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?

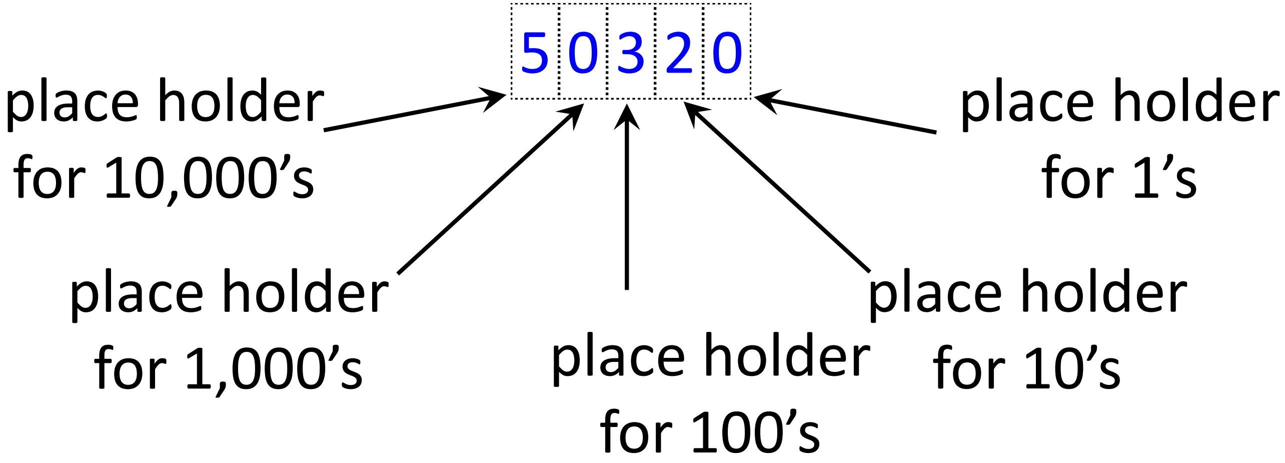
- Originally used base 10.
- Led to complicated designs in the age of vacuum tubes.
- Have to be able to distinguish between 10 different states.
- Konrad Zuse's mechanical computer Z1 (developed 1935 – 1938) was the first to use a binary representation.
- It led to a much *simpler design.*
- Bonus: 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

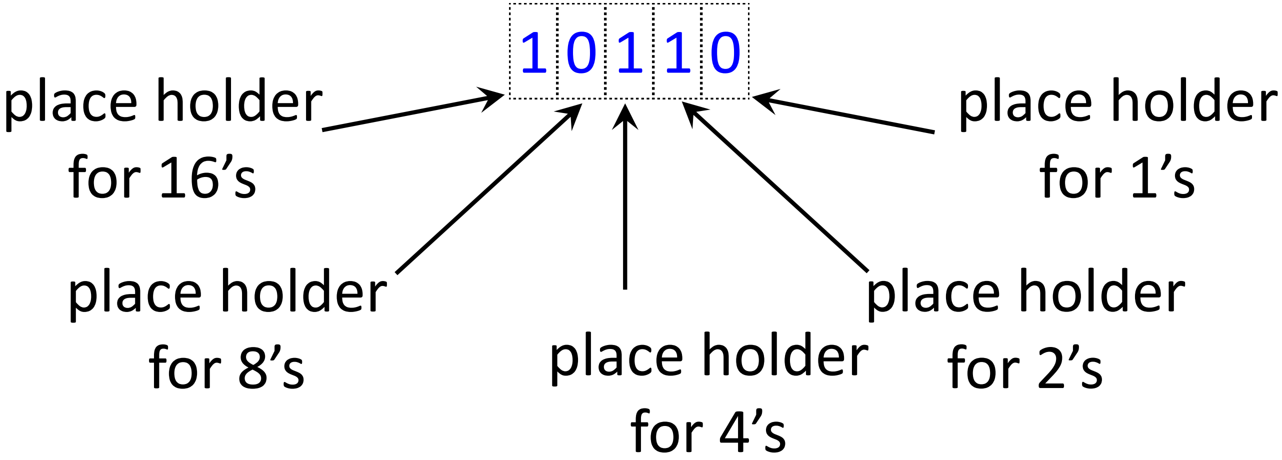
- *key idea:* each time you move over one digit from right to left, multiply the placeholder *by 10*

Unsigned Integers

Binary Representation

$$1\,0110_2 = 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 0\cdot2^0 = 22_{10}$$

$$1\,0110_2 = 1\cdot16 + 0\cdot8 + 1\cdot4 + 1\cdot2 + 0\cdot1 = 22_{10}$$



1 0 1 1 0

place holder for 16's

place holder for 8's

place holder for 4's

place holder for 2's

place holder for 1's

- *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 → 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 $50320_{\mathbf{10}}$ 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 $50320_{\mathbf{10}} = 50320_{\mathbf{10}}$

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

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

$$\begin{array}{rccccc} & \multicolumn{5}{c}{\longleftarrow} \\ & & & 1 & 1 & \\ & 0 & 0 & 0 & 0 & 1_2 \\ + & 0 & 1 & 0 & 1 & 1_2 \\ \hline & 0 & 1 & 1 & 0 & 0_2 \end{array} \qquad \begin{array}{r} 1_{10} \\ +11_{10} \\ \hline 15_{10} \end{array}$$

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	→	1111	→	0000	0_{10}
1_{10}:	0001	→	1110	→	1111	-1_{10}
4_{10}:	0100	→	1011	→	1100	-4_{10}
7_{10}:	0111	→	1000	→	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.

$$0\cdot(-2^3) + 0\cdot2^2 + 1\cdot2^1 + 2\cdot2^0 = 2+1 = 3$$

$$1\cdot(-2^3) + 1\cdot2^2 + 1\cdot2^1 + 1\cdot2^0 = -8+4+2+1 = -1$$

$$1\cdot(-2^3) + 0\cdot2^2 + 1\cdot2^1 + 1\cdot2^0 = -8+3 = -5$$

$$1\cdot(-2^3) + 0\cdot2^2 + 0\cdot2^1 + 0\cdot2^0 = -8$$

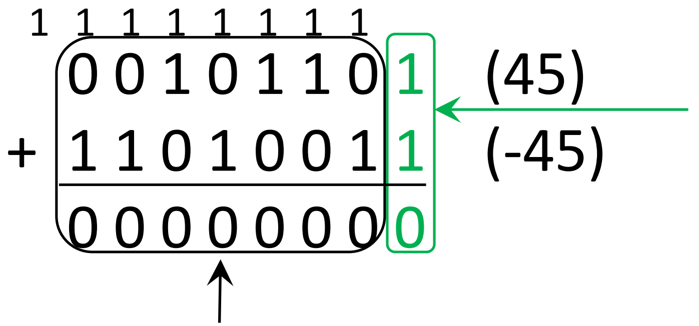
Two's Complement

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	3: 0011
-2: 1110	-3: 1101
4: 0100	5: 0101
-4: 1100	-5: 1011
6: 0110	7: 0111
-6: 1010	-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)

~~1~~0001 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)

~~0~~1111 -1

ignore last carry bit

Two’s Complement: Overflow

Example 3: 5 + 3

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

$$\begin{array}{r} \cancel{0}\;\,1\;1\;1 \\ 0101 \\ +0011 \\ \hline 1000 \end{array} \qquad\qquad \begin{array}{r} 5 \\ +3 \\ \hline -8 \end{array}$$

- 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

$0000_{\text{bin}} = 0_{\text{hex}}$

$0001_{\text{bin}} = 1_{\text{hex}}$

$0010_{\text{bin}} = 2_{\text{hex}}$

$0011_{\text{bin}} = 3_{\text{hex}}$

$0100_{\text{bin}} = 4_{\text{hex}}$

$0101_{\text{bin}} = 5_{\text{hex}}$

$0110_{\text{bin}} = 6_{\text{hex}}$

$0111_{\text{bin}} = 7_{\text{hex}}$

$1000_{\text{bin}} = 8_{\text{hex}}$

$1001_{\text{bin}} = 9_{\text{hex}}$

$1010_{\text{bin}} = a_{\text{hex}}$

$1011_{\text{bin}} = b_{\text{hex}}$

$1100_{\text{bin}} = c_{\text{hex}}$

$1101_{\text{bin}} = d_{\text{hex}}$

$1110_{\text{bin}} = e_{\text{hex}}$

$1111_{\text{bin}} = f_{\text{hex}}$

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

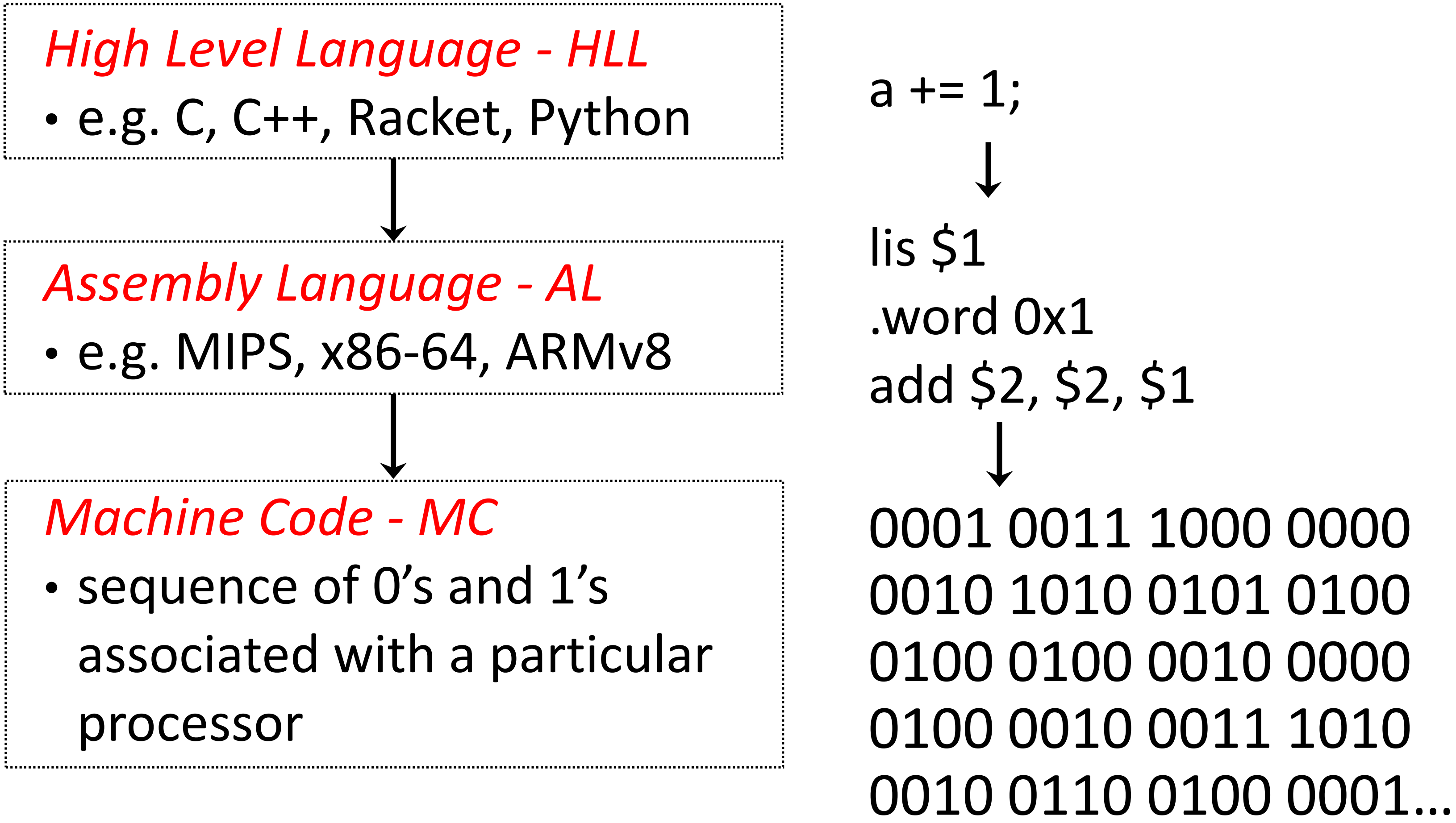
Key Ideas

- 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

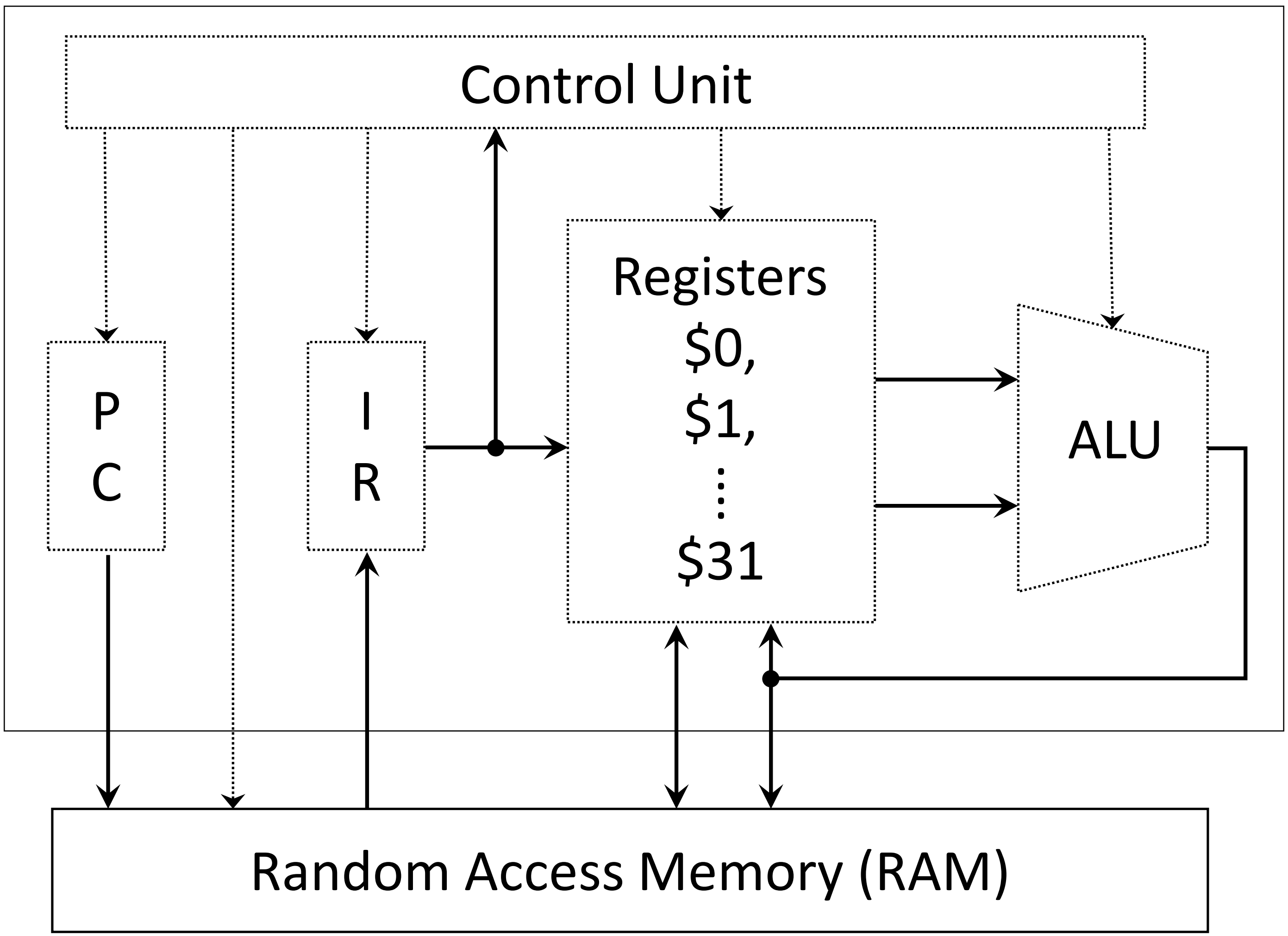
Constants

- 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 0th element, A[1] the 1st 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 0th element is called the *base address.*
- *The address of the ith 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

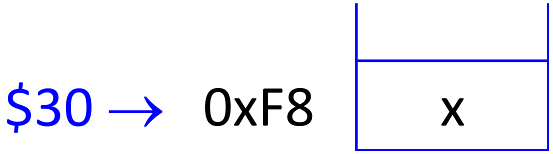
Example: store and then restore the values in $2 and $3 on the stack and the initial value of the SP ($30) is 0xF8…

The Call Stack

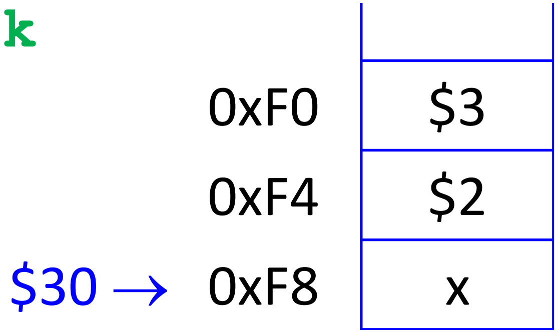
Saving $2 and $3 on the Stack

Stack

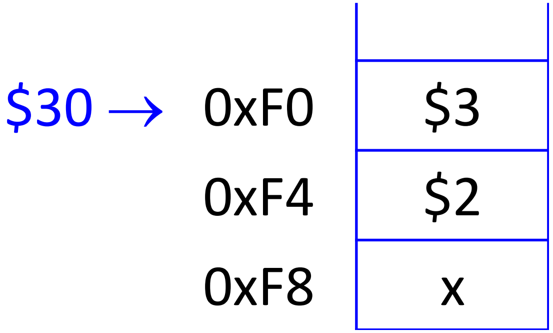
```
;; 0. Initially
```



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

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



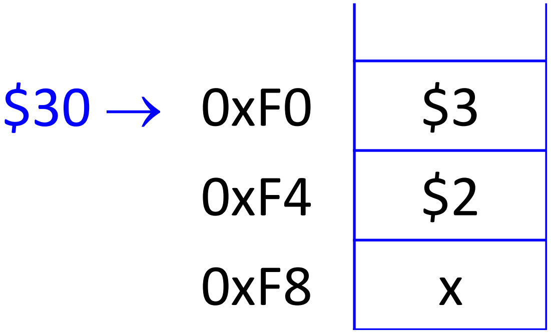
```
;; 2. Decrement the stack pointer

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



The Call Stack

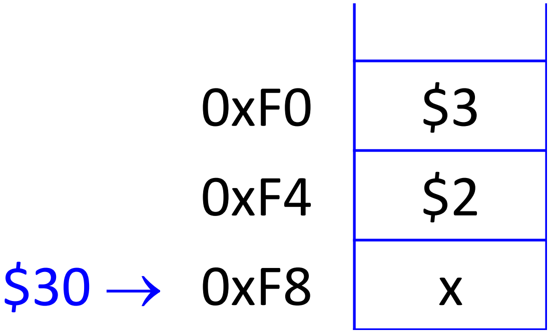
Restoring $2 and $3 from the Stack

```
;; 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

Covered
in Lecture

- Example 7: calling a recursive subroutine
  a) calling code
  b) recursive subroutine

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)

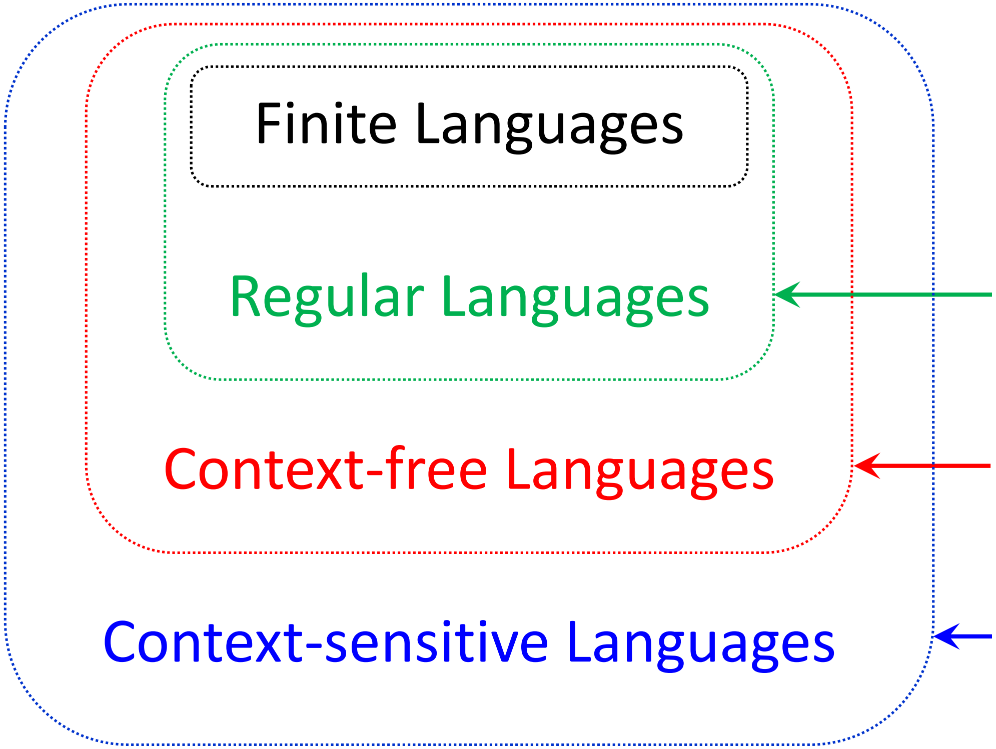
The Compiler

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



Finite Languages

Regular Languages

Context-free Languages

Context-sensitive Languages

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* Σ = { 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. αε = εα = α
- 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. 1*x* = *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

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

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

- $R^i = R \cdot R^{i\text{-}1}$, 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.

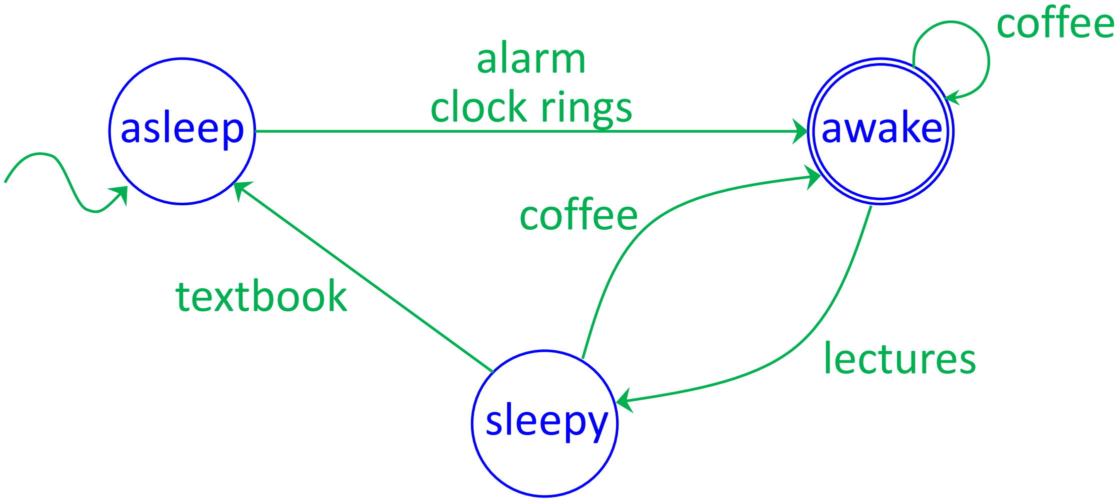
Deterministic Finite Automata (DFA)

- 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

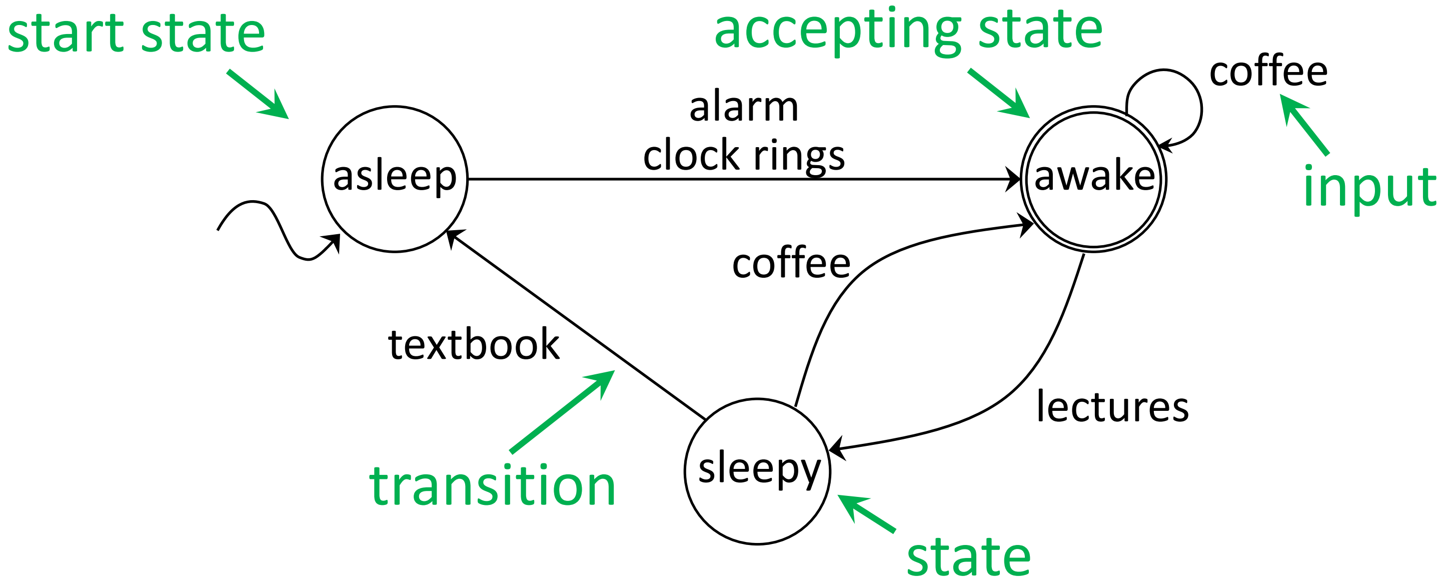
- Start state: asleep (has a curvy arrow pointing to it)
- Accepting state (a.k.a. end state): awake (has a double circle)
- Transitions: change states when input occurs: e.g. if you are in a sleepy state and drink coffee, go to the awake state.



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.

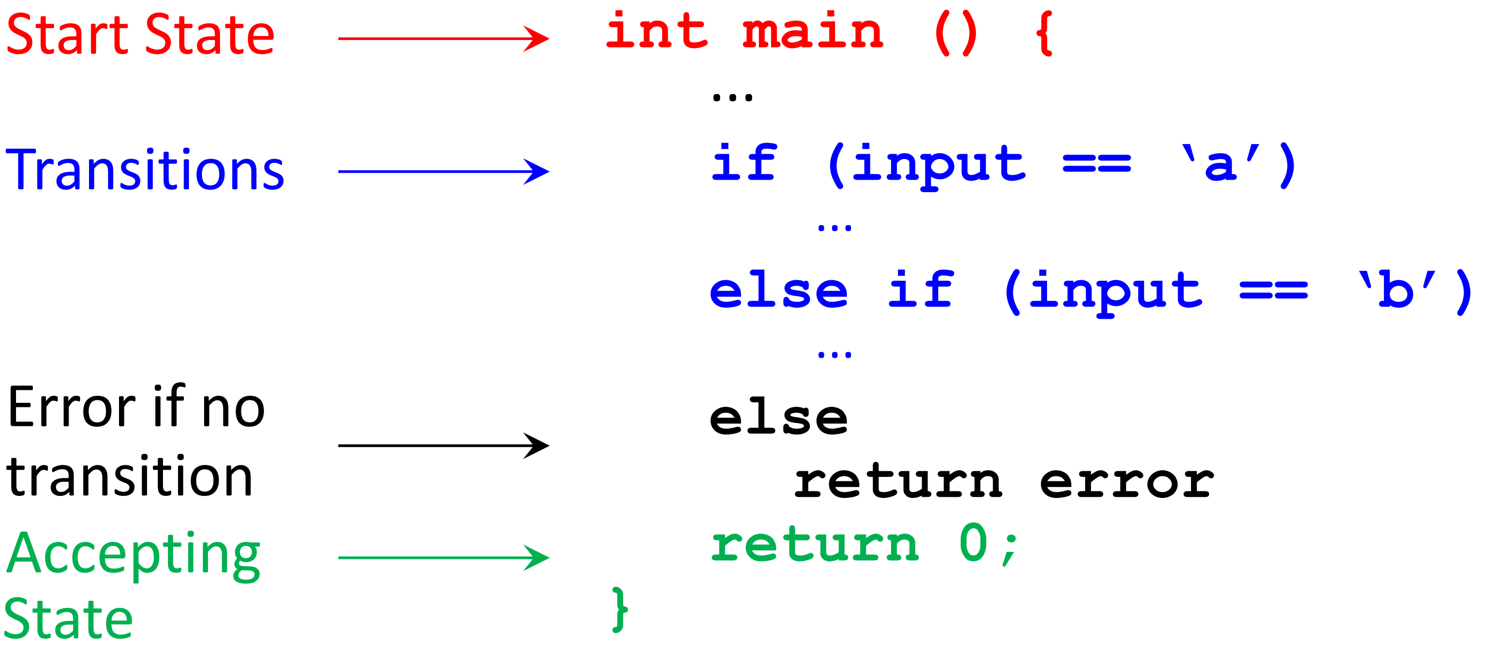


Parts of a DFA

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

Transitions

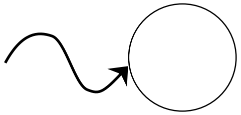
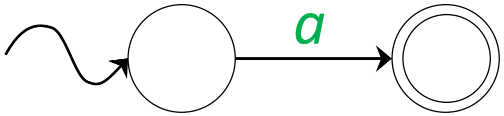
Error if no transition

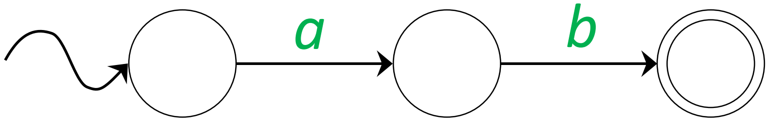
Accepting State

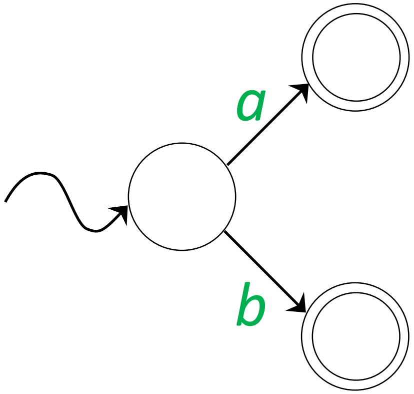
```
int main () {
     ...
     if (input == ‘a’)
          ...
     else if (input == ‘b’)
          ...
     else
        return error
     return 0;
}
```

Deterministic Finite Automata (DFA)

Examples of DFAs

- Accepts nothing: 
- Accepts {*a*} : 

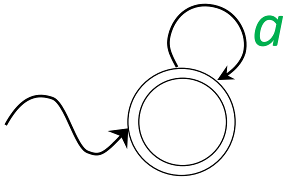
- Accepts {*ab*}: (concatenation) 

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

Deterministic Finite Automata (DFA)

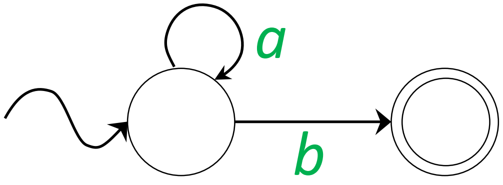
Examples of DFAs

- Accepts a*: (repetition)



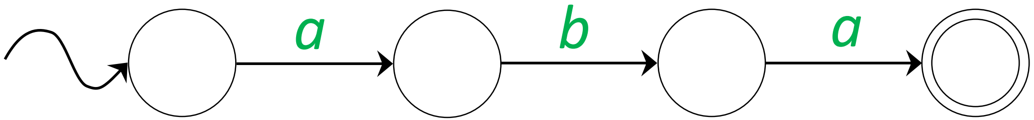
0 or more *a*’s

- 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.*

seen an *a*

seen *ab*

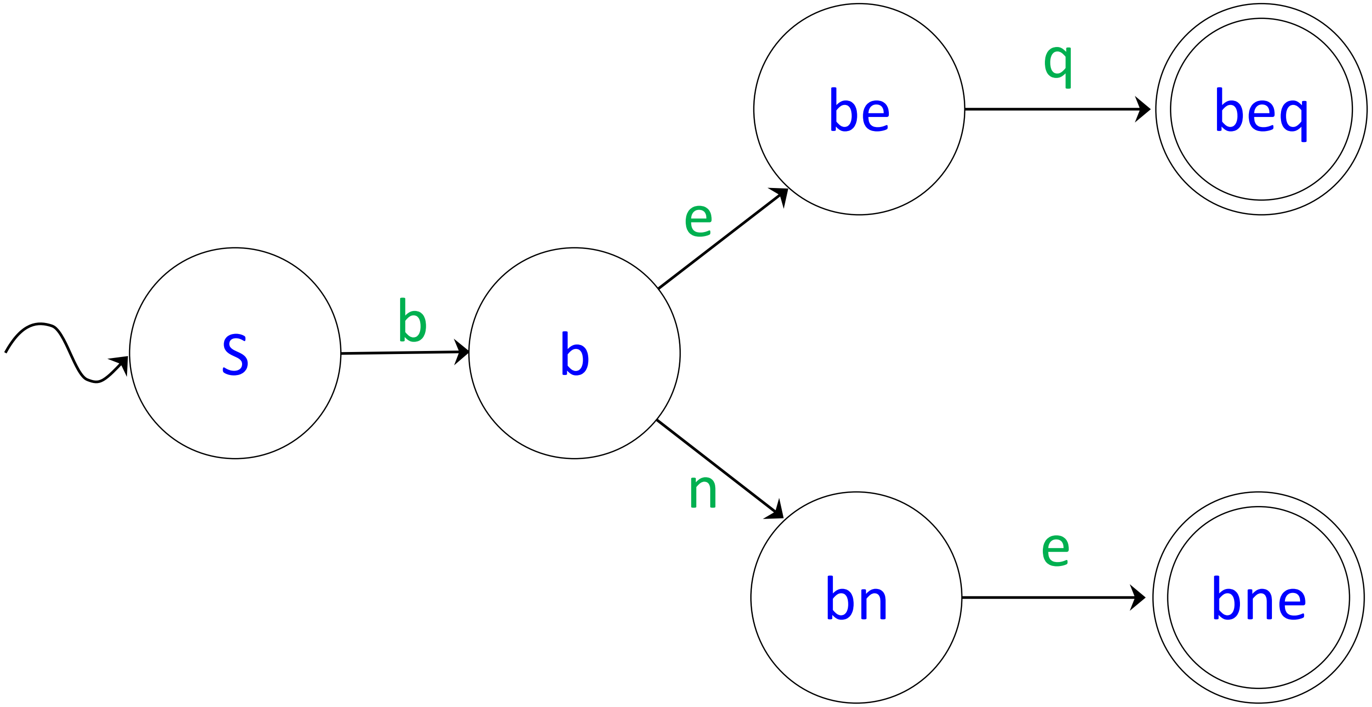
seen *aba*

- 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

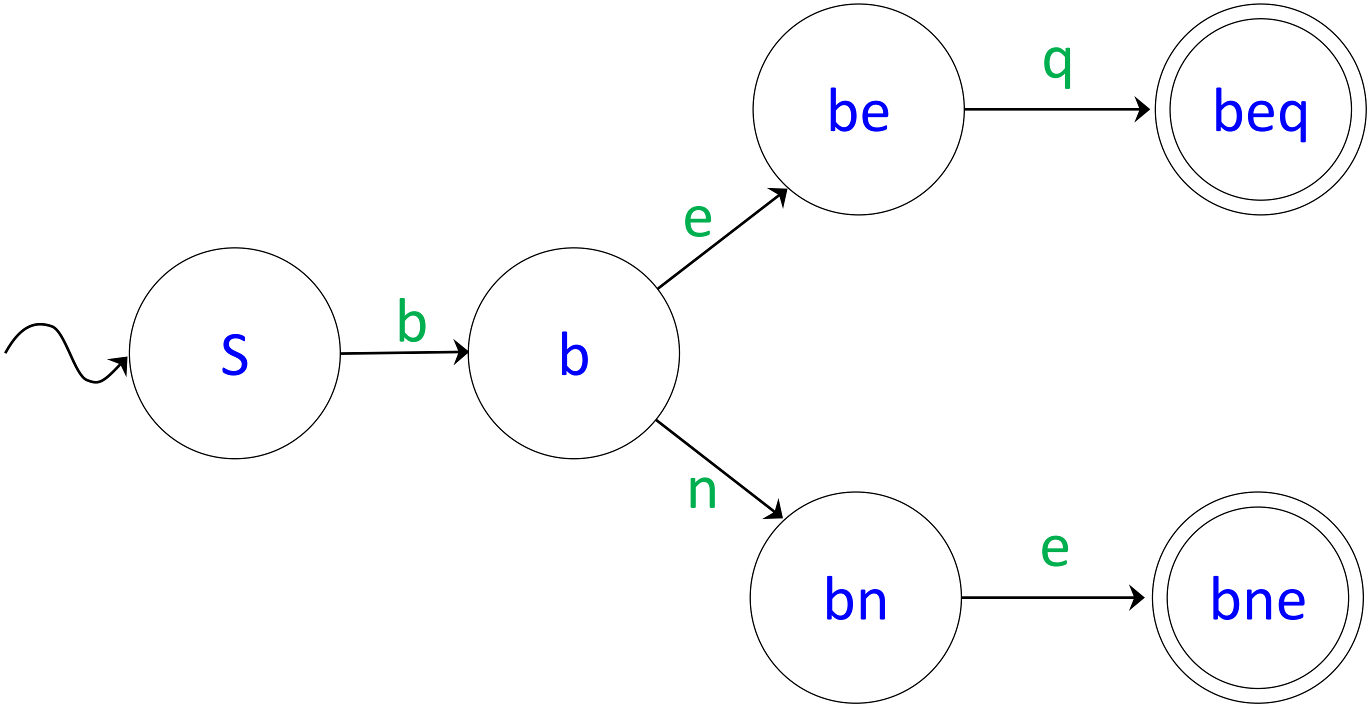
Let Σ ={a,b,c}

- 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 ← $q_0$                          // start in the start state
for i = 1 to n do:                   // for each character in the input
    state ← $\delta$ (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

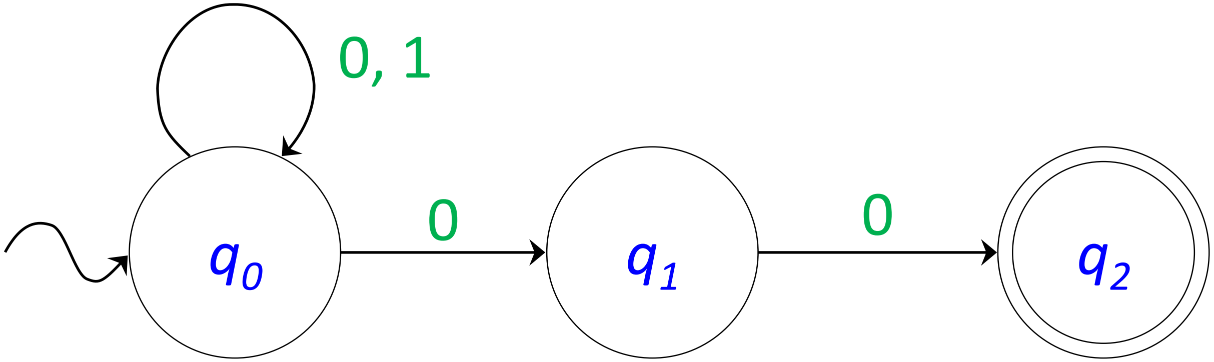
References

- *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 $\{q_0, q_1\}$.

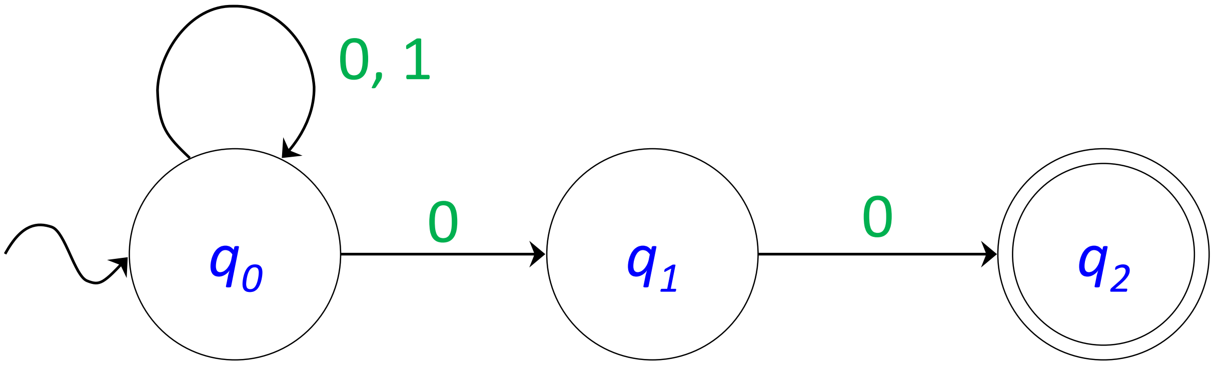


Non-deterministic Finite Automata (NFA)

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}*·{00}, i.e. the language of strings over the alphabet {0, 1} that end with 00.

