COP 3502: Computer Science I
Spring 2004

– Day 6 –
Recursion

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Tracing Recursive Functions

- Many times it will be necessary to trace the execution of a recursive program. While in many ways this is similar to tracing any other code, tracing a recursive program requires a bit more care and caution.

- The next two pages show recursive functions; let’s trace their execution.

Tracing Recursive Functions (cont.)

```
int  f (int x, int y)
{
if (x == 0  && y >= 0)
        return y + 1;
else if (x > 0 && y == 0)
        return (f(x–1,1));
else if (x > 0 && y > 0)
        return (f(x–1, f(x, y–1)));
}
```

the function

Trace for call f(1,3)

```
x = f(1, 3)
        = f(0, f(1, 2))
        = f(0, f(0, f(1, 1)))
                = f(0, f(0, f(0, f(1,0))))
                = f(0, f(0, f(0, f(0,1))))
        = f(0, f(0, f(0, 2)))
        = f(0, f(0, 3))
        = f(0,4)
   = 5
```

the execution trace

Tracing Recursive Functions (cont.)

```
int  f (int x, int y)
{
if (y == 0 || x == y  && x >= 0)
    return 1;
else
    return (f(x–1, y) + f(x–1, y–1));
}
```

the function

Trace for call f(5,3)

$$
\begin{aligned}
x = f(5, 3) &= f(4, 3) + f(4, 2) \\
&= f(3, 3) + f(3, 2) + f(4, 2) \\
&= 1 + f(2, 2) + f(2, 1) + f(4, 2) \\
&= 1 + 1 + f(1, 1) + f(1,0) + f(4, 2) \\
&= 1 + 1 + 1 + 1 + f(3, 2) + f(3, 1) \\
&= 4 + f(2, 2) + f(2, 1) + f(3,1) \\
&= 4 + 1 + f(1, 1) + f(1, 0) + f(3,1) \\
&= 5 + 1 + 1 + f(2, 1) + f(2,0) \\
&= 7 + f(1, 1) + f(1, 0) + f(2, 0) \\
&= 7 + 1 + 1 + 1 \\
&= 10
\end{aligned}
$$

the execution trace

Practice Constructing Recursive Functions

- Below are several practice problems that you should implement as recursive functions. Sample solutions are at the end of this set of notes.

1. Construct a recursive function that returns $\sum_{i=1}^{n} i$
2. Construct a recursive function that will count the number of times a particular character appears in a string. Example: rcount (‘s’, “Mississippi sassafras”)

Binary Number System

- Base or radix 2 number system
- **B**inary dig**it** is called a bit.
- Numbers are 0 and 1 only.
- Numbers are expressed as powers of 2.
- $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 16$, $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$, $2^{11} = 2048$, $2^{12} = 4096$, $2^{12} = 8192$, …

Binary Number System (cont.)

Conversion of binary to decimal (base 2 to base 10)

Example: convert $(1000100)_2$ to decimal

$= (1 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$

$= 64 + 0 + 0 + 0 + 4 + 0 + 0$

$= (68)_{10}$

Binary Number System (cont.)

Conversion of decimal to binary (base 10 to base 2)

Example: convert $(68)_{10}$ to binary

68 ÷ 2 = 34 remainder is 0

34 ÷ 2 = 17 remainder is 0

17 ÷ 2 = 8 remainder is 1

8 ÷ 2 = 4 remainder is 0

4 ÷ 2 = 2 remainder is 0

2 ÷ 2 = 1 remainder is 0

1 ÷ 2 = 0 remainder is 1

Answer = 1 0 0 0 1 0 0

Note: the answer is read from bottom (MSB) to top (LSB) as 1000100_2

Octal Number System

- Base or radix 8 number system.
- 1 octal digit is equivalent to 3 bits.
- Octal numbers are 0-7.
- Numbers are expressed as powers of 8.
  - $8^0 = 1$, $8^1 = 8$, $8^2 = 64$, $8^3 = 512$, $8^4 = 4096$

Octal Number System (cont.)

