underset{(5)}{\Rightarrow} 10 + D \underset{(6)}{\Rightarrow} 10 + 1$$

  - $E \Rightarrow^* D0 + E$ w/ productions: 1, 3, 5, 7
  - $E \Rightarrow^* 10 + 1$ w/ productions: 1, 3, 5, 7, 6, 3, 5, 6

- *Informally, derives* means it takes 0 or more derivation steps.

CFGs and Derivations

Formal Definition: Derives the Word

- The grammar G *derives the word* $w \in T^*$ if $S \Rightarrow^* w$
  - w is a concatenation of terminals (i.e. no non-terminals)
  - S is the start symbol
- *Informally*, the grammar G *derives a word* w if you can derive w from the start symbol.
  - e.g. E $\Rightarrow *$ 10 + 1 w/ productions: 1, 3, 5, 7, 6, 3, 5, 6
- The *language* $\mathcal{L}$ *(G)* = $\{w \in T^*$ i: $S \Rightarrow^* w\}$.
- *Informally,* the *language described by the grammar G* is the set of concatenations of terminal symbols that can be derived from the start symbol.
- Given a CFG G and word w, you can think of $S \Rightarrow^* w$ as a proof that w is in the language $\mathcal{L}(G)$.

CFGs and Derivations

Formal Definition: Context-free

- A *language L is context-free* if there exists a context-free grammar *G, such that* $\mathcal{L}(G) = L$.
- *Informally,* a set of strings is context-free if there is some context free grammar that describes the language.

- Given $uAvC\gamma$ where
  - $u, v \in T^*$ i.e. a finite number of terminals
  - $A, C \in N$ i.e. a single non-terminal
  - $\gamma \in (N \cup T)^*$ i.e. a mixture of both
  - then a *leftmost derivation* must rewrite *A*.
- *Informally*, rewrite the leftmost non-terminal first.

Topic 12 – Top-Down Parsing

Key Ideas

- Parsing
- Top-down and bottom-up parsing
- LL(1) Parsing
- Creating a Predict Table
- Helper Functions: First(), Follow(), Nullable()
- Limitations of LL(1) Parsing

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.7 to 3.10, 3.12

Parsing

What is Parsing

- *Parsing:* Given a grammar *G* and a word *w*, *derive* *w* using the grammar G.
- Analogous to Regular Expressions (which are used to *specify* tokens) and DFAs and Simplified Maximal Munch (which are used to *recognize* tokens)
- Here we use CFGs to *specify* a grammar and parsing algorithms *derive* the program.
- There are algorithms (which you *do not* have to know about) that work for any CFG once it is put in a particular form
  - e.g. the CYK algorithm, which runs in O(n^3 |G|) where n is the size of the input and |G| is the size of the grammar.

Parsing Algorithms

General Approaches.

- We will look at two *linear-time* approaches:
  1. *Top-down:* Find a non-terminal (e.g. S) and replace it with the right-hand side (e.g. for rule S $\rightarrow$ A*y*B replace S with A*y*B), e.g. LL(1)
  2. *Bottom-up:* replace a right-hand side (e.g. *ab*) with a non-terminal: (e.g. for rule A $\rightarrow$ *ab* replace *ab* with A), e.g. LR(0) and SLR(1).
- These algorithms don't work for all CFGs, so when we create a grammar for a programming language we must check that it can be parsed by one of these linear-time algorithms
- In both of these strategies, we have to decide which rule to apply next at each step of the derivation.