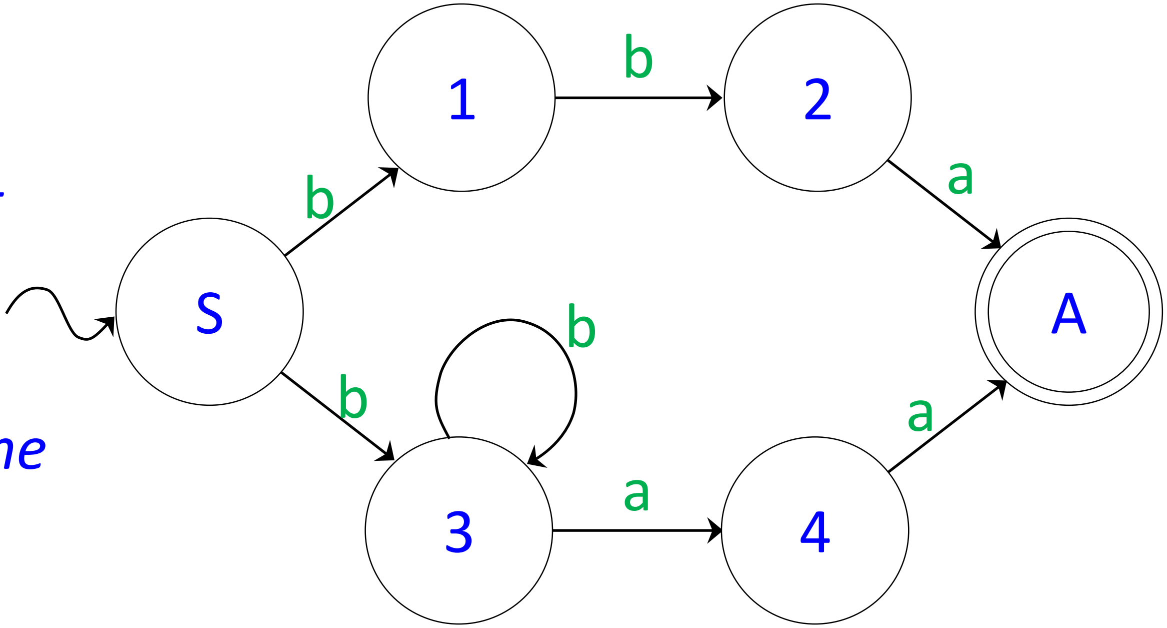
- It is often easier to design an NFA rather than an equivalent—but more complex—DFA (e.g. to tokenize input).
- Algorithms exist to convert an NFA to an equivalent DFA.



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. $c_1c_2 \cdots c_n$

```
states ← {$q_0$}                    // start in the start state
for $c_i$ 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, $c_i$)       // 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’ ← { }                         // 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’ ← 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

```
1. states ← {q_0}
2. for c_i in input do:
3.     s' ← { }
4.     for s in states do:
5.         s' ← s' U T(s, c_i)
6.     states ← s'
```

Value of Various Variables

states = $\{q_0\}$

c_1= 0

s' = { }

s = q_0

s' = { } U $T(q_0, 0) = \{q_0, q_1\}$

states = $\{q_0, q_1\}$

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

Non-deterministic Finite Automata (NFA)

Example 1

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

```
for $c_i$ in input do:
    $s' \leftarrow \{ \}$
    for s in states do:
        $s' \leftarrow s' \cup T(s, c_i)$

    states $\leftarrow s'$
return (states $\cap A \neq \{ \}$ )
```

$c_2 = 0$

$s' = \{ \}$

s in $\{q_0, q_1\}$

$s' = \{ \} \cup T(q_0, 0) = \{q_0, q_1\}$

$s' = \{q_0, q_1\} \cup T(q_1, 0) = \{q_0, q_1, q_2\}$

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

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

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

```
for c_i in input do:
    s' ← { }
    for s in states do:
        s' ← s' ∪ T(s, c_i)

    states ← s'
return (states ∩ A ≠ { } )
```

c_3= 1

$s' = \{ \}$

s in $\{q_0, q_1, q_2\}$

$s' = \{ \} \cup T(q_0,1) \cup T(q_1,1) \cup T(q_2,1)$

$s' = \{ \} \cup \{q_0\} \cup \{ \} \cup \{ \} = \{q_0\}$

states = $\{q_0\}$

$\{q_0\} \cap \{q_2\} = \{ \}$

$\{ \} \neq \{ \}$ 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)

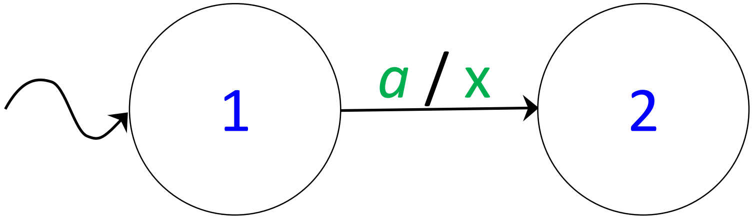
Where are DFA's used?

- 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.

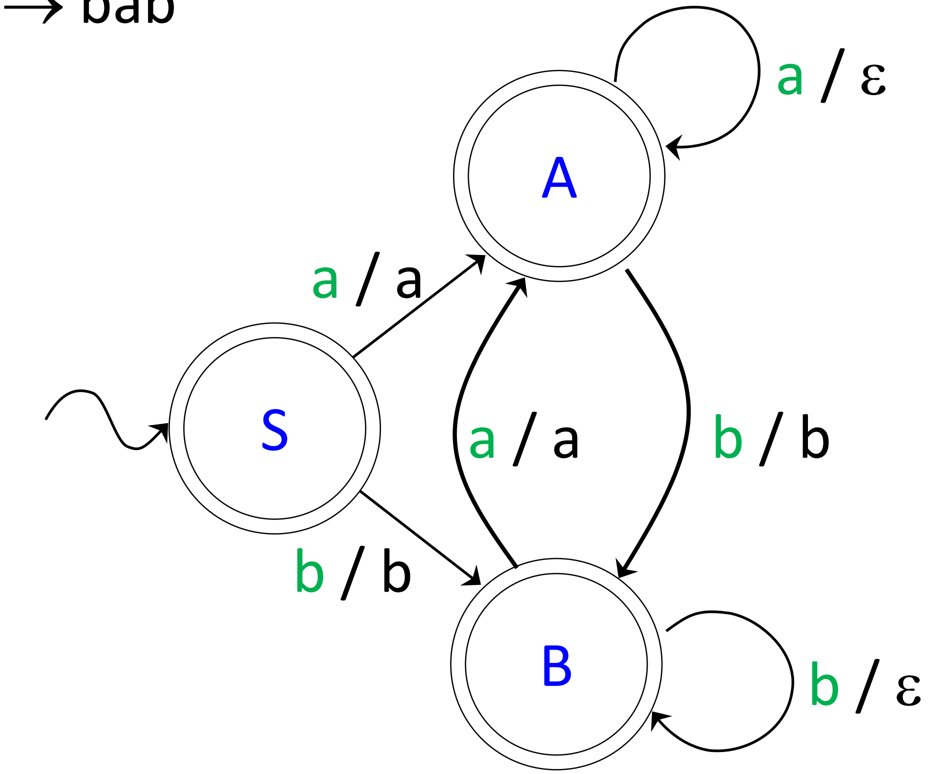


- for a lexer / scanner the output will be a token

Extensions

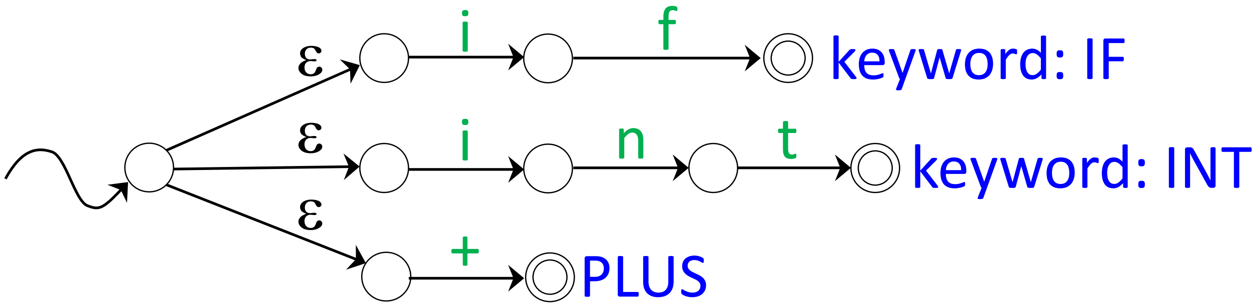
Transducers

- 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

- $\emptyset$ 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 $\emptyset$ concatenated with anything yields $\emptyset$.

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*|…|*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 = $q_0$                                 // start state of the DFA
loop:
    if ( i < k ):                             // 1: if not at end of input
        next_state = $\delta$(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 $\notin$ accepting_states): // 2a: not a valid token
            report a fatal error and exit     //   error in input
        if (state $\neq$ White_space):        // 2b: skip white space
            output token                      //   2c: output token
        state = $q_0$                         // 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

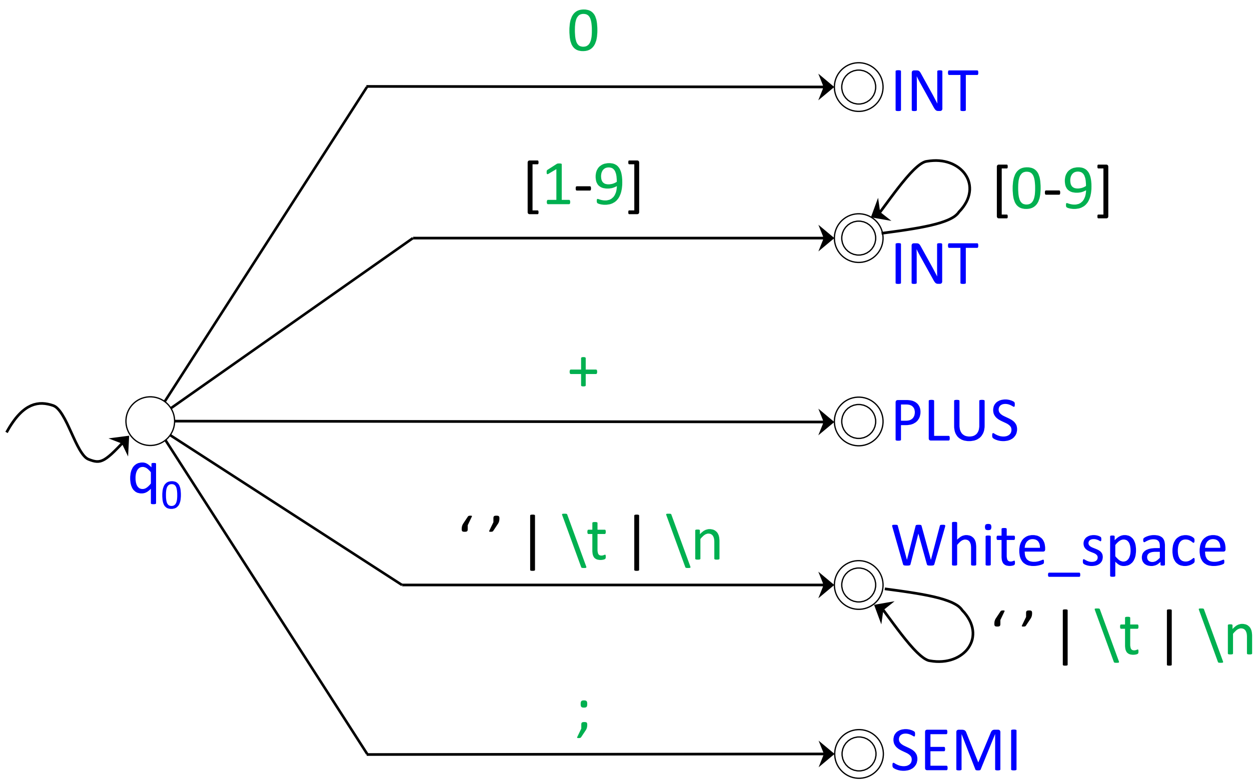
- If next_state == ERROR (lines 9-15):
  If you get an ERROR (line 8) the char counter *i* is not incremented, but the DFA does go to the start state (line 13) and *you reconsider the ith character as the start of the next token*.

- 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*: { ( ) { } , : }

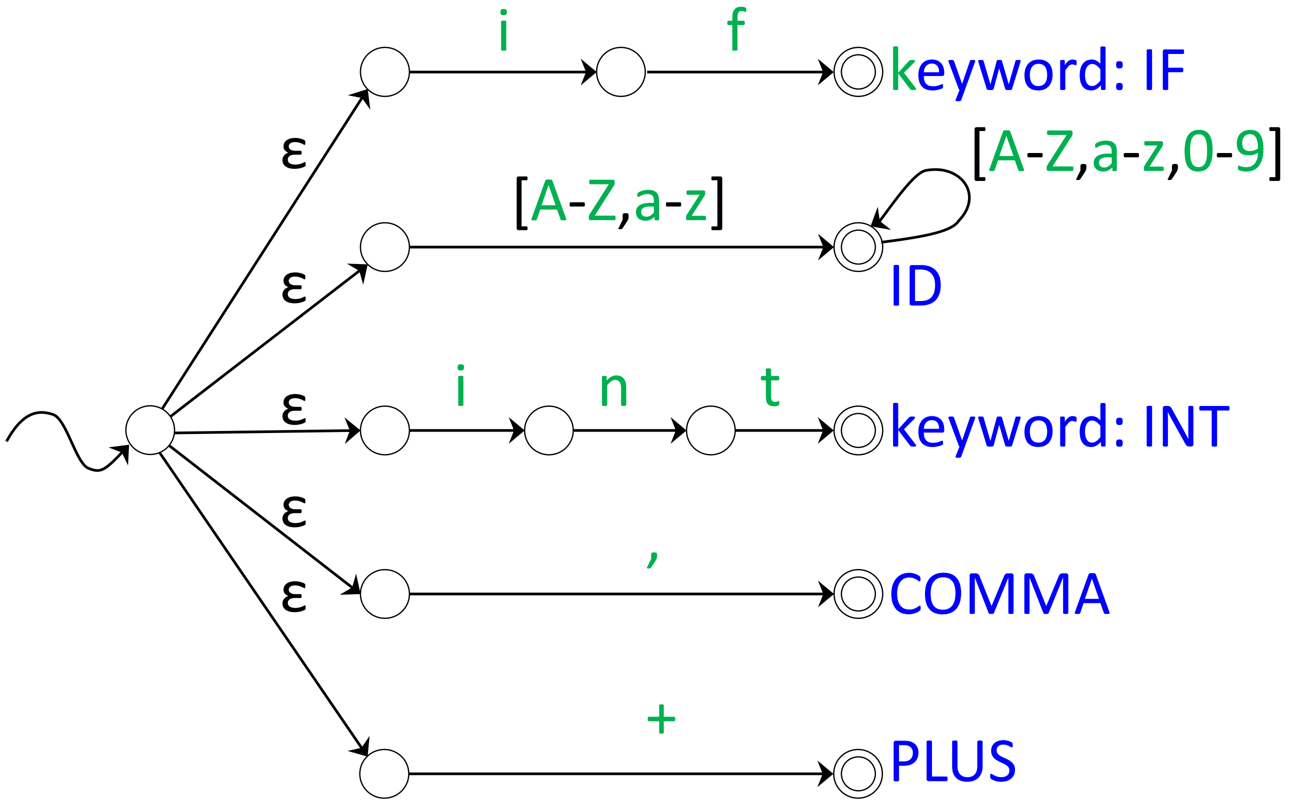
Scanners and NFAs

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

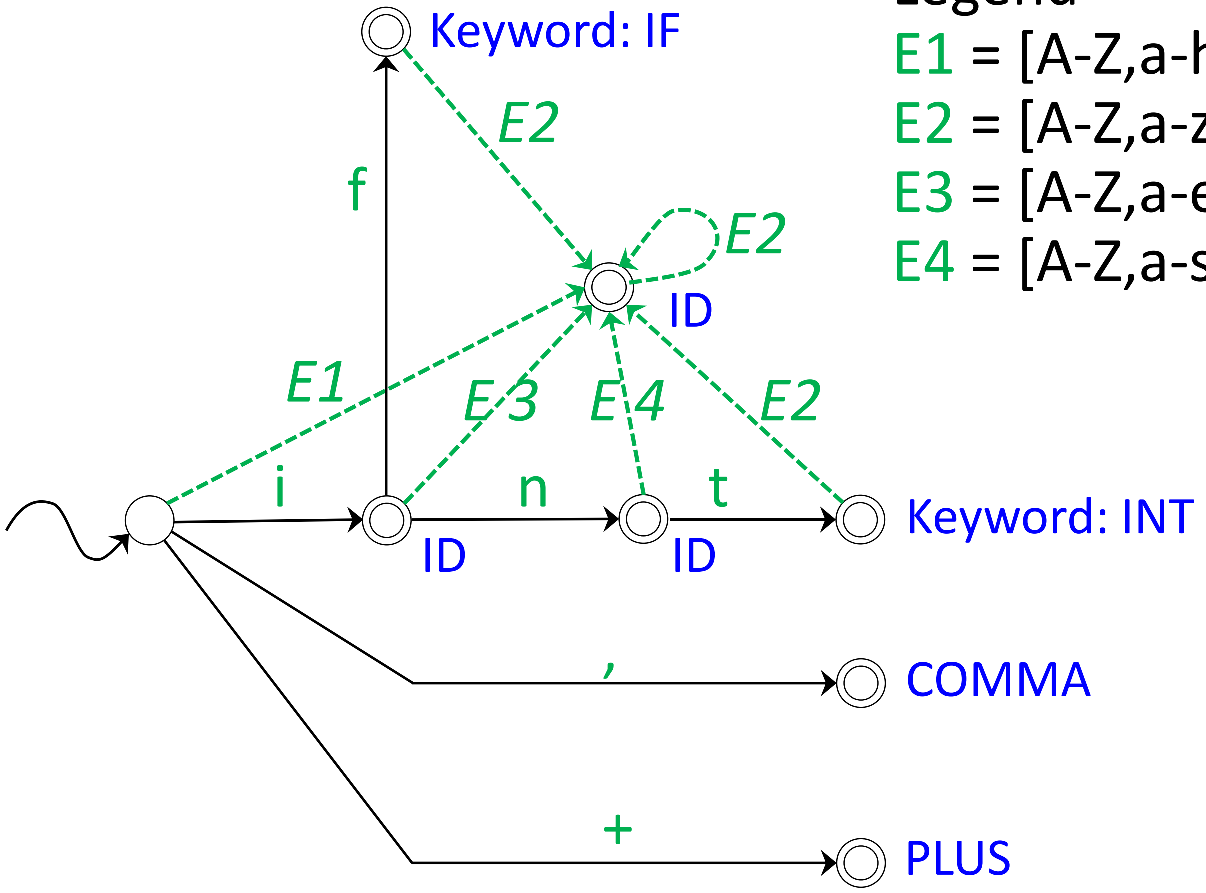
Scanners and NFAs

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]

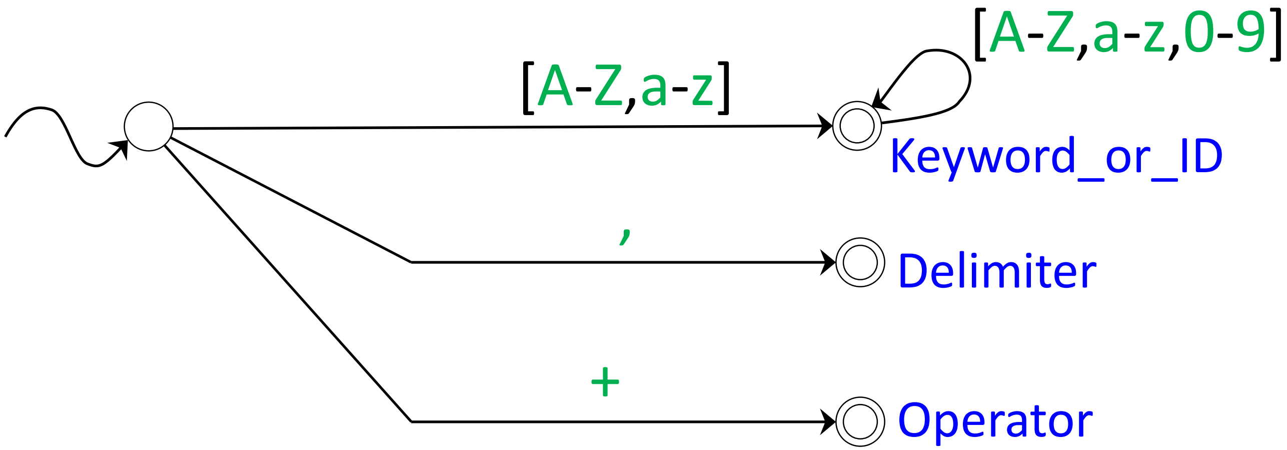
E3 = [A-Z,a-e,g-m,o-z,0-9]

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

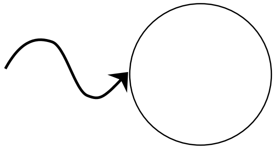
- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 2.1 to 2.5.

Regular Expressions (RE) to ε-NFAs

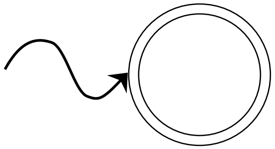
Convert an RE to an ε-NFA

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

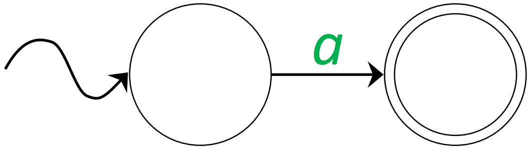
- If the RE is ∅ then the ε-NFA is:
  - no accepting state



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

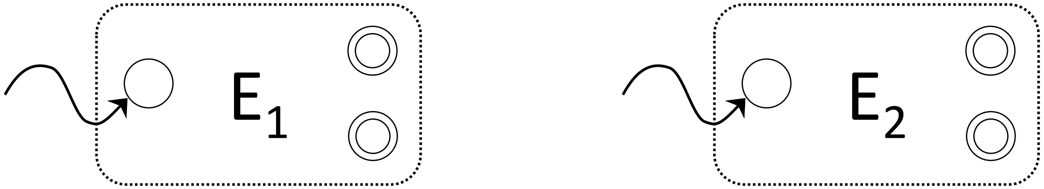


- If the RE is *a* then the ε-NFA is:

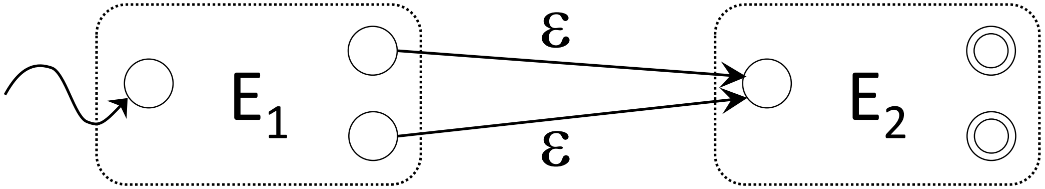


Regular Expressions (RE) to ε-NFAs

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.



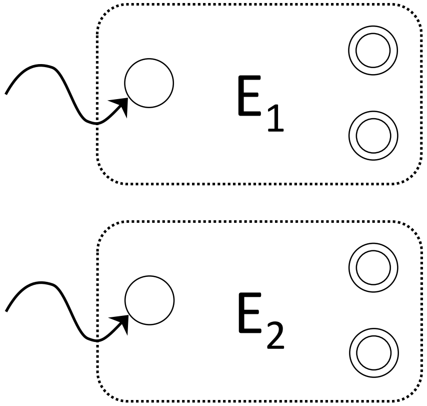
- Note: expressions and automata occur *in sequence*

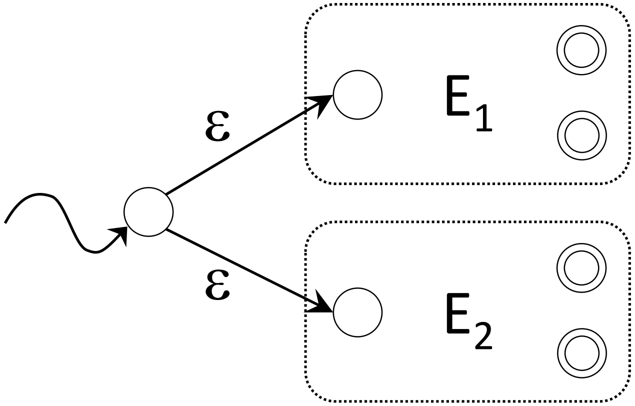
Regular Expressions (RE) to ε-NFAs

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*



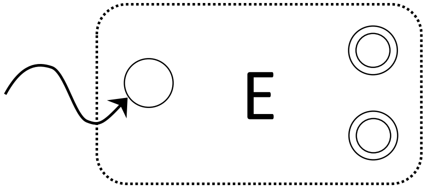


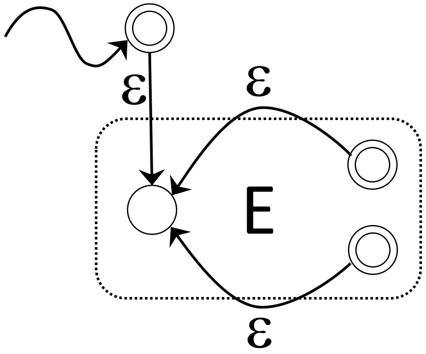
Regular Expressions (RE) to ε-NFAs

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.*

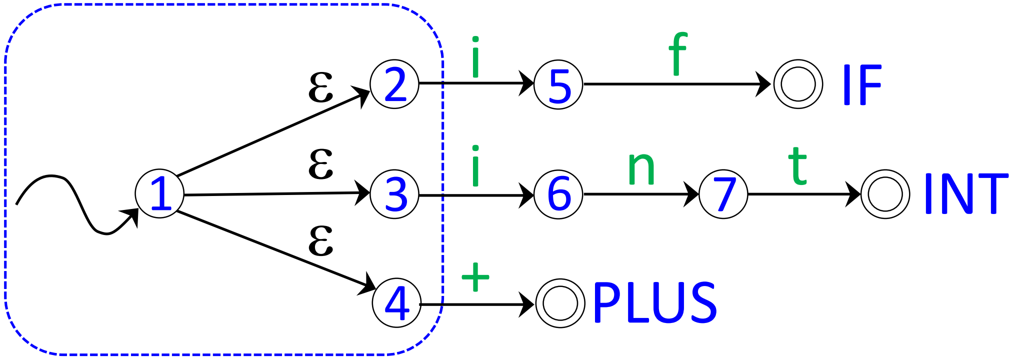




Converting ε-NFA to NFA

ε-closure

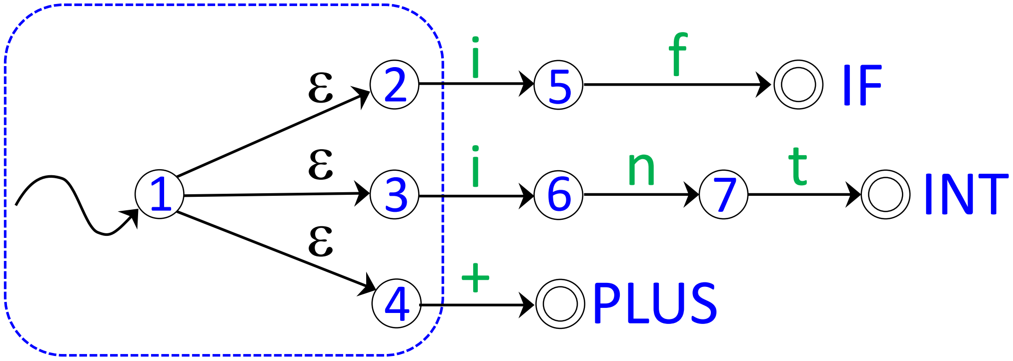
- The *ε-closure* of a state (or set of states) is the set of states that can be reached from that state (or set of states) by ε-transitions.



- 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.

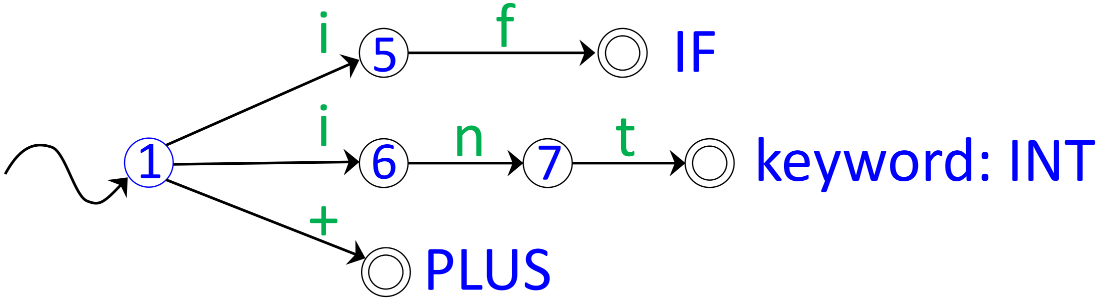
ε-Non-deterministic Finite Automata (ε-NFA)

Converting an ε-NFA to a NFA



E.g. ε-closure({1}) = {1,2,3,4}

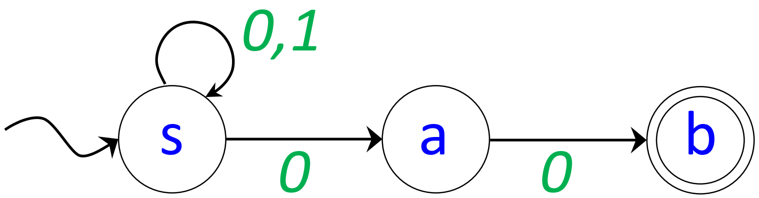
- input i: go from {1,2,3,4} to {5,6} and ε-closure({5,6}) = {5,6}.
- input +: go from {1,2,3,4} to {PLUS} and ε-closure({PLUS}) = {PLUS}.



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

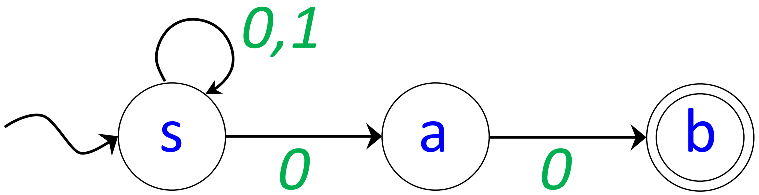
1. From State: for each possible input, track the set of Next States that can be reached.
2. If the Next State is new set of states, add it to the table and repeat step 1 for that new set of states.
3. Continue until any set that appears in the Next State column also appears in State column.

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}.

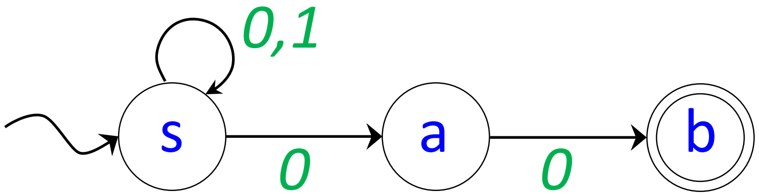


State	Input	Next State
{s}	0	{s, a}
	1	{s}
{s, a}	0	
	1	

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

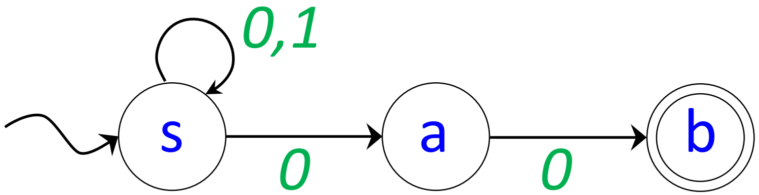
- From state {s, a, b}

  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.



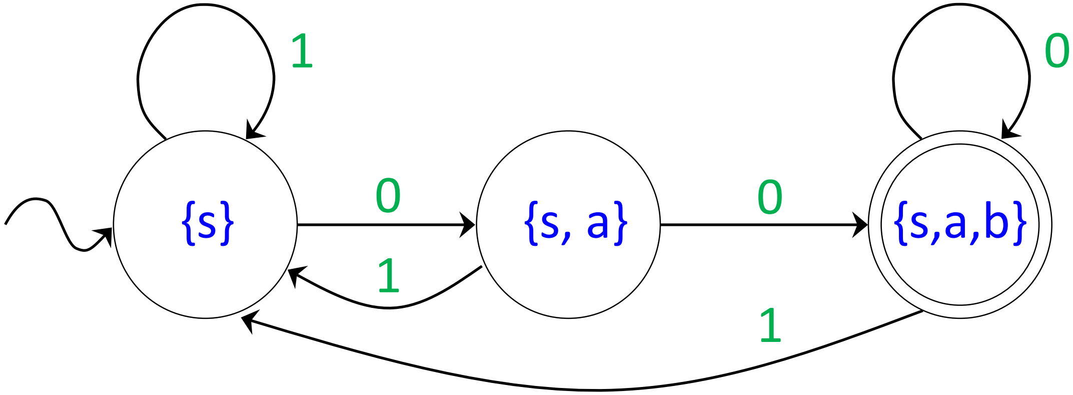
State	Input	Next State
{s}	0	{s, a}
	1	{s}
{s, a}	0	{s, a, b}
	1	{s}
{s, a, b}	0	{s, a, b}
	1	{s}

Converting NFA’s to DFA’s

Subset Construction: Example 1

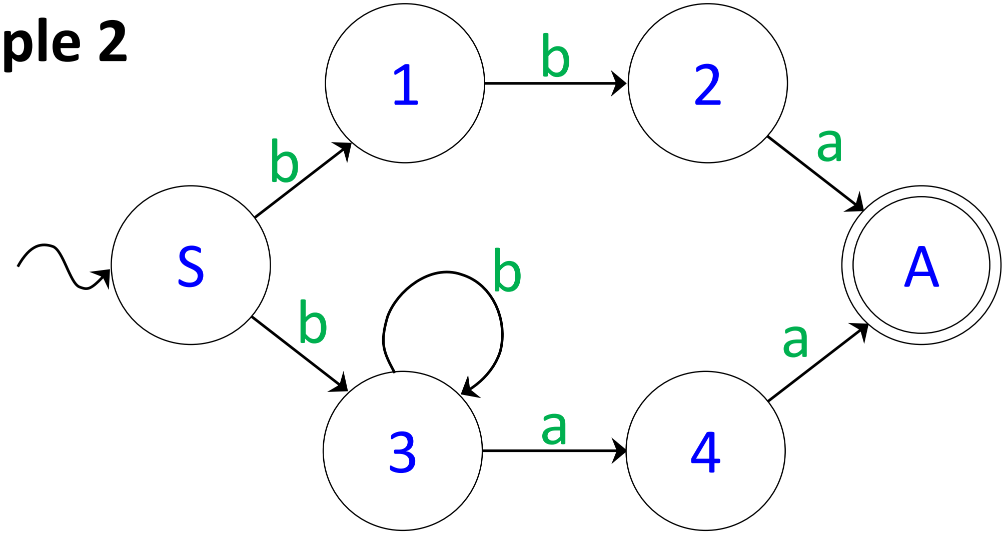
- Connect up the states with their corresponding transitions and inputs.
- The state that just contains the start state of the NFA, {s}, is also the start state of the DFA.
- Any DFA state that contains an accept state of the NFA (i.e. b) is also an accept state in the DFA.

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}

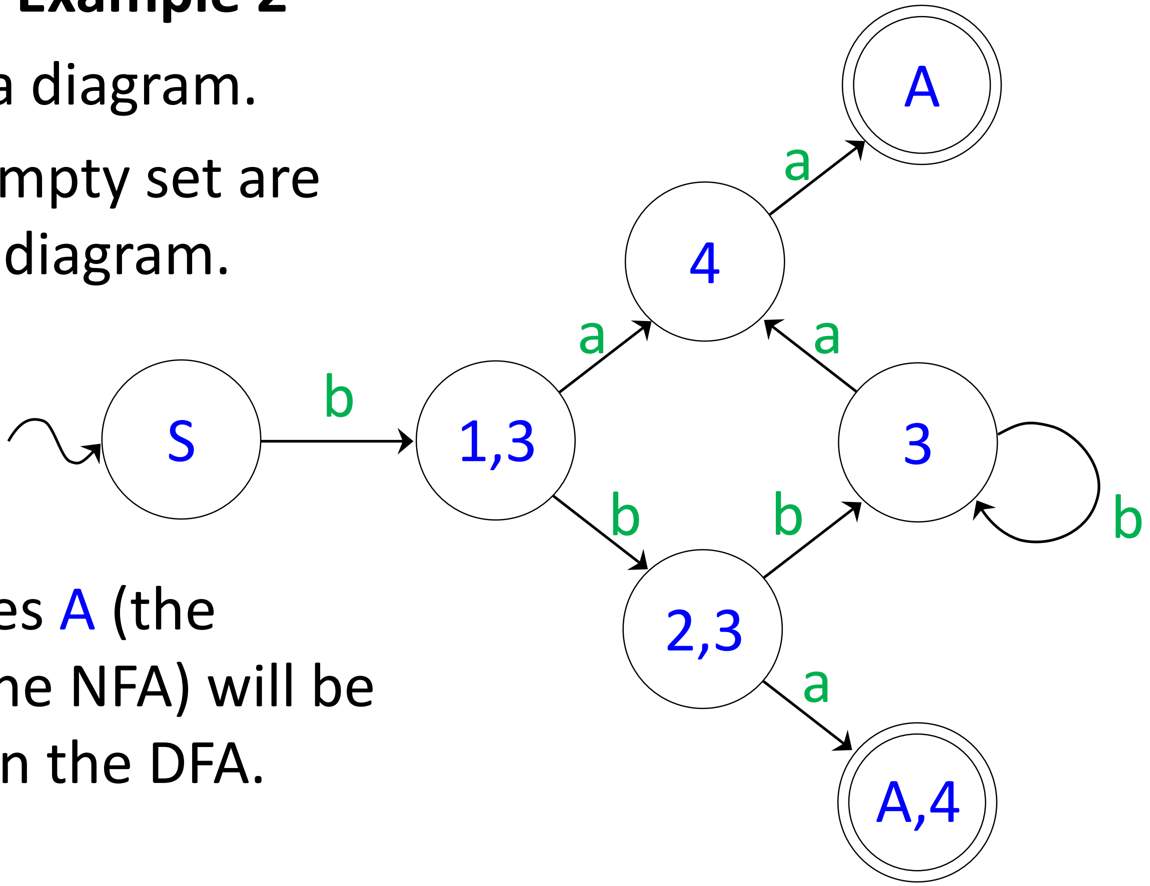
Now create a DFA with seven states using this table.

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.



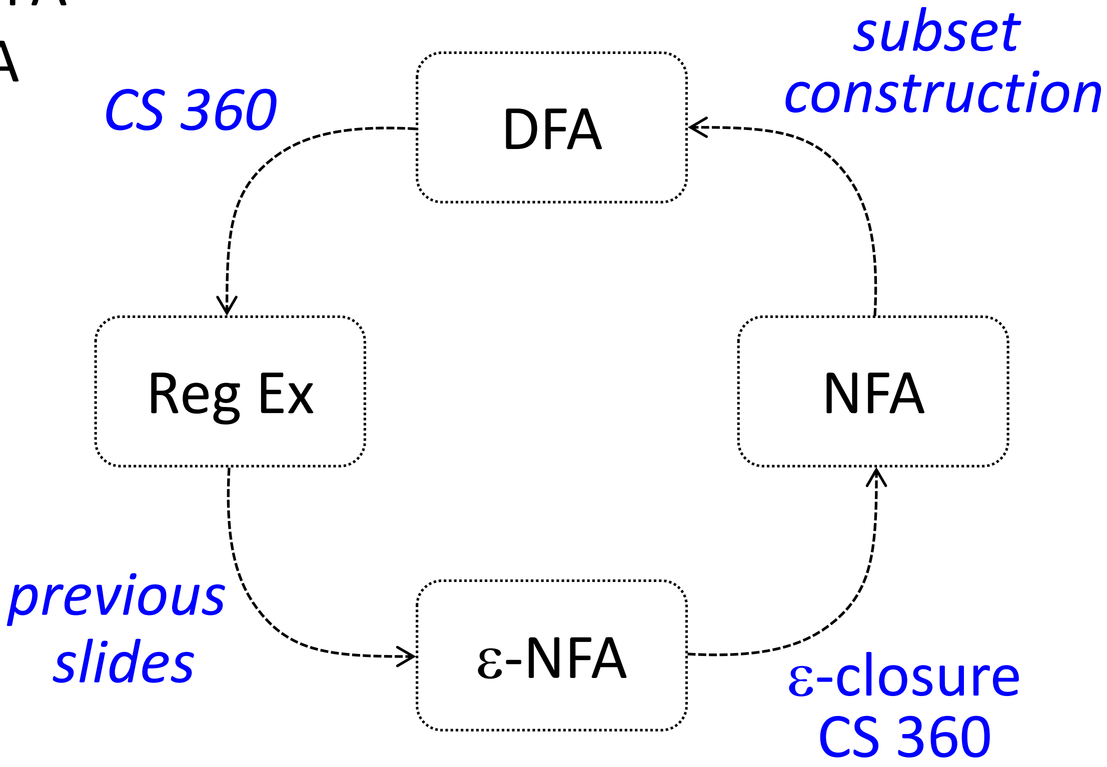
- Any sets the includes A (the accepting state in the NFA) will be an accepting state in the DFA.

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

(( ) (( )))

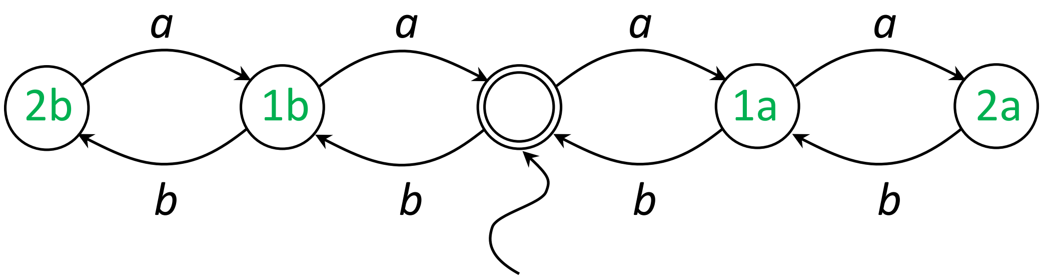
balanced braces