Conversion of octal to decimal (base 8 to base 10)

Example: convert $(632)_8$ to decimal

$= (6 \times 8^2) + (3 \times 8^1) + (2 \times 8^0)$

$= (6 \times 64) + (3 \times 8) + (2 \times 1)$

$= 384 + 24 + 2$

$= (410)_{10}$

Octal Number System (cont.)

Conversion of decimal to octal (base 10 to base 8)

Example: convert $(177)_{10}$ to octal

177 ÷ 8 = 22 remainder is 1

22 ÷ 8 = 2 remainder is 6

2 ÷ 8 = 0 remainder is 2

Answer = 2 6 1

Note: the answer is read from bottom to top as $(261)_8$, the same as with the binary case.

Hexadecimal Number System

- Base or radix 16 number system.
- 1 hex digit is equivalent to 4 bits.
- Numbers are 0-9, A, B, C, D, E, and F.
  - $(A)_{16} = (10)_{10}$, $(B)_{16} = (11)_{10}$, $(C)_{16} = (12)_{10}$, $(D)_{16} = (13)_{10}$, $(E)_{16} = (14)_{10}$, $(F)_{16} = (15)_{10}$
- Numbers are expressed as powers of 16.
- $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, $16^3 = 4096$, $16^4 = 65536$, …

Hexadecimal Number System (cont.)

Conversion of hex to decimal (base 16 to base 10)

Example: convert $(F4C)_{16}$ to decimal

$= (F \times 16^2) + (4 \times 16^1) + (C \times 16^0)$

$= (15 \times 256) + (4 \times 16) + (12 \times 1)$

$= 3840 + 64 + 12$

$= (3916)_{10}$

Hexadecimal Number System (cont.)

Conversion of decimal to hex (base 10 to base 16)

Example: convert $(4768)_{10}$ to hex.

= *4768* ÷ 16 = 298 remainder 0

= 298 ÷ 16 = 18 remainder 10 (A)

= 18 ÷ 16 = 1 remainder 2

= 1 ÷ 16 = 0 remainder 1

Answer: 1 2 A 0

Note: the answer is read from bottom to top as $(4D)_{16}$, the same as with the binary case.

Decimal, Binary, Octal, and Hex Numbers

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversion from Hex or Octal to Binary

- Conversion of octal and hex numbers to binary is based upon the the bit patterns shown in the table on page 17 and is straight forward.
- For octal numbers, only three bits are required. Thus $6_8 = 110_2$, and $345_8 = 11100101_2$.

$37254_8 = 011\,|\,111\,|\,010\,|\,101\,|\,100_2 = 11111010101100_2$

- For hex numbers, four bits are required. Thus $E_{16} = 1110_2$, and $47D_{16} = 10001111101_2$.

$57DE4_{16} = 0101\,|\,0111\,|\,1100\,|\,1110\,|\,0100_2$

$= 1010111110011100100_2$

Conversion from Binary to Hex or Octal

- Conversion of binary numbers to octal and hex simply requires grouping bits in the binary numbers into groups of three bits for conversion to octal and into groups of four bits for conversion to hex.
- Groups are formed beginning with the LSB and progressing to the MSB.
- Thus, $11|100|111_2 = 347_8$
- $11|100|010|101|010|010|001_2 = 3025221_8$
- $1110|0111_2 = E7_{16}$
- $1|1000|1010|1000|0111_2 = 18A87_{16}$

Binary Addition

+	0	1
0	0	1
1	1	10

result is 0 with a carry of 1

Example:

Carry		1	11	11	11
Addend	10011	10011	10011	10011	10011
Augend	+ 110	110	110	110	110
Sum	1	01	001	1001	11001

Binary Subtraction

subtrahend

minuend

–	0	1
0	0	1 with borrow from next column
1	1	0

Example:

Borrow		01	01	01	0	0
Minuend	10100	10100	10100	10100	10100	10100
Subtrahend	- 1001	1001	1001	1001	1001	1001
Difference		1	11	011	1011	01011