Stack-based Parsing

Using a Stack

- For top-down parsing, we use a stack to remember information about our derivations or processed input.
- Recall that CFGs are recognized by a DFA with a stack
- e.g. for language of paired parentheses
  - if input is ‘(‘, push it on the stack
  - if input is ‘)’ pop the stack
  - if you *pop when the stack is empty*: ERROR
  - if the *stack is not empty when you are finished* processing the input: ERROR
- e.g (( ) ( ))
- because we want to detect the end of our input we need to *augment* our grammar …

Augmenting Grammars

New Symbols

- We augment our grammars by adding three unique characters
  - a *new start symbol* S' that only appears in one rule
  - *the beginning* of the input: ⊢ (also called *BOF*)
  - *the end* of the input: ⊣ (also called *EOF*)
- Formally, augmenting the grammar (N, T, P, S) yields $\{N \cup \{S'\}, T \cup \{\vdash, \dashv\}, P \cup \{S' \to \vdash S \dashv\}, S'\}$

1. $S' \to \vdash S \dashv$
2. $S \to AyB$
3. $A \to ab$
4. $A \to cd$
5. $B \to z$
6. $B \to wz$

Example: Leftmost derivation of the word ⊢ abywz ⊣

$$\begin{aligned} S' &\Rightarrow \vdash S \dashv && \text{rule (1)} \\ &\Rightarrow \vdash AyB \dashv && \text{rule (2)} \\ &\Rightarrow \vdash abyB \dashv && \text{rule (3)} \\ &\Rightarrow \vdash abywz \dashv && \text{rule (6)} \end{aligned}$$

Top-Down Parsing

Definition of an Augmented Grammars

- the start symbol occurs as the LHS of exactly one rule
- that rule must begin and end with a terminal

Parsing Algorithm: Two Actions

- to start, push the start symbol, S’, on the stack
- when a *non-terminal* is at the top of the stack:
  - *expand* the non-terminal using a production rule where the RHS of the rule matches the input (e.g. if the rule is $S' \rightarrow \vdash S \dashv$ then pop S’ off the stack and push $\vdash$ S $\dashv$ onto the stack)
- when it is a *terminal* at the top of the stack: *match* with input
  - pop the terminal off of the stack
  - read the next character from the input

Top-Down Parsing

Parsing the Input

- To start, push S’ on the stack

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	

- When it is a *non-terminal* at the top of stack: *expand* the non-terminal (using a production rule) so that the new top of the stack matches the first symbol of the input.
  - in this case use rule 1 (S’ → ⊢ S ⊣) because the first symbol of the input matches the RHS of rule 1 (they are both ‘⊢’)

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	expand (1)
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	

Top-Down Parsing

Parsing the Input

- Since the top of the stack matches the first char of the input, pop ⊢ off the stack and read the next char of input

	Derivation	Read	Input	Stack	Action
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	

- The top of the stack in a non-terminal so expand it using rule 2 (S $\rightarrow$ AyB). There is only one choice of rule to use.

	Derivation	Read	Input	Stack	Action
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	expand (2)
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	

Top-Down Parsing

Parsing the Input

- The top of the stack in a non-terminal so expand it.
- There are two possible rules to use: 3 (A $\rightarrow$ ab) and 4 (A $\rightarrow$ cd) but only the RHS of rule 3 matches the input a.

	Derivation	Read	Input	Stack	Action
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	expand (3)
5	⊢ AyB ⊣	⊢	**a**bywz ⊣	> **a** b y B ⊣	match

- Read from input and pop the next three chars, which match.

	Derivation	Read	Input	Stack	Action
6	⊢ abyB ⊣	⊢ a	**b**ywz ⊣	> **b** y B ⊣	match
7	⊢ abyB ⊣	⊢ ab	**y**wz ⊣	> **y** B ⊣	match
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	

Top-Down Parsing

Parsing the Input

- Again, the top of the stack is a non-terminal so expand it.
- There are two possibilities: 5 B $\rightarrow$ z or 6 B $\rightarrow$ wz, but only the RHS of rule 6 matches the current input w.

	Derivation	Read	Input	Stack	Action
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	expand (6)
9	⊢ abywz ⊣	⊢ aby	wz ⊣	> **w** z ⊣	

- Pop off the stack and read the next two chars, which match.