```
{
   {
   }
}
```

Motivation for CFG's

Limitations of Regular Languages

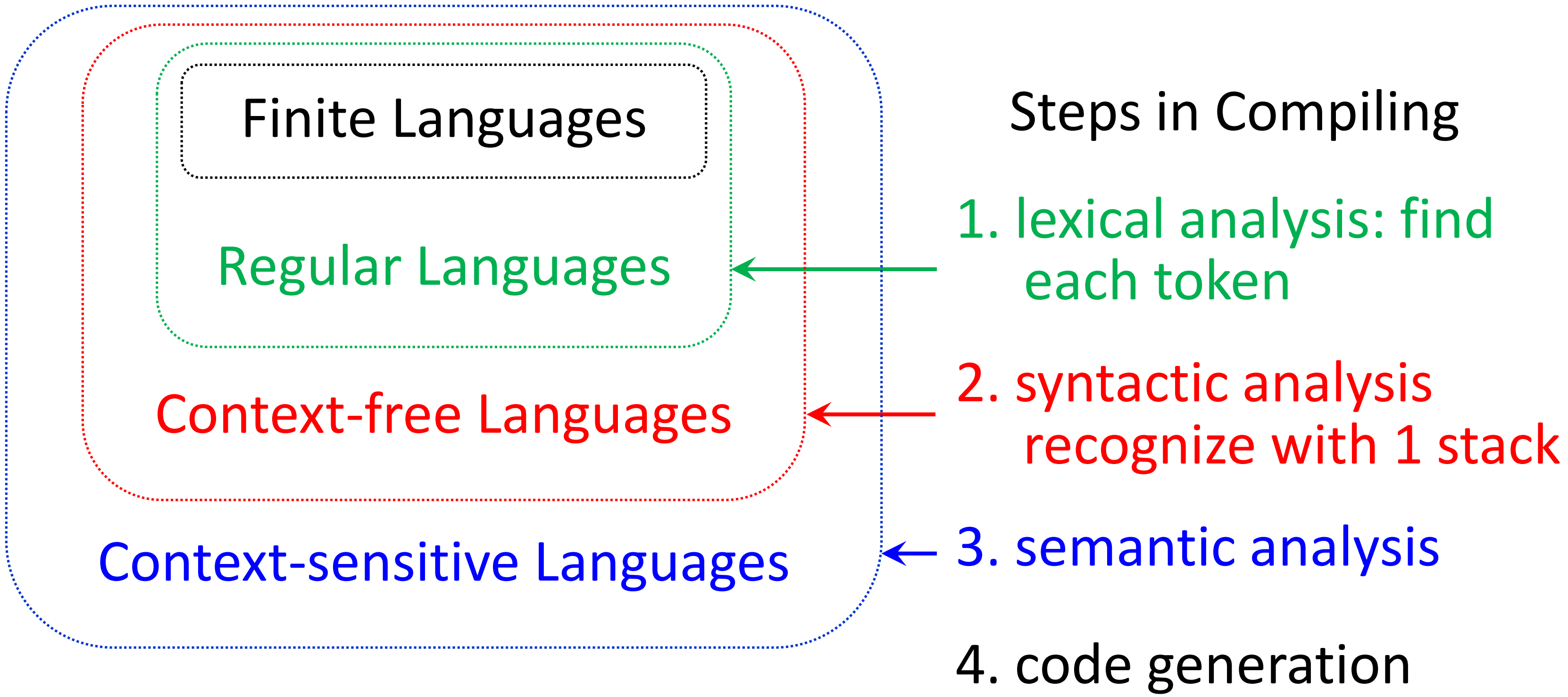
- Create a DFA that recognizes the language $\mathcal{L}$ = {*w*: number of *a*'s in *w* = the number of *b*'s in w} over alphabet Σ ={*a*, *b*}.
- Easy if the difference in the number of *a*'s and *b*'s is fixed, say 2.



- *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



Finite Languages

Regular Languages

Context-free Languages

Context-sensitive Languages

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) $D \rightarrow cD$ // D on both sides of a rule is *recursion*
(4) $D \rightarrow \varepsilon$

- 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 '→' for rules and '⇒' for derivations
- Derivation: $S \underset{1}{\Rightarrow} aSb \underset{2}{\Rightarrow} aDb \underset{3}{\Rightarrow} acDb \underset{3}{\Rightarrow} accDb \underset{4}{\Rightarrow} accb$

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 $D \rightarrow$

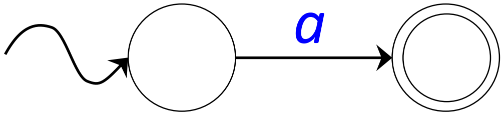
Derivation: $S \underset{1}{\Rightarrow} aSb \underset{2}{\Rightarrow} aDb \underset{3}{\Rightarrow} acDb \underset{3}{\Rightarrow} accDb \underset{4}{\Rightarrow} accb$

- 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

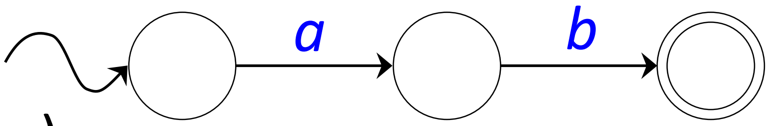
Regular Expressions vs. DFAs vs. Context-free Grammars

- *a*



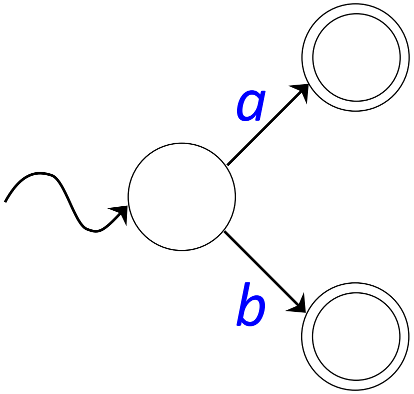
(1) S → *a*

- *ab*
  (concatenation)



(1) S → *ab*

- *a*|*b*
  (union)



(1) S → *a*

(2) S → *b*

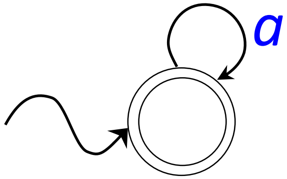
or as

(1) S → *a* | *b*

Example CFGs

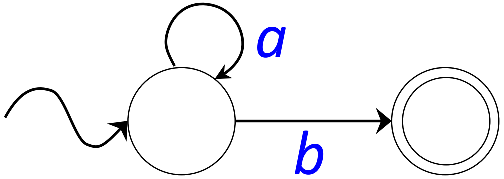
Regular Expressions, DFAs and Context-free Grammars

- a^*



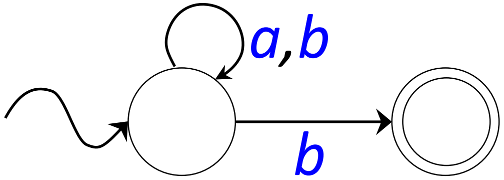
(1) $S \to Sa$
(2) $S \to \varepsilon$

- a^*b



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

- $(a|b)^*b$



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

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

  } *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 → 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	→	*a*S*b*	where A is S and β is aSb
S	→	D	A is S and β is D
D	→	*c*D	A is D and β is *c*D
D	→	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) $D \rightarrow cD$
(4) $D \rightarrow \varepsilon$

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	→	S*b*
	(2)	S	→	D*a*
	(3)	D	→	D*a*
	(4)	D	→	D*b*
	(5)	D	→	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*

$$\begin{array}{llcl} G: & (1) & S \rightarrow & bS \\ & (2) & S \rightarrow & Sb \\ & (3) & S \rightarrow & aSa \\ & (4) & S \rightarrow & \varepsilon \end{array} \left.\vphantom{\begin{array}{l}1\\2\\3\\4\end{array}}\right\}$$

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

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.

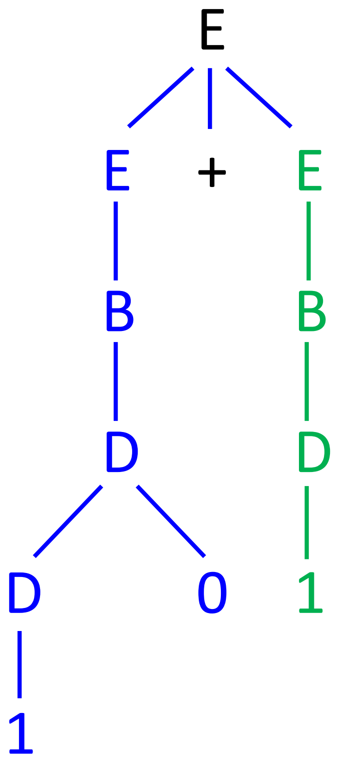
Parse Trees

E ⇒* 10 + 1 Parse Tree

The derivation

E ⇒ E + E using rule E → E + E
⇒ B + E using rule E → B
⇒ D + E using rule B → D
⇒ D0 + E using rule D → D0
⇒ 10 + E using rule D → 1
⇒ 10 + B using rule E → B
⇒ 10 + D using rule B → D
⇒ 10 + 1 using rule D → 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?

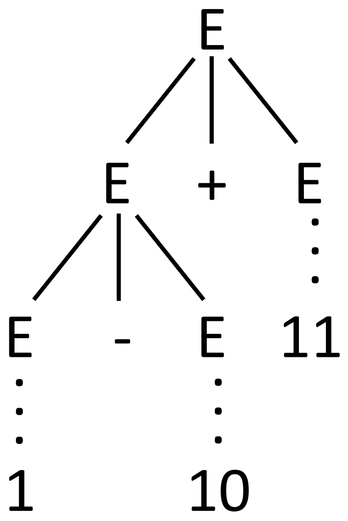
Ambiguous Grammars

Parse Trees for E ⇒* 1 - 10 + 11

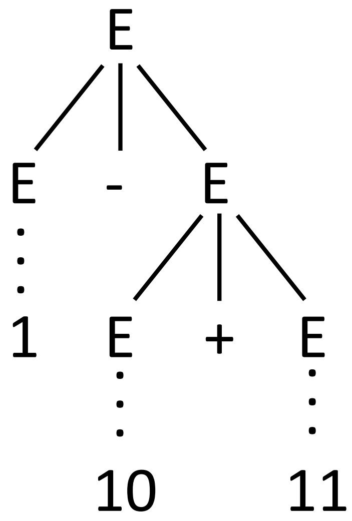
- The same string can have two different parse trees.
- If a grammar can generate at least one string that has two different parse trees, then the grammar is *ambiguous.*

R1 E → E + E
R2 E → 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

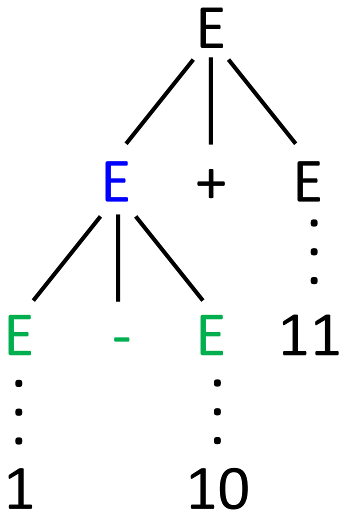
Ambiguous Grammars

Parse Trees for E ⇒* 1 - 10 + 11

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

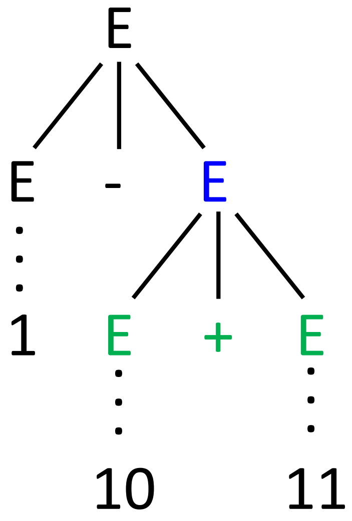
E ⇒ E+E ⇒ E-E+E ⇒ B-E+E ⇒ D-E+E ⇒ 1-E+E ⇒ …

yields this parse tree



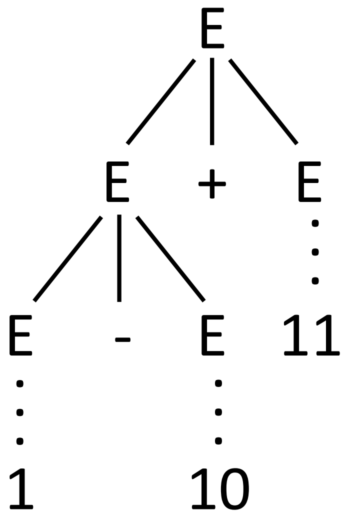
E ⇒ E-E ⇒ B-E ⇒ D-E ⇒ 1-E ⇒ 1-E+E ⇒ 1-B+E ⇒ 1-D+E …

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.*

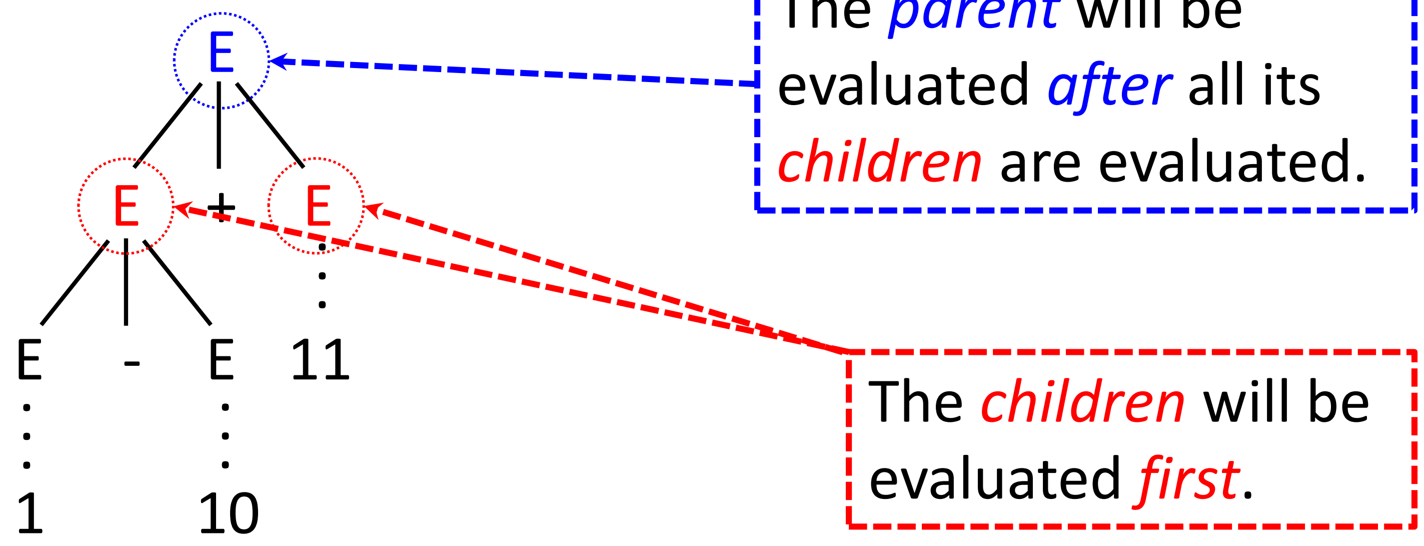
Parse trees are processed using a *post-order depth first* traversal for arithmetic expressions.

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



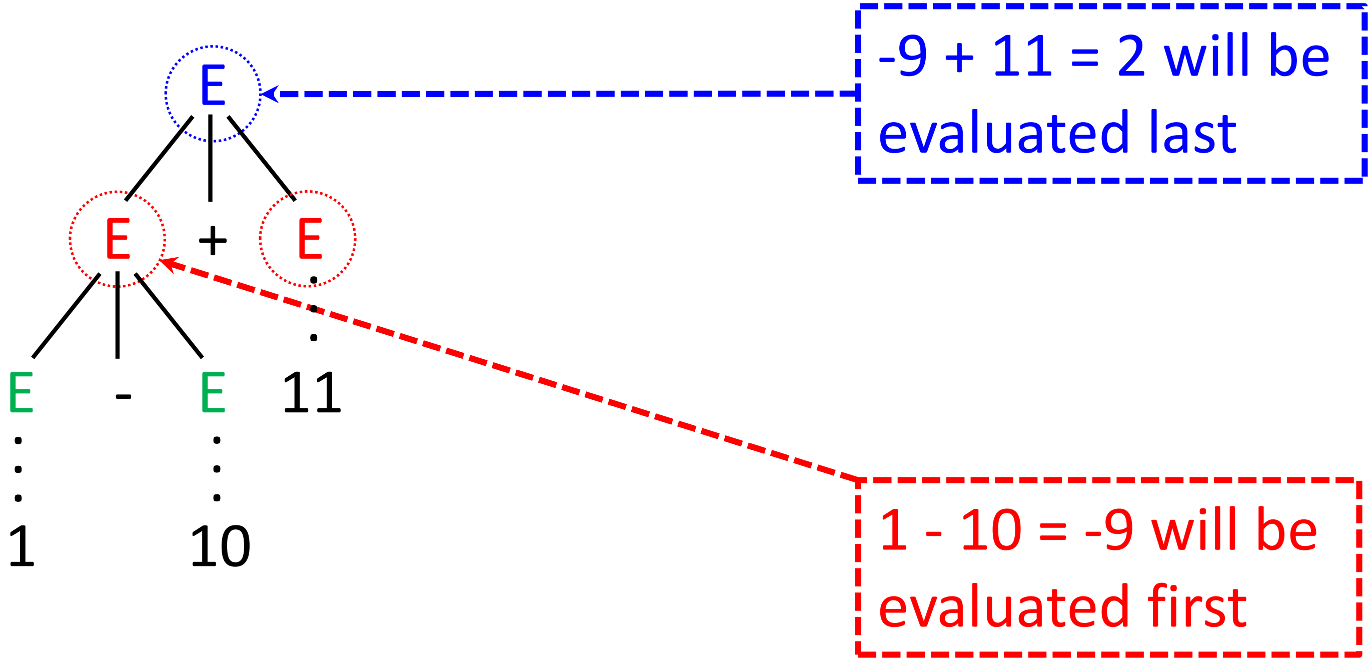
The *parent* will be evaluated *after* all its *children* are evaluated.

The *children* will be evaluated *first*.

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

Processing Order in a Parse Tree

Properties of a Post-Order Traversal



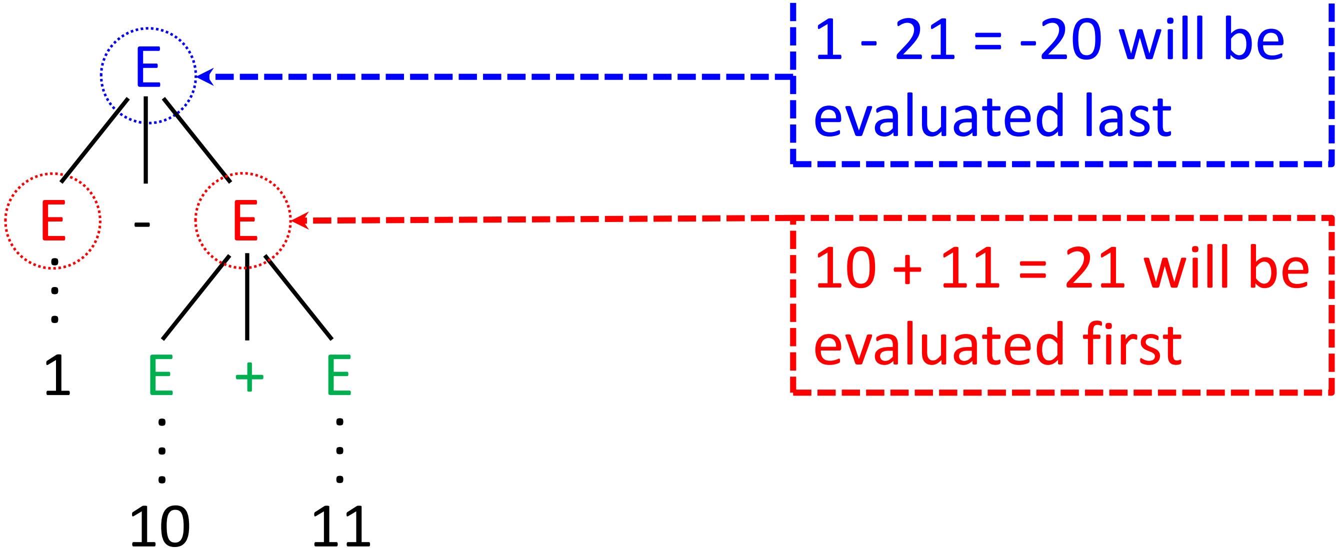
-9 + 11 = 2 will be evaluated last

1 - 10 = -9 will be evaluated first

- 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



1 - 21 = -20 will be evaluated last

10 + 11 = 21 will be evaluated first

- 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 → E + E
    2. E → 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 → ~~E + E~~ B + E	5.	B → D
2.	E → ~~E - E~~ B - E	6.	D → 1
3.	E → B	7.	D → D0
4.	B → 0	8.	D → 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.

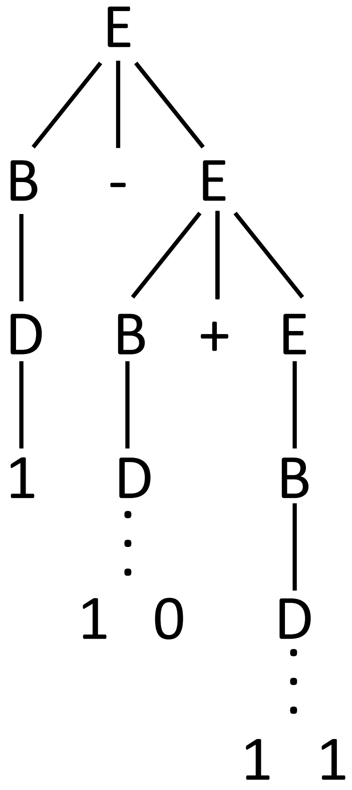
Unambiguous Grammars

Binary Expressions

- Change the first two productions

1.	E → B + E	5.	B → D
2.	E → B - E	6.	D → 1
3.	E → B	7.	D → D0
4.	B → 0	8.	D → 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

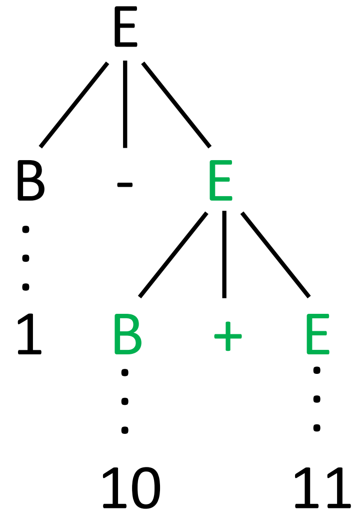
- CFGs can generate *balanced parentheses and implicit order of evaluating expressions* in the absence of parentheses.
- *associativity*: grouping equivalent operations
  - example: 6 - 3 + 4
  - is it read as (6 - 3) + 4 or 6 - (3 + 4)?
  - we want left associativity, i.e. evaluate from left to right (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. $E \rightarrow B + E$
2. $E \rightarrow B - E$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$



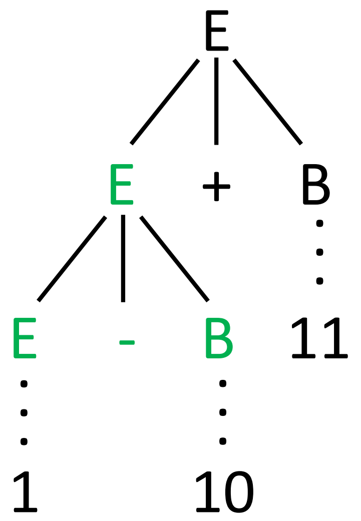
- Consider the tree corresponding to E ⇒ B - E ⇒ B - B + E
- The expression gets longer by adding more operators and digits (i.e. expressions) on the *right* hand side.
- Since *the children get evaluated before the parent*, 10 + 11 will be evaluated before 1 - ( )
- These rules enforce associativity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

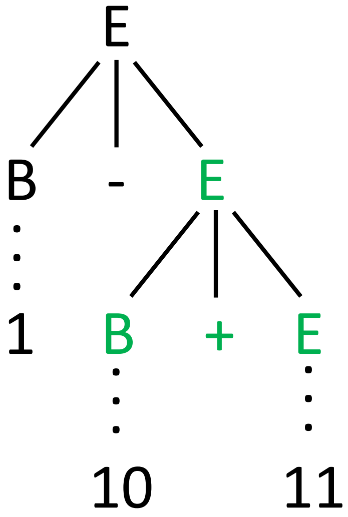
Associativity

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

1. E → B + E
2. E → B - E

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

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

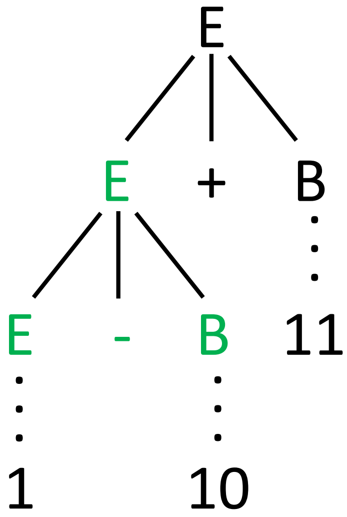


When our grammar is *left recursive,* i.e.

1. E → E + B
2. E → E - B

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

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

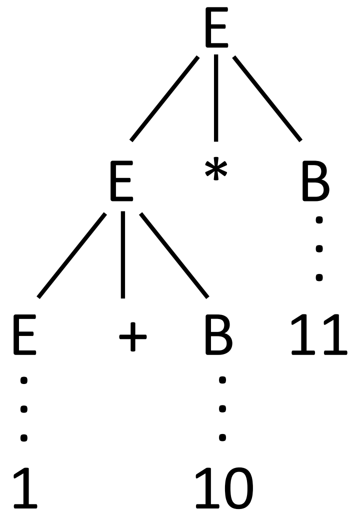


Precedence

Binary Expressions

- Now include multiplication and division.

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



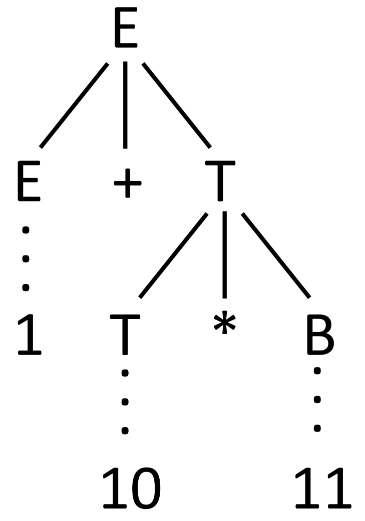
- Consider the derivation E ⇒ E * B ⇒ E + B * B
- This grammar will evaluate the expression 1+10*11 as (1+10)*11 which *ignores the standard rules of precedence.*
- *Idea: have multiplication occur with children of E* (rather than with E itself) by creating a new non-terminal *T*.

Precedence

Binary Expressions

- Introduce a new non-terminal T

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



- Consider the derivation E ⇒ E + T ⇒ 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. $E \rightarrow E + E$
  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. $D \rightarrow D1$

- 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* $A \rightarrow \gamma$ 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 $\rightarrow$ 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 $A \rightarrow \gamma$
- i.e. it doesn’t matter what α and β are, as long as there is a production rule $A \rightarrow \gamma$, 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 → 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 → *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 ∪ {S’}, T ∪ {⊢,⊣}, P ∪ {S’ → ⊢S⊣}, S’}

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

Example: Leftmost derivation of the word ⊢ abywz ⊣

S’	⇒ ⊢ S ⊣	rule (1)
	⇒ ⊢ AyB ⊣	rule (2)
	⇒ ⊢ abyB ⊣	rule (3)
	⇒ ⊢ abywz ⊣	rule (6)

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’ → ⊢S⊣ then pop S’ off the stack and push ⊢ S ⊣ 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 → 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 → ab) and 4 (A → 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 → z or 6 B → 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’ → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → 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: $S' \;{}^{1}\!\Rightarrow \vdash S \dashv \;{}^{2}\!\Rightarrow \vdash AyB \dashv \;{}^{3}\!\Rightarrow \vdash abyB \dashv \;{}^{6}\!\Rightarrow \vdash abywz \dashv$
- 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: |Predict (N, c)| ≤ 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 ⇒* c ⋯

1. S’ → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → 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 ∈ {S', S, A, B} which terminals that can begin a string derived from N, i.e. N ⇒* c···

Row 1: S'

1. S' → ⊢ S ⊣ First (S') = {⊢} by rule 1

Row 2: S

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

Constructing a Predict Table

Helper Function: First( )

Row 3: A

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

Row 4: B

5. $B \rightarrow z$
6. $B \rightarrow wz$ 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 → ε
- Now we can derive:
  $S' \,{}^{1}\!\Rightarrow \vdash S \dashv \,{}^{2}\!\Rightarrow \vdash AyB\dashv \,{}^{3}\!\Rightarrow \vdash abyB\dashv \,{}^{7}\!\Rightarrow \vdash aby\dashv$
- *key point:* ⊣ can appear after the B *but there is no derivation* $B \Rightarrow^{*} \dashv$
- i.e. *using First() is not sufficient*
  - the symbol '⊣' came from rule 1: S' → ⊢ S ⊣
  - the symbol B came from rule 2: S → AyB
  - and B derives ε with rule 7: B → ε
- *conclusion:* ⊣ is in the follow set of B

1. S' → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz
7. B → ε

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’ → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz
7. B → ε

	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 ⊢ abyB ⊣ ⇒ ⊢ aby ⊣
- 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’ ⇒* ⊢abyB⊣
- *Informally:* Follow(N) is the set of terminals c that can follow N in some derivation; that is, S ⇒* ··· Nc ···
- *Formally:* for any non-terminal N, Follow(N) = { c | S’ ⇒* αNcβ}
  - 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 ⇒* ε 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 $a \in w$ {
    while (top of stack is a non-terminal N ) {   // 1st try expand
        if ( Predict(N, a) == ($N \rightarrow \alpha$) )
            pop N
            push $\alpha$ on stack (in reverse)
        else
            reject                                 //  no rule found
    }
    c = pop_stack()                                //  2nd try match
    if ($c \neq a$)
        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 ⊣	⊢	cdy ⊣	> c d y B ⊣	match
6	⊢ cdyB ⊣	⊢ c	dy ⊣	> 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 $S' \to \vdash S \dashv$ 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 $S' \to \cdots B \dashv$
- However $\dashv \in$ Follow(S), there is a rule $S \to ABC$ and Nullable(C) = true.
- More generally if $N \to B_1 B_2 \ldots B_i B_{i+1} \ldots B_n$ 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.

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

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**$(B_i) = \{ c \mid S \Rightarrow^* \alpha B_i c \beta \}$

```
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($B_i$) U 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($B_i$) U 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 $N \to \alpha$ | $c \in \text{First}(\alpha)$ } ∪

  { the rule $N \to \beta$ | $c \in \text{Follow}(N)$ and $\text{Nullable}(\beta)$ = 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 | $\alpha \Rightarrow^* a\beta$}

A: a $\in$ First (A) since A $\overset{3}{\Rightarrow}$ aA

B: b $\in$ First (B) since B $\overset{5}{\Rightarrow}$ bB

S: a $\in$ First (S) since S $\overset{2}{\Rightarrow}$ AB $\Rightarrow$ aAB

b $\in$ First (S) since S $\overset{2}{\Rightarrow}$ AB $\Rightarrow$ B $\Rightarrow$ bB

1. S' → ⊢ S ⊣
2. S → AB
3. A → aA
4. A → ε
5. B → bB
6. B → 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 → $B_1 B_2 \ldots B_{i-1} B_i B_{i+1} \ldots B_n$:

(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' → ⊢ S ⊣
2. S → AB
3. A → aA
4. A → ε
5. B → bB
6. B → ε

S: ⊣ ∈ Follow(S) since S' → ⊢ S ⊣ and ⊣ ∈ First(⊣) by (i)

B: ⊣ ∈ Follow(B) since S → AB and ⊣ ∈ Follow(S) by (ii)

A: ⊣ ∈ Follow(A) since S → AB, Nullable(B) and ⊣ ∈ Follow(S) by (ii)

b ∈ Follow(A) since S → AB and b ∈ First(B) by (i)

Example of Constructing a Predict Table

The Predict Table

- Let $N \in \{S, A, B\}$ and let $c \in \{a, b, \vdash, \dashv\}$
- 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. $S' \rightarrow \vdash S \dashv$
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 → $B_1 B_2 \ldots B_k$            // check rules
    if (k = 0) or (Nullable($B_1$) = ··· = Nullable($B_k$) = true)
      then Nullable(A) = true
until nothing changes
```

R1 $S' \to \vdash S \dashv$

R2 $S \to b\ S\ d$

R3 $S \to p\ S\ q$

R4 $S \to C$

R5 $C \to c\ C$

R6 $C \to \varepsilon$

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 $B_1B_2\cdots B_k$
  if ($B_i$ is a non-terminal) then                   // $B_i$ is a non-terminal
    answer = answer ∪ First($B_i$)
    if (not Nullable($B_i$)) then break               // go to next rule
  else                                                // $B_i$ is a terminal
    answer = answer ∪ {$B_i$}
    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(A) **for a Non-terminal** A

R1 S’ → ⊢ S ⊣
R2 S → b S d
R3 S → p S q
R4 S → C
R5 C → c C
R6 C → 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

```
for each non-terminal A except S': Follow(A) = { }  // initialize
repeat
    for each rule $A \rightarrow B_1 B_2 \cdots B_k$                // check rules
        for i = 1 … k
            if ($B_i$ is a non-terminal)                  // $B_i$ is a non-terminal
                Follow($B_i$) = Follow($B_i$) ∪ First*($B_{i+1} \cdots B_k$)  // case 1
                if (Nullable($B_{i+1} \cdots B_k$)) then
                    Follow($B_i$) = Follow($B_i$) ∪ Follow(A)      // case 2
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’ → ⊢ S ⊣
R2 S → b S d
R3 S → p S q
R4 S → C
R5 C → c C
R6 C → ε

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 $C \rightarrow \varepsilon$

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 → a b
2. S → 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 → a b
2. S → a c b

LL(1)

G’: 1’. R → a T
2’. T → b
3’. T → 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).

Topic 13 – Bottom-up Parsing

Key Ideas

- limitations of LL(k) parsing
- LL vs. LR Parsing
- LR Operations: shifting, reducing
- using a transducer to parse
- building an LR automaton
- LR Parsing Algorithm
- shift-reduce and reduce-reduce conflicts
- SLR(1) Parsing
- building parse trees bottom-up

References

- *Basics of Compiler Design* by T. Mogensen sections 3.14- 3.15

Non-LL(k) Grammars

A Non-LL(1) Grammar

G: $\mathcal{L} = \{a^n b^m \mid n \geq m \geq 0\}$

- i.e. the number of a’s is greater or equal to the number of b’s
- $\mathcal{L}$ is not LL(*k*) for any *k:* just make the run of a’s larger than *k*

Ambiguous Version of Grammar

G_1: $S \rightarrow \varepsilon$
$S \rightarrow aS$
$S \rightarrow aSb$

e.g. $S \Rightarrow aS \Rightarrow aaSb \Rightarrow aab$
$S \Rightarrow aSb \Rightarrow aaSb \Rightarrow aab$

Unambiguous Version of Grammar

G_2: 1. $S \rightarrow\ \vdash A \dashv$
2. $A \rightarrow aA$
3. $A \rightarrow B$
4. $B \rightarrow aBb$
5. $B \rightarrow \varepsilon$

A generates excess a’s
B generates pairs of a’s and b’s

Non-LL(k) Grammars

E.g. for LL(4) Grammar

Stack	Next 4	Action
> S	⊢aa⊣	expand (1)
> ⊢ A ⊣	⊢aa⊣	match ⊢
> A ⊣	aa⊣	expand (2)
> aA ⊣	aa⊣	match a
> A ⊣	a⊣	expand (2)
> aA ⊣	a⊣	match a
> A ⊣	⊣	expand (3)
> B ⊣	⊣	expand (5)
> ⊣	⊣	match ⊣

G_2: 1. S → ⊢ A ⊣
2. A → aA
3. A → B
4. B → aBb
5. B → ε

- Match the *a's using rule 2*: A → aA and not 4: B → aBb

Non-LL(k) Grammars

E.g. for LL(4) Grammar

Stack	Next 4	Action
> S	⊢ab⊣	expand (1)
> ⊢ A ⊣	⊢ab⊣	match ⊢
> A ⊣	ab⊣	expand (3)
> B ⊣	ab⊣	expand (4)
> aBb ⊣	ab⊣	match a
> Bb ⊣	b⊣	expand (5)
> b ⊣	b⊣	match b
> ⊣	⊣	match ⊣

G_2: 1. S → ⊢ A ⊣
2. A → aA
3. A → B
4. B → aBb
5. B → ε

- Match the *a's using rule 4*: B → aBb and not 2: A → aA

Non-LL(k) Grammars

E.g. for LL(4) Grammar

Stack	Next 4	Action
> S	⊢aaa	expand (1)
> ⊢ A ⊣	⊢aaa	match ⊢
> A ⊣	aaaa	what next???

G_2: 1. S → ⊢ A ⊣
2. A → aA
3. A → B
4. B → aBb
5. B → ε

- Expand by 2 and have aA⊣ on the stack?
- Expand by 3 then 4 and have B⊣ then aBb⊣ on the stack?
- It depends on how many b's are in the input.
  - Is the input aaaa⊣ or aaaabbbb⊣?
  - You don't know. You can only lookahead 4 symbols.
- Increasing the lookahead doesn't help because there is always some input where that size of lookahead is insufficient.

LR Parsing

LL vs. LR

- Recall that a stack in LL/top-down parsing is used in the following way:
  - the derivation progresses from the top of the parse tree (S’) down to the bottom, i.e. a *top-down derivation*
  - current step in derivation = *input processed + stack*
  - the stack is read from *top to bottom*
- For LR/bottom-up parsing, we have
  - the derivation progresses from the bottom of the parse tree up to the top (i.e. S’), i.e. a *bottom-up derivation*
  - current step in derivation: *stack + input to be read*
  - stack is read from *bottom to top*

Sample CFG

Sample Grammar

- Recall our Augmented Grammar

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

Rightmost Derivation

S’	⇒ ⊢ S ⊣	(1)
	⇒ ⊢ AyB ⊣	(2)
	⇒ ⊢ Aywz ⊣	(6)
	⇒ ⊢ abywz ⊣	(3)

- LL parsing is intuitive: *read* from the *left, parse* from the *left*
- For *LR parsing, read* from the *left* and *parse* from the *right*
  - i.e. parse using a rightmost derivation

Recall: Example of *LL(1) Parsing*

LL(1) Parsing

	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 ⊣	⊢	**a**bywz ⊣	> **a** b y B ⊣	match
6	⊢ abyB ⊣	⊢ a	**b**ywz ⊣	> **b** y B ⊣	match
7	⊢ abyB ⊣	⊢ ab	**y**wz ⊣	> **y** B ⊣	match
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	expand (6)
9	⊢ abywz ⊣	⊢ aby	**w**z ⊣	> **w** z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	**z** ⊣	> **z** ⊣	match
11	⊢ abywz ⊣	⊢ abywz	**⊣**	> **⊣**	ACCEPT

Example of *LR Parsing*

LR Parsing

	Derivation	Stack	Read	Input	Action
1	⊦ abywz ⊣	<		⊦ abywz ⊣	shift ⊦
2	⊦ abywz ⊣	⊦ <	⊦	abywz⊣	shift a
3	⊦ abywz ⊣	⊦ a <	⊦ a	bywz⊣	shift b
4	⊦ abywz ⊣	⊦ ab <	⊦ ab	ywz⊣	reduce (3)
5	⊦ Aywz ⊣	⊦ A <	⊦ ab	ywz⊣	shift y
6	⊦ Aywz ⊣	⊦ Ay <	⊦ aby	wz⊣	shift w
7	⊦ Aywz ⊣	⊦ Ayw <	⊦ abyw	z⊣	shift z
8	⊦ Aywz ⊣	⊦ Aywz <	⊦ abywz	⊣	reduce(6)
9	⊦ AyB ⊣	⊦ AyB <	⊦ abywz	⊣	reduce (2)
10	⊦ S ⊣	⊦ S <	⊦ abywz	⊣	shift ⊣
11	⊦ S ⊣	⊦ S ⊣ <	⊦ abywz ⊣		reduce (1)
12	S’	S’ <	⊦ abywz ⊣		ACCEPT

Comparing LL vs. LR Parsing

LL vs. LR

- *Derivation Column*
  - in LL: it goes from S’ to ⊢ abywz ⊣ (i.e. down the parse tree)
  - in LR: it goes from ⊢ abywz ⊣ to S’ (i.e. up the parse tree)
- *Top of the Stack*
  - in LL: the top of the stack is on the left when we read it
  - in LR: the top of the stack is on the right when we read it
- *Terminals in the Stack*
  - in LL: at one stage, the stack had many of the terminals from the beginning of the input on the stack: > a b y B ⊣
  - in LR: at one stage the stack had many of the terminals from the end of the input on the stack: ⊢ A y w z <

LR Parsing

LR Operations

There are two operations in LR Parsing

1. *Shift*
   - move a character from the input file to the stack
   - we’ll also include it in the “Read” column to keep track of what has been read so far.
2. *Reduce*
   - If there is a production rule of the form S → AyB and AyB is on the stack then reduce (i.e. replace) AyB to S
   - this step is the act of applying a production rule to simplify what is on the stack

Bottom-Up Parsing

Parsing the Input

- To start, keep on shifting input onto the stack until you have a match with the right hand side (RHS) of some production rule.

	Derivation	Stack	Read	Input	Action
1	⊢ abywz ⊣	<		⊢ abywz ⊣	shift ⊢
2	⊢ abywz ⊣	⊢ <	⊢	abywz⊣	shift a
3	⊢ abywz ⊣	⊢ a <	⊢ a	bywz⊣	shift b
4	⊢ abywz ⊣	⊢ ab <	⊢ ab	ywz⊣	

- Now there is a match between the top of the stack and the RHS of rule 3, A → ab, so reduce (i.e. replace) what is on the stack, ab, with the left hand side (LHS) of that same rule, i.e. A.

4	⊢ abywz ⊣	⊢ ab <	⊢ ab	ywz⊣	reduce (3)
5	⊢ Aywz ⊣	⊢ A <	⊢ ab	ywz⊣	

Bottom-Up Parsing

Parsing the Input

- Again, keep on shifting input onto the stack until you have a match with the RHS of some production rule.

	Derivation	Stack	Read	Input	Action
5	⊢ Aywz ⊣	⊢ A <	⊢ ab	ywz⊣	shift y
6	⊢ Aywz ⊣	⊢ A y <	⊢ aby	wz⊣	shift w
7	⊢ Aywz ⊣	⊢ Ayw <	⊢ abyw	z⊣	shift z
8	⊢ Aywz ⊣	⊢ Aywz <	⊢ abywz	⊣	

- Now there is a match between the top of the stack and the RHS of rule 6, B ⟶ wz, so reduce wz to the LHS of rule 6, i.e. B.

8	⊢ Aywz ⊣	⊢ Aywz <	⊢ abywz	⊣	reduce(6)
9	⊢ AyB ⊣	⊢ AyB <	⊢ abywz	⊣	

Bottom-Up Parsing

Parsing the Input

- After that reduction there is yet another match with the RHS of a production rule, so there is no need to shift.