Octal Addition Table

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

Octal Addition & Subtraction

Addition Example:

Carry		1	11	11
Addend	1775	1775	1775	1775
Augend	+ 670	670	670	670
Sum	5	65	665	2665

Subtraction Example: (just like decimal with the borrows)

Borrow		3	13	5	
Minuend	1643	1643	1643	1643	1643
Subtrahend	- 256	256	256	256	256
Difference		5	65	365	1365

Hexadecimal Addition Table

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
1	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10
2	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11
3	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12
4	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
5	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14
6	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15
7	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16
8	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17
9	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18
A	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19
B	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A
C	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Hexadecimal Addition & Subtraction

Addition Example:

Carry		1	1	
Addend	A27	A27	A27	A27
Augend	+ 3CF	3CF	3CF	3CF
Sum		6	F6	DF6

Subtraction Example: (just like decimal with the borrows)

Borrow		B 13	B	
Minuend	AC3	AC3	AC3	AC3
Subtrahend	- 604	604	604	604
Difference		F	BF	4BF

Negative Number Representation

- There are several alternative conventions that can be used to represent negative (as well as positive) integers, all of which involve treating the MSB as a sign bit.
- Typically, if the MSB is 0, the number is positive; if the MSB is 1, the number is negative.
- The simplest form of representation that employs a sign bit is the *sign-magnitude* representation. In an *n*-bit word, the right-most *n*-1 bits represent the magnitude of the integer, and the left-most bit represents the sign of the integer. For example, in an 8-bit word the value of $+24_{10}$ is represented by: 00011000_2, while the value of $–24_{10}$ is represented by 10011000_2.

Negative Number Representation (cont.)

- There are several disadvantages to sign magnitude representation.
- One is that addition and subtraction operations require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation.
- Another disadvantage is that there are two representations of 0. Using an 8-bit word, both 00000000_2 and 10000000_2 represent 0 (the first +0, the latter –0). This makes logical testing for equality on 0 more complex (two values need to be tested).
- Because of these disadvantages, sign-magnitude representation is rarely used in implementing the integer portion of the ALU.

Negative Number Representation (cont.)

Two's Complement

- Like sign-magnitude, two's complement uses the MSB as a sign bit, thus making it easy to test if an integer is positive or negative.
- Two's complement differs from sign-magnitude in the way the remaining n-1 bits (of an n-bit word) are interpreted.
- Two's complement representation has only a single representation for the value of 0. The two's complement of a binary number is found by subtracting each bit of the number from 1 and adding 1.

Two's Complement Representation (cont.)

- An alternate way of performing a two's complementation (does exactly the same thing the addition does without thinking about doing the subtraction and the addition) is as follows:
- Beginning with the LSB and progressing toward the MSB, leave all 0 bits unchanged and the first 1 bit unchanged, after encountering the first 1 bit, complement all remaining bits until the MSB has been processed. The resulting number is the two's complement of the original number.

Example:

Binary:	11011000100100	01011011
2's comp	00100111011100	10100101

Why Two's Complement?

- Two's complement arithmetic allows you to perform addition operations when subtraction is the actual desired operation.
- This means that any expression of the form: A – B can be computed as $A + B_C$ where B_C represents the two's complement form of B.
- This fact allows the Arithmetic Logic Unit (ALU) inside the CPU to be more compact since circuitry for subtraction is not included.

Why Two's Complement? (cont.)

- Example using 2's complement:

Suppose that our problem (in decimal) is: 7 + (-3).

Representing these numbers in 4 bits we have:

$7_{10} = 0111_2$ $3_{10} = 0011_2$ 2's comp form = 1101_2

```
  0111
+ 1101
 ------
 10100
```

10100 ignoring the overflow (extra bit) we have our answer = $0100_2 = 4_{10}$

Why Two's Complement? (cont.)

- Although it may seem that with two's complement we have found nirvana as far as representing negative numbers inside a computer is concerned, we unfortunately, have not.
- For any