	Derivation	Read	Input	Stack	Action
9	⊢ abywz ⊣	⊢ aby	wz ⊣	> **w** z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	z ⊣	> **z** ⊣	match
11	⊢ abywz ⊣	⊢ abywz	⊣	> **⊣**	

Top-Down Parsing

Parsing the Input

- The last character in the input matches the last character on the stack, pop it off the stack and accept the string.

	Derivation	Read	Input	Stack	Action
11	⊢ abywz	⊢ abywz	⊣	> ⊣	match
12	⊢ abywz ⊣	⊢ abywz⊣		>	ACCEPT

- The next slide shows the complete parsing of abywz using the grammar:

1. S’ $\rightarrow$ ⊢ S ⊣
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd
5. B $\rightarrow$ z
6. B $\rightarrow$ wz

Top-Down Parsing

Parsing the Input

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	expand (1)
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> S ⊣	expand (2)
4	⊢ AyB ⊣	⊢	abywz ⊣	> A y B ⊣	expand (3)
5	⊢ abyB ⊣	⊢	abywz ⊣	> a b y B ⊣	match
6	⊢ abyB ⊣	⊢ a	bywz ⊣	> b y B ⊣	match
7	⊢ abyB ⊣	⊢ ab	ywz ⊣	> y B ⊣	match
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> B ⊣	expand (6)
9	⊢ abywz ⊣	⊢ aby	wz ⊣	> w z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	z ⊣	> z ⊣	match
11	⊢ abywz ⊣	⊢ abywz	⊣	> ⊣	ACCEPT

Top-Down Parsing

Different Formats for Tables

- You may see two different formats for the tables having to do with the location of the action column.
- Here the action, expand (1), states which action was taken to *get to the next line*.

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	expand (1)
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	

- Here the action, expand (1), states which action was taken to get to the *current line.*

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	expand (1)

Top-Down Parsing

Top-down parsing with a stack

- *invariant* (i.e. true throughout the entire process)
  derivation = input already read + stack (read top-down) , e.g.
  - Line 1: S’
  - Line 2: ⊢ S ⊣
  - Line 4: ⊢ AyB ⊣
  - Line 6: ⊢ a byB ⊣
  - Line 9: ⊢ aby wz ⊣
  - Derivation: S’ $^1\Rightarrow$ ⊢ S ⊣ $^2\Rightarrow$ ⊢AyB⊣ $^3\Rightarrow$ ⊢abyB⊣ $^6\Rightarrow$ ⊢abywz⊣
- How do we know when we are done?
  - both stack and input contain ⊣
- How do we know which rule to use?
  Our Goal: to be able to correctly predict which rule applies!

LL(1) Parsing

Meaning of LL(1)

- first 'L' means process the input from Left to right
- second 'L' means find a Leftmost derivation
- 1 means the algorithm is allowed to look ahead 1 token

Goal: Unambiguous Prediction

- Find what rule applies if N (a non-terminal) is on the stack and c (a terminal) is the next symbol in the input to be read
- Implement Predict(N, c) as a table.
- For LL(1) grammars
  - for all non-terminals N and all terminals c: $|\text{Predict}(N, c)| \le 1$
  - i.e. given an N on the top of the stack and an c as the next input character at most one rule can apply.

Constructing a Predict Table

Approach

- Question: How do we implement Predict(N, c)?
- Recall our two actions for top-down parsing
  - to *match*: pop a terminal off the stack and get the next char from input: so we don't need to make a choice
  - to *expand*: we need to know which rule to choose
- In order to implement Predict(N, c) we use three helper functions
  1. *First*( )
  2. *Follow*( )
  3. *Nullable*( ) or *Empty*( )
- Naturally we will look at First() first!
- We will use First() to fill our Predict Table.