	Derivation	Stack	Read	Input	Action
9	⊢ AyB ⊣	⊢ AyB <	⊢ abywz	⊣	

- There is a match between the top of the stack and the RHS of rule 2, S → AyB, so reduce AyB to the LHS of rule 2, i.e. S.

9	⊢ AyB ⊣	⊢ AyB <	⊢ abywz	⊣	reduce (2)
10	⊢ S ⊣	⊢ S <	⊢ abywz	⊣	

- Again, keep on shifting input onto the stack until you have a match with the RHS of some production rule.

Bottom-Up Parsing

Parsing the Input

	Derivation	Stack	Read	Input	Action
10	⊦ S ⊣	⊦ S <	⊦ abywz	⊣	shift ⊣
11	⊦ S ⊣	⊦ S ⊣ <	⊦ abywz ⊣		

- There is a match between what is on the stack and the RHS of rule 1, S’ → ⊦ S ⊣, so reduce ⊦ S ⊣ to S’ using rule 1.

11	⊦ S ⊣	⊦ S ⊣ <	⊦ abywz ⊣		reduce (1)
12	S’	S’ <	⊦ abywz ⊣		ACCEPT

- The start symbol, S’, is now the only symbol on the stack so the input ⊦ abywz ⊣ has been derived from S’ and so it is a string in the language generated by the grammar.

Shift / Reduce

When to Shift, When to Reduce

- *Key Question:* How do you know when to shift and when to reduce?
  - for LL(1) parsing, we have a predictor table
  - for LR parsing, we have a transducer
    - i.e. a DFA that recognizes strings and may produce output during a transition from one state to another
    - you will need to review / recall transducers for the next assignment
- In 1965 Donald Knuth proved a theorem that we can construct a DFA (really, a transducer) for LR grammars

Shift / Reduce

When to Shift, When to Reduce

- *Key Question:* How do you know when to shift and when to reduce?
- *Key Idea:* Introduce the symbol "•" as a place holder to help keep track of where we are in the RHS of a production rule, e.g.

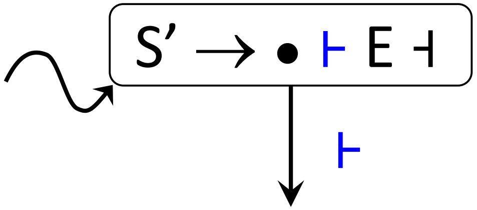
  S' → • ⊢ E ⊣
  S' → ⊢ • E ⊣
  S' → ⊢ E • ⊣
  S' → ⊢ E ⊣ •

- We *create a finite automaton* to track the progress of the placeholder through the various production rules
- How to build the automaton: there is a *different state each time the place holder moves over one symbol* in the production rule.

Building an LR(0) automaton

Sample Grammar

G: 1. S’ → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*



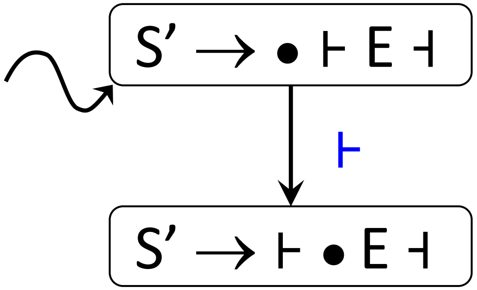
S’ → • ⊢ E ⊣

- *Start state:* make the start state the first rule, with a dot (•) in front of the leftmost symbol of the RHS, e.g. S’ → • ⊢ E ⊣
- *For each state:* create a transition out of that state with the symbol that follows the “•”
- Here the BOF symbol “⊢” follows the “•” so have a transition out of the start state labelled ⊢ and move the “•” symbol forward one character in that rule.

Building an LR(0) automaton

Sample Grammar

G: 1. S’ → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*

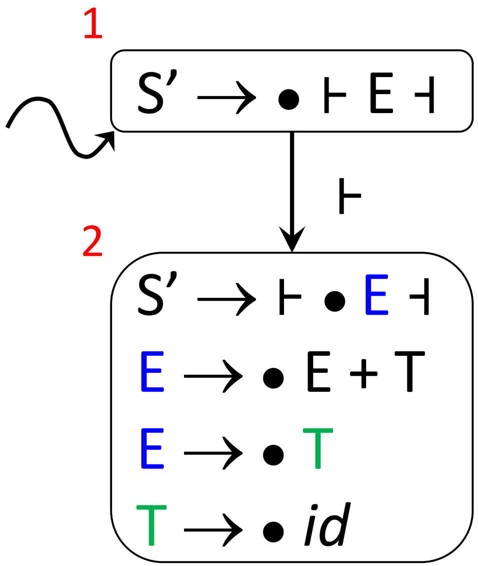


- Here the RHS of the start state is “• ⊢ E ⊣”
- Advancing the “•” forward by one character creates the new state “S’ → ⊢ • E ⊣”
- This transition is saying with input ⊢ the automaton will advance from state “S’ → • ⊢ E ⊣” to state “S’ → ⊢ • E ⊣”
- A rule with a “•” somewhere on the RHS is called an *item*. It indicates a partially completed rule.

Building an LR(0) automaton

Sample Grammar

G: 1. $S' \to \vdash E \dashv$
2. $E \to E + T$
3. $E \to T$
4. $T \to id$



1

$S' \to \bullet \vdash E \dashv$

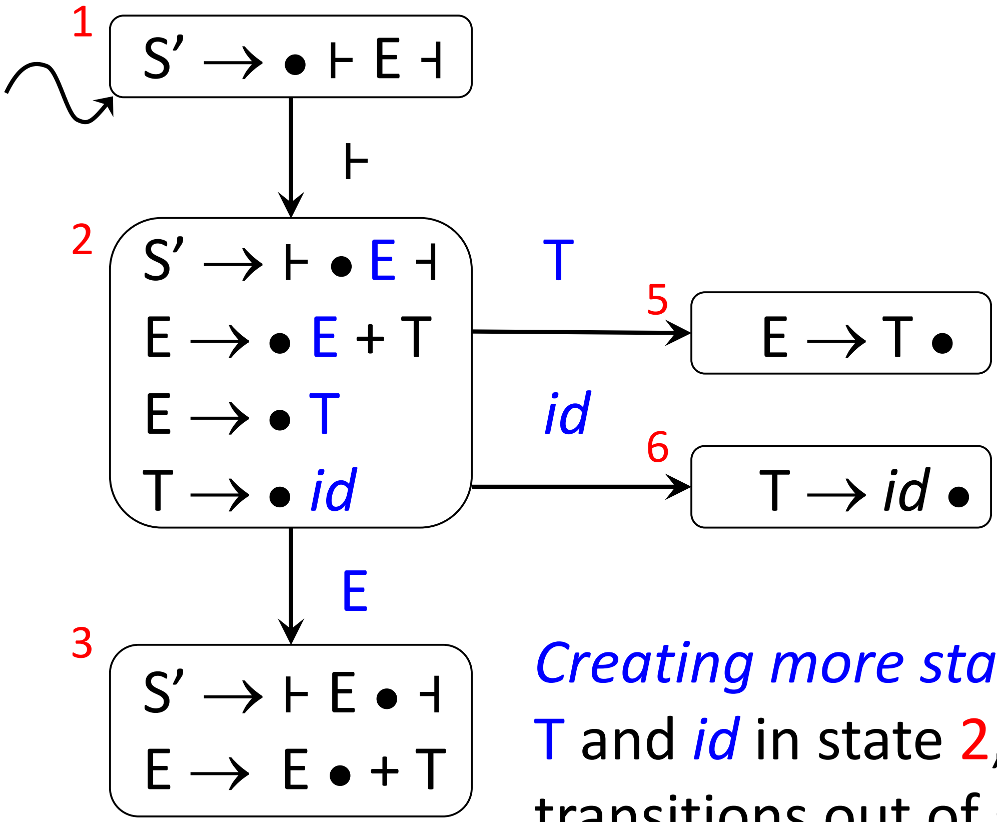
$\vdash$

2

$S' \to \vdash \bullet E \dashv$
$E \to \bullet E + T$
$E \to \bullet T$
$T \to \bullet id$

- *For non-terminals:* If "•" precedes a non-terminal (in this case E) add all productions with that non-terminal on the LHS to the current state (and place the "•" in the leftmost position of these rules).
- E.g. In state 2, "•" precedes the non-terminal E, so add all the rules that have E on the LHS, i.e. $E \to E + T$ and $E \to T$
- Now "•" also proceeds T, so add all the rules with T on the LHS as well, i.e. $T \to id$

Building an LR(0) automaton



Sample Grammar

G: 1. S’ → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*

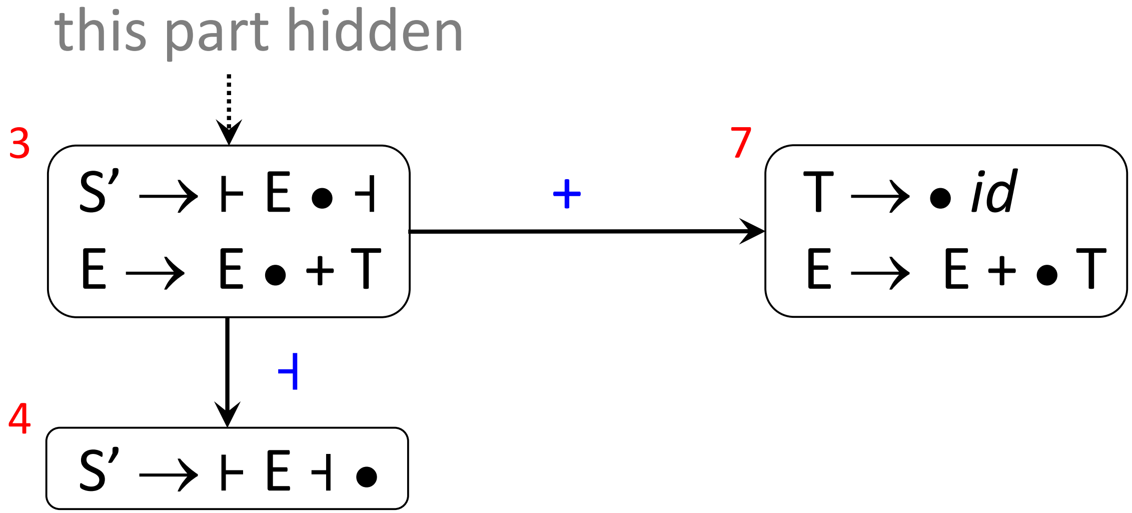
Creating more states: Since the “•” precedes E, T and *id* in state 2, there will be three transitions out of state 2, labelled E (to state 3), T (to state 5) and *id* (to state 6). In each new state, the “•” will move forward one symbol.

Building an LR(0) automaton

Creating more states: Since the "•" precedes ⊣ and + in state 3, there will be two transitions out of state 3, labelled ⊣ (to state 4) and + (to state 7). In each of these two new states, the "•" will move forward one symbol.

Sample Grammar

G: 1. S' → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*

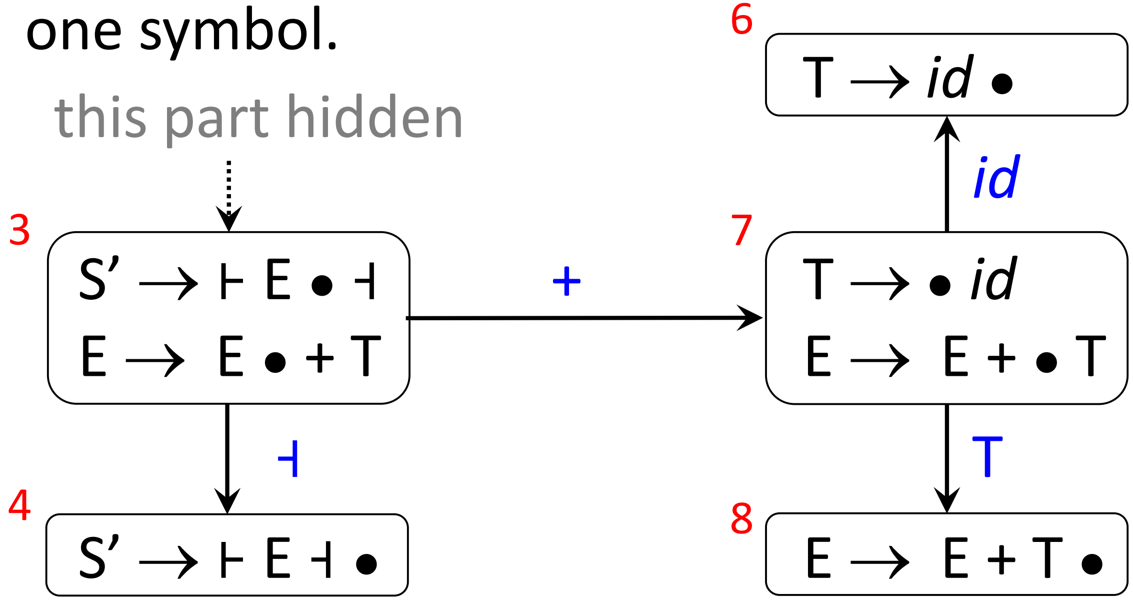


Building an LR(0) automaton

Creating more states: Since the "•" precedes *id* and T state 7, there will be two transitions out of state 7, one labelled *id*, to the already existing state 6, and the other labelled T (to a new state 8). In each of these states, the "•" will move forward one symbol.

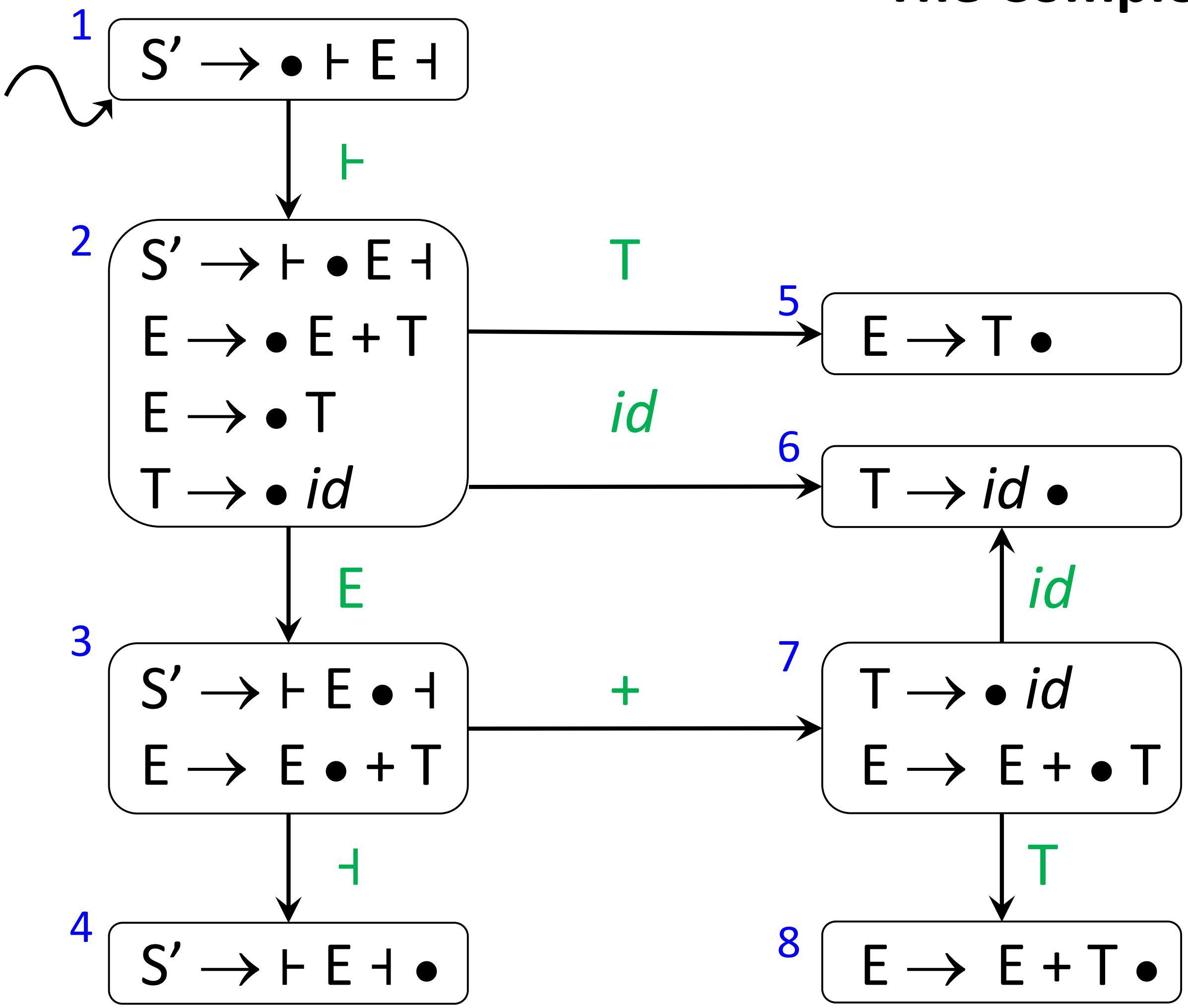
Sample Grammar

G: 1. S' → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*



Building an LR(0) automaton

The Complete LR(0) Automaton



1

S′ → • ⊢ E ⊣

⊢

2

S′ → ⊢ • E ⊣

E → • E + T

E → • T

T → • *id*

T

5

E → T •

id

6

T → *id* •

E

3

S′ → ⊢ E • ⊣

E → E • + T

+

7

T → • *id*

E → E + • T

id

⊣

4

S′ → ⊢ E ⊣ •

T

8

E → E + T •

G: 1. S′ → ⊢ E ⊣
2. E → E + T
3. E → T
4. T → *id*

We will be moving to blue for states and green for terminals and non-terminals.

LR(0) Parsing Algorithm

```
shift ⊢ onto the symbol_stack                                  //initialize
push $\delta(q_0, \vdash)$ onto the state_stack
for each token a in the input {
    while (current state has 1 item: reduction $A \rightarrow \gamma\bullet$ ) {   // reduce
        pop $|\gamma|$ symbols off the symbol_stack
        pop $|\gamma|$ states off the state_stack
        push A on the symbol_stack
        push $\delta$(state_stack.top, A) onto the state_stack
    }
    shift a onto the symbol_stack                               // shift
    if ($\delta$(state_stack.top, a) == undefined) report parse error
    else push $\delta$(state_stack.top, a) onto the state_stack
}
if (⊣ has been shifted, i.e. ⊢ S ⊣ is on the symbol_stack )
    then ACCEPT
```

E.g. states 4, 5, 6 and 8 on the previous slide

LR(0) Parser

LR(0) Parsing Algorithm

- Lines 1-2: *initialize* the automaton by taking the first transition
  - the automaton is always in the state state_stack.top
  - **push** δ(q0, ⊢) onto the state_stack means take the transition for input ⊢ from the start state, q0
- Lines 4-9: first try to *reduce*
- Lines 10-12: only *shift* input after any potential reductions have been performed
- Lines 14-15: accept when you have shifted ⊣ because that means the input has been derived from the original start symbol S (or E in the next example).
- The algorithm uses two stacks that are always kept in synch.

Using the LR(0) Automaton

Stacks

- (1) a symbol_stack to track the symbols from the input stream and (2) a state_stack to track the states that the automaton has been in.
- each time a symbol is pushed on or popped off the symbol_stack, a state is pushed on or popped off the state_stack.

Sample Grammar

G: 1. $S' \rightarrow \vdash E \dashv$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow \mathit{id}$

- *Task:* Use the grammar G and the automaton to parse the input $\vdash \mathit{id}+\mathit{id} \dashv$

Using the LR(0) Automaton

Simulation

	Symbol Stack	**State Stack**	**Input Read**	**Unread Input**	**Action**
1		1		⊢ *id*+*id* ⊣	shift ⊢
2	⊢	1 2	⊢	*id*+*id*⊣	shift *id*
3	⊢ *id*	1 2 6	⊢ *id*	+*id* ⊣	reduce *id*

1. Start in q_0 (i.e. state 1) and shift ⊢, i.e. push ⊢ onto the symbol_stack (line 1 of algorithm)

   Move to state δ(1, ⊢) = 2 in the automaton, i.e. push 2 onto the state_stack, now: 1 2 (line 2)

2. Shift: push *id* onto the symbol_stack and move to state δ(2, *id*) = 6 in the automaton, i.e. push 6 onto the state_stack, now: 1 2 6 (lines 10-11)

Using the LR(0) Automaton

	Symbol Stack	State Stack	Input Read	Unread Input	Action
3	⊢ *id*	1 2 6	⊢ *id*	+*id* ⊣	reduce *id*
4	⊢ T	1 2 5	⊢ *id*	+*id* ⊣	reduce T

3. Reduce: State 6 has only one item in it (T → *id* •) and "•" is the rightmost symbol, so reduce *id* by rule 4 (lines 4-9 in the algorithm)
   - pop *id* off of the symbol_stack , now: ⊢
   - pop |*id*| = 1 state off of the state_stack, now: 1 2
   - push the LHS of the rule you have just reduced (the rule was T → *id* •, so push T) onto the symbol_stack, now: ⊢ T
   - move to state δ(2, T) = 5 in the automaton, i.e. push 5 onto the state_stack, now: 1 2 5

Using the LR(0) Automaton

	Symbol Stack	**State Stack**	**Input Read**	**Unread Input**	**Action**
4	⊢ T	1 2 5	⊢ *id*	+*id* ⊣	reduce T
5	⊢ E	1 2 3	⊢ *id*	+*id* ⊣	shift +

4. Reduce: State 5 has only one item in it (E → T •) and "•" is the rightmost symbol, so reduce T (lines 4-9 in the algorithm)
   - pop T off of the symbol_stack , now: ⊢
   - pop |T| = 1 state off of the state_stack, now: 1 2
   - push the LHS of the rule you have just reduced (the rule was E → T •, so push E onto the symbol_stack, now: ⊢ E
   - move to state δ(2, E) = 3 in the automaton, i.e. push 3 onto the state_stack, now: 1 2 3

Using the LR(0) Automaton

	Symbol Stack	**State Stack**	**Input Read**	**Unread Input**	**Action**
5	⊢ E	1 2 3	⊢ *id*	+*id* ⊣	shift +
6	⊢ E +	1 2 3 7	⊢ *id*+	*id* ⊣	shift *id*
7	⊢ E + *id*	1 2 3 7 6	⊢ *id*+*id*	⊣	reduce *id*

5. Shift: push + onto the symbol_stack, now: ⊢ E + and move to state δ(3, +) = 7 in the automaton, i.e. push 7 onto the state_stack, now: 1 2 3 7 (lines 10-11).
6. Shift: push *id* onto the symbol_stack, now: ⊢ E + *id* and move to state δ(7, *id*) = 6 in the automaton, i.e. push 6 onto the state_stack, now: 1 2 3 7 6 (lines 10-11).
7. Reduce: State 6 has only one item in it (T → *id* •) and "•" is the rightmost symbol, so reduce *id* (lines 4-9 in the algorithm).

Using the LR(0) Automaton

	Symbol Stack	State Stack	Input Read	Unread Input	Action
7	⊢ E + *id*	1 2 3 7 6	⊢ *id*+*id*	⊣	reduce *id*
8	⊢ E + T	1 2 3 7 8	⊢ *id*+*id*	⊣	

- pop *id* off of the symbol_stack, now: ⊢ E +
- pop |*id*| = 1 state off of the state_stack, now: 1 2 3 7
- push the LHS of the rule (T → *id* •, so push T) onto the symbol_stack, now: ⊢ E + T
- move to state δ(7, T) = 8 in the automaton, i.e. push 8 onto the state_stack, now: 1 2 3 7 8

8. Reduce: State 8 has only one item in it (E → E + T •) and "•" is the rightmost symbol, so reduce (lines 4-9 in the algorithm).
    - pop E + T off of the symbol_stack, now: ⊢

Using the LR(0) Automaton

	Symbol Stack	State Stack	Input Read	Unread Input	Action
8	⊢ E + T	1 2 3 7 8	⊢ *id*+*id*	⊣	reduce E + T
9	⊢ E	1 2 3	⊢ *id*+*id*	⊣	shift ⊣
10	⊢ E ⊣	1 2 3 4	⊢ *id*+*id* ⊣		reduce ⊢ E ⊣

- pop |E + T| = 3 states off of the state_stack, now: 1 2
- push the LHS of the rule (E → E + T •, so push E) onto the symbol_stack, now: ⊢ E
- move to state δ(2, E) = 3 in the automaton, i.e. push 3 onto the state_stack, now: 1 2 3

9. Shift: push ⊣ onto the symbol_stack, now: ⊢ E ⊣ and move to state δ(3, ⊣) = 4 in the automaton, i.e. push 4 onto the state_stack, now: 1 2 3 4 (lines 10-11).

Using the LR(0) Automaton

	Symbol Stack	State Stack	Input Read	Unread Input	Action
10	⊢ E ⊣	1 2 3 4	⊢ *id*+*id* ⊣		reduce ⊢ E ⊣
11	S’	1	⊢ *id*+*id* ⊣		ACCEPT

10. Reduce: use rule 1, S’ → ⊢ E ⊣

- pop ⊢ E ⊣ off of the symbol_stack, now: ε
- pop |⊢ E ⊣| = 3 states off of the state_stack, now: 1
- push the LHS of the rule (S’ → ⊢ E ⊣ •, so push S’) onto the symbol_stack, now: S’

11. S’ is on the stack so ACCEPT

The next two slides illustrate the entire parsing of ⊢ *id*+*id*⊣ …

Using the LR(0) Automaton

	Symbol Stack	State Stack	Input Read	Unread Input	Action
1		1		⊢ *id+id* ⊣	shift ⊢
2	⊢	1 2	⊢	*id+id*⊣	shift *id*
3	⊢ *id*	1 2 6	⊢ *id*	+*id* ⊣	reduce *id*
4	⊢ T	1 2 5	⊢ *id*	+*id* ⊣	reduce T
5	⊢ E	1 2 3	⊢ *id*	*+id* ⊣	shift +
6	⊢ E +	1 2 3 7	⊢ *id*+	*id* ⊣	shift *id*
7	⊢ E + *id*	1 2 3 7 6	⊢ *id+id*	⊣	reduce *id*
8	⊢ E + T	1 2 3 7 8	⊢ *id+id*	⊣	reduce E + T
9	⊢ E	1 2 3	⊢ *id+id*	⊣	shift ⊣
10	⊢ E ⊣	1 2 3 4	⊢ *id+id* ⊣		reduce ⊢ E ⊣
11	S’	1	⊢ *id+id* ⊣		ACCEPT

LR Parsing Limitations

Reducing the Time Complexity

- Observation: you can recreate the state stack from the symbol stack, e.g. for line 7 if you process the input ⊢ E + *id* with the automaton you will go through states 1 2 3 7 6.

	Symbol Stack	**States Stack**	**Input Read**	**Unread Input**	**Action**
7	⊢ E + *id*	1 2 3 7 6	⊢ *id*+*id*	+*id* ⊣	reduce *id*

- This situation is an invariant for the parsing algorithm.
- So why have a State Stack?
  - To reduce the time complexity from O(n^2) to O(n).
  - If the symbol stack had *n* symbols, you would move through the stack and automaton *n* steps to go to the next state.

LR Parsing Limitations: Conflicts

However

- The LR(0) parsing algorithm does have some limitations …

Shift-Reduce Conflict

- Problem 1: What if the state looks like this?

$$\boxed{\begin{array}{l} A \to \alpha\bullet c\beta \\ B \to \gamma\bullet \end{array}}$$

- Question: Do we …
  - *shift* the next character c (as suggested by $A \to \alpha\bullet c\beta$) or
  - *reduce* γ to B (as suggested by $B \to \gamma\bullet$)?
- This situation is known as a *shift-reduce conflict*. i.e. when a state has both a shift and a reduction in it.

LR Parsing Limitations: Conflicts

Reduce-Reduce Conflict

- Problem 2: What if the state looks like this?

$$\boxed{\begin{array}{l} A \rightarrow \alpha\bullet \\ B \rightarrow \beta\bullet \end{array}}$$

- Question: Do we …
  - *reduce* α to A (as suggested by $A \rightarrow \alpha\bullet$) or
  - *reduce* β to B (as suggested by $B \rightarrow \beta\bullet$)?
- This is known as a *reduce-reduce conflict*.

Causes of Conflicts

- If any item $A \rightarrow \alpha\bullet$ (i.e. the placeholder is at the end) occurs in a state in which *it is not alone* then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0).

LR Parsing Limitations: Conflicts

Sample Grammar with Conflict

- Consider right-associative expressions. Modify our previous grammar slightly (i.e. reverse RHS of second rule):

G_R: 1. $S' \to \vdash E \dashv$

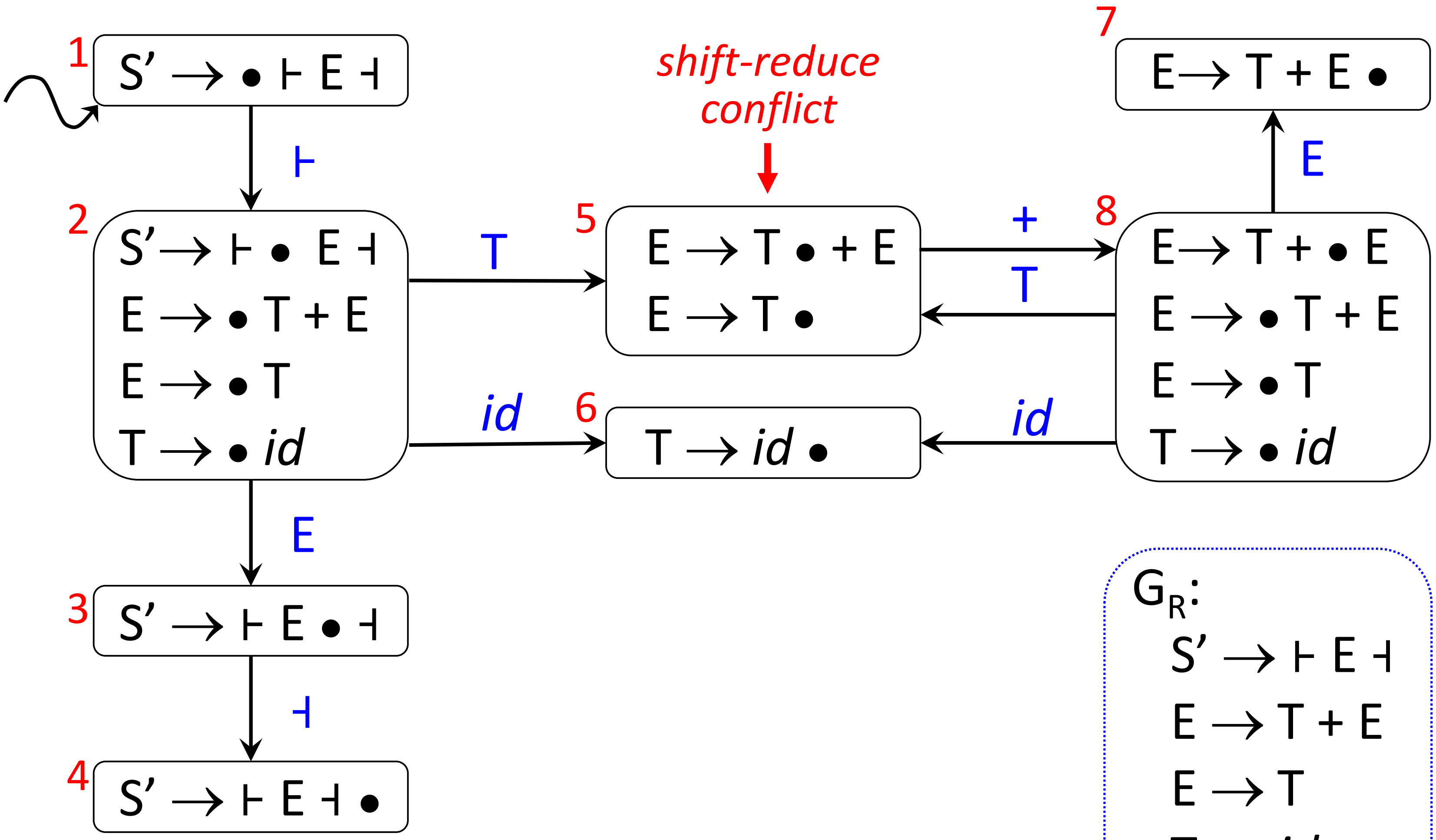
2. $E \to T + E$ (was $E \to E + T$)

3. $E \to T$

4. $T \to \mathit{id}$

- Now build an automaton based on this modified grammar.

Conflicts: New LR(0) automaton



G_R:

S' → ⊢ E ⊣

E → T + E

E → T

T → *id*

LR Parsing Limitations: Conflicts

Sample Conflict

- Input starts with ⊢ *id* …
- Consider the stack (initially empty)

$$\boxed{} \xrightarrow{\text{shift } \vdash} \boxed{\vdash } \xrightarrow{\text{shift } id} \boxed{\vdash id} \xrightarrow{\text{reduce } id} \boxed{\vdash T}$$

- Should we now reduce T to E (i.e. use rule E → T)?
- Answer: it depends
  - If the input is ⊢ *id* ⊣ then YES.
  - If the input is ⊢ *id* + … ⊣ then NO. Keep shifting to get T + E and then reduce using rule E → T + E instead

G_R:

S’ → ⊢ E ⊣
E → T + E
E → T
T → *id*

Resolving Conflicts

Sample Conflict

- Solution: *add a lookahead token* to the automaton to resolve the conflict
- For each $A \rightarrow \alpha$, attach Follow(A), e.g.
  - Follow(E) = { ⊣ }
  - Follow(T) = { +, ⊣ }

G_R:

$S' \rightarrow \vdash E \dashv$

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id$

$E \rightarrow T \bullet$
$E \rightarrow T \bullet + E$

becomes

$E \rightarrow T \bullet \quad \{\dashv\}$
$E \rightarrow T \bullet + E$

- Interpretation: the reduce action $\boxed{A \rightarrow \alpha \bullet \quad \{X\}}$ *applies only if the next token is* X, where X=Follow(A).
- $E \rightarrow T \bullet \quad \{\dashv\}$ applies when the next token is "⊣"
- $E \rightarrow T \bullet + E$ applies when the next token is "+"

SLR(1) Parser

SLR(1) Parsing

- When we add one character of lookahead, we have an *SLR(1)* (Simple LR with 1 character lookahead) parser
- We modify our existing LR(0) automaton as follows
  - When you are in a state that has a rule of the form $A \rightarrow \alpha \bullet \{X\}$ (which calls for a reduction) if the next symbol is X reduce using that rule, otherwise shift.
- Allowing for lookahead, we now have the following algorithm (which is the similar to LR)
- The *only difference is line 4* i.e. use the lookahead to decide whether to reduce or not.

SLR(1) Parsing Algorithm

```
shift ⊢ onto the symbol_stack                                        // initialize
push $\delta(q_0, ⊢)$ onto the state_stack
for each token a in the input {
    while (there is a reduction  $A \to \gamma\bullet$  {a} in state_stack.top ) {
        pop |$\gamma$| symbols off the symbol_stack                  // reduce
        pop |$\gamma$| states off the state_stack
        push A on the symbol_stack
        push $\delta$(state_stack.top, A) onto the state_stack
    }
    shift a onto the symbol_stack                                    // shift
    if ($\delta$(state_stack.top, a) == undefined) report parse error
    else push $\delta$(state_stack.top, a) onto the state_stack
}
if (⊣ has been shifted, i.e. ⊢ S ⊣ is on the symbol_stack )
    then ACCEPT
```

LR Parsing

Outputting a Rightmost Derivation

- *Idea:* each time a reduction is done, output the rule that was used.
- *Modification:* since LR parsing is bottom-up, list the rules in reverse order.
- For our ⊢ abywz ⊣ derivations, it would be rules 1, 2, 6, 3

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

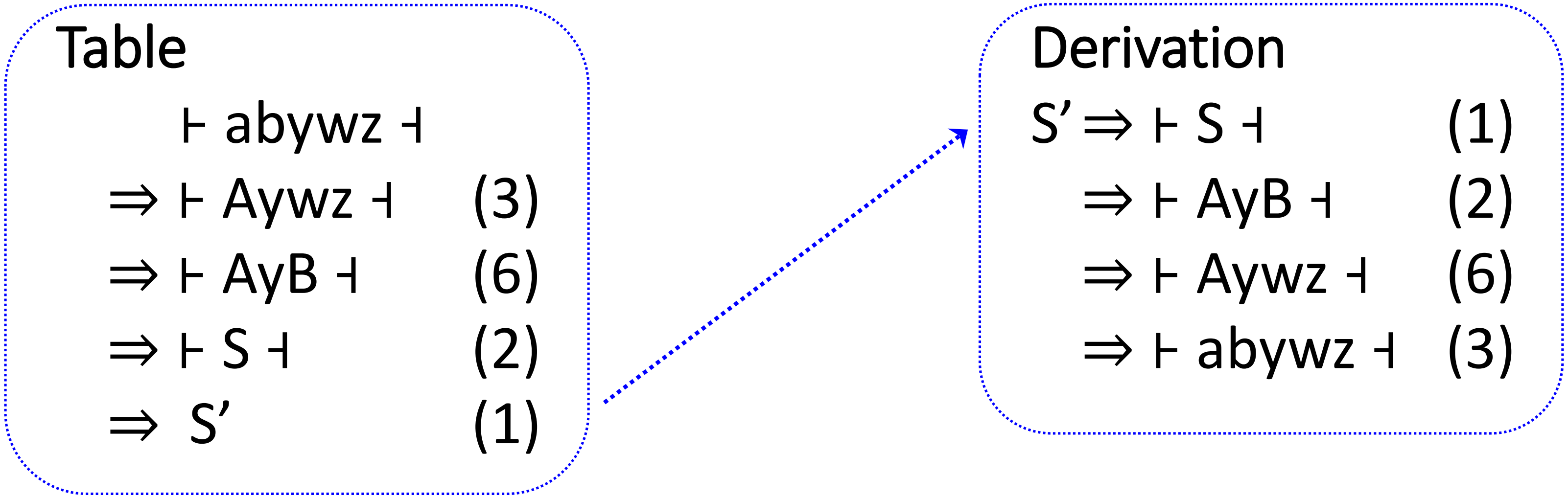
Derivation

S’⇒ ⊢ S ⊣	(1)
⇒ ⊢ AyB ⊣	(2)
⇒ ⊢ Aywz ⊣	(6)
⇒ ⊢ abywz ⊣	(3)

LR Parsing

Outputting a Rightmost Derivation

- In the table we are expanding the leftmost terminal.
- When we output the rules in reverse, the list now expands on the rightmost terminal first.



Table

⊢ abywz ⊣	
⇒ ⊢ Aywz ⊣	(3)
⇒ ⊢ AyB ⊣	(6)
⇒ ⊢ S ⊣	(2)
⇒ S’	(1)

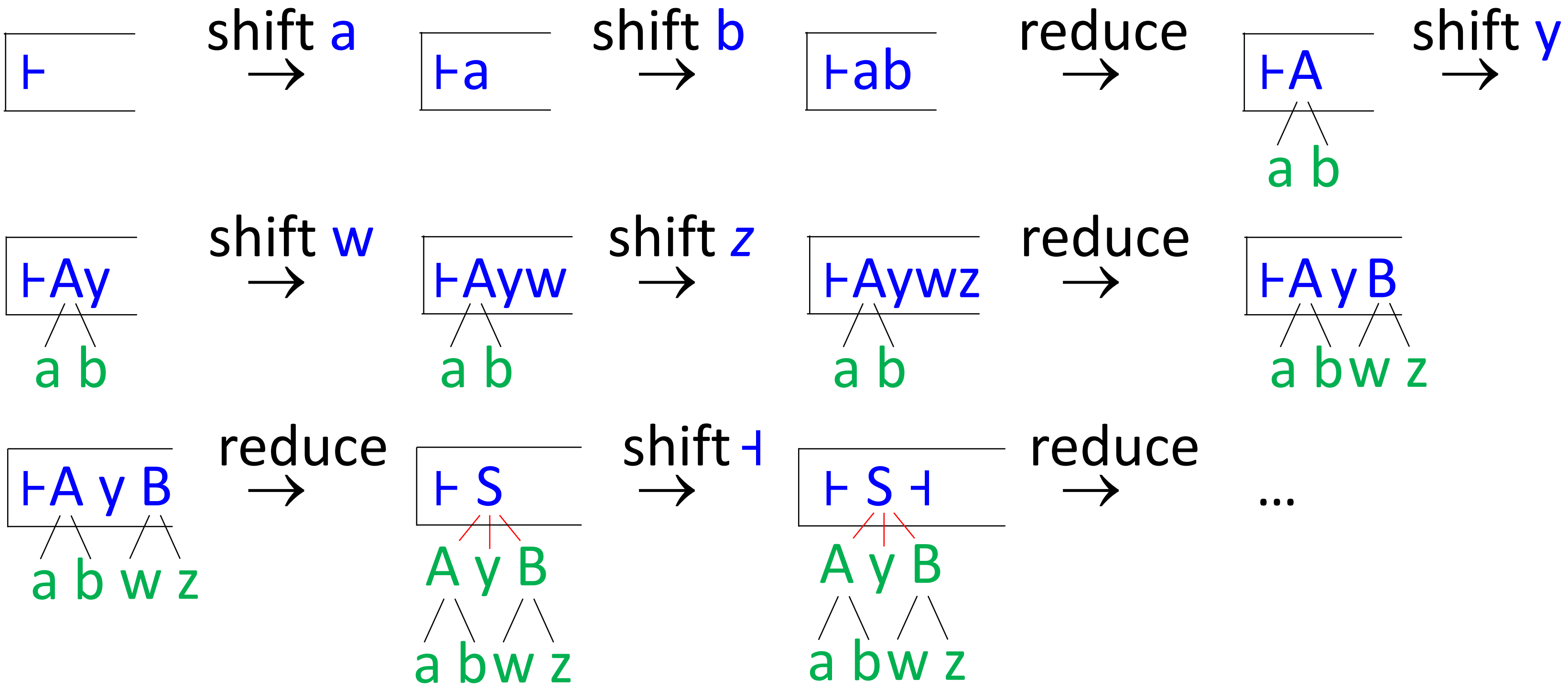
Derivation

S’ ⇒ ⊢ S ⊣	(1)
⇒ ⊢ AyB ⊣	(2)
⇒ ⊢ Aywz ⊣	(6)
⇒ ⊢ abywz ⊣	(3)

LR Parsing

Outputting a Parse Tree

- create a tree stack
- each time we reduce, the items popped off the stack become the children and the item pushed on becomes the parent.



⊦ shift a → ⊦a shift b → ⊦ab reduce → ⊦A shift y →

a b

⊦Ay shift w → ⊦Ayw shift z → ⊦Aywz reduce → ⊦A y B

a b a b a b a b w z

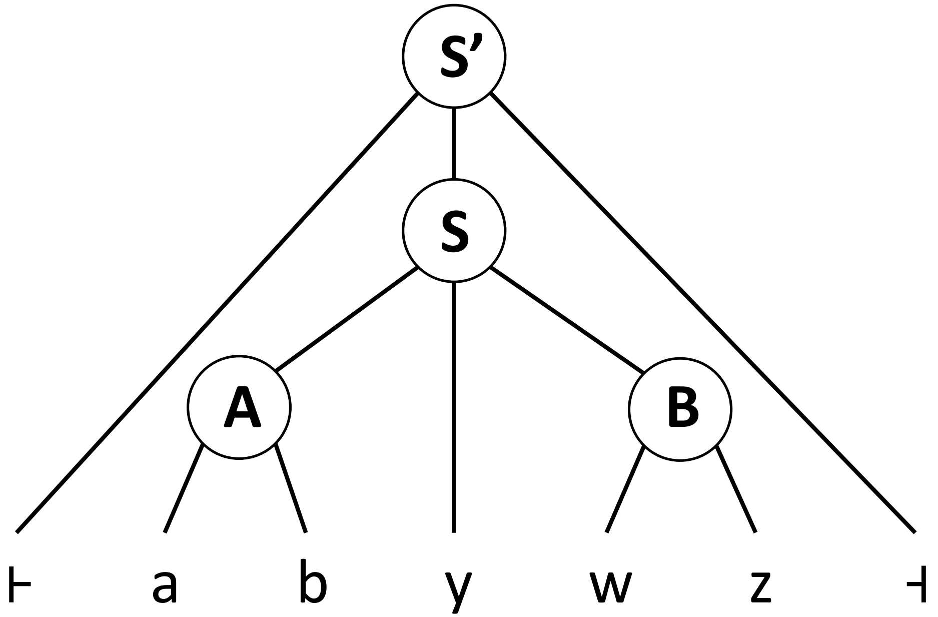
⊦A y B reduce → ⊦ S shift ⊣ → ⊦ S ⊣ reduce → …

a b w z A y B A y B

a b w z a b w z

LR Parsing

The Parse Tree



Grammar

1. S' → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz

Derivation

S' ⇒ ⊢ S ⊣ (1)
⇒ ⊢ AyB ⊣ (2)
⇒ ⊢ Aywz ⊣ (6)
⇒ ⊢ abywz ⊣ (3)

Non- LL(1) Grammars

Use LR to Parse our Non-LL(1) Grammar

G: $L = \{a^n b^m \mid n \geq m \geq 0\}$ is not in LL(k) for any k

1: S’ → ⊢ A ⊣
2: A → a A
3: A → B
4: B → aBb
5: B → ε

if input = ⊢, shift

if input = a, shift

if input = b, reduce by (5)
then repeatedly shift
and reduce by (3)

if input = ⊣, reduce by (2) …

Stack	**Input**	**Action**
	⊢ aaabb ⊣	shift ⊢
⊢	aaabb ⊣	shift a
⊢ a	aabb ⊣	shift a
⊢ aa	abb ⊣	shift a
⊢ aaa	bb ⊣	reduce 5
⊢ aaaB	bb ⊣	reduce 4
⊢ aaB	b⊣	reduce 4
⊢ aB	⊣	reduce 3
⊢ aA	⊣	reduce 2
⊢ A	⊣	shift ⊣
⊢ A ⊣		accept

SLR(1) Parser

LR Parsing

- SLR(1) resolves many, but not all, conflicts.
- Can create increasing more sophisticated automatons
  - e.g. LALR(1) (used in YACC and Bison) and LR(1) parsers
  - each is more complex
  - each can parse more grammars
  - the parsing algorithm and the format of the automaton is the same, but the method used to create the automaton is different, e.g. how you calculate the follow set
    - per non-terminal (e.g. the follow set for E) or
    - per rule (e.g. the follow set for the rule E $\rightarrow$ T + E)

Assignment 6

Hints

- P1 and P2: no programming required, create these cfg-r files yourself
- P3: Given a description of a DFA, a current state, and a single input (in lr1 format) take one transition
- P4: Create a parser based on solution to P3
  - read in a CFG, a DFA and an input string (in lr1 format)
  - output a derivation (in cfg-r format)
- P5: Write a parser for WLP4
  - read in tokens, build a parse tree bottom-up and output a leftmost derivation (in wlp4i format) along with tokens and lexemes

Assignment 6

Hints

- P5: Write a parser for WLP4
  - can no longer read the lr1 file in from stdin (i.e. the file that describes the WLP4 LR(1) automaton and grammar)
    - read in from separate file
    - embed as a (big) string constant
  - stdin is now used to read in the sequence of tokens (from a scanner for WLP4)

Assignment 6 Hints

Three File Formats

- Lots of file formats: cfg-r, lr1, wlp4i

P1, P2

- cfg-r like CFG but is a reverse rightmost derivation

P3

lr1 file format: CFG + LR(1) machine + sequence to be parsed, e.g.

- “0 BOF shift 6”

  when in state 0 (i.e. state_stack.top == 0) if the lookahead character is BOF then shift the input onto the symbol stack and goto state 6 (push it on the state_stack)

Assignment 6 Hints

Three File Formats

- “4) reduce 1”
  when in state 4 (i.e. state_stack.top == 4) if the lookahead character is ‘)’ then reduce using rule 1

P4

- input is an lr1 file,
- output is an cfg-r file

P5

- input is a token stream generated by a scanner (as in A5)
- output is wlp4i file
  - like a cfg file (from A5)
  - also include tokens and lexemes
  - defined recursively

Topic 14 – Context-sensitive Analysis

Key Ideas

- variable and procedure declarations
- scope
- type checking
- well-typed expressions

References

- *Basics of Compiler Design* by T. Mogensen sections 4.1- 4.2 (Scope and Symbol Tables), 6.1-6.7 (Type Checking)
- WLP4 Language Spec and Type rules

  https://www.student.cs.uwaterloo.ca/~cs241/wlp4/WLP4.html

  https://www.student.cs.uwaterloo.ca/~cs241/wlp4/typerules.pdf

What is Next?

Basic Compilation Steps

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

↓ *1. WLP4 program*

(A5) WLP4 Scan: lexical analysis (regular languages)

↓ *2. tokens*

(A6) WLP4 Parse: syntactic analysis (context-free grammars)

↓ *3. parse tree*

(A7) WLP4Gen: semantic analysis

↓ *4. augmented parse tree + symbol table*

(A8-A9) code generation

↓ *5. MIPS assembly language*

What is Next?

Basic Compilation Steps

WLP4 Input file:

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

$\xrightarrow[A5]{}$

Sequence of Valid Tokens
i.e. (Kind, Lexeme) Pairs:

```
INT int
WAIN wain
LPAREN (
INT int
ID a
COMMA ,
INT int
ID b
RPAREN )
LBRACE {
    ⋮
```

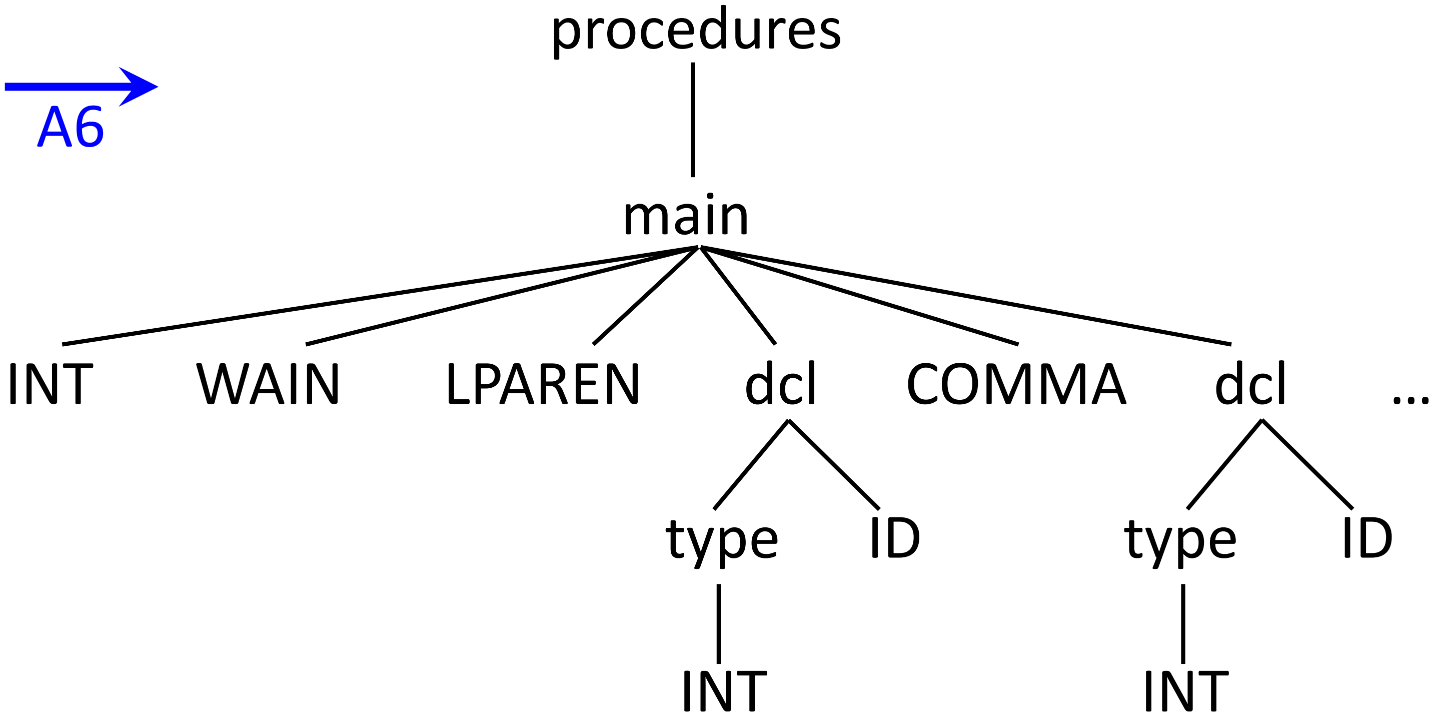
$\xrightarrow[A6]{}$

The postcondition for A5 and the precondition for A6 is that the output of lexical analysis is a sequence of *valid* tokens.

What is Next?

Basic Compilation Steps

Parse Tree



A6

procedures

main

INT WAIN LPAREN dcl COMMA dcl …

type ID type ID

INT INT

- unlike a binary tree, nodes can have more than two children

Context-Sensitive Analysis

Syntax vs. Semantics

- *Context-free*
  - e.g. *cannot detect* if a variable is used before it is declared
- *Context-sensitive*
  - e.g. *can detect* if a variable is used before it is declared
- Input: a parse tree
- Precondition: the program is syntactically valid
- Output:
  if input is semantically valid
  then output an augmented parse tree + symbol table
  else output ERROR

Context-Sensitive Analysis

Errors that a Context-Sensitive Analysis Finds

- If a program is syntactically valid, what else can go wrong?
  - *variables* can be
    - undeclared, used before they were declared
    - have multiple declarations
  - *procedures* can be
    - undeclared, used before they were declared
    - have multiple declarations
  - *types*
    - return value of procedures
    - parameter lists
    - operators
  - *scope*
    - scope of variables in (and out of) procedures

Variable Declaration Issues

How to Solve Variable Declaration Issues

- Answer: a *Symbol Table*
  - similar to what we did for our MIPS assembler and labels
  - track: *Name* and *Location*
    - which we also did for our MIPS assembler
  - but also track: *Type* (e.g. int and int*)
    - did not track this information with our MIPS assembler
    - programming languages generally have many more types, bool, char, short int, int, long, long long, float, double, long double, void …

Variable Declaration Issues

How to Solve Variable Declaration Issues

- e.g. test001.wlp4

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

- *When using a variable*, make sure it is in the symbol table
  - i.e. it exists
- "`return c;`" is
  - lexically valid,
  - syntactically valid,
  - but is semantically invalid (i.e. a semantic error) if `c` has not been declared somewhere, i.e. if we do not know what RAM location `c` represents

Variable Declaration Issues

How to Solve Variable Declaration Issues

- e.g. test002.wlp4

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

- *When declaring a variable*
  - check that it is not already in the symbol table
  - if it is not, then added it
  - if it is then report an error
  - similar to what we did with label definitions for the MIPS assembler

Checking Variable Declarations

First Check for Multiple Declarations

- *recursively traverse* the parse tree and track any declarations
- *search* for nodes with rule *dcl* → TYPE ID
  - extract the name (e.g. **a**) and the type (e.g. int)
  - *check* **if** the name is already in the symbol table

    then ERROR

    else *add* name and type to symbol table

Next Check for Undeclared Variables

- *recursively traverse* the parse tree and track the use of variables
- *search* for nodes with the rules
  - *factor* → ID
  - *lvalue* → ID
- *check* **if** ID's name is not in the symbol table **then** ERROR

Checking Variable Declarations

Scope

- must also consider the concept of *scope*
- both `f` and `wain` can declare and use the local variable `a`

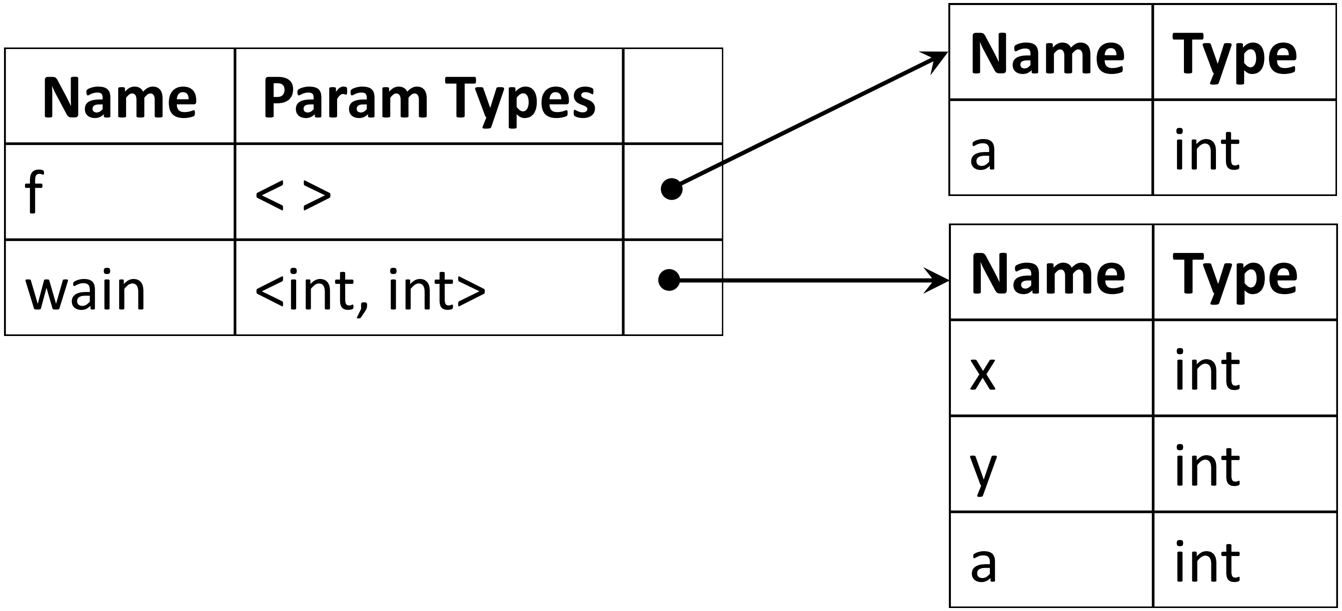
```
int f() {
  int a=0;
  return a;
}

int wain(int x, int y) {
  int a=1;
  return x+a;
}
```

- clearly we need a more sophisticated version of a symbol table, i.e. a *hierarchical symbol table*

Variable Declaration Issues

How to Implement Scope for Variables

- have a *global symbol table* for procedure names and types
- have separate symbol tables for each procedure to track its parameters and local variables
- note: WLP4 does not have global variables
- note: the return type of all WLP4 procedures is int



Name	Param Types	
f	< >	●
wain	<int, int>	●

Name	Type
a	int

Name	Type
x	int
y	int
a	int

Variable Declaration Issues

Obtaining Signatures

- Procedures have *signatures,* i.e.
  - *names (called IDs)* which must be extracted
  - *return types* which is always `int` in WLP4
  - *parameters lists* with possibly a mixture of `int` and `int*` types

- *Finding procedures* in the parse tree
  - *traverse* the parse tree and *search* for procedures declarations i.e. nodes with one of these two rules
    - procedure → INT ID LPAREN params RPAREN LBRACE…
    - main → INT WAIN LPAREN dcl COMMA dcl RPAREN LBRACE…

Variable Declaration Issues

Obtaining Signatures

- once you have found one of these rules declaring procedures
  - *main* → INT WAIN LPAREN dcl COMMA …
  - *procedure* → INT ID LPAREN params LBRACE …
- **if** the procedure name is already in the global symbol table
  then report ERROR
  else add it and create a new symbol table for that procedure
- for procedures we store its signature in the symbol table
- these are captured by the following production rules
  paramlist → dcl
  paramlist → dcl COMMA paramlist
  dcl → TYPE ID

Type Checking

Why Types Matter

- Recall: looking at a pattern of bits will not tell us what they represent
- in WLP4 there are only two types: int and int*
- Types help us
  - remember what a variable *means*
  - interpret the pattern of 0's and 1's stored in memory
  - delimit how a value can be used
  - catch if we have used the value improperly (sometimes)
  - e.g. in WLP4

```
int *aPtr = NULL;
aPtr = 7;              // ERROR: assigning an int to an int*
```

Type Checking Quiz

Well-typed Expressions

- Given the following declarations

```
int  i = 1, j = 2;
int *p =&i,*q =&j;
```

- Which of the following assignments violate C++’s type rules?

```
i = i + j;           p = i + j;
i = i + p;           p = i + p;
i = p + i;           p = p + i;
i = p + q;           p = p + q;
i = p - q;           p = p - q;
```

- *Hint:* if it makes sense for some situation then allow it.
- Note: WLP4 has the same type rules.

Type Checking

Working with Type Rules

- See “WLP4 Semantic Rules” handout
- Notation or rules: $\dfrac{\text{assumptions}}{\text{consequences}}$ or $\dfrac{\text{preconditions}}{\text{postconditions}}$
- To type-check:
  - ensure that the WLP4 Semantic Rules are followed *when computing the type of an expression*
  - set the left-hand side’s type to the right-hand side’s type for rules such as

$$\left.\begin{array}{l} \text{expr} \rightarrow \text{term} \\ \text{term} \rightarrow \text{factor} \\ \text{factor} \rightarrow \text{ID} \\ \text{factor} \rightarrow \text{NUM} \\ \text{factor} \rightarrow \text{NULL} \end{array}\right\} \quad \text{type(LHS) = type(RHS)}$$

Type Checking

Working with Type Rules

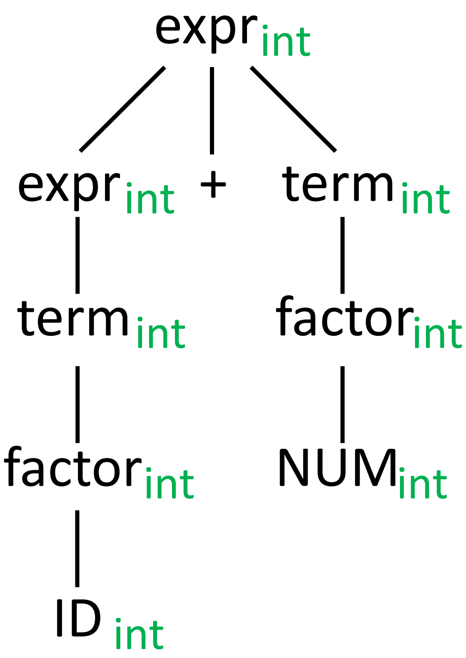
- To type-check:
  - decorate the parse tree with types
  - also called an augmented parse tree
  - propagate from the leaves up

    factor → NUM

    factor → ID

    term → factor

    expr → term
  - ensure that rules are followed
  - e.g int + int is an int
  - *we need a method to specify type rules*


expr_{int}
expr_{int}
+
term_{int}
term_{int}
$\text{factor}_{\text{int}}$
$\text{factor}_{\text{int}}$
NUM_{int}
ID_{int}

Type Checking

Working with Type Rules

- *must check if types are being used properly*
- *notation:* we'll introduce the variable τ to represent a type
- recall that only has two types: int or int*
- use τ to talk about types without mentioning a specific type, e.g.
  - "$E_1 : \tau$ and $E_2 : \tau$" means E_1 and E_2 have the same type, i.e. they are either both int or both int*
  - "$E_1 : \tau_1$ and $E_2 : \tau_2$" means E_1 and E_2 may or may not have the same type
- allowing both integers and pointers creates a challenge , i.e. we must track if the type is int or int*

Type Checking

Working with Type Rules

- Rule: $\dfrac{<\mathit{id.name}, \tau> \in \mathit{dcl}}{\mathit{id.name} : \tau}$
- Meaning:
  - **if** id.name was declared to have type τ
  - **then** id.name has type τ
  - true whether τ = int or τ = int *

- Rules: $\dfrac{}{\text{NUM} : \text{int}}$ $\dfrac{}{\text{NULL} : \text{int*}}$ $\dfrac{\text{E} : \tau}{\text{(E)} : \tau}$
- Meaning:
  - NUM is always of type int (no assumptions are needed)
  - NULL is always of type int* (no assumptions are needed)
  - putting parenthesis around an expression preserves its type

Type Checking

Type Rules for Pointer Types

- Rules: $\dfrac{E : \text{int}}{\&E : \text{int*}}$ $\qquad$ $\dfrac{E : \text{int*}}{*E : \text{int}}$
- Meaning:
  - when you take the address of an int type, you get an int*
  - when you dereference an int* type (put a * in front of it) , you get an int.

- Rule: $\dfrac{E : \text{int}}{\text{new int[E]} : \text{int*}}$
- Meaning:
  - when you create a new array (of size E) you get an int* (i.e. a pointer to the first element in the array)

Type Checking

Type Rules for Arithmetic Operations

- Rules: $\dfrac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 * E_2 : \text{int}}$ $\quad \dfrac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 / E_2 : \text{int}}$ $\quad \dfrac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 \% E_2 : \text{int}}$
- Meaning:
  - if E_1 and E_2 are int's then the result of multiplying them, dividing them or finding the remainder is also an int.

- Rules: $\dfrac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 + E_2 : \text{int}}$ $\quad \dfrac{E_1 : \text{int*} \quad E_2 : \text{int}}{E_1 + E_2 : \text{int*}}$ $\quad \dfrac{E_1 : \text{int} \quad E_2 : \text{int*}}{E_1 + E_2 : \text{int*}}$
- Meaning:
  - When you add two int's, the sum is an int.
  - When you add an int and an int*, the sum is an int*. You *cannot add two* int*'s, i.e. there is no rule for this operation.

Type Checking

Type Rules for Arithmetic Operations

- Rules: $\dfrac{E_1 : \text{int} \quad E_2 : \text{int}}{E_1 - E_2 : \text{int}}$ $\quad \dfrac{E_1 : \text{int*} \quad E_2 : \text{int*}}{E_1 - E_2 : \text{int}}$ $\quad \dfrac{E_1 : \text{int*} \quad E_2 : \text{int}}{E_1 - E_2 : \text{int*}}$
- Meaning:
  - When you subtract two int's, or two int*'s, the result is an int.
  - An int* minus an int is an int*.
  - You *cannot* subtract an int* from an int

- Rules: $\dfrac{< f, (\tau_1, \ldots \tau_n) > \; \in \text{procedures-decl} \; E_1{:}\tau_1, \ldots E_n{:}\tau_n}{f(E_1, \ldots, E_n) : \text{int}}$
- Meaning:
  - If a function with *n* parameters has been declared, its return type is int (no matter what its parameter types are).

Type Checking

Well-typed Expressions

- Some structures (e.g. while loops or statements) don't have types, so we check that the structure is *well-typed* e.g. the components have the right types
- Rules: $\dfrac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\ E_1 == E_2)}$ $\quad$ $\dfrac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\ E_1 < E_2)}$
- Meaning:
  - If E_1 and E_2 are of the same type, then the comparisons $E_1 == E_2$ and $E_1 < E_2$ are *well-typed*.
  - There are six comparisons: ==, !=, <, <=, >, >=
  - WLP4 allows comparisons of pointers
  - These comparisons are referred to as *tests.*

Type Checking

Well-typed Expressions

- Rules: $$\frac{E_1 : \tau \quad E_2 : \tau}{\textit{well-typed}(\ E_1 = E_2)}$$
- Meaning:
  - When you assign a value to a variable, then the types must match.
  - This is referred to as an *assignment.*

- Rules: $$\frac{E : \text{int*}}{\textit{well-typed}(\ \text{delete [] } E)}$$
- Meaning:
  - You can deallocate memory if it is a pointer to an int

Type Checking

Well-typed Expressions

- Rules:

$$\frac{\textit{well-typed}(S_1) \qquad \textit{well-typed}(S_2)}{\textit{well-typed}(S_1\ S_2)}$$

- Meaning:
  - *Here S_1 and S_1 are statements.*
  - A concatenation of statements is *well-typed* if both the prefix and the suffix are *well-typed*.

Type Checking

Well-typed Expressions

- Rules:

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

- Meaning:
  - *Here T is a test and S are statements.*
  - A while loop is *well-typed* if what is enclosed by parenthesis is a *well-typed* test and what is enclosed by braces is a *well-typed* statement(s).

Type Checking

Well-typed Expressions

- Rules: $$\frac{\textit{well-typed}(T)\quad \textit{well-typed}(S_1)\quad \textit{well-typed}(S_2)}{\textit{well-typed}(\ \textit{if}\ (T)\ \{S_1\}\ \textit{else}\ \{S_2\}\ )}$$
- Meaning:
  - *Here T is a test, S_1 and S_2 are statements.*
  - An if statement is *well-typed* if what is enclosed by parenthesis is a *well-typed* test and what is enclosed by braces are *well-typed* statements.

Assignment 7

Input

- a .wlp4i file (the output format of A6)

P1-P4: Create a Symbol Table(s)

- Create and output symbol tables.
- For P1 initially you will only have *one procedure*, i.e. this rule

  procedures → main

  and with P2-P4 you handle the rule which generates additional procedures:

  procedures → procedure procedures

P1

- output a symbol table
- check for multiple declarations of identifiers
- check for identifiers used before declared

Assignment 7

P2

- allow for other procedures but only process `wain`

P3

- process signatures of other procedures (but not their local variables/parameters)

P4

- process all procedures, their parameters and local variables

P5 Type Checking

- type check expressions (expr) and lvalues (lvalue)

P6 Type Checking

- type check everything on the Semantic Rules handout (e.g. add statements and tests)

Topic 15 – Code Generation

Key Ideas

- syntax-directed translation
- stack frames
- frame pointer (fp)
- MIPS register conventions (for CS 241)
- use of $5 and stack for intermediate results

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen section 7.4
- CS241 – WLP4 Programming Language Specification
- CS241 Assignment 8

Code Generation: Overview

Recall: Basic Compilation Steps

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

WLP4 text file
↓
WLP4 tokens + lexemes
↓
parse tree
↓
augmented parse tree
+ symbol table
↓
MIPS
Assembly Language

A5 *lexical analysis*: identify the tokens

A6 *syntax analysis*: parse

A7 *context-sensitive analysis:* create symbol table and type checking

A8-A9 *code generation*

Code Generation: Overview

Overview

- *Input:*
  - an augmented parse tree + a symbol table
- *Preconditions:*
  - the program has no syntax errors
  - the program has no semantic errors, *i.e.* types rules have been followed (it is *well-typed*), variable and procedures are properly declared, scope has been utilized properly
- *Output:*
  - MIPS assembly language program equivalent to the WLP4 code(same input → same output and return value)
  - many possible answers, *i.e.* append "**`add $1,$1,$0`**" to the program any number of times

Code Generation: Overview

Key Issues

- *Correctness*
  - compiler must create an equivalent program
  - compiler must be correct for all valid inputs (*i.e.* valid programs)
- *Ease of (or simplicity of) writing the compiler*
  - especially for CS 241
- *Efficiency of the compiler:*
  - time to compile a program, O(*n*) for *n* lines of code
- *Efficiency of the compiled code:*
  - minimize resources (time and space) required
  - called *code optimization* (e.g. how to assign registers effectively) which is only touched on in this course

Code Generation: Overview

Approach

- *Syntax-directed Translation*
  - create a translation function for each syntactic category, *e.g.* for loops, if-else statements, assignment, expressions
    - e.g. a code() function for each grammar rule/production
  - translation closely follows the syntactic structure of the code (*i.e.* parse tree) with some additional information as needed (*i.e.* symbol table)
  - recursively traverse the parse tree to gather the needed information
  - first you must understand exactly what we mean by …

    code(expr) = code(expr_1) + code(term_2) + code(expr_1 - term_2)

Syntax-directed Translation

Example

```
1 - 2;
```

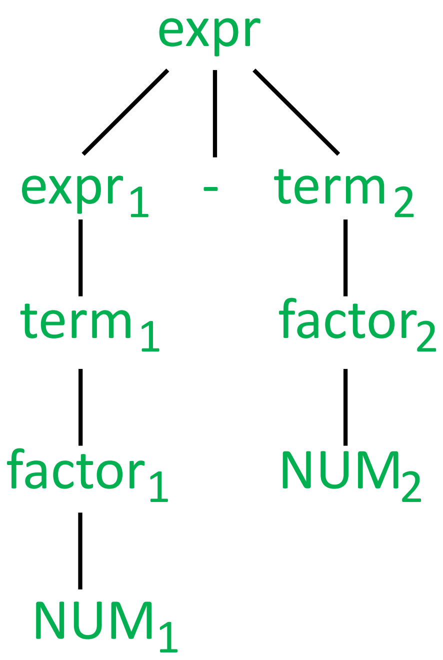
- We have these production rules

  $\text{expr} \rightarrow \text{expr}_1 \text{ - } \text{term}_2$

  $\text{expr}_i \rightarrow \text{term}_i$

  $\text{term}_i \rightarrow \text{factor}_i$

  $\text{factor}_i \rightarrow \text{NUM}_i$



expr

expr_1 - term_2

term_1 factor_2

factor_1 NUM_2

NUM_1

- So we would generate this code recursively using the following rules

  $\text{code}(\text{expr}) = \text{code}(\text{expr}_1) + \text{code}(\text{term}_2) + \text{code}(\text{expr}_1 \text{ - } \text{term}_2)$

  $\text{code}(\text{expr}_i) = \text{code}(\text{term}_i)$

  $\text{code}(\text{term}_i) = \text{code}(\text{factor}_i)$

  $\text{code}(\text{factor}_i) = \text{code}(\text{NUM}_i)$

Syntax-directed Translation

Example

- we would eventually get (greatly simplified) something like

```
;; code(expr) =
lis $3                    ; code($expr_1$)
.word 1
lis $5                    ; code($term_2$)
.word 2
sub $3, $3, $5            ; code($expr_1$ - $term_2$)
```

- which we would express as
  code(expr) = code($expr_1$) + code($term_2$) + code($expr_1$ - $term_2$)
  i.e. the code for expr equals the code for $expr_1$ concatenated with the code for $term_2$ concatenated with the code for $expr_1$ - $term_2$
- post order traversal (code for children before code for parent)

Syntax-directed Translation

Another Example

- For a more complicated rule like

  main → INT WAIN LPAREN dcl_1 COMMA dcl_2 RPAREN LBRACE dcls statements RETURN expr SEMI RBRACE

- we would have

  code(main) = code(dcl_1) + code(dcl_2) + code(dcls) + code(statements) + code(expr)

- i.e. we don’t generate code for the delimiters like commas, semicolons, left and right parentheses.

Storing Variables: A8P1

Example a)

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

Output

```
add $3, $1, $0      ; move 1st parameter to register
jr $31              ; that holds the return value and
                    ; return control to OS
```

Conventions

- $1 and $2 hold the parameters for the **wain** function
  - think of the loaders mips.twoints and mips.array from A2
- $3 holds the return value
- $31 holds the address (of the operating system) that we return to when our program exits

Storing Variables: A8P1

Example b)

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

Output

```
add $3, $2, $0      ; move 2nd parameter to register
jr $31              ; that holds the return value and
                    ; return control to OS
```

Observation

- Examples a) and b) both have the same parse tree.
- How do we differentiate the programs?
- Add a Location column to the Symbol Table

Symbol Table		
Symbol	**Type**	**Location**
a	int	$1
b	int	$2

Storing Variables

Attempt 1: Store Variables in Registers

- Idea : each variable gets its own register
- Problem: what if there are more that 32 variables

Attempt 2: Store Variables in RAM

- Idea : Store variables in RAM using the .word directive
  - each variable x gets its own label “x” in MIPS
- Problems
  - more costly to access variables
  - must be able to differentiate local variables that both share the same ID and label
  - will not work for recursive functions

Storing Variables

Attempt 3: Store Variables in Stack

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

- store parameters **a**, **b** and local variable **c** on the stack with the locations relative to $30

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

```
;;; prolog
lis $4
.word 4
sw $1,-4($30)   ; push a
sub $30,$30,$4
sw $2,-4($30)   ; push b
sub $30,$30,$4
;;; body
sw $0,-4($30)   ; push c
sub $30,$30,$4
lw $3,8($30)    ; return a
;;; epilog
lis $12         ; pop
.word 12        ; c, b, a
add $30,$30,$12;
jr $31
```

Storing Variables

Attempt 4: Store in Stack Frame

- the value of the stack pointer changes with each push
- idea: allocate a *stack frame all at once*
- subtract 4*n* where *n* is the size of the symbol table

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

```
;;; prolog
lis $12              ; push
.word 12             ; stack
sub $30,$30,$12;     frame
sw $1,8($30)         ; save a
sw $2,4($30)         ; save b

;;; body
sw $0,0($30)         ; declare c
lw $3,8($30)         ; return a

;;; epilog
lis $12              ; pop
.word 12             ; stack
add $30,$30,$12;     frame
jr $31
```

Storing Variables

Comments

- We now have
  - a *stack frame* to store parameters and local variables
  - a *prolog* (to set up the stack frame, any constants needed etc.)
  - a *body* do the task required
  - an *epilog* (to pop off the stack frame)
- but be aware that
  - the stack pointer may change value in the body the code
  - if the body has complicated expressions like

    (a+b) - (4*a*c) / (2*a)

    then intermediate results, like (a+b), are stored on the stack

Storing Variables

Frame Pointer

- Problem
  - cannot use the stack for temporary storage after pushing the stack frame because if we change the value of the stack pointer then the offsets in the symbol table will all need to be updated
- Solution: *frame pointer (fp)*
  - reserve $29 to *point to the first element of the stack frame* (for this procedure)
  - offsets in symbol table will be relative to the frame pointer
  - the frame pointer does not change value as the stack is used for temporary values in the body of the function

Storing Variables

Attempt 5: Use Frame Pointer $29

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

- store **a**, **b** and **c** on the stack with location offsets *relative to the frame pointer,* $29

Symbol Table		
Name	**Type**	**Location**
a	int	0
b	int	-4
c	int	-8

```
;;; prolog
lis $4
.word 4
sub $29,$30,$4  ; init fp
lis $12         ; push
.word 12        ; stack
sub $30,$30,$12; frame
sw $1,0($29)    ; store a
sw $2,-4($29)   ; store b
;;; body
sw $0,-8($29)   ; declare c
lw $3, 0($29)   ; return a
;;; epilog      ; pop
add $30,$29,$4  ; stack
jr $31          ; frame
```

Storing Variables

Attempt 5: Use Frame Pointer $29

- Prolog: have the frame pointer ($29) point to the next available stack location ($30 - 4).
- Refer to parameters and local variables based on the frame pointer ($29) which does not change value during the function.
- Epilog: have the stack pointer ($30) point to its previous value before the function call ($29+4).

```
;;; prolog
lis $4
.word 4
sub $29,$30,$4  ; init fp
lis $12         ; push
.word 12        ; stack
sub $30,$30,$12; frame
sw $1,0($29)    ; store a
sw $2,-4($29)   ; store b
;;; body
sw $0,-8($29)   ; declare c
lw $3, 0($29)   ; return a
;;; epilog      ; pop
add $30,$29,$4  ; stack
jr $31          ; frame
```

Storing Variables

Attempt 5: Use Frame Pointer $29

- Now all references to arguments and local variables are based on frame pointer.
- The stack can be used in the body of the function to store intermediate values.
- *This approach is recommended.*

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

```
;;; prolog
lis $4
.word 4
sub $29,$30,$4  ; init fp
lis $12         ; push
.word 12        ; stack
sub $30,$30,$12; frame
sw $1,0($29)    ; store a
sw $2,-4($29)   ; store b
;;; body
sw $0,-8($29)   ; declare c
lw $3, 0($29)   ; return a
;;; epilog      ; pop
add $30,$29,$4  ; stack
jr $31          ; frame
```

Conventions

Code Gen Conventions (for CS 241)

- We will use the following conventions in CS 241

$0	always 0 (or false)
$1	**`wain`**'s 1st argument (a1)
$2	**`wain`**'s 2nd argument (a2)
$3	result (and intermediate results) of calculations
$4	constant 4, useful for pushing and popping the stack
$5	previous intermediate results
$11	always 1
$29	frame pointer (fp)
$30	stack pointer (sp)
$31	return address (ra)

Conventions

Code Gen Conventions (for CS 241)

- Program Prolog
  - initialize constants (store 4 in $4 and 1 in $11)
  - store return address ($31) on stack
  - initialize frame pointer ($29) and create stack frame
  - store arguments ($1 and $2) in stack frame
- Program Body
  - initialize local variables in stack frame
  - generate code for body of function
- Program Epilog
  - pop stack frame
  - restore previous return address to $31

Some Examples: A8P2

Code

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

Output

```
;; same prolog
lw $3, 0($29)  ; load a from stack(based on fp)
;; same epilog
```

Rationale

- for the rule : factor → LPAREN expr RPAREN

code(factor) = code(LPAREN) + code(expr) + code(RPAREN)
= code(expr)

Some Examples: A8P3

Code

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

Output

```
add $3, $1, $2
```

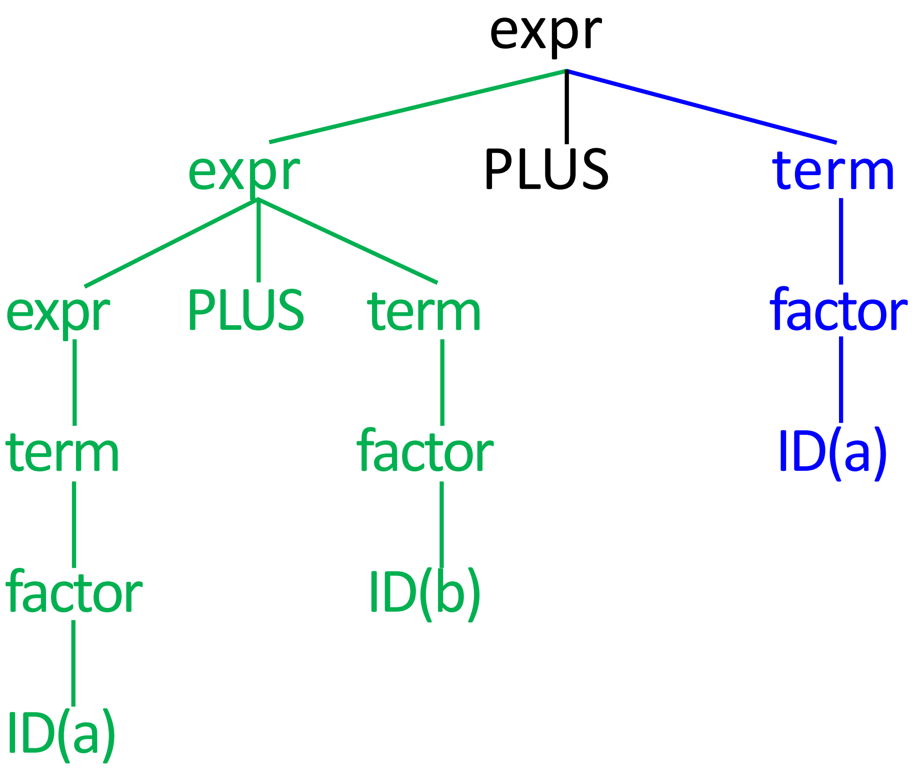
Does this approach always work?

- What about a+b+a?
- What about (a1-a2)*(b1-b2)?
- What about a * (b * (c*d))?
- What about (a1-a2) * ((b1-b2) * ((c1-c2) * (d1-d2)))?
- and so on ...

Some Examples: A8P3

Parse Tree

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



Problem

- If we assign $3 to the value of the left subtree (expr) what register do we assign to the right subtree (term)?
- If our plan is to use another register, and if there are many nested subexpressions, we will run out of registers.
- Key Point: Our approach must work for an *arbitrary* number of expressions.

Some Examples: A8P3

Approach for Binary Operations

- Rather than write out all the MIPS assembly language instructions, I'll use two bits of pseudocode

```
push($3)        ;; push the value stored in $3
                ;; onto the system stack
                sw $3,-4($30)
                sub $30,$30,$4
$5 = pop()      ;; pop the value off of the system
                ;; stack and store it in register $5
                add $30,$30,$4
                lw $5,-4($30)
```

- When coding is up for yourself *replace the pseudocode by the actual assembly language instructions*

Some Examples: A8P3

Approach for Binary Operations

- for rule: *$expr_1 \rightarrow expr_2 + term$* the code is

Output

```
;; code(expr1) =
code(expr2)     ; $3 ← result of expr2
push($3)        ; pseudocode to push $3 onto stack
code(term)      ; $3 ← result of term
$5 = pop()      ; $5 ← expr2, pseudocode to pop stack
add $3, $5, $3  ; $3 ← expr2 + term
```

Rationale

- use $3 for all intermediate (as well as final) results
- use the stack to temporarily store the result of expr2 and then pop the result of expr2 into $5 when it is needed
- only need one other register, $5.