Constructing a Predict Table

Using First( ) to Construct the Predict Table

- *Informally:* For each non-terminal N, First(N) is the set of terminals that can begin a string derived from N; that is $N \Rightarrow^* c \cdots$

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$

	a	b	c	d	y	w	z	⊢	⊣
S'								1	
S	2		2						
A	3		4						
B						6	5		

- Using the table: First (S') = {⊢} by rule 1 so the entry at (S', ⊢) is 1.
  - i.e. if S' is on the stack and the input is ⊢, expand using rule 1.
- Empty cells are error states.
- Hmm, reminds me of a DFA table.

Constructing a Predict Table

Helper Function: First( )

- To fill a row of the table: start with that row's non-terminal and try all applicable rules, tracking which terminal symbols *eventually* appear as the first character of a string
- *Question:* For each non-terminal N $\in$ {S', S, A, B} which terminals that can begin a string derived from N, i.e. N $\Rightarrow^*$ c···

Row 1: S'

1. S' $\rightarrow$ ⊢ S ⊣ First (S') = {⊢} by rule 1

Row 2: S

2. S $\rightarrow$ AyB First (S) = {a, c} by rule 2 (then 3 or 4)
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd

Constructing a Predict Table

Helper Function: First( )

Row 3: A

3. $A \to ab$
4. $A \to cd$ First(A) = {a, c} by rules 3 and 4

Row 4: B

5. $B \to z$
6. $B \to wz$ First(B) = {z, w} by rules 5 and 6

- You can generalize First( ) to talk about α where α is any string of terminals and non-terminals, or possibly ε …
- *Formally* First(α) = { c | $\alpha \Rightarrow^* c\beta$} where c is a terminal and $\alpha, \beta \in$ (terminals | non-terminals)*.
- Now consider the next helper function Follow( )…

Constructing a Predict Table

Helper Function: Follow( )

- To understand Follow(), we need to add a rule to our original grammar where a non-terminal derives ε, e.g. rule 7: $B \rightarrow \varepsilon$
- Now we can derive:
  $S' \overset{1}{\Rightarrow} \vdash S \dashv \overset{2}{\Rightarrow} \vdash AyB\dashv \overset{3}{\Rightarrow} \vdash abyB\dashv \overset{7}{\Rightarrow} \vdash aby\dashv$
- *key point:* $\dashv$ can appear after the B *but there is no derivation* $B \Rightarrow^* \dashv$
- i.e. *using First() is not sufficient*
  - the symbol '$\dashv$' came from rule 1: $S' \rightarrow \vdash S \dashv$
  - the symbol B came from rule 2: $S \rightarrow AyB$
  - and B derives ε with rule 7: $B \rightarrow \varepsilon$
- *conclusion:* $\dashv$ is in the follow set of B

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$
7. $B \rightarrow \varepsilon$

Constructing a Predict Table

Using Follow( ) to Construct the Predict Table

- The Predict Table for our new grammar has a new entry Predict(B, ⊣) = 7 (the rest is the same)

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$
7. $B \rightarrow \varepsilon$

	a	b	c	d	y	w	z	⊢	⊣
S’								1	
S	2		2						
A	3		4						
B						6	5		7

- We used rule 7 to take the step $\vdash abyB \dashv \;\Rightarrow\; \vdash aby \dashv$
- So if B is on the stack and the next input symbol is ‘⊣’ then expand with rule 7, i.e. have B derive the empty string.

Constructing a Predict Table

Helper Function: Follow( )

- The terminal symbol '⊣' is in Follow(B) because there is a derivation from the start symbol $S' \Rightarrow^* \vdash abyB\dashv$
- *Informally:* Follow(N) is the set of terminals c that can follow N in some derivation; that is, $S \Rightarrow^* \cdots Nc \cdots$
- *Formally:* for any non-terminal N, $\text{Follow}(N) = \{ c \mid S' \Rightarrow^* \alpha N c \beta \}$
  - where α and β are (possibly empty) sequences of terminals and non-terminals
- But Follow(N) is only relevant if there is a derivation $N \Rightarrow^* \varepsilon$ so we need to check if N can derive the empty string.
- We need yet another helper function Nullable()…