Some Examples: A8P3

Approach for Binary Operations

- Key Idea: for instructions that *require two source registers* (e.g. add, sub, mult, multu, div, divu, slt, sltu, beq, bne) the source registers will always be *$3* and *$5*
- *store the first result (result1) on the stack, calculate result2, then get the previous result (result1) from the stack and put it in $5.*

Output

```
;; code(result) =
code(expr2)      ; $3 ← result1
push($3)         ; stack ← $3 (i.e. result1)
code(term)       ; $3 ← result2 (reuse $3)
$5 = pop()       ; $5 ← result1 (from stack)
add $3, $5, $3   ; $3 ← result1 + result2
```

Some Examples: A8P4

Approach

- for rule: *statements* → *PRINTLN LPAREN expr RPAREN*

Output

- **`print`** prints whatever is in $1 on the screen, followed by a newline
- it overwrites (i.e. destroys) the contents of $1 and $31
- it is a library interface with the OS provided by the compiler
- print.merl has to be linked in, e.g.

```
./wlp4gen < source.wlp4i > source.asm
cs241.linkasm < source.asm > source.merl
linker source.merl print.merl > exec.mips
```

- the directive **`.import print`** must be added to the prolog
- more about importing, linking and merl later…

Some Examples: A8P4

Approach

- for rule: *statements* → *PRINTLN LPAREN expr RPAREN*

Output

```
;; code(println(expr)) =
   ;; Prolog
   .import print       ; imports the subroutine print
   ;; Body
   code(expr)          ; evaluate expr: $3 ← expr
   add $1, $3, $0      ; copy to $1: $1 ← expr
   lis $10             ; $10 ← print addr
   .word print         ;
   jalr $10            ; call print subroutine
   ;; Epilog
   ;; $31 restored
```

Some Examples: A8P4

The print Subroutine

- the print subroutine overwrites the contents of $1
- three ways to deal with this situation

1. try a different calling convention (say read from $3)
   but then older code needs to be changed
2. let the value in $1 be lost
   but it may be important
3. save and restore $1 on the system stack before calling print

- we generally store $1 and $2 on the stack
- later on when we take calling procedures, we'll set it up so that procedures save and restore any registers whose values they overwrite

Some Examples: A8P5

Rules for Assignment

- dcls → dcls dcl BECOMES NUM SEMI
- dcl → type ID
- e.g. `int total = 0;`

Notes

- code(NUM)
  - put the number, NUM, into register $3, i.e. **$3** ← **NUM**
- code(dcl BECOMES NUM SEMI)
  - load NUM into $3 (use `lis $3` and `.word`)
  - look up the offset of ID in the symbol table (i.e. the offset relative to the frame pointers $29)
  - generate the code: `sw $3, ID_offset($29)`

Some Examples: A8P5

Rules for Assignment

- statement → lvalue BECOMES expr SEMI
- lvalue → ID
- e.g. `total = a+1;`
- For A8 lvalue is an ID (not a pointer). That will change for A9.

Notes

- code(statement)
  - evaluate the expression expr by calling code(expr)
  - the results should be stored in register $3
  - look up the offset of the ID in the symbol table (i.e. the offset relative to the frame pointer $29)

```
code(statement) = code(expr)
                  sw $3, ID_offset($29)
```

Some Examples: A8P6

Rules for Comparison Test

- test → expr_1 LT expr_2

Notes

- there are two control structures in WLP4:
  - (1) while loops and (2) if-then-else statements
- both rely on comparison tests

Conventions

- \$0 ← 0, no choice here, it's hardwired into MIPS
- \$11 ← 1, we must add this to the prolog
- recall: when evaluating multiple expressions, in a recursively friendly way
  - results are returned in \$3
  - use stack (to store) and \$5 (to retrieve) intermediate results

Some Examples: A8P6

Rules for Comparison Test

- test → $expr_1$ LT $expr_2$

Generating Code

- evaluate the 1st expression, $expr_1$ (the results will be in $3) and then push $3 on the stack

```
code(expr₁)            ; $3 ← expr₁
push($3)               ; stack ← expr₁
```

- evaluate the 2nd expression, $expr_2$ (the results will be in $3)

```
code(expr₂)            ; $3 ← expr₂
```

- pop off the stack results into $5 and complete the test

```
$5 = pop()             ; $5 ← expr₁
slt $3, $5, $3         ; set $3 if expr₁ < expr₂
```

Some Examples: A8P7

Rules for Comparison Test

- test → expr_1 GT expr_2

Generating Code

- *note:* ($3 > $5) is the same as ($5 < $3)
- so by swapping the order of the source registers, e.g.

```
    slt $3, $3, $5    ; $3 < $5
```

vs

```
    slt $3, $5, $3    ; $3 > $5
```

- we can obtain the other comparison using one instruction
- So the code for test → expr_1 GT expr_2
  - is very similar to the code for test → expr_1 LT expr_2
  - except the order of the source registers are swapped

Some Examples: A8P7

Rules for Comparison Test

- test → expr_1 GE expr_2
- test → expr_1 LE expr_2

Generating Code

- *note:* ($3 ≥ $5) is the same as **not** ($3 < $5)
- *note:* ($3 ≤ $5) is the same as **not** ($3 > $5)
- Since the result of a **`slt`** comparison is either 0 or 1
- to take the *not* of the result, subtract it from 1 (i.e. $11)

```
sub $3, $11, $3    ; $3 ← not($3)
```

- Why?
  - if $3==1 (true), then 1-$3==0 (false)
  - if $3==0 (false), then 1-$3==1 (true)
  - by CS241 convention, we will always store 1 in $11

Some Examples: A8P7

Rules for Comparison Tests

- test → $expr_1$ NE $expr_2$

Code Generation

```
;; code(test) =
code($expr_1$)          ; $3 ← $expr_1$
push($3)                ; stack ← $expr_1$
code($expr_2$)          ; $3 ← $expr_2$
$5 = pop()              ; $5 ← $expr_1$
slt $6, $3, $5          ; $6 ← $expr_2$ < $expr_1$
slt $7, $5, $3          ; $7 ← $expr_1$ < $expr_2$
add $3, $6, $7          ; $6 and $7 cannot both be 1
```

- if $expr_1$ == $expr_2$, then both `slt` commands will return 0 and sum is 0. If one of the `slt` tests returns 1, the sum will be 1.

Some Examples: A8P7

Rules for Comparison Tests and the NOT operation

- test → expr_1 EQ expr_2

Code Generation

- do the code for expr_1 != expr_2 followed by the statement

```
sub $3, $11, $3
```

- recall $11 contains 1 and $3 contains our results (a 0 or 1)
- again, subtraction (in this case) is equivalent to the NOT operation on the value in $3.
  - it will flip a 0 to a 1 and a 1 to a 0, i.e.
    - if $3 == 0 then $11 - $3 == 1
    - if $3 == 1 then $11 - $3 == 0

Some Examples: A8P6 and P8

Automatically Generating Labels

- for control structures such as *while* loops and *if-else* statements you will need to be able to *generate unique labels*
- *idea:* have a function like label()
  - recall that the leading character must be a letter
  - each time it gets called, a variable gets incremented
  - its value is concatenated to a letter
    - e.g. **`L1`**, **`L2`**, **`L3`**, …

Some Examples: A8P6

Rules for While Loops

- statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

Notes

- create a series of unique labels: **L1**, **L2**, etc.

Code

```
;; code(statement) =
L1:
     code(test)               ; $3 ← test
     beq $3, $0, L2           ; if test false exit loop
     code(statements)
     beq $0, $0, L1           ; retest condition
L2:
```

Some Examples: A8P6

Rules for While Loops

- statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

Notes

- limited to branch 2^{15}-1 instructions forward
- for assignments, no need to branch any farther
- in general, it limits the number of instructions created by the code(statements) line
- otherwise must do something like the following to jump farther

```
lis $6
.word L3
jr $6
```

Some Examples: A8P8

Rules for If Statements

- statement → IF LPAREN test RPAREN LBRACE statements_1 RBRACE ELSE LBRACE statements_2 RBRACE

Notes

- continue using the unique labels: **L4**, **L5**, … etc.

Code

```
;; code(statement) =
     code(test)             ; $3 ← test
     beq $3, $0, L4         ; if test false do statements₂
     code(statements₁)
     beq $0, $0, L5         ; skip statements₂
L4:
     code(statements₂)
L5:
```

Summary

Notes

- you now have all the ideas to generate code for Assignment 8!
- You can handle a single function that always takes two parameters and returns an integer.
- inside the body of the function you can have
  - additional *declarations* and *assignments* (e.g. a=1;)
  - *control structure*s {if-else, while loops}
  - using a variety of *comparison tests*: { <, <=, >, >=, ==, !=}
  - various *arithmetic operations* {+, -, *, /, %}

Summary

Notes

- *Hint:* generate comments with your code to aid debugging
- automatically generated code is harder to follow
- there can be a lot of it
- *Hint:* test your code
- create a bunch of small programs that test a single aspects of your code
- you are missing
  - pointers / memory allocation and deallocation
  - multiple procedures (i.e. one procedure calling another)

Topic 16 – Code Generation: Pointers

Key Ideas

- lvalue
- representing NULL
- pointer arithmetic
- pointer comparisons

References

- CS241 – WLP4 Programming Language Specification
- CS241 Assignment 9: P1- P4

Preview of A9

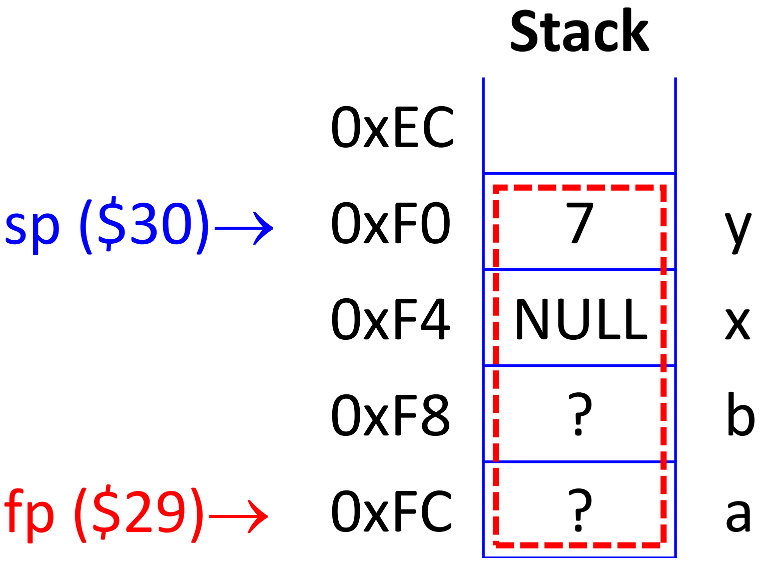
Overview

- *Our Goal:* generate a MIPS assembly language program that is equivalent to the WLP4 version (same input → same output and return value)
- We have two flavours of loaders
  - mips.twoints
  - mips.array
- WLP4 allows *arrays to be declared, initialized, dynamically allocated and destroyed*
  - represent an array as an **`int*`** that points to the first element of the array
- can also use pointers on their own (without involving arrays)

An Example

Pointers

```
int wain(int a, int b) {
  int *x = NULL;
  int y = 7;
  x = &y;
  return (*x);
}
```



Stack

	0xEC		
sp ($30)→	0xF0	7	y
	0xF4	NULL	x
	0xF8	?	b
fp ($29)→	0xFC	?	a

- What does this program do?
- How do we implement it in MIPS?
- *Hint:* let our grammar rules be our guide, i.e. syntax-directed translation

Pointer Specifications

Specifications

- *The WLP4 compiler must support:*
  - Dynamically allocating and deallocating (heap) memory
  - Assignment through pointers
  - Dereferencing (*) and address-of (&) operators
  - pointer arithmetic
  - pointer comparisons
  - the NULL pointer

Example: A9 P1

Dereferencing a Pointer

- $\text{factor}_1 \rightarrow \text{STAR factor}_2$
- Example:

  *p
- Solution:
  - here you are dereferencing the pointer p
  - i.e. returning the contents of the address stored in p
  - generate the code for factor_2, then interpret the results (which is in $3) as an address and load the contents of that address into $3

  ```
  code(factor₁) = code(factor₂)
                  lw $3, 0($3)
  ```

Example: A9 P1

Code for NULL

- factor → NULL
- Requirement:
  dereferencing a NULL pointer should crash the MIPS machine
- Solution: make NULL = 0x01,
  - not word aligned, i.e. the address is not divisible by 4
  - any attempt to use this address (with the `lw` or `sw` MIPS instruction) will crash the machine
  - implementation: move 0x01 into register $3

    code(NULL) = `add $3, $0, $11`
  - in most other languages NULL is 0x0 and it is the OS that prevents using 0x0 as a address.

Example: A9 P1

Lvalues

Informally, there are two ways to think about *lvalues*

1) An lvalue is something that can appear on the *left hand side of an assignment*, i.e. it can be assigned a value.

These are Correct

```
int a = 0;
int *p = NULL;
a = b - (c + 2);
p = &a;
```

These are Incorrect

```
0 = a;
NULL = *p;
b - (c + 2) = a;
&b = p;
```

Here a, p and *p are lvalues.

0, NULL, b-(c+2) and &a are not lvalues.

Example: A9 P1

Lvalues

2) An lvalue is a value that *has a location* in RAM and *a type associated with that location*, i.e. a location value, e.g.
   - a=1 means store the value 1 in the location specified by a
   - p=&a means p now refers to the same location as a refers to

- In different programming languages, lvalues can have slightly different meanings
- Even in the same language, it can mean different things in different standards:
  - In C89 the meaning is closer to version 2) above
  - Recognizing that a variable can be declared `const` in C, C99 is closer to a combination of versions 1) and 2)

Example: A9 P1

Lvalues

- In WLP4, lvalue appears in five production rules
  1) statement → lvalue BECOMES expr SEMI
     you can assign to it
  2) factor→ AMP lvalue
     it has a address

how it is used

  3) lvalue → ID
     it can be an ID
  4) lvalue → STAR factor
     it can be a dereferenced factor
  5) lvalue → LPAREN lvalue RPAREN
     putting parenthesis around an lvalue is still an lvalue

what it is

Example: A9 P1

Code for Address-of

- factor → AMP lvalue
- lvalue has an address
- it cannot be something like NULL or 3
- the rule is factor → AMP lvalue rather than factor → AMP factor in order to prohibit code like “&NULL” or “&3”
- Question: What directly derives from an lvalue?
- Answer: 3 cases

1. lvalue → ID e.g. `a = b`
2. lvalue → STAR factor e.g. `*p = b`
3. lvalue → LPAREN lvalue RPAREN e.g. `(a) = b` or `(*p) = b`

Example: A9 P1

Code for Case 1: Address-of

- factor → AMP lvalue
- lvalue → ID
- the statement "`&y`" is asking for the address where the variable `y` is stored, so look it up in the symbol table
- the address is stored as an offset from the frame pointer ($29) so get the actual address by adding the variable's offset to $29
- use "lis $3" and the ".word" directive
- implementation:

  code(factor) = lookup the offset of ID in the symbol table

```
lis $3
.word ID_offset
add $3, $3, $29
```

Example: A9 P1

Program

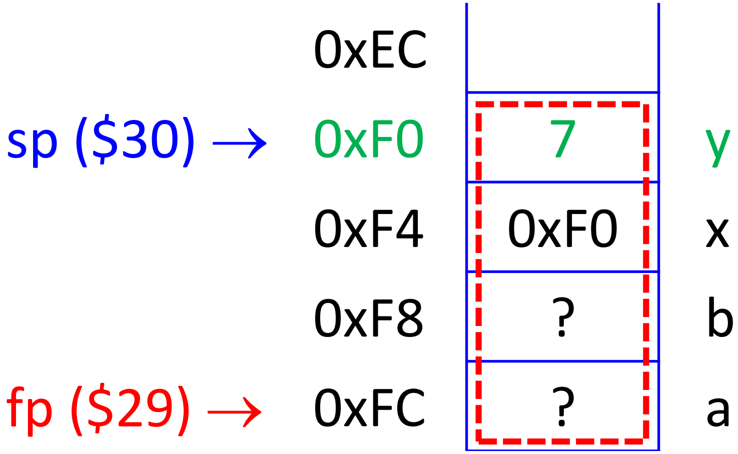
```
int wain(int a, int b) {
  int *x = NULL;
  int y = 7;
  x = &y;
  return (*x);
}
```

E.g. for the statement "&y"

- y's offset is -0xC
- &y = $29 + y's offset from fp
- &y = 0xFC + (-0xC) = 0xF0

```
  lis $3
  .word -0xC
  add $3, $3, $29
```

Stack



0xEC

sp ($30) → 0xF0 | 7 | y

0xF4 | 0xF0 | x

0xF8 | ? | b

fp ($29) → 0xFC | ? | a

Symbol Table

Name	Type	Offset from fp
a	int	0x0
b	int	-0x4
x	int*	-0x8
y	int	-0xC

Example: A9 P1

Code for Case 2: Address-of

- $\text{factor}_1 \rightarrow$ AMP lvalue
- lvalue $\rightarrow$ STAR factor_2
- we will say "`&(*y) = y`", i.e. the two operators cancelled each other out
- implementation:
  $\text{code}(\text{factor}_1) = \text{code}(\text{factor}_2)$

Code for Case 3: Address-of

- factor $\rightarrow$ AMP lvalue_1
- $\text{lvalue}_1 \rightarrow$ LPAREN lvalue_2 RPAREN
- here "`&(y) = &y`", i.e. parenthesis do not change the lvalue
- implementation:
  $\text{code}(\text{lvalue}_1) = \text{code}(\text{lvalue}_2)$

Example: A9 P1

Assignment to a Pointer

- lvalue → STAR factor
- recall what happened in A8P5 for the production rule statement → lvalue BECOMES expr SEMI

  code(statement) = code(expr) ; $3 ← expr

  ```
  sw $3, ID_offset($29)
  ```
- i.e. store the value of the expression at the address of the variable, i.e. frame pointer ($29) plus variable's offset
- works if expr is type int and lvalue is an int variable *but not if lvalue is an int* variable*
- e.g. *p = 2;
- for this rule you must know the lvalue type to generate the code

Example: A9 P1

Assignment to a Pointer

- statement → lvalue BECOMES expr SEMI
- lvalue → STAR factor
- calculate the value (address) of lvalue
- then store the result of expr at that address.
- Solution
  - calculate the code for expr and push the result onto the stack
  - calculate the code for lvalue (an address) and leave in $3
  - pop stack into $5 and store the results at the address in $3

```
code(statement) = code(expr)          ; $3 ← expr
                  push($3)            ; stack ← expr
                  code(lvalue)        ; $3 ← lvalue
                  $5 = pop()          ; $5 ← expr
                  sw $5, 0($3)
```

Background for A9 P1

A Simple Array

```
int wain(int *a, int n) {
  return *a;
}
```

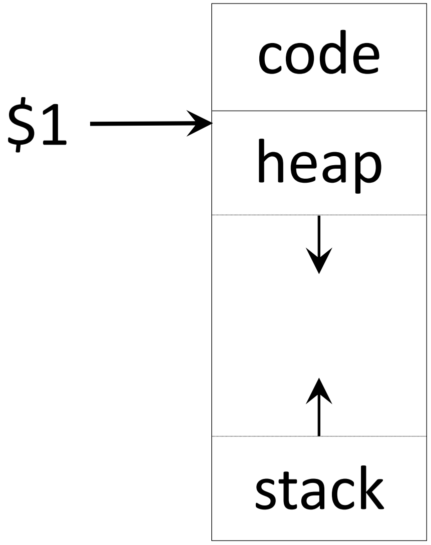
- E.g. format for mips.array loader back in A2
- What does the program do
  - Answer: return the first element of the array
- How do we do this in MIPS?
  - *find the base address for the array* (in $1) and copy it over to $3

    ```
    lw $3, 0($1)
    ```
- What is mips.array actually doing?

Background for A9 P1

A Simple Array

```
% cat ex1.wlp4 | wlp4scan | wlp4parse | ./wlp4gen |
   cs241.binasm > ex1.mips
% mips.array ex1.mips
  Enter length of array: 3
  Enter array element 0: 10
  Enter array element 1: 11
  Enter array element 2: 12
```



- What is mips.array actually doing?
- *It allocates memory on the heap* and then calls **`wain`** with the location of the array ($1) and its size ($2) as parameters

Background for A9 P1

Another Simple Array

```
int wain(int *a, int n) {
  return *(a+1);
}
```

- What does this program do?
  - Answer: it returns element a[1] of the array,
  - a[1] = *(a+1) = *(1+a)
  - the size of each element in the array (an int) is 4 bytes so we are actually adding 4 to the base address to get the address of the next element a[1]

Example: A9 P2

Dynamic Memory Allocation

- factor → NEW INT LBRACK expr RBRACK
- statement → DELETE LBRACK RBRACK expr SEMI
- we (CS241) provide the library routines that handles memory management
- you must include the following directives in the prolog
  - `.import init`
  - `.import new`
  - `.import delete`
  - call **`init`** to initialize the heap
    - see assignment for details on parameters for **`init`**
  - link in alloc.merl (which we provide for you) as the last object file to link in (we’ll talk about how linking works later)

Example: A9 P2

Dynamic Memory Allocation

- factor → NEW INT LBRACK expr RBRACK
- statement → DELETE LBRACK RBRACK expr SEMI
- **`init`** initializes the data structures within the dynamic memory module
- **`new`** allocates memory from the heap
  - $1 is the size of the array requested
  - it returns
    - the address of 0th element (*base address*) in $3 if successful
    - 0 in $3 if memory is exhausted
- **`delete`** frees up the memory
  - $1 is the base address of the array
  - must delete the whole array (not part of it)
  - it does not check if $1=NULL

Example: A9 P3

Pointer Arithmetic: PLUS

- $expr_1$ → $expr_2$ PLUS term
- **if** type($expr_2$) == int and type(term) == int

  then do as in A8: code($expr_2$), push on stack, code(term), pop stack into $5 and append instruction add $3, $5, $3

  else if type($expr_2$) == int* and type(term) == int

```
code($expr_1$) =  code($expr_2$)        ; $3 ← $expr_2$
                  push($3)              ; stack ← $expr_2$
                  code(term)            ; evaluate term
                  mult $3, $4           ; multiply term by 4
                  mflo $3               ;  i.e. the size of one word
                  $5 = pop()            ; $5 ← $expr_2$
                  add $3, $5, $3
```

Example: A9 P3

Pointer Arithmetic: PLUS

- **else if** type(expr$_2$) == int and type(term) == int*
  - left as an exercise
- Notes:
  - *you must know the types of the children* expr$_2$ and term
  - typically you would *store type info in the parse tree nodes*
  - the code for “int*, int” is much the same as for “int, int” with the exception of the additional statements in red
  - this statement is used to index into an array, so you need to consider the width of the elements in the array
  - we take a similar approach for subtraction

Example: A9 P3

Pointer Arithmetic: MINUS

- $\text{expr}_1 \rightarrow \text{expr}_2$ MINUS term
- **if** type(expr_2) == int and type(term) == int

  then do as in A8: code(expr_2), push on stack, code(term), pop stack into $5 and append instruction add $3, $5, $3

  else if type(expr_2) == int* and type(term) == int

```
code(expr_1) = code(expr_2)              ; $3 ← expr_2
               push($3)                  ; stack ← expr_2
               code(term)                ; evaluate term
               mult $3, $4               ; multiply term by 4
               mflo $3                   ;  i.e. the size of one word
               $5 = pop()                ; $5 ← expr_2
               sub $3, $5, $3
```

Example: A9 P3

Pointer Arithmetic: MINUS

- **else if** type($expr_2$) == int* and type(term) == int*
  same as integer subtraction *but divide result by 4*

```
code($expr_1$) = code($expr_2$)          ; $3 ← $expr_2$
                 push($3)                ; stack ← $expr_2$
                 code(term)              ; evaluate term
                 $5 = pop()              ; $5 ← $expr_2$
                 sub $3, $5, $3
                 div $3, $4              ; divide result by 4
                 mflo $3
```

Example: A9 P4

Pointer Comparisons

- test → expr_1 LT expr_2
- since the code has already successfully passed through the context-sensitive analysis phase before reaching the code generation phase, the types of expr_1 and expr_2 match
- What needs to change if type(expr_1) == *int ?
  - for A8 (integers) you used the instruction slt $3, $5, $3
  - for pointers use the instruction sltu $3, $5, $3
  - addresses / pointers are unsigned integers
  - they can range from 0 to 2^{32}-4

Topic 17 - Code Generation: Procedures

Key Ideas

- procedure prologs and epilogs
- three tasks
  1. saving registers values between function calls
  2. saving the frame pointer
  3. passing function arguments
- handling namespace collisions

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 10.1-10.5
- CS241 – WLP4 Programming Language Specification

Review: Prologs and Epilogs

Recall from our Discussion of Code Generation

For the procedure **`wain`**

- Prolog
  - initializations (constants, **`.import`**'s, and call **`init`**
  - push the return address ($31) on the stack
  - push a stack frame and store args ($1 and $2) in the frame
- Body of Procedure
  - generate code for the body of the procedure
- Epilog
  - pop frame (local variables and arguments) off the stack
  - restore previous return address to $31
- *Key Challenge: How to handle (1) registers (2) frame pointer and (3) passing arguments* for functions calling other functions?

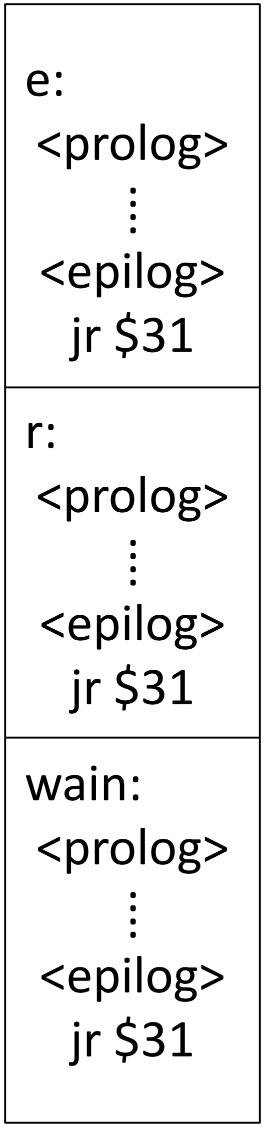
A9 P5: Multiple Procedures

Prologs and Epilogs

- *Question:* What is handled in the prologs and epilogs of procedures **`e()`**, **`r()`** and **`wain()`**?

```
int e(…) {…}          // the callee
int r(…) {…}          // the caller
int wain(…) {…}
```

- *Handled once* in **`wain`**'s prolog
  - **`.import`**'s
  - initializations ($4 ←4, $11 ←1, **`init`** etc.)
- *Handled in each procedure's* prolog and epilog
  - frame and frame pointer, $29
  - save and restore our caller's return address, $31
  - save and restore the other registers



e:
<prolog>
⋮
<epilog>
jr $31

r:
<prolog>
⋮
<epilog>
jr $31

wain:
<prolog>
⋮
<epilog>
jr $31

A9 P5: Multiple Procedures

Q1: Saving Register Values: Three Approaches

Question 1: who saves what registers?

- say procedure r( ) calls procedure e( ), i.e.
  int r(…) { … e(…)… }

a) *the caller* r( ) *saves any register values it needs*

- r( ) saves all the registers that have values that need to be saved (e.g. intermediate results)
- r( ) may be saving registers that e( ) will not modify

b) *the callee* e( ) *saves any register values it modifies*

- e( ) saves all registers whose values it will overwrite
- e( ) may be saving register that r( ) no longer needs

```
int e(…) {
  ⋮
}

int r(…) {
  ⋮
  e();
  ⋮
}
```

A9 P5: Multiple Procedures

Q1: Saving Register Values: Three Approaches …

Question 1: who saves what registers?

c) *hybrid (recommended approach)*

- caller saves some registers and callee saves others
  - caller saves $31 (because its value is overwritten when the instruction `jalr` is executed)
  - callee saves the registers whose values it will modify
- this is the approach we've been following so far
- other hybrid approaches are possible

```
int e(…) {
  ⋮
}
```

```
int r(…) {
  ⋮
  e();
  ⋮
}
```

A9 P5: Multiple Procedures

Q2: Who Saves the Frame Pointer, $29?

- *if the callee* e( ) *saves $29* then it saves the registers and updates $29 to point to the start of its frame
  - *if we update $29 first* (before saving the registers):
    - then we've changed $29's value before saving it
  - *if registers are saved first* (before $29 is updated):
    - saving the registers on the stack will change the value of the stack pointer, $30
    - since we are saving the registers in the stack frame, now we need to track how many registers we have saved to calculate the start of the stack frame
    - i.e. $29 = $30 + 4 × (number of registers saved)
    - doable, but must track number of registers saved

A9 P5: Multiple Procedures

Q2: Who Saves the Frame Pointer, $29?

- *the caller* r( ) *saves $29*
  - r( ) saves its value for $29
  - e( ) updates the value of $29 (based on the stack pointer)
  - easier to implement
- Answer: (i.e. *recommended approach*)
  - caller saves $29 and $31, then calls procedure
  - when procedure returns, caller restores $31 and $29

```
int e (…) {
  sub $29,$30,$4
  ⋮
}

int r (…) {
  ⋮
  push($29)
  push($31)
  lis $5
  .word e
  jalr $5
  $31 ← pop()
  $29 ← pop()
  ⋮
}
```

A9 P6: Passing Arguments

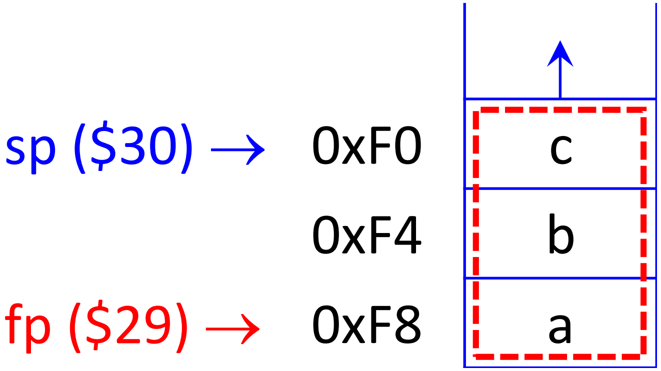
Program

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

Currently for `wain`

- save arguments (always 2 of them) and local variables on the stack
- frame pointer ($29) points to the beginning of the frame
- stack pointer ($30) points to the top of the stack
- locations in the symbol table are relative to the frame pointer ($29)

Stack



Symbol Table

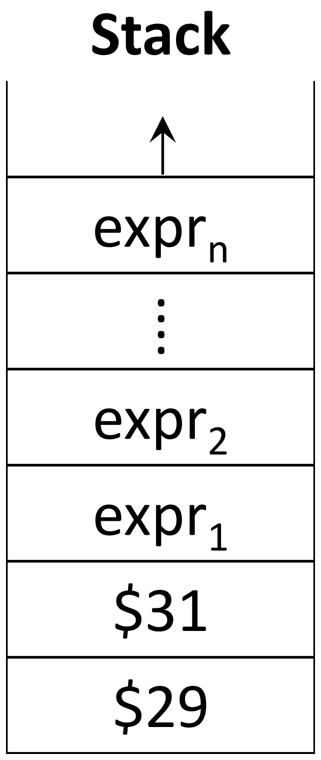
Name	Type	Offset
a	int	0x0
b	int	-0x4
c	int	-0x8

A9 P6: Passing Arguments

Passing a Varying Number of Arguments

- *Problem:* Could use registers for arguments but what if there are a lot of them? e.g. factor → ID($expr_1$, $expr_2$, …, $expr_n$)
- Solution: caller *loads arguments on the stack*, e.g.

```
code(factor) =
    push($29)
    push($31)
    code($expr_1$)
    push($3)
        ⋮
    code($expr_n$)
    push($3)
    lis $5
    .word ID
    jalr $5
    pop args
    $31 = pop()
    $29 = pop()
```



A9 P6: Passing Arguments

Generating Code for a Procedure

- procedure → INT ID(params) { dcls stmts RETURN expr ; }
- Note: The caller has already placed the params on the stack.

Output

```
code( procedure ) =
    sub $29, $30, $4                     ; update frame pointer
    push registers (that will be used)   ; callee saves registers
    code(dcls)                           ; that will be overwritten
    code(stmts)
    code(expr)
    pop registers (that were saved)      ; callee restores registers
    add $30, $29, $4                     ; pop stack frame
    jr $31                               ; return to caller
```

A9 P6: Passing Arguments

Q3: Where to Save Registers

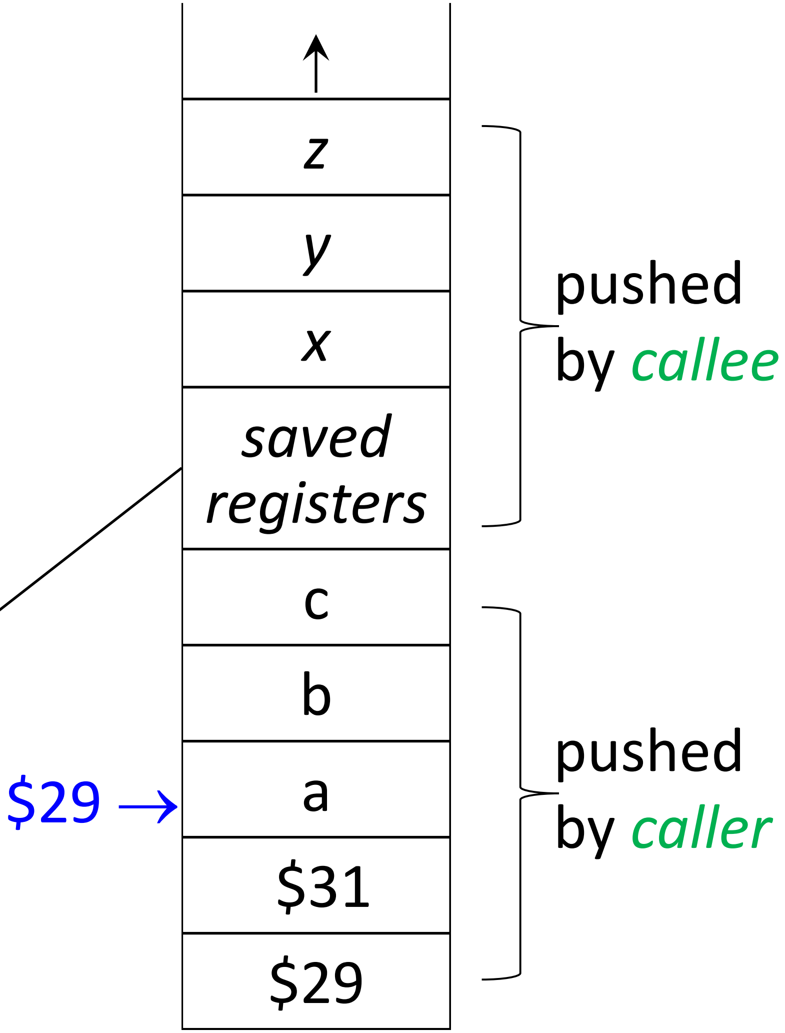
```
int e(int a, int b, int c) {
  int x = 0;
  int y = 0;
  int z = 0;
  ...
}
```

Symbol Table

Name	Type	Offset
a	int	0x0
b	int	-0x4
c	int	-0x8
x	int	~~-0xC~~
y	int	~~-0x10~~
z	int	~~-0x14~~

values depend on # of registers saved

Stack

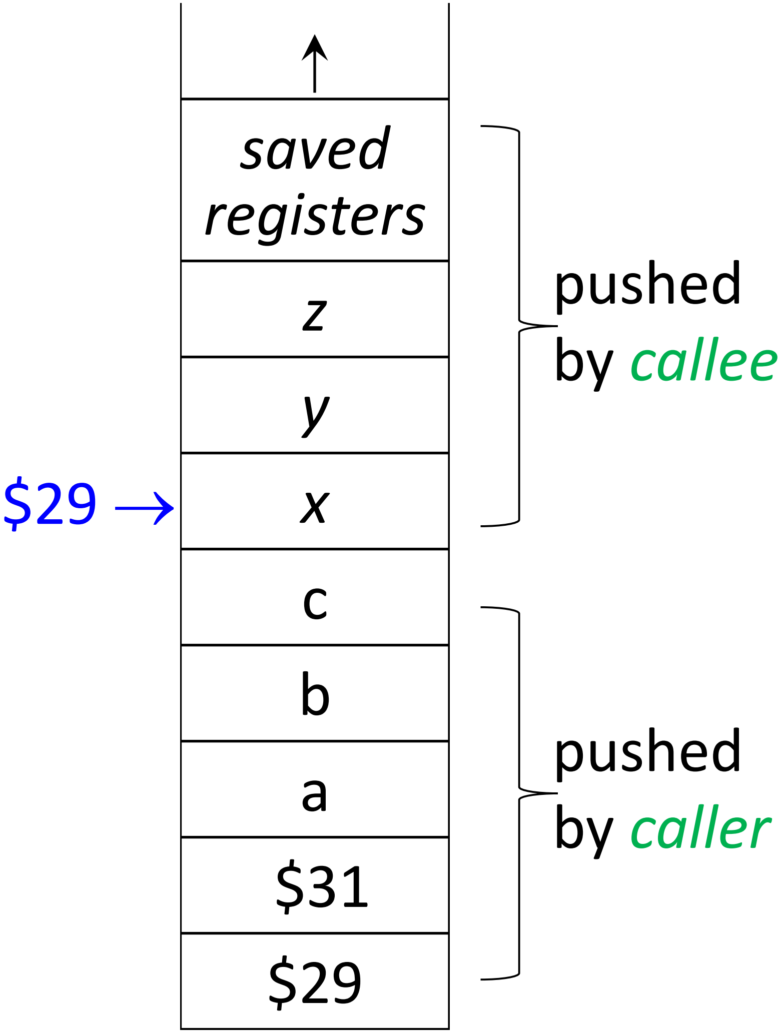


A9 P6: Passing Arguments

Q3: Where to Save Registers

- *Problem* (on previous slide): the arguments for *e*( ), i.e. a, b, c, and its local variables i.e. x, y, z, are separated by the saved registers
- some of the values in the symbol table (on the previous slide) are now incorrect
- *Solution:* could save registers after local variables
- or convert the old symbol values to the new symbol values: add (the number of parameters × 4)

Stack



A9 P6: Passing Arguments

Q3: Where to Save Registers

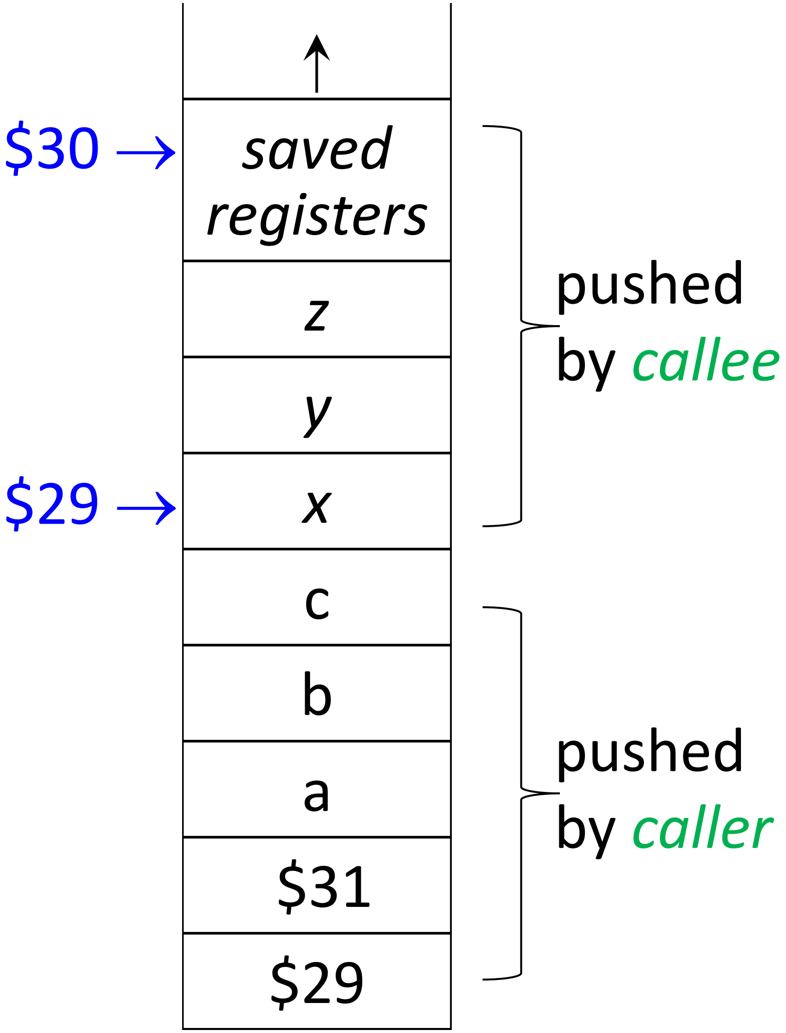
If saving after local variables

- *positive offsets* are the arguments
- *zero and negative offsets* are the local variables

Symbol Table

Name	Type	Offset
a	int	0xC
b	int	0x8
c	int	0x4
x	int	0x0
y	int	-0x4
z	int	-0x8

Stack



$30 →

saved registers

z

y

$29 →

x

pushed by *callee*

c

b

a

$31

$29

pushed by *caller*

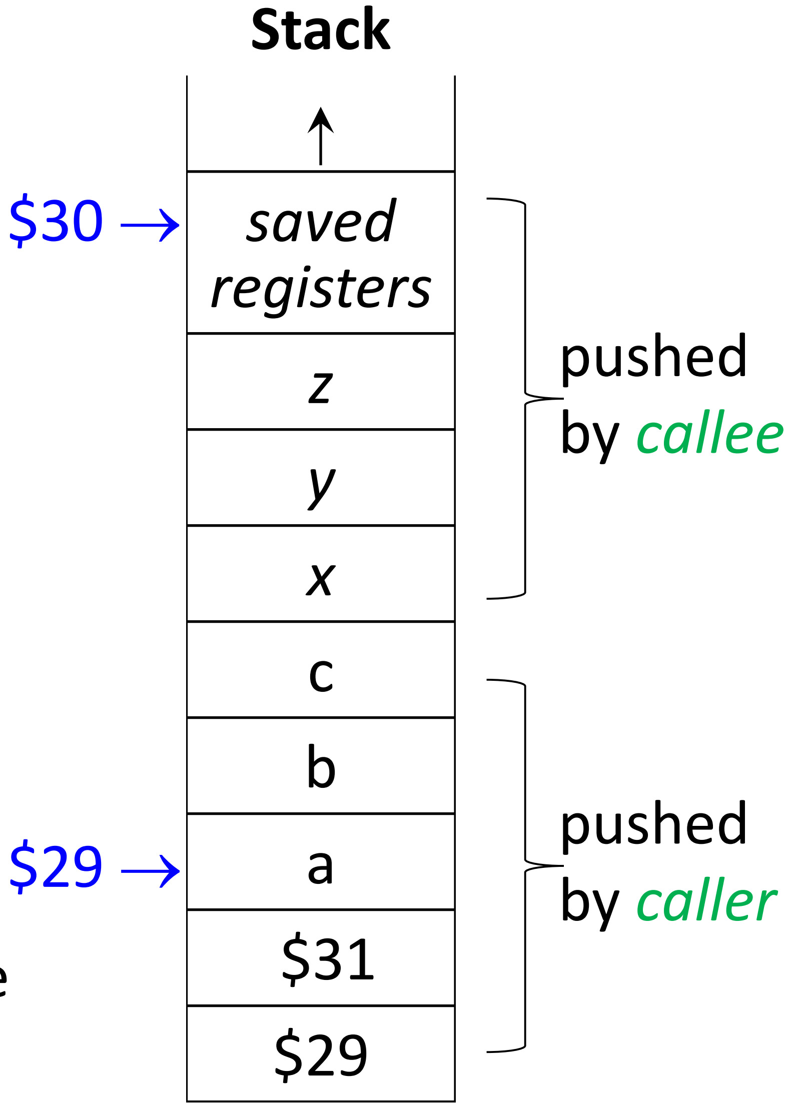
A9 P6: Passing Arguments

Q3: Saving Registers

- *alternative:* could keep $29 pointing at the first argument, i.e. at “a”.
- having the caller save the registers is **not** a good idea especially if one procedure, say f(), calls another procedure multiple times, e.g.

```
int f()  {
   g(1);
   g(2);
   g(3);
   }
```

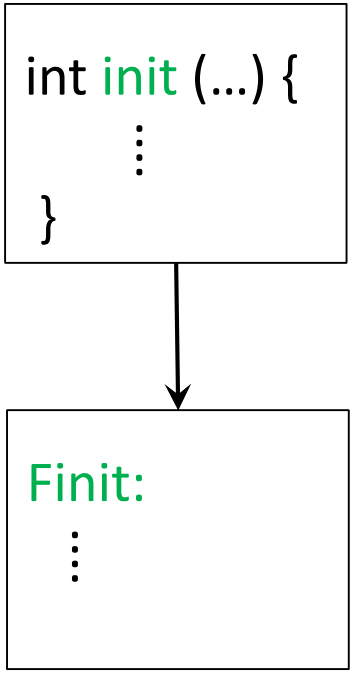
- reason: bigger size, i.e. repeating the same save and restore code 3 times.



Multiple Procedures: Namespace Collision

Namespace Collisions

- *Question:* If names of procedures map onto labels, what if a procedure uses the same name as a label in the runtime environment?
- called a *namespace collision*
- e.g. you have a function called init() and the underlying system already uses init as a label
- Solution: reserve the letter F for functions
- when processing WLP4 procedure names append the letter F in front of the corresponding MIPS assembly language label
- e.g. the procedure “int init(…) { … }” in WLP4 becomes “Finit: …” in MIPS assembly language.



```
int init (…) {
     ⋮
}
```

```
Finit:
  ⋮
```

Summary

Caller Pushes

- frame pointer $29
- return address $31
- arguments

onto the stack.

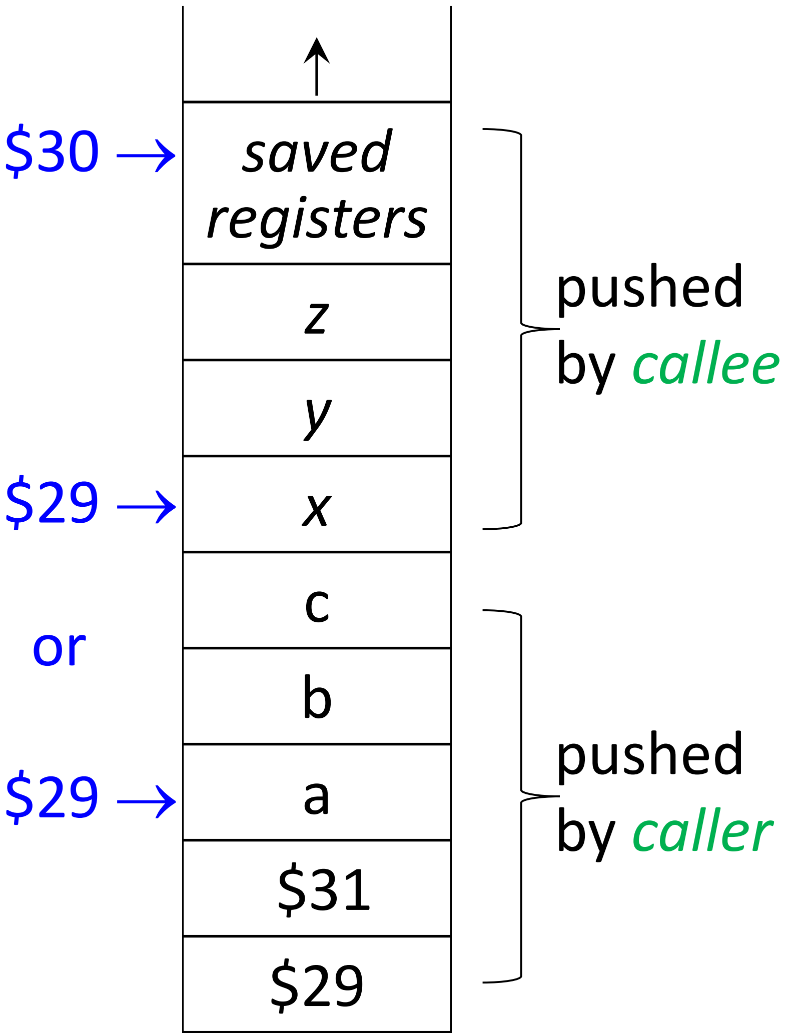
Callee Pushes

- local variables
- register values it will modify

onto the stack.

The frame pointer can point to the first argument (a) or the first local variable (x).

Stack



$30 →

saved registers

z

y

$29 →

x

c

or

b

$29 →

a

$31

$29

pushed by *callee*

pushed by *caller*

Topic 18 – Optimization

Key Ideas

- Common Subexpression Elimination
- Register Allocation
- Constant Folding
- Constant Propagation
- Dead-code Elimination
- Strength Reduction
- Inlining Procedures
- Tail Recursion

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 11.1 – 11.7 for more detailed explanation

Optimization (A9 Bonus)

Overview

- Recall: for any WLP4 program there are an infinite number of equivalent MIPS assembly language programs.
- What *criteria d*o we use to decide if one compiled version of a WLP4 program is better than another?
  - Answer: the time it takes for the program to run
- Finding the equivalent program with the minimum runtime is incomputable, so we must…
- Use *heuristics*: i.e. *recognize* a pattern of instructions and *replace* them with an equivalent set that
  - runs quicker or
  - (as an approximation) uses a smaller number of instructions

Optimization (A9 Bonus)

Overview

- *Key Point:* These patterns do not necessarily appear in the WLP4 source code
  - They may appear because code is generated by looking at one single node in the parse tree at a time
- *Observation:* for the code x = x+1;
  - in the subtree on the *left hand side* of the '=' sign the parser will generate code that gets the address of 'x'
  - on the subtree on the *right hand side* of the '=' sign the parser will generate code that gets the address of 'x'
  - the parser created code to calculate the same value twice
  - this observation leads to one form of optimization...

Optimization: Common Subexpression

Common Subexpression Elimination

- *Idea:* store the results of *common subexpressions* (often generated by the compiler not the programmer) in registers
- For example: (a+b) * (a+b),
  - calculate the answer to a+b and store in $3 then
    mult $3, $3
    mflo $3
- For "x = x+1" calculate the address of x once, use it twice
- *Caution:* it may not work with functions, e.g. f(1)+f(1) since the functions may have side effects, such as print output
- *Note:* It takes resources to find these common subexpressions
- e.g. the "g++" command runs much quicker than "g++ -O3"

Optimization: Register Allocation

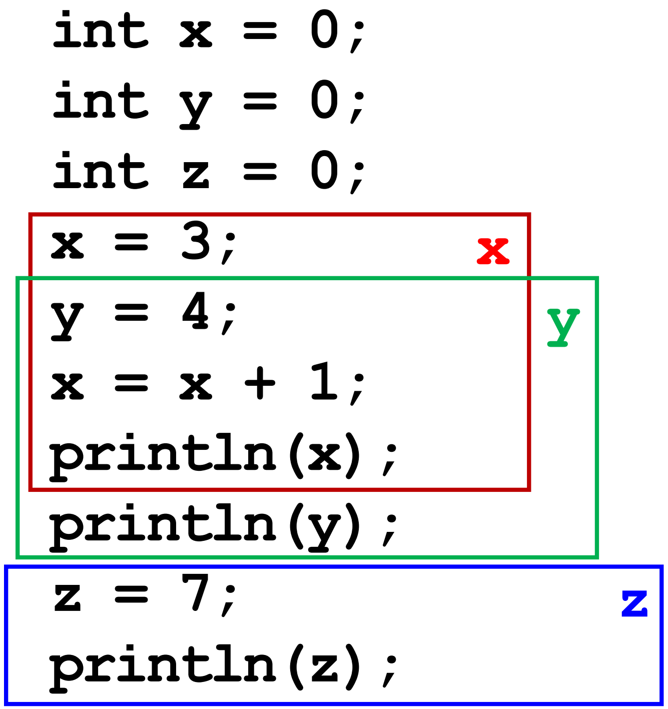
Register Allocation

- *Observation:* accessing a register is much quicker than accessing the stack or RAM in general
- using registers also eliminates the code that pushes and pops from the stack, or lw and sw instructions for accessing RAM
- our code generator does not use registers $14-$28
- *Challenge:* must decide how to allocate them if there are more than 15 variables, typically “most used”, or “most recently used”
- allocating these registers wisely is a key optimization strategy
- *Caution:* you cannot use the address-of operator on a register location, only a RAM location, so push these values into RAM

Optimization: Register Allocation

Register Allocation

- *Idea:* keep track of the *live ranges* of each variable: from where it is assigned a value to the location where it is used with that value.
- If the live ranges of two variables intersect, then you must use two different registers.
- If the live ranges do not intersect, you can reuse the register.



```
int x = 0;
int y = 0;
int z = 0;
x = 3;
y = 4;
x = x + 1;
println(x);
println(y);
z = 7;
println(z);
```

The live ranges of **x** and **y** intersect. The live range of **z** does not intersect with **x** or **y**.

Optimization: Register Allocation

Register Allocation

- *Idea:* **`code()`** specifies available registers in avail and returns where the result is located, e.g. for $expr_1 \rightarrow expr_2 + term$
- after generating the code for $expr_2$, the result is in **`s`**
- when generating the code for *term*, the set avail minus the register s is available for use
- *Enhancement:* provide the ability to specify where you want the result stored

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

// new way
code(expr1, avail)=
s = code(expr2, avail)
t = code(term, avail\{s})
add $s, $s, $t
return s
```

Optimization: Constant Folding

Example: Code for 2+3

- reduce the number of instructions by calculating answers involving constants at compile time

Currently 9 Instructions vs. *Only 2 Instructions*

```
code(2+3) =
     lis $3                ; load 2
     .word 2
     sw $3, -4(30)         ; push 2 on stack
     sub $30, $30, $4
     lis $3                ; load 3
     .word 3
     lw $5, 0($30)         ; pop 2 off stack
     add $30, $30, $4
     add $3, $5, $3        ; answer
```

```
code(2+3) =
     lis $3
     .word 5
```

Optimization: Constant Propagation

Constant Propagation

WLP4 Code:

```
int x = 2;
// value of x does not change
return x + x;
```

- *Approach*: Recognize that the value doesn't change and return 4.
- If it is the only place that **x** is used, it does not need a stack entry.
- What our compiler currently does:
  - load the value 2 into $3 (2 instructions): lis and .word
  - store result in **x** (1 instruction): sw
  - push value stored in **x** on stack (3 instructions): lw, sw and sub
  - load value stored in **x** into $3 (1 instruction): lw
  - move **x** from stack to $5 (2 instructions) : lw and sub
  - then add $5 and $3 (1 instruction): add

Optimization: Constant Propagation

Constant Propagation

WLP4 Code:
```
int x = 2;
// value of x does not change
return x + x;
```

- Since x is always 2, the compiler could do the following

```
lis $3              ; load 2 into x
.word 2
sw $3, -12($29)     ; where the offset to x is -12
;; do other stuff
lis $3              ; return value is 4
.word 4
jr $31              ; return from function
```

Optimization: Constant Propagation

Constant Propagation

- *Challenge:* need a way to detect and propagate constants
- *Solution:* The function code() could return an order pair (encoding, value) e.g.
  - (register, 3) would say the result is in $3 (this has been the only option so far)
  - (const, 2) would say the result is the constant 2
- E.g. if the rule $expr_1 \rightarrow expr_2 + term$ had $expr_2$ and $term$ both evaluate to constants, e.g. (const, 2) and (const, 3), then $expr_1$ would evaluate to (const, 5)
- (const, 5) would result in two lines of code

```
lis $3
.word 5
```

Optimization: Dead-code Elimination

Dead code

- Sometimes when code is generated, dead code is created.
- *Dead code* is
  - code that is never executed, e.g.
  - because a logical test is always false
  - because it occurs after a return statement (not in WLP4)
  - code that is executed but whose results are never used
- *Idea:* detect and do not output dead code.

Optimization: Strength Reduction

Strength Reduction

- *Approach:* some operations can be replaced by faster ones
- *Observation:* for CS241 addition is quicker than multiplication by two.

Currently 8 Instructions vs.

```
code(n*2;) =
        sw $3, 0(30)        ; push n on stack
        sub $30, $30, $4
        lis $3              ; load 2 into $3
        .word 2
        lw $5, 0($30)       ; pop n off stack
        add $30, $30, $4
        mult $3, $5         ; multiply 2 * n
        mflo $3             ; load answer in $3
```

Only 1 Instruction

```
code(n*2;) =
        add $3, $3, $3
```

Optimization: Inlining Procedures

Inlining Procedures

Inlining replace a function call with the body of the function, i.e.

- Replace

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

  with

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

- Pros:
  - if all calls to `f` are in-lined, no need to generate code for `f` at all
  - save overhead of creating a stack frame for `f`
- Con:
  - if `f` is big or used often, then we generate a lot of extra code
  - difficult to do for recursive functions

Optimization: Tail Recursion in Procedures

Tail Recursion

```
int fact(int n, int a){
  if(n == 0)
    return a;
  else
    return fact(n-1,n*a);
}
```

- *Note :* the very last instruction the function does is a recursive call, i.e. **`else return fact(…);`**
- *Optimization:* The content of the current stack frame (local variables etc.) will not be used again in the call of the function, therefore ⇒ reuse the stack frame for the next recursive call
- Won’t work for WLP4: only one return statement is allowed

Optimization

Intermediate Code

- *Challenge:* one of the challenges that many of these approaches have is that it is difficult to find patterns such as common subexpressions
- Approach: generally, but beyond the scope of this course, after the lexical, syntactic, and semantic analysis stages, an *intermediate code* (rather than assembly language) is generated with the idea that this code is easier to optimize than the final assembly language
- After optimization the intermediate code is converted to assembly language for a particular processor.
- Would only need to change the final step to create code for different processors (x86-64 vs. ARM-8 vs. MIPS)

Topic 19 – Heap Management

Key Ideas

- system stack vs. heap
- components of heap management
- automatic vs. manual memory management
- fragmentation: internal and external
- allocation strategies: first fit, best fit, worst fit
- dlmalloc
- garbage collection
  - reference count
  - mark and sweep
  - copying collectors

Code Gen for New and Delete

Rules in WLP4 that Deal with Arrays and Pointers

- *Recall*: in WLP4 we had two functions to deal with memory management: **`new`** and **`delete`**

```
int* ia = NULL;
  ⋮
ia = new int[100];
  ⋮
delete [] ia;
```

- The underlying system, alloc.merl, supported three subroutines

  1. **`init`**
  2. **`new`**
  3. **`delete`**

Overview of Heap Management

The Challenge

- Procedure arguments, return values, and local variables can all be handled elegantly with the system stack.
- Stack frames for nested procedure calls and returns follow a last in first out (LIFO) pattern suitable for a stack.
- Dealing with **`new`** and **`delete`** (i.e. *dynamic allocation* and *reclamation*) is a much more problematic issue.
- *The Problem:* they can be called in an unpredictable pattern
  - they may be called within **`if`** statements
  - they don't necessarily follow a pattern like Last In First Out
  - e.g. if **`new`** was called in the order: **`new a; new b; new c;`** **`a`**, **`b`** and **`c`** could be deleted in any order.

Overview of Heap Management

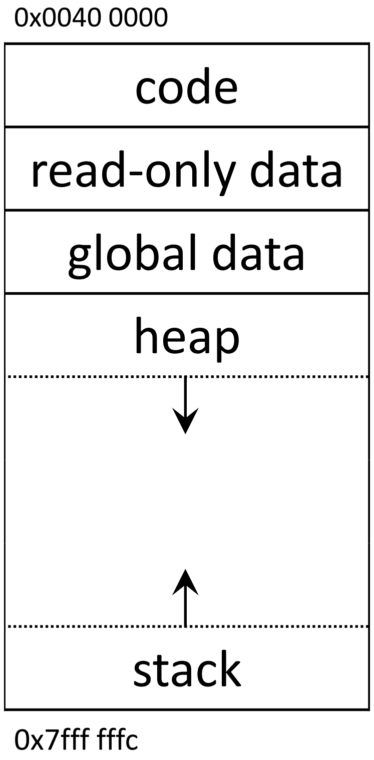
The Challenge

- *Key differences:* Local variables disappear once the function that they are declared in returns, but dynamically allocated arrays can remain even after the function has returned.
- Many data structures can grow and shrink dynamically (e.g. a linked list), i.e. their size is not known at compile time
- *Consequences:* Because of these differences, it is not efficient to store dynamically allocated memory in the system stack.
- *Solution:* Instead another region of memory is reserved for dynamically allocated memory: the *heap*
- Here heap means RAM available for dynamic allocation (not a balanced binary tree for finding the max or min element).

Overview of Heap Management

Solution: Typical Layout in Memory

- The *code, read-only data* and *global data* have relatively low addresses (near 0x0040 0000) and they do not change size as the program runs.
- The *stack* has relatively high values for addresses (0x7fff fffc) and grows towards the heap.
- The size of the stack will change as functions get called and return.
- The *heap* is located near the global data and grows towards the stack.
- The size of the heap can grow if a program requires a lot of dynamic memory.



Overview of Heap Management

The Components

- There are three tasks to consider for memory management ...

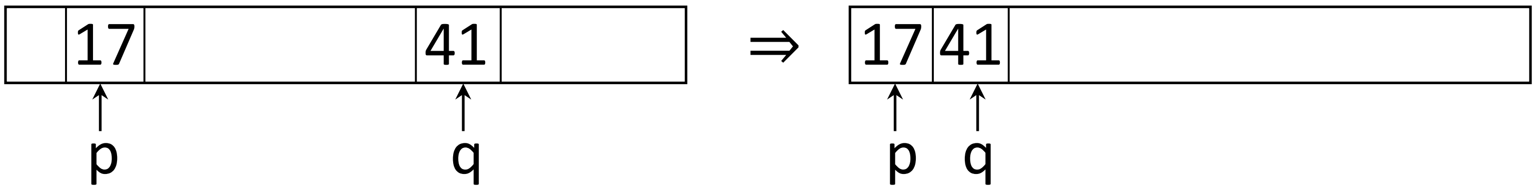
	System Stack	Heap
1. *Initialization*	done by O/S	init
2. *Allocation*	push()	new
3. *Reclamation*	pop()	delete

- The operating system (O/S) initializes the system stack.
- Procedures are implemented to manage the allocation and reclamation of the system stack efficiently.
- How the heap is managed varies: there are many possibilities ...

Overview of Heap Management

Varieties of Heap Management

- memory can be *allocated* implicitly (it just happens) or explicitly (i.e. the function **`new`** is called).
- memory can be *reclaimed* implicitly (it just happens) or explicitly (i.e. the function **`delete`** is called).
- memory can be *allocated* in one size only (a *fixed size*) or in many sizes (a *variable size*)
- some languages (not WLP4 or C++) allow pointers to be *relocated* in order to fill in spaces between allocated memory



Overview of Heap Management

Implicit vs. Explicit

- Many languages (such as Racket, Java and Python) have implicit / *automatic memory management*
  - the program creates new objects and a procedure runs in the background that decides when to free up the memory for the object because it is no longer being used (a.k.a. *garbage collection*).
- Other languages (like WLP4, C, and C++) have explicit / *manual memory management*
  - the programmer calls **`delete`** on any memory that is no longer needed
  - the risk is you can call **`delete`** too early, too late or too often

Overview of Heap Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce…

- *dangling pointer* errors: using memory that has been freed, i.e. **`delete`** has been called *too early*

```
int* ia = NULL;
ia = new int[100];
delete [] ia;
 ⋮
ia[0] = 17;     // error: dangling pointer!
```

- Risks: if that memory location is being used by another data structure, you are unintentionally modifying that data structure in an unpredictable way.

Overview of Heap Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce…

- *memory leaks:* you allocate memory but then have no pointers pointing to it, i.e. **delete** has been called *too late*

```
int* ia = NULL;
ia = new int[100];
 ⋮
ia = NULL;       // error: access to memory is lost!
```

- the program slowly uses up more and more memory
- risks: memory exhaustion (i.e. running out of memory)
- the risk increases if the program runs for a long time

Overview of Heap Management

Pros of Automatic Memory Management

With automatic memory management you avoid or substantially reduce…

- *deleting twice:* you call **`delete`** on the same memory location multiple times, i.e. **`delete`** has been called *too often.*

```
int* ia = NULL;
ia = new int[100];
delete [] ia;
 ⋮
delete [] ia;    // error: freeing twice!
```

- risks: can crash the system

Overview of Heap Management

Cons of Automatic Memory Management

With automatic memory management you

- use more resources (i.e. time to track memory usage)
- may have a performance impact
- possible stalls in program execution (i.e. not good for some real time programming applications)

Manual and Automatic Memory Management Commonalities

- With both you still need to track which locations in RAM are
  - *being used* (*a.k.a. live heap objects*)
  - *free* (available for use)

Explicit Memory Management

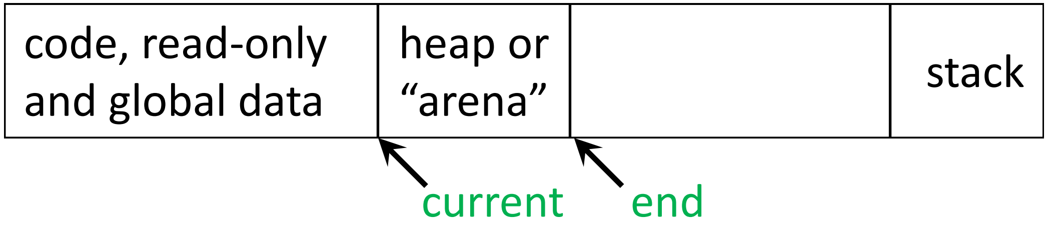
Basic Approach

- Carve out an *arena* from RAM, i.e. a large contiguous area of memory that gets allocated once and then is handed out in pieces called *blocks* using calls such as **`new`** or **`malloc`**
  - perhaps from the stack during the prolog for wain()
  - or the O/S provides it for you
  - we call this arena the *heap*
- this arena provides an area of memory that the **`new`** and **`delete`** procedures manage
- ***`new`*** (or **`malloc`**) *is easy if you don't have* ***`delete`*** (or **`free`**) and don't reuse the memory

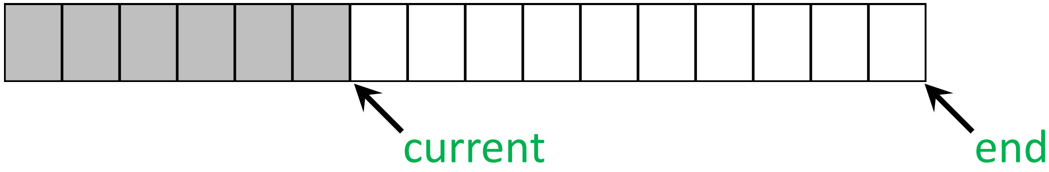
Explicit Memory Management

Approach 1: No Reclamation

- Features: fixed size, explicit allocation, *no reclamation*, no relocation
- Initialization: O(1) - set up current and end pointers



- Allocate: O(1) - move current forward (grey used, white available)

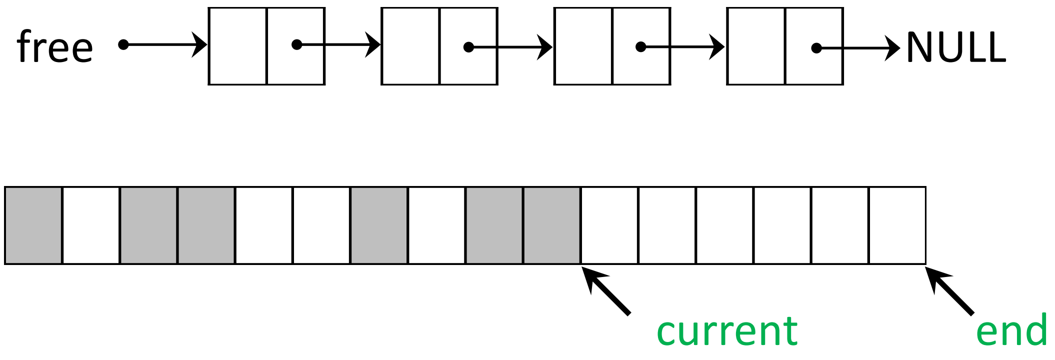


- Reclaim: do nothing
- Limitations: can exhaust memory quickly if it is not reused

Explicit Memory Management

Approach 2: Explicit Reclamation

- Features: fixed size, explicit allocation, *explicit reclamation,* no relocation
- Reclaiming Memory: keep a free list (list of locations that are available from the start of the heap until current)
- allocate from the free list first and only move the current pointer closer to end and use that new spot if the free list is empty



Explicit Memory Management

Approach 2: Explicit Reclamation

- Allocate: O(1)

```
if (free ≠ NULL) // free list not empty
    remove first block from free list
    return first block
else if (current ≠ end)
    return current and then increment it
else
    ERROR: memory exhausted
```

- Reclaim: O(1)

```
add to free list
```

Explicit Memory Management

Approach 3: Variable-sized blocks

Features: *variable-sized block*, explicit allocation, explicit reclamation, no pointer reallocation.

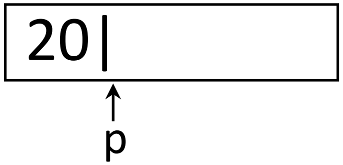
- idea: create a linked list of free blocks O(1)
- *init:* initially the entire heap is free and the linked list contains one entry (say 1024 bytes)
- free → [1024| NULL]

- Allocate: find a chunk of memory that is big enough

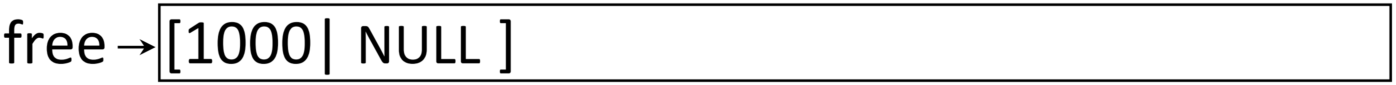
Explicit Memory Management

Variable-sized Blocks: Allocation

- if 20 bytes are requested
  - allocate 24 bytes:
    - the 1st part (4 bytes) stores *the size of the block*
    - the 2nd part (20 bytes) stores *the data*
  - return a pointer to the start of the data portion



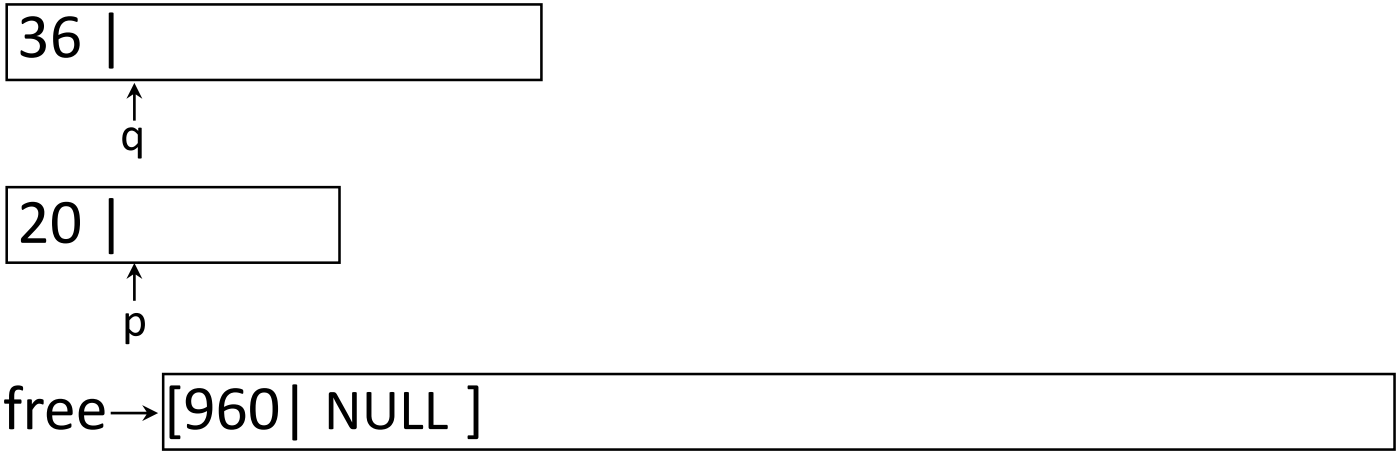
  - the free list now contains 1000 bytes



Explicit Memory Management

Variable-sized Blocks: Allocation

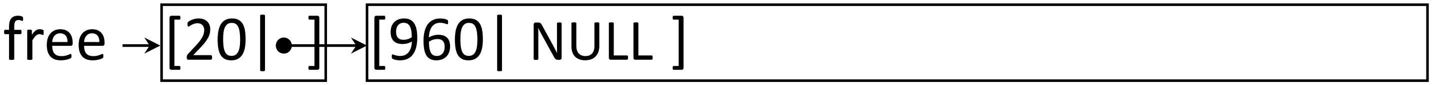
- if 36 bytes are requested next
  - allocate 40 bytes, store the size in the 1st part and return a pointer to the start of the 2nd part



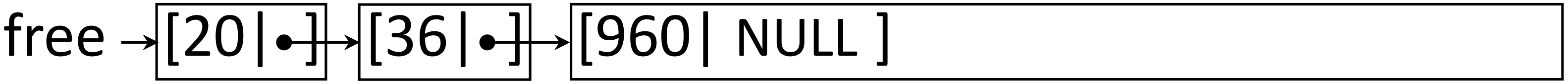
Explicit Memory Management

Variable-sized Blocks: Reclamation

- Suppose the *first block is freed,* i.e. **`delete [] p;`**
  - **`delete`** checks p[-1] to determine how much memory has been freed and adds it to the free list.


free → [20|•] → [960| NULL]

- Suppose the *second block is freed,* i.e. **`delete [] q;`**
  - **`delete`** checks q[-1] to determine how much memory has been freed and adds it to the free list


free → [20|•] → [36|•] → [960| NULL]

- If the free list is sorted by address, the system can recognize that these blocks are adjacent in RAM and merge them together.

Allocation and Fragmentation

Variable-sized Blocks: Reclamation

- When inserting q into the free list, check if q's predecessor can coalesce with q and if q's successor can coalesce with q, i.e.

if (my address + my size == address of next block in list)

then *coalesce* // i.e. join the two smaller blocks into a bigger one

free→ [1024 | NULL]

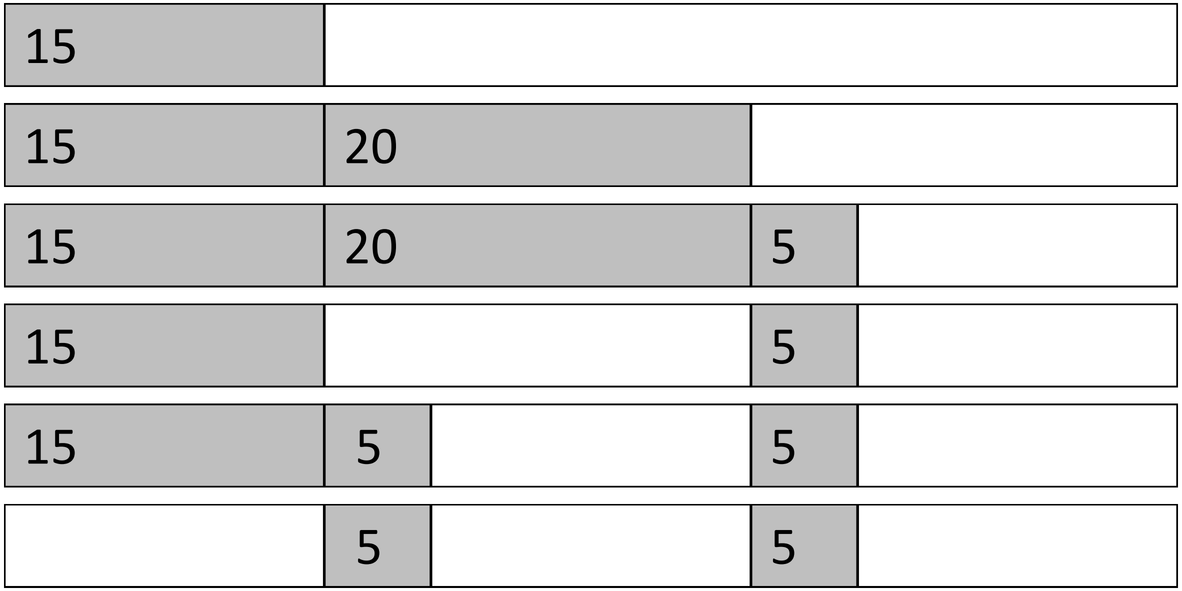
Fragmentation

- *Problem:* repeated allocation and reclamation can create gaps in the heap
- called *fragmentation*, i.e. even though there are *n* bytes free in the heap, you *may not be able to allocate a block of n contiguous bytes*

Allocation and Fragmentation

Fragmentation

- alloc 15
- alloc 20
- alloc 5
- free 20
- alloc 5
- free 15



- There are 15+15+15=45 free but cannot allocate 16 in a single block
- *Idea:* to reduce fragmentation don’t always choose the first block of RAM big enough to satisfy the request

Allocation and Fragmentation

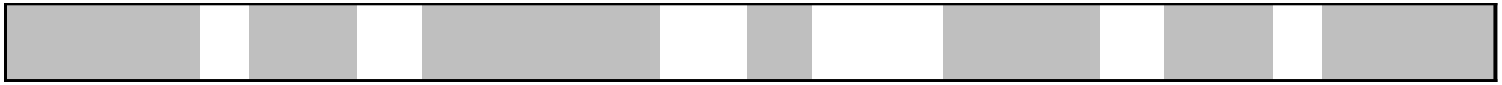
Allocation Strategies

- *first fit:* find the first hole it fits in
  - generally works fairly well in practice
  - fast
- *best fit:* find the location that has the least amount of leftover space
- *worst fit:* pick the biggest hole, so that a relatively large hole remains, which can easily satisfy another request
  - most fragmentation / least utilization in practice
- all approaches are O(log *n*), where *n* is the number of free blocks, with some sort of search tree

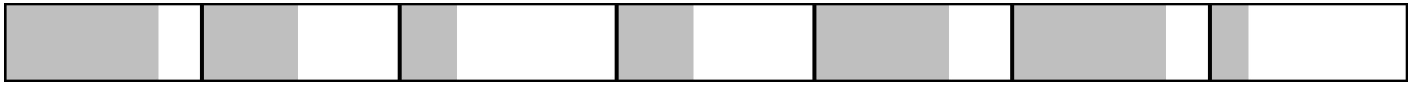
Allocation and Fragmentation

Types of fragmentation

- *external fragmentation*
  - unused memory (white) between allocated blocks (grey)
  - only happens in variable-sized block



- *internal fragmentation:*
  - unused memory (white) within a block (black rectangle)
  - e.g. asked for 100 bytes but all blocks are 128 bytes, so use a 128 byte block and waste 28 bytes
  - can happen both in fixed-sized and variable-sized blocks (when sizes are binned as we'll see with dlmalloc…)



Explicit Memory Management: dlmalloc

Allocation and Deallocation Strategies

- named after its creator, Douglas Lea
- used in C since 1987 (with modifications to allow for multithreaded code)
- *key idea:* distinguish between small allocations, called *smallbin requests* (512 bytes or less), medium (typically 513B to 256KB or less) and large sized requests (greater than 256KB)
- smallbin requests have bins of various sizes, all multiples of 16 starting at 32 bytes, i.e. 32, 48, 64, 80, … 512
- *key idea:* have multiple free lists for holes of different sizes
- medium and large sizes have more sophisticated data structures such as tries

Explicit Memory Management: dlmalloc

Allocation and Deallocation Strategies

- For small bin requests, each block tracks 8 bytes of info, its status (is it in use) and two copies of it's size (one at the beginning of the space allocated and one at the end).

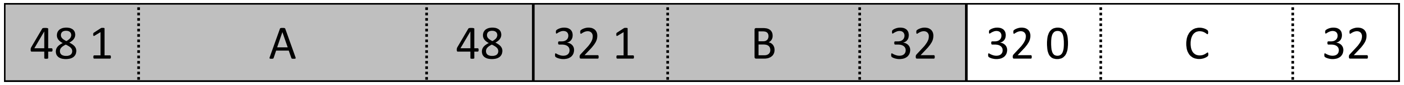
32 1	Your data here	32

- Since the lowest bin size is 32, a request for 1 to 24 bytes results in an allocation of 32 bytes because 8 bytes are reserved for overhead.
- *When deallocating,* you can check the neighbour on either side (neighbours in RAM not in the free list) and if free they can coalesce with you to create a larger block.
- *When allocating,* if there isn't a small block available to fulfill a request, break up a larger block into smaller ones.

Explicit Memory Management: dlmalloc

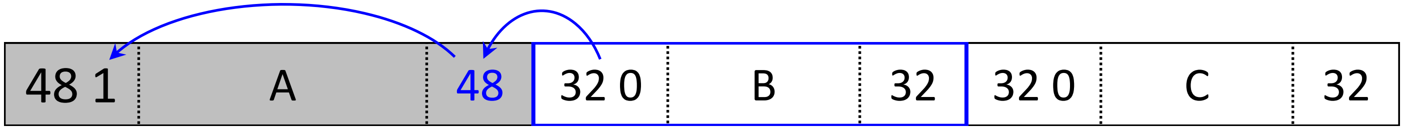
Coalescing

- Check the neighbour on either side (in RAM) for deallocation.
- Here A and B are allocated and C is free.



48 1	A	48	32 1	B	32	32 0	C	32

- If the middle block, B, is deallocated, then check the size of the previous block. Its size is in the 4 bytes before the start of block B. Use the size of A to determine the start of A and check if A has been allocated.



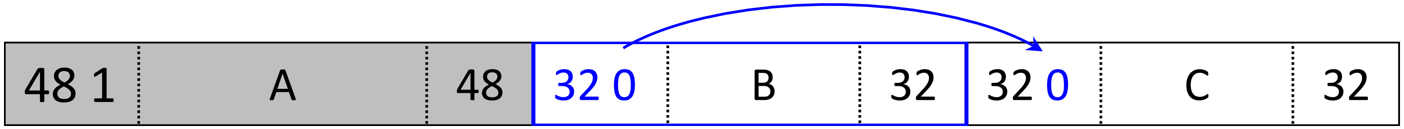
48 1	A	48	32 0	B	32	32 0	C	32

- Since A is allocated, B cannot coalesce with it.

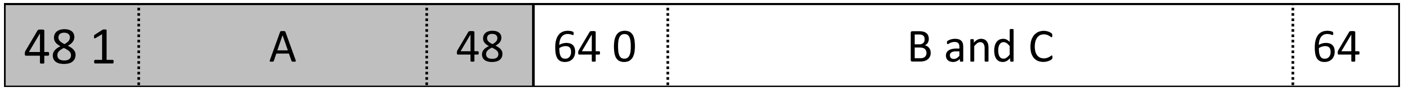
Explicit Memory Management: dlmalloc

Coalescing

- Next use B's size to determine the start of the next block, C, and check if C has been allocated.



- C has not been allocated so B and C can coalesce.



- In constant time it has been determined that B could not coalesce with A but could coalesce with C.

Overview of Heap Management

Pros of Automatic Memory Management

- *The problem:* pointer values can be assigned or changed.