Constructing a Predict Table

Helper Function: Nullable( )

- Sometimes called Empty( )
- *Informally*: Nullable(N) indicates that N can derive the empty string, i.e. $N \Rightarrow^* \varepsilon$
- *More generally, ask* if α can derive the empty string where α is in (terminals | non-terminals)* and B_i is a single terminal or non-terminal.
- *Formally*: Nullable(α) = true if $\alpha \Rightarrow^* \varepsilon$
  - False if α has a terminal in it (only non-terminals can derive ε)
  - True if there is a rule $\alpha \rightarrow \varepsilon$
  - For any rule of the form $\alpha \rightarrow B_1 B_2 \cdots B_n$
    Nullable(α) is true if each of Nullable(B_1), Nullable(B_2), ..., Nullable(B_n) is true.

LL(1) Parsing

```
Input: w
push S' (start symbol) on stack
for each a ∈ w {
    while (top of stack is a non-terminal N ) {   // 1st try expand
        if ( Predict(N, a) == (N → α) )
            pop N
            push α on stack (in reverse)
        else
            reject                                 //  no rule found
    }
    c = pop_stack()                                //  2nd try match
    if (c ≠ a)
        reject                                     //  no match found
}
accept w
```

Example of LL(1) Parsing

LL(1) Parsing: Parse ⊢ cdy ⊣

	Derivation	Read	Input	Stack	Action
1	S'		⊢ cdy ⊣	> **S'**	predict(S', ⊢) = 1
2	⊢ S ⊣		**⊢** cdy ⊣	> **⊢** S ⊣	match
3	⊢ S ⊣	⊢	cdy ⊣	> **S** ⊣	predict(S, c) = 2
4	⊢ AyB ⊣	⊢	cdy ⊣	> **A** y B ⊣	predict(A, c) = 4
5	⊢ cdyB ⊣	⊢	**c**dy ⊣	> **c** d y B ⊣	match
6	⊢ cdyB ⊣	⊢ c	**d**y ⊣	> **d** y B ⊣	match
7	⊢ cdyB ⊣	⊢ cd	**y** ⊣	> **y** B ⊣	match
8	⊢ cdyB ⊣	⊢ cdy	⊣	> **B** ⊣	predict(B,⊣) = 7
9	⊢ cdy ⊣	⊢ cdy	⊣	> **⊣**	match
10	⊢ cdy ⊣	⊢ cdy ⊣		>	ACCEPT

More about Follow()

Helper Function: Follow( ) is Complicated

- Need a different grammar to see this fact.
- In the grammar on the right $\dashv \in$ Follow(S) since $S' \rightarrow \vdash S \dashv$ and $S \Rightarrow ABC \Rightarrow BC \Rightarrow C \Rightarrow \varepsilon$
- But we also have the derivation $S' \Rightarrow \vdash S \dashv \Rightarrow \vdash ABC \dashv \Rightarrow \vdash aBC \dashv \Rightarrow \vdash aB \dashv$ and Nullable(B) = true so $\dashv \in$ Follow(B)
- But there is no rule of the form $S' \rightarrow \cdots B \dashv$

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow ABC$
3. $A \rightarrow aA$
4. $A \rightarrow \varepsilon$
5. $B \rightarrow bB$
6. $B \rightarrow \varepsilon$
7. $C \rightarrow cC$
8. $C \rightarrow \varepsilon$

- However $\dashv \in$ Follow(S), there is a rule $S \rightarrow ABC$ and Nullable(C) = true.
- More generally if $N \rightarrow B_1B_2 \ldots B_iB_{i+1} \ldots B_n$ and Nullable($B_{i+1}B_{i+2} \ldots B_n$) then Follow($B_i$) = Follow($B_i$) U Follow(N) i.e. if the RHS of B_i *is nullable, then what follows* N *can also follow* B_i.

More about Follow()

Helper Function: Follow( ) is Complicated

- *Asking:* Starting from the start symbol, does the terminal c ever occur immediately following B_i.
- Here c is a terminal; A, N are non-terminals; B_i is a single terminal or non-terminal; $\alpha, \beta \in$ (terminals | non-terminals)*
- **Follow**(B_i) = { c | $S \Rightarrow^* \alpha B_i c \beta$ }

  Initialize: Follow(N) = { } for all non-terminals N // *the empty set*

  for each rule of the form $A \rightarrow B_1 B_2 \ldots B_{i-1} B_i B_{i+1} \ldots B_k$:

      for i = 1 to k:

          if (B_i is a non-terminal) // *what can appear after* B_i

              Follow(B_i) = Follow(B_i) ∪ First ($B_{i+1} B_{i+2} \ldots B_k$)

          if (Nullable($B_{i+1} B_{i+2} \ldots B_k$)) // *what can appear after* A

              Follow(B_i) = Follow(B_i) ∪ Follow(A)

Constructing a Predict Table

Constructing Predict(N, c)

- *Asking:* If N is on the top of the stack and c is the next symbol in the input, which rule should be used to expand N?
- Here $\alpha, \beta \in$ (terminals | non-terminals)*

  c is a terminal, N is a non-terminal
- **Predict**(N, c) = { the rule $N \to \alpha$ | c $\in$ First(α) } ∪

  { the rule $N \to \beta$ | c $\in$ Follow(N) and Nullable(β) = true }
- *In summary:* To fill out the Predict Table, i.e. calculate which rule to use for Predict(N, c), we need to consider
  - First(α) for all rules of the form $N \to \alpha$
  - Follow(N) for all rules of the form $N \to \beta$ whenever Nullable(β) is true.

Example of Constructing a Predict Table

First()

First(α) ={ a | $\alpha \Rightarrow^*$ aβ}

A: a $\in$ First (A) since A $^3\Rightarrow$ aA

B: b $\in$ First (B) since B $^5\Rightarrow$ bB

S: a $\in$ First (S) since S $^2\Rightarrow$ AB $\Rightarrow$ aAB

b $\in$ First (S) since S $^2\Rightarrow$ AB $\Rightarrow$ B $\Rightarrow$ bB

1. S' $\rightarrow$ ⊢ S ⊣
2. S $\rightarrow$ AB
3. A $\rightarrow$ aA
4. A $\rightarrow$ ε
5. B $\rightarrow$ bB
6. B $\rightarrow$ ε

Nullable()

Nullable(α) = true if $\alpha \Rightarrow^* \varepsilon$

A: Nullable(A) = true since A $\Rightarrow \varepsilon$ by rule 4

B: Nullable(B) = true since B $\Rightarrow \varepsilon$ by rule 6

S: Nullable(S) = true since S $\Rightarrow$ AB $\Rightarrow$ B $\Rightarrow \varepsilon$ starting with rule 2

Example of Constructing a Predict Table

Follow()

Recall: Follow(B_i) = { c | S' $\Rightarrow^*$ $\alpha B_i c \beta$}

If Nullable(B_i) we need to consider Follow(B_i)

for rules N $\rightarrow$ $B_1 B_2 \ldots B_{i-1} B_i B_{i+1} \ldots B_n$:

(i) Follow(B_i) = Follow(B_i) U First ($B_{i+1} B_{i+2} \ldots B_n$)

(ii) if (Nullable($B_{i+1} B_{i+2} \ldots B_n$))
Follow(B_i) = Follow(B_i) U Follow(N)

1. S' $\rightarrow$ ⊢ S ⊣
2. S $\rightarrow$ AB
3. A $\rightarrow$ aA
4. A $\rightarrow$ ε
5. B $\rightarrow$ bB
6. B $\rightarrow$ ε

S: ⊣ $\in$ Follow(S) since S' $\rightarrow$ ⊢ S ⊣ and ⊣ $\in$ First(⊣) by (i)

B: ⊣ $\in$ Follow(B) since S $\rightarrow$ AB and ⊣ $\in$ Follow(S) by (ii)

A: ⊣ $\in$ Follow(A) since S $\rightarrow$ AB, Nullable(B) and ⊣ $\in$ Follow(S) by (ii)

b $\in$ Follow(A) since S $\rightarrow$ AB and b $\in$ First(B) by (i)

Example of Constructing a Predict Table

The Predict Table

- Let N $\in$ {S, A, B} and let c $\in$ {a, b, ⊢, ⊣}
- For the entries due to First(N), use rule N $\rightarrow \alpha$ where c $\in$ First(α)
- For the entries due to Follow(N) use rule N $\rightarrow \alpha$ where c $\in$ Follow(N) and Nullable(α) = true

Grammar

1. S' $\rightarrow$ ⊢ S ⊣
2. S $\rightarrow$ AB
3. A $\rightarrow$ aA
4. A $\rightarrow \varepsilon$
5. B $\rightarrow$ bB
6. B $\rightarrow \varepsilon$

Predict Table

	a	b	⊢	⊣
S'			1	
S	2	2		2
A	3	4		4
B		5		6

Computing Nullable

Nullable( )

for each non-terminal A: Nullable(A) = false // initialize
repeat
  for each rule $A \rightarrow B_1B_2 \ldots B_k$ // check rules
    if (k = 0) **or** (Nullable(B_1) = ⋯ = Nullable(B_k) = true)
      then Nullable(A) = true
until nothing changes

R1 $S' \rightarrow \vdash S \dashv$
R2 $S \rightarrow b\ S\ d$
R3 $S \rightarrow p\ S\ q$
R4 $S \rightarrow C$
R5 $C \rightarrow c\ C$
R6 $C \rightarrow \varepsilon$

Iteration	**0**	**1**	**2**	**3**
S’	false	false	false	false
S	false	false	true	true
C	false	true	true	true

Computing First

First(A) **for a Non-terminal** A

for each non-terminal A: First(A) = { } // initialize
repeat
  for each rule $A \rightarrow B_1B_2\cdots B_k$ // check rules
    for i = 1 … k
      if (B_i is a non-terminal) // B_i is a non-terminal
        First(A) = First(A) ∪ First(B_i)
        if (**not** Nullable(B_i)) **then** break; // go to next rule
      else // B_i is a terminal
        First(A) = First(A) ∪ {B_i};
        break // go to next rule
until nothing changes

General Idea: keep processing $B_1B_2\cdots B_k$ until you encounter a terminal or a symbol that is not Nullable. Then go to the next rule.

Computing First

First*$(B_1B_2\cdots B_k)$ **for a Concatenation of Symbols**

// Before you considered each rule, now just consider $B_1B_2\cdots B_k$.

```
answer = { }                                  // initialize
for i = 1 … k                                 // check B1B2···Bk
  if (Bi is a non-terminal) then              // Bi is a non-terminal
    answer = answer ∪ First(Bi)
    if (not Nullable(Bi)) then break          // go to next rule
  else                                        // Bi is a terminal
    answer = answer ∪ {Bi}
    break;                                    // go to next rule
until nothing changes
```

General Idea: keep processing $B_1B_2\cdots B_k$ until you encounter a terminal or a symbol that is not Nullable. Then go to the next rule.

Computing First

First(A) **for a Non-terminal** A

R1 S’ $\rightarrow$ ⊢ S ⊣
R2 S $\rightarrow$ b S d
R3 S $\rightarrow$ p S q
R4 S $\rightarrow$ C
R5 C $\rightarrow$ c C
R6 C $\rightarrow$ ε

Iteration	**0**	**1**	**2**	**3**
S’	{ }	{⊢}	{⊢}	{⊢}
S	{ }	{b, p}	{b, c, p}	{b, c, p}
C	{ }	{c}	{c}	{c}

- Iteration 0: set all to empty set (line 1)
- Iteration 1: With rules R1, R2, R3, and R5 set the values For S’, S and C using lines 8-9 with i=1.
- Iteration 2: c becomes part of First(S) using line 6 and R4 namely First(S) = First(S) ∪ First{C}
- Iteration 3: nothing changes so terminate

Computing Follow

Follow(A) **for a Non-terminal** A

1. **for each** non-terminal A except S': Follow(A) = { } // initialize
2. **repeat**
3. **for each** rule A $\rightarrow$ $B_1B_2\cdots B_k$ // check rules
4. **for** i = 1 … k
5. **if** (B_i is a non-terminal) // B_i is a non-terminal
6. Follow(B_i) = Follow(B_i) ∪ First*($B_{i+1}\cdots B_k$) // case 1
7. **if** (Nullable($B_{i+1}\cdots B_k$)) **then**
8. Follow(B_i) = Follow(B_i) ∪ Follow(A) // case 2
9. **until** nothing changes

- No terminal can follow S', so no need to calculate its follow set.
- Have two cases for Follow(B_i): 1) First*($B_{i+1}\cdots B_k$)
  2) Nullable($B_{i+1}\cdots B_k$)

Computing Follow

Follow(A) **for a Non-terminal** A

R1 $S' \rightarrow \vdash S \dashv$
R2 $S \rightarrow b\ S\ d$
R3 $S \rightarrow p\ S\ q$
R4 $S \rightarrow C$
R5 $C \rightarrow c\ C$
R6 $C \rightarrow \varepsilon$

Iteration	**0**	**1**	**2**
S	{ }	{⊣, d, q}	{⊣, d, q}
C	{ }	{⊣, d, q}	{⊣, d, q}

- Iteration 0: set all to empty set (line 1)
- Iteration 1: with R1, R2 and R3 set the values S (lines 3-6)
  with R4 Follow(C) = Follow(C) ∪ Follow{S} (line 8)
- Iteration 3: nothing changes so terminate

Example of Constructing a Predict Table

The Predict Table

- Let N $\in$ {S', S, C} and let c $\in$ {b, c, d, p, q, ⊢, ⊣}
- For the entries due to First(N), use rule N $\rightarrow \alpha$ where c $\in$ First(α) (blue entries in table).
- For the entries due to Follow(N) use rule N $\rightarrow \alpha$ where c $\in$ Follow(N) and Nullable(α) = true (black entries in table).

Grammar

R1 S' $\rightarrow$ ⊢ S ⊣
R2 S $\rightarrow$ b S d
R3 S $\rightarrow$ p S q
R4 S $\rightarrow$ C
R5 C $\rightarrow$ c C
R6 C $\rightarrow$ ε

Predict Table

	b	c	d	p	q	⊢	⊣
S'						1	
S	2	4	4	3	4		4
C		5	6		6		6

Non-LL(1) Grammars

A Non-LL(1) Grammar

G: 1. $S \rightarrow a\ b$
   2. $S \rightarrow a\ c\ b$

	a	b	c
S	1,2		

- L(G) = {ab, acb}
- Not in LL(1).
- The predict table is ambiguous, i.e. Predict(S, a) ={1, 2}
- *Must look ahead to the second symbol* in order to tell which rule to use. The predict table must consider pairs of terminals.
- G is in LL(2).

	aa	ab	ac	ba	bb	bc	ca	cb	cc
S		1	2						

Non-LL(1) Grammars

Converting a Non-LL(1) Grammar

LL(2)

G: 1. $S \rightarrow a\ b$

2. $S \rightarrow a\ c\ b$

LL(1)

G': 1'. $R \rightarrow a\ T$

2'. $T \rightarrow b$

3'. $T \rightarrow cb$

	a	b	c
R	1		
T		2	3

- Rewrite overlapping productions (1 and 2) so that
  - one rule contains the common prefix (a) and
  - a new non-terminal (T) produces the different suffixes (b and cb).