```
1   int* ia = NULL;
2   int* ip = NULL;
3   ia = new int[100];
4   ...                          // What happened here?
5   ip = ia;
6   ...                          // What happened here?
7   delete [] ip;
8   ...                          // What happened here?
```

- Question: Is line 7 an error?
- Answer: it depends on what happened on lines 4, 6 and 8. Was delete called on **`ia`**? Is **`ip[1]`** or **`ia[1]`** accessed after the delete? Was **`ia`**’s or **`ip`**’s value modified? If you did “**`ip = ip+1`**” did you lose the original value of **`ip`**?

Automatic Memory Management

Recall Manual Memory Management

- The compiler cannot tell for sure if it is an error or not because what happens in 4, 6 and 8 could depend on the input.
  - *e.g.* there could statements that say:

```
if (user closes browser tab) {
    delete [] ia;
}
```

- *conclusion:* don't try to detect if new and delete are properly paired up at compile time as pointer values can be assigned (i.e. copied) or modified.

Automatic Memory Management

Approaches

- *Challenge:* need to identify all the pointers
- *Solution 1:* monitor memory access and the values of pointers at runtime (e.g. valgrind)
  - slows down the program so should only be used during testing
  - good testing relies on selecting good test cases
  - hard to guarantee you've caught all errors
- *Solution 2*: decide when to free up memory automatically, typically called *garbage collection*. Here are three approaches:
  1. track if pointing to block: *reference counting*
  2. search for unused memory and reclaim it: *mark and sweep*
  3. search for used memory and reclaim the rest: *copying collectors*

Automatic Memory Management

Approach 1: Reference Counting

- for each heap block, keep track of its *reference count,* i.e. the number of pointers that point to it
- this means you must keep track of every pointer and update the references counts each time a pointer is reassigned
- if a block's reference count is 0, then reclaim it
- problem: circular references
  - a pointer in block 1 is pointing to block 2
  - a pointer in block 2 is pointing to block 1
  - if no other pointers are pointing to block 1 or 2 then their reference count is both 1 but collectively they are inaccessible
- older method

Automatic Memory Management

Approach 2: Mark and Sweep

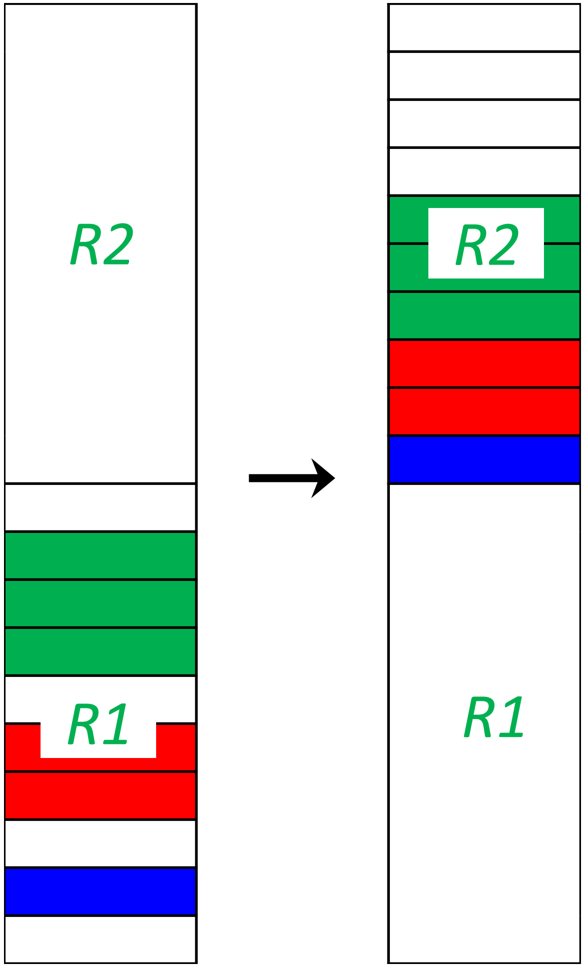
```
scan global variables and the entire stack for pointers
for each non-NULL pointer found (a.k.a. a live heap object)
   mark the block in the heap that the pointer is referring to
   if the heap object contains pointers (e.g. node in a parse tree)
     then follow those pointers as well
scan the heap
   reclaim any blocks not marked
clear all marks
```

- since we are following pointers to blocks that could contain more pointers we are searching on a graph, e.g. need some sort of graph traversal algorithm (a CS341 topic), e.g. depth first search
- older method

Automatic Memory Management

Approach 3: Copying Collector

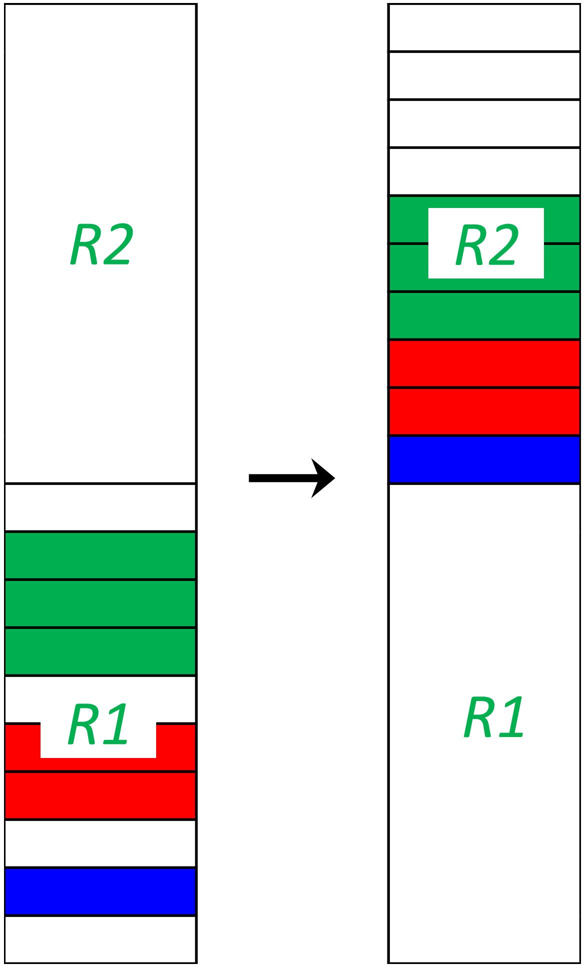
- informally: find all the good stuff (live objects) and disregard the rest
- heap has two regions: 1) *R1* 2) *R2*
- initially allocate only from *R1*
- when *R1* fills up, all reachable data is copied from *R1* to *R2*
  - **scan** for non-NULL pointers (similar to mark and sweep)
  - **copy** data (*live heap objects*) from one region to the other and adjust the pointer values



Automatic Memory Management

Approach 3: Copying Collector

- then reverse the roles of *R1* and *R2* (i.e. now only allocate from *R2*)
- *pros:* no fragmentation: after copying, all reachable data will occupy continuous memory
- *pros:* new and delete are quick
- *cons:* only half the heap is in use at a time (variants have 3 or 4 regions one of which is always free)
- currently a widely used method with many variants



Topic 20 – Linkers and Loaders

Key Ideas

- relocating addresses
- loaders and linkers
- linking files
- MERL file format
  - relocation addresses (REL)
  - external symbol references (ESR)
  - external symbol definitions (ESD)

References

https://www.student.cs.uwaterloo.ca/%7Ecs241/merl/merl.html

https://www.student.cs.uwaterloo.ca/~cs241/slides/link_algorithm.pdf

Overview

The Need for Relocation

- So far we have been assuming that each executable was created from an individual file, however typically many files are combined in to one large executable file.
- When creating individual files, we start at address 0x0.
- Imagine we create a file, print.asm that contains the **`print`** subroutine starting at 0x0

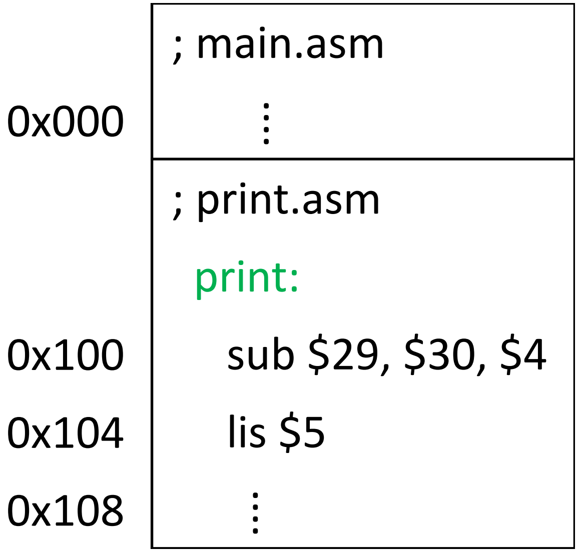
```
0x0  print: sub $29,$30,$4    ; init frame ptr
0x4         lis $5            ; push frame
0x8         .word 36
0xc         …
```

- What happens if this file is combined with another file, main.asm, that is 0x100 bytes in size and print.asm comes after main.asm? (i.e. one file must occur after the other).

Overview

The Need for Relocation

- Initially the label `print` referred to address 0x0.
- When combined with main.asm, the label `print` should now refer to location 0x100.
- Any references to `print` need to be adjusted (i.e. *relocated*) if the procedure is moved to a different location in memory.
- To understand this better, *we'll first investigate a simple case, a single file and a relocating loader.*



```
       ; main.asm
0x000      ⋮
       ; print.asm
         print:
0x100      sub $29, $30, $4
0x104      lis $5
0x108      ⋮
```

Loaders

What is Loading?

- You now know how to convert an assembly language program to a machine language program (via an assembler).
- *But how do you actually run the program?*
- Some other program must be responsible for copying it from *secondary storage* (HDD or SSD) into *primary storage* (RAM) and then starting to execute the instructions in that program.
  - Processors can only execute code located in RAM.
- The *loader* is the program responsible for loading other programs into primary storage and preparing them for execution.

Loaders

Types of Loaders

- We'll look at a very simple loader called a *relocating loader* which determines where in RAM the program will reside and adjusts any references to labels.
- This task is roughly what mips.twoints and mips.array do.
- A more modern approach: the *linker* (which we'll discuss soon) determines the location (in virtual memory, a CS350 topic) and then the loader loads the file into RAM.
- In either case, we need to understand the concept of *relocating addresses*.

Loaders

A Simple Loader

```
loader(P)
  // P is the program to load and run, P = P[0], P[1], …
  for i = 0 to codelength-1          // copy P into memory starting at 0x0
      MEM[i] = P[i]
  $30 ← 0x01000000                   // set addr of stack
  jalr $0                            // start executing P
```

- *Key Problem:* What if programs are not loaded into RAM at location 0x0.
- *Solution:* Addresses need to be adjusted (i.e. *relocated*) depending on where in RAM the program is loaded.

Loaders

A Relocating Loader

```
loader(P)
   // P is the program to load and run,  P = P[0], P[1], …
   // determine memory needed, n, and a location in RAM, $\alpha$
   n = codeLength + space for heap and stack
   $\alpha$ = findRAM(n)                  // $\alpha$ is the starting address
   **for** i = 0 to codeLength-1     // copy P into RAM starting at $\alpha$
      MEM[$\alpha$ + i] $\leftarrow$ P[i]
   $30 = $\alpha$ + n                    // set addr of stack
   place $\alpha$ into $3              // start executing P
   jalr $3
```

Note: the program is no longer loaded starting at address 0x0.

Loaders

A Relocating Loader: the Details

- determine the size of program P, i.e. the codeLength
- *allocate RAM* starting at, say address α, for the code and a stack (and possibly a heap)
- *copy the program* from secondary storage (HDD or SSD) into primary storage (RAM) starting at α,
- possibly set up the program, e.g. pass parameters to the program by placing them in registers or in P's stack
- load the address, α, into some register, say $3.
- *start executing the program* (jalr $3)
- possibly do some work at the end, e.g. **`mips.twoints`** will print out all the register values

Relocation

Changing a Program's Location

- *Key Problem: If a program gets relocated in memory, it affects the values of certain labels*

Assembly Language		Relocated Machine Code	
20	`lis $3`	α+20	0x0000 1814
24	`.word p`	α+24	0x0000 0040
28	`jalr $3`	α+28	0x0060 0009
⋮	⋮	⋮	⋮
40 `p:`	`sw $2,-4($30)`	α+40	0xAFC2 FFFC

- Initially the label `p` referred to address 0x40 but when the code gets relocated to α, it should refer to address α + 0x40

Relocation

Which Values Get Changed?

- *When .word refers to a location, you must add α to it.*

24		`.word p`	α+24	0x0000 00~~40~~
⋮		⋮	⋮	⋮
40	`p:`	sw $2, -4($30)	α+40	0xAFC2 FFFC

- When .word refers to a constant: *do nothing.*

```
0   lis $4
4   .word 4
8   sub $29, $30, $4
```

- For beq, bne: *do nothing,* they jump forward or backward *i* instructions not to a certain address.
- All other instructions: *do nothing.*

Relocation Example

Assembly	**Machine Code**		**Loaded at α=0x0**	
lis $1	0x0	0000 0814	0x0	0000 0814
.word 1	0x4	0000 0001	0x4	0000 0001
⋮	⋮	⋮	⋮	⋮
lis $3	0x20	0000 1814	0x20	0000 1814
.word p	0x24	0000 0040	0x24	0000 0040
jalr $3	0x28	0060 0009	0x28	0060 0009
⋮	⋮	⋮	⋮	⋮
p: sw $2, -4($30)	0x40	AFC2 FFFC	0x40	AFC2 FFFC
⋮	⋮	⋮	⋮	⋮
jr $31	0x5C	03E0 0008	0x5c	03E0 0008

Relocation Example

Assembly	Machine Code		Loaded at α=0x100	
lis $1	0x0	0000 0814	0x100	0000 0814
.word 1	0x4	0000 0001	0x104	0000 0001
⋮	⋮	⋮	⋮	⋮
lis $3	0x20	0000 1814	0x120	0000 1814
.word p	0x24	0000 0040	0x124	0000 0140
jalr $3	0x28	0060 0009	0x128	0060 0009
⋮	⋮	⋮	⋮	⋮
p: sw $2, -4($30)	0x40	AFC2 FFFC	0x140	AFC2 FFFC
⋮	⋮	⋮	⋮	⋮
jr $31	0x5C	03E0 0008	0x15C	03E0 0008

Relocation Example

	Assembly	Machine Code		Loaded at α=0x2000	
	lis $1	0x0	0000 0814	0x2000	0000 0814
	.word 1	0x4	0000 0001	0x2004	0000 0001
	⋮	⋮	⋮	⋮	⋮
	lis $3	0x20	0000 1814	0x2020	0000 1814
	.word p	0x24	0000 0040	0x2024	0000 2040
	jalr $3	0x28	0060 0009	0x2028	0060 0009
	⋮	⋮	⋮	⋮	⋮
p:	sw $2, -4($30)	0x40	AFC2 FFFC	0x2040	AFC2 FFFC
	⋮	⋮	⋮	⋮	⋮
	jr $31	0x5C	03E0 0008	0x205C	03E0 0008

Relocation

Finding those Values

- Problem: Machine code is just a sequence of bits
- Question: *How do we know which words are addresses that must be adjusted* (vs. constants or instructions which do not need to be adjusted).
- Answer: We don't know without additional information.
- Approach: We *must augment the machine code* with information about which words need adjusting if the code is relocated.
- This enhancement of machine code with additional information is called *object code*.

MERL

What is MERL?

- MERL is the format for a program's machine code that *includes information about what words need to be adjusted if the program in loaded into a location other than 0x0.*
- MERL = **M**IPS **E**xecutable **R**elocatable **L**inkable file
- It's CS241's own simplified format.
- Aside: Linux uses ELF and Linux provides tools (i.e. commands) like readelf that understand the ELF format.
- MERL has three parts:
  1. a header
  2. the MIPS machine code
  3. the relocation information (with more coming later).

MERL

Part 1: The MERL Header

The header consists of three words (12 bytes)

1. *Cookie:*
   - the value is 0x1000 0002
   - it identifies the type of file
   - it can be interpreted as the MIPS instruction beq $0 ,$0, 2, which would skip over the header if executed
2. *FileLength:* the length of the MERL file in bytes
3. *CodeLength:* the length of the header plus the MIPS machine code (which is also the offset to the Relocation Table)

MERL

Part 2: The Body: MIPS Program

- *This is the program in MIPS machine code.*
- It works correctly if the program is loaded into RAM location 0x0c (i.e. the location immediately following the header).

Part 3: Relocation and External Symbol Table

- *It contains relocation information.*
- Format: the word 0x01 followed by the location of a word in the MERL file that needs to be adjusted if the file is relocated.
- called a *REL* or *Relocation Entry*.
- This part also contains external symbol definitions and external symbol references (which we'll discuss later).

MERL Example

Assembly	Addr	MERL file	Comments
`beq $0, $0, 2`	0x00	0x1000 0002	; 1 - Header
`.word endfile`	0x04	0x0000 003c	; file length
`.word endcode`	0x08	0x0000 002c	; code + header
`lis $3`	0x0c	0x0000 1814	; 2 - Body
`.word 0x4`	0x10	0x0000 0abc	; no REL
`lis $1`	0x14	0x0000 0814	
`r1: .word A`	0x18	0x0000 0024	; needs a REL
`jr $1`	0x1c	0x0020 0008	
`B: jr $31`	0x20	0x03e0 0008	
`A: beq $0 ,$0, B`	0x24	0x1000 fffe	
`r2: .word B`	0x28	0x0000 0020	; needs a REL

MERL Example

Assembly	Addr	MERL file	Comments
`endcode:`			; 3 - Relocation Table
`.word 0x1`	0x2c	0x0000 0001	; REL format code
`.word r1`	0x30	0x0000 0018	; location
`.word 0x1`	0x34	0x0000 0001	; REL format code
`.word r2`	0x38	0x0000 0028	; location
`endfile:`			

Comments about Relocation Entries

- the instructions at `r1:` and `r2:` need to be relocated because `A` and `B` are addresses of instructions (not constants)
- the instruction at no REL does not, because 0x4 is a constant

Loader Pseudocode

Loading a CS 241 MERL File

```
read in MERL header
$\alpha$ = findRAM(codeLength)        // space for code + heap + stack
for i = 0 .. codelength-1             // copy into RAM
    MEM[$\alpha$ + i] = instruction[i]
for each REL entry                    // relocate REL addresses
    MEM[$\alpha$ + location] += $\alpha$
initialize $30                        // stack pointer
place $\alpha$ into $3                // start executing code
jalr $3
```

A MERL Assembler

Modifications to Create a MERL Assembler

For Pass 1

- record the size of the file
- start counting addresses at 0x0c (rather than 0x0)
- when you encounter a `.word <label>` instruction
  - record the location

For Pass 2

- output the header
- output the MIPS machine code (already do this step)
- output the relocation table

Loader Notes

Loading in CS 241 MIPS Program

- Notice how mips.twoints works:

```
% mips.twoints
Usage: mips.twoints <filename> [load_address]
```

- i.e. you can select the load address

Official Description of MERL

- The official description of the MERL format is in the CS241 web site in the Resource Material section. https://www.student.cs.uwaterloo.ca/%7Ecs241/merl/merl.html

Assemblers, Loaders and Linkers

What They Do

- Assemblers
  - *what:* need two passes to translate labels
  - *why:* so labels can be used before they are defined
- Relocating Loader
  - *what*: need to track and adjust labels that were used in a *.word* assembler directive.
  - *why:* allows a program to be loaded anywhere into RAM
- Linker
  - *what:* use multiple files for code
  - *why: …*

Linking

Why Link Object Code Files?

- Answer*: so we can break up a large program into several modules* (i.e. easier to manage pieces).
- Why break-up large programs?
- Answers: For the same reasons we do so for high level languages.
  - *Procedural Abstraction*: programmers just need to know interface not how the subroutine is implemented.
  - Collect related subroutines together.

Linking

Why Link Object Code Files?

- Why break-up large programs?
  - Create a collection of subroutines (i.e. a library) that can be used in many programs.
  - Errors are easier to track down.
  - Different people/ groups can be responsible for different modules.
  - Avoid duplication of effort (e.g. same print integer subroutine created many times)

Linking Files

How to Link: Attempt 1

- Recall Goal: *use multiple files for code.*
- Attempt 1: just combine (i.e. concatenate) all the small assembly language files into one big one and then assemble.
- A small change in one small file would mean redoing everything.
- May just want to distribute the object code not the assembly language code.
- Requirement #1: *We need a tool that works with multiple MERL files as input.*

Linking Files

How to Link: Attempt 2

- Attempt: Assemble all the MERL files then concatenate (i.e. join) together.
- Problem: When assembling, we start at address 0x0, so all files would start at the same location. This will not be true when linking together multiple MERL files.
- Consequence: If you concatenate two MERL files, the result is not a valid MERL file.
- Requirement #2: *We need a tool that outputs the MERL format.*
- Requirement #3: *We need a tool that works with labels (representing subroutines) defined in one file and used in another.*

Linking Files

How to Link: The External Symbol Reference (ESR)

- *Create a directive, .import, that tells the assembler that this symbol (i.e. label) occurs in another file* (i.e. externally).
- The assembler does not translate this directive into an instruction. The directive provides information to the assembler.
- For example *.import notify_nsa* means that the symbol *notify_nsa* is defined in another file.
- When assembling, initially assign the value of 0 to this symbol, but make a note in the MERL file that this symbol is not yet defined.
- If you never find it, after linking is complete, then report an error.

Linking Files

The External Symbol Reference (ESR) Format

- *In the Relocation and External Symbol Table section of MERL file create an ESR entry.*
- There is only one ASCII char per word to represent the chars in the symbol (here a label) in order to make it easy to implement
- It is in the following format

word 1:	0x11	; this is an ESR entry
word 2:	location	; where the symbol is used
word 3:	length	; of the symbol in bytes (say n)
word 4:	1st char of symbol (in ASCII)	
word 5:	2nd char of symbol (in ASCII)	
…	…	
word n+3:	nth char of symbol (in ASCII)	

Linking Files

The External Symbol Reference (ESR) Format

- The first word is always 0x11 which signifies that whatever follows is an ESR.
- Concern: *What if multiple files use the same symbol?*

file1.asm

```
.import abc
lis $1
.word abc
```

file2.asm

```
; abc is a loop
abc:
  …
beq $1, $2, abc
```

file3.asm

```
; abc is a proc
abc:
  sw $1, -4($30)
  sw $2, -8($30)
```

Linking Files

The External Symbol Definition (ESD)

- Requirement: Need a way to *provide information hiding.*
- We want to differentiate between a symbol meant for local use (within a file) and one meant for global use (external to the file).
- Use the *.export directive* to indicate that other files may use (i.e. refer to) this symbol.
- A symbol can only be defined once, but may be referenced many times.

Linking Files

The External Symbol Definition (ESD) Format

- Using *.export* is like declaring a variable global.
- The *.import .export* pair links the definition in one file to its reference in another.

file1.asm

```
.import abc
lis $1
.word abc
```

file2.asm

```
; abc is a loop
abc:
  …
beq $1, $2, abc
```

file3.asm

```
; abc is a proc
.export abc
abc:
   sw $1, -4($30)
   sw $2, -8($30)
```

Linking Files

The External Symbol Definition (ESD) Format

- *In the Relocation and External Symbol Table section of MERL file create an ESD entry.*
- It is similar in format to the ESR entry except the entry type is now 0x05 (rather than 0x01 or 0x11).

word 1:	0x05	; this is an ESD entry
word 2:	location	; where the symbol refers to
word 3:	length	; of the symbol in bytes (say n)
word 4:	1st char of symbol (in ASCII)	
word 5:	2nd char of symbol (in ASCII)	
...	...	
word n+3:	nth char of symbol (in ASCII)	

Linking Files

Modifications to Handle External References

Prior Pass 1 Tasks (just handle RELs)

- record the size of the file
- when you encounter a `.word <label>` instruction
  - record the location

Additional Pass 1 Tasks (also handle ESRs and ESDs)

- when you encounter an `.import <symbol>` directive
  - record each symbol that needs importing and the locations where it is referenced
- when you encounter an `.export <symbol>` directive
  - record each symbol that needs exporting and the location where it is defined

Linking Files

Modifications to Handle External References

Prior Pass 2 Tasks (just handle RELs)

- output the MERL header
- output the MIPS machine code
- output the Relocation and External Symbol Table
  - create a Relocation Entry for each relocatable address

Additional Pass 2 Tasks (also handle ESRs and ESDs)

- when outputting the Relocation and External Symbol Table
  - for each symbol that is imported, create an ESR entry for each location where it is referenced
  - create an ESD entry for each symbol that is exported

Linker Pseudocode

Goal: handle multiple files and external symbols

1. Concatenate the programs.
2. Combine and adjust ESDs with new locations.
3. Use new ESDs to update old ESRs and replace them by RELs (i.e. the reference is no longer an external reference it is now a relocation entry).
4. Relocate addresses both in the body of the code and in the Relocation table for RELs.

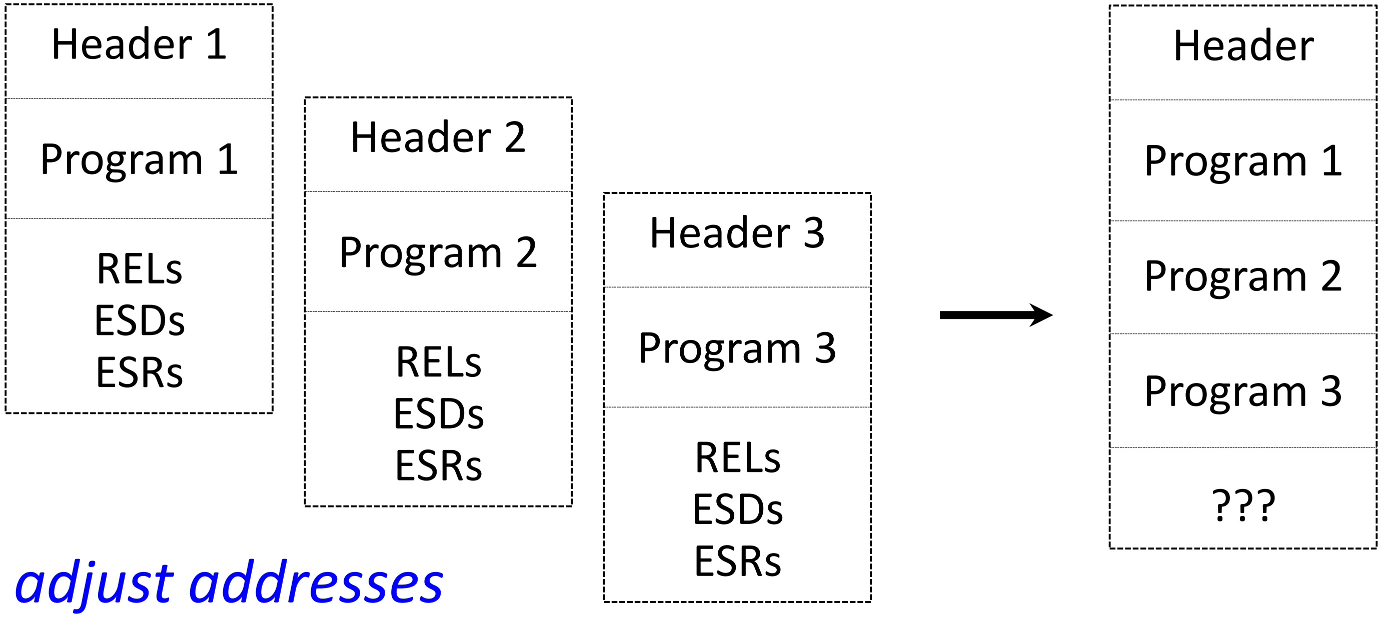
Key Task: like loading, addresses need to be adjusted.

If file2.asm is added to the end of file1.asm then the addresses in file2.asm need to be adjusted to take in account that they now occur after file1.asm.

Linker Pseudocode

Step 1: Concatenate Programs

- You will not be able to finalize the header and the ESRs and ESDs initially.



Header 1

Program 1

RELs
ESDs
ESRs

Header 2

Program 2

RELs
ESDs
ESRs

Header 3

Program 3

RELs
ESDs
ESRs

Header

Program 1

Program 2

Program 3

???

Key Task: adjust addresses

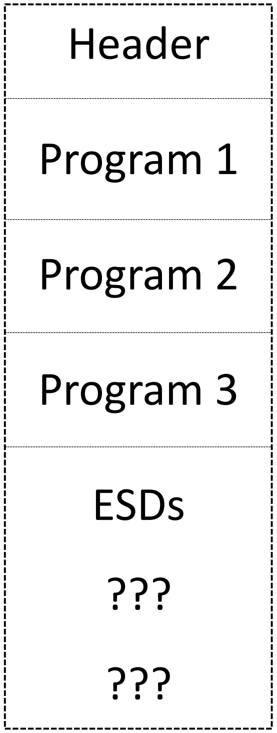
$\mathrm{Addr}_{2\ \mathrm{new}} = \mathrm{Addr}_{2\ \mathrm{old}} + |\mathrm{Prog}\ 1|$

$\mathrm{Addr}_{3\ \mathrm{new}} = \mathrm{Addr}_{3\ \mathrm{old}} + |\mathrm{Prog}\ 1| + |\mathrm{Prog}\ 2|$

Linker Pseudocode

Step 2: Combine and Adjust ESDs

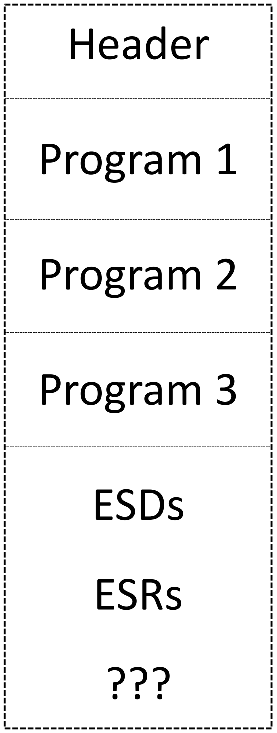
- Combine all the External Symbol Definitions (ESDs)
  - Program 1's ESDs have no change.
  - *Programs 2's ESDs have to be shifted down by the size of Program 1*, i.e. $\text{ESD}_{2\ \text{new}} = \text{ESD}_{2\ \text{old}} + |\text{Prog } 1|$
  - Programs 3's ESDs have to be shifted down by the size of Program 1 + the size of Program 2, i.e. $\text{ESD}_{3\ \text{new}} = \text{ESD}_{3\ \text{old}} + |\text{Prog } 1| + |\text{Prog } 2|$
- You can get the size of each program from its original header.



Linker Pseudocode

Step 3: Use new ESDs to update old ESRs

```
for each old ESR
    look up the new ESD
    if found
        update the value at the location + offset
        (i.e. it is no longer referenced externally)
        convert the ESR to an REL (relocation entry)
    else
        adjust the new ESRs with the new offset, e.g.
        $ESR_{2\ new} = ESR_{2\ old} + |Prog\ 1|$
        $ESR_{3\ new} = ESR_{3\ old} + |Prog\ 1| + |Prog\ 2|$ …
```



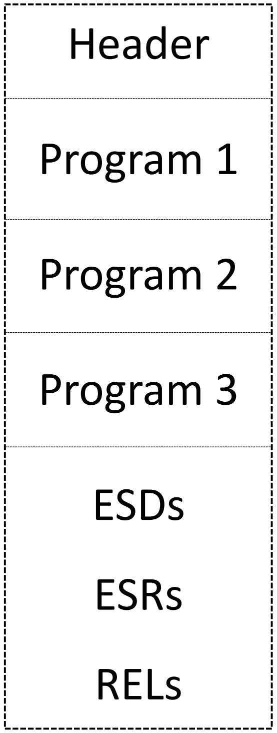
Linker Pseudocode

Step 4: Relocate addresses (internally)

- just like what was done for loading, any *relocation entries in programs 2, 3, etc. need to be relocated*.
- for each relocation entry
  - add the appropriate offset in the code
  - add the appropriate offset in the relocation entry

    $\text{Addr}_{2\text{ new}} = \text{Addr}_{2\text{ old}} + |\text{Prog } 1|$

    $\text{Addr}_{3\text{ new}} = \text{Addr}_{3\text{ old}} + |\text{Prog } 1| + |\text{Prog } 2|$



Header

Program 1

Program 2

Program 3

ESDs

ESRs

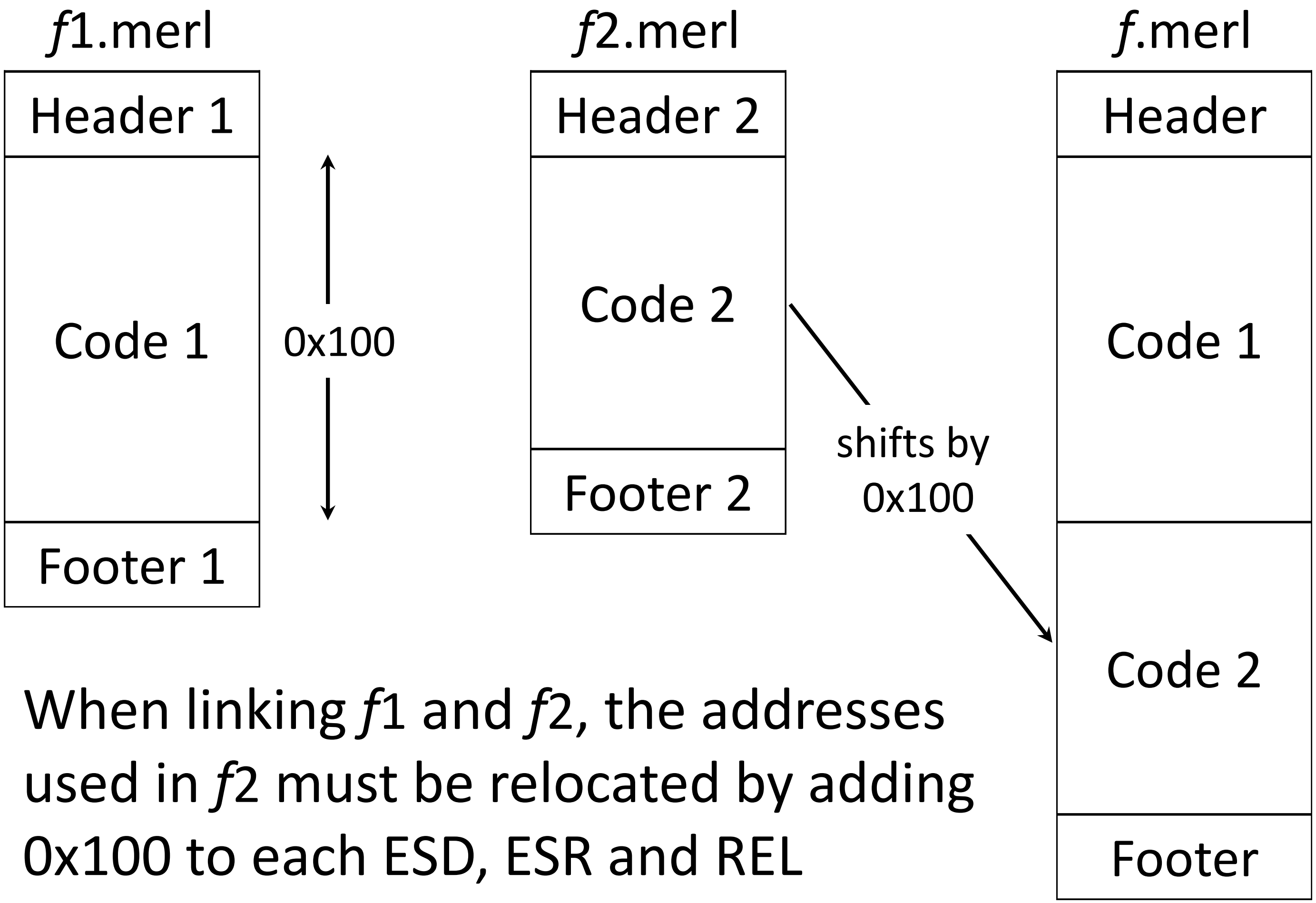
RELs

Linking Example

In the following example we'll be linking together two files

- *f*1.merl has a 0x100 bytes of MIPS instructions
- *f*2.merl has 0x80 bytes of MIPS instructions
- *the code from f2.merl will be added to the end of the code from f1.merl*
- the resulting file will be called *f*.merl

Linking *f*1 and *f*2



When linking *f*1 and *f*2, the addresses used in *f*2 must be relocated by adding 0x100 to each ESD, ESR and REL (relocation entry).

Linking *f*1 and *f*2

Memory Math

Because memory locations start at 0 and each word / instruction is 4 bytes, storing data works as follows.

- To store 1 word (i.e. 4 bytes) at address 0x0, locations 0x0–0x3 are occupied. 0x4 is the address of the first free location and 0x4 is also the length of the entry.

X	X	X	X				
0	1	2	3	**4**	5	6	7

- To add 2 more words (i.e. 8 bytes), we have 4 + 8 = 0xC (i.e. 12) bytes, locations 0x0–0xB are used. 0xC is the address of the first free location and 0xC is also the total length of the entries.

X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y				
0	1	2	3	**4**	5	6	7	**8**	9	A	B	**C**	D	E	F

Linking *f*1 and *f*2

Memory Math

- If entry *x* is 0xC bytes long and starts at address 0x100 then the last used location is 0x10B and the next free location is 0x10C

X	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*		
100	101	102	103	**104**	105	106	107	**108**	109	10A	10B	**10C**	10D

- In general you must take into account the fact that
  - we start filling out memory at location 0
  - each word occupies 4 bytes.

- We are now ready to link our two files…

Linking *f*1 and *f*2

assembly language

```
           .import pr
0x0C      lis $1
               ⋮
0x30      lis $2
0x34      .word a
0x38      jalr $2
               ⋮
0x50      lis $3
0x54      .word pr
0x58      jalr $3
               ⋮
0x70  a:  sw $4, -4($30)
               ⋮
0x108       jr $31
0x10C     ; Footer

; code 0x100 bytes long
```

f1.merl		Header
0x000	0x1000 0002	cookie
0x004	0x128	file size: C + 100 + 1C
0x008	0x10C	header + code = C+100
0x00C	0x0000 0814	Code *f1*
⋮	⋮	
0x108	0x03e0 0008	
		Footer
0x10C	0x1	REL (relocation entry)
0x110	0x34	location referencing a
0x114	0x11	Ext Symbol Reference
0x118	0x54	location referencing pr
0x11C	0x2	length of symbol
0x120	0x70	ASCII p
0x124	0x72	ASCII r

Linking *f*1 and *f*2

assembly language

```
           .export pr
0x0C     lis $1
             ⋮
0x20     lis $1
0x24     .word b
0x28     jalr $1
             ⋮
0x40  b:  sw $2, -4($30)
             ⋮
0x60  pr: sw $3, -4($30)
             ⋮
0x88      jr $31
0x8C    ; Footer

; code 0x80 bytes long
```

f2.merl		
		Header
0x000	0x1000 0002	cookie
0x004	0x108	file size: C + 80 + 1C
0x008	0x08C	header + code = C + 80
0x00C	0x0000 0814	Code *f2*
⋮	⋮	
0x088	0x03e0 0008	
		Footer
0x08C	0x1	REL (relocation entry)
0x090	0x24	location referencing b
0x094	0x05	Ext Symbol Definition
0x098	0x60	location of pr
0x09C	0x2	length of symbol
0x100	0x70	ASCII p
0x104	0x72	ASCII r

Linking *f*1 and *f*2

	f.merl	Header
0x000	0x1000 0002	cookie
0x004	0x1B8	file length
0x008	0x18C	header + code length: 0xC+0x100+0x80
0x00C	0x0000 0814	Code *f1* - 0x100 long, not shifted
⋮	⋮	⋮
0x108	0x03e0 0008	
0x10C	0x0000 0814	Code *f2* - 0x80 long, shifted by 0x100
⋮	⋮	⋮
0x188	0x03e0 0008	
0x18C		Footer…

- Note: The file length cannot be determined until the footer is finalized.

Linking *f*1 and *f*2

	f.merl	Footer
0x18C	0x05	Ext Symbol Definition
0x190	0x160	ESD address (of pr) (+100)
0x194	0x2	length of symbol
0x198	0x70	ASCII p
0x19C	0x72	ASCII r
0x1A0	0x1	REL (relocation entry)
0x1A4	0x54	location (of pr)
0x1A8	0x1	REL (relocation entry)
0x1AC	0x34	location (of a)
0x1B0	0x1	REL (relocation entry)
0x1B4	0x124	location (of b) (+100)

- Note: The entries in the Footer (RELs, ESDs, and ESRs) can be in any order.
- The file length is now known: *f*.merl is 0x1B8 bytes long.

Linking *f*1 and *f*2

Pass 3: Edits to the Code: Resolving ESRs

In Pass 3, the ESR on line 0x54 (originally in *f1*.merl) gets resolved, i.e. the label pr refers to location 0x160.

- So the value 0x160 gets written to location 0x54 in *f*.merl.

	assembly language		*f1.merl*		*f.merl*
	.import pr				
	⋮		⋮		⋮
0x50	lis $3	0x50	0x0000 0814	0x50	0x0000 0814
0x54	.word pr	0x54	0x0000 0000	0x54	0x0000 0160
0x58	jalr $3	0x58	0x0060 0009	0x58	0x0060 0009
	⋮		⋮		⋮

Linking *f*1 and *f*2

Pass 4: Edits to the Code: Updating RELs

In Pass 4, since *f2* has been relocated by 0x100 bytes …

- All values corresponding to the RELs in the body of *f2* have to be relocated in *f*.merl by adding the appropriate offset (i.e. 0x100).
- Hence, 0x100 is added to the value 0x40 (stored at location 0x024 + 0x100) to reflect the fact that the subroutine b has been relocated.

assembly language		*f2.merl*		*f.merl*	
0x020	lis $1	0x020	0x0000 0814	0x120	0x0000 0814
0x024	.word b	0x024	0x0000 0040	0x124	0x0000 0140
0x028	jalr $1	0x028	0x0020 0009	0x128	0x0020 0009
	⋮		⋮		⋮
0x040	b: sw $2, -4($30)	0x040	0xafc3 fffc	0x140	0xafc3 fffc

Topic 21 – Concluding Remarks

Key Ideas

- what we did
- why we did it
- preparing for the final
- course evaluations

Concluding Remarks

What we did

- *all the steps that happen after creating a WLP4 program* → having the code running on a MIPS processor
- it is a difficult task
- need to know about a lot of topics: data representation (hexadecimal, 2's complement, ASCII), assembly language, finite automata (deterministic and non-deterministic) and regular expressions, context free grammars, parsing, parse trees, symbol tables, type checking, the heap, the stack, stack frames, object files, linking and loading…

Concluding Remarks

Why we did it

- programming languages are the interface between a programmer’s idea and a computer running a program
- C, C++, Racket, Java etc. aren’t naturally occurring phenomena
  - they were created by (some fairly bright) humans
- now you understand how they work
- sometimes they have features we don’t like
- sometimes they are missing features we do like
- hopefully, you will now think more critically about programming and programming languages.
- *You can modify an existing language or create a new one!*

Concluding Remarks

Preparing for the Final

- I will make a complete copy of my slides available on Learn (in the next few days).
- I have a pinned post in Piazza listing any typos for existing slides.
- Will have Final Exam [official] post in Piazza.
- I will have extra office hours just before the final (will post in Piazza).
- We are having review sessions before the final.
- We will be monitoring and answering questions in Piazza.
- *Good luck on the final!*
- *Good luck with computer programming!*
- *Thank-you for your attention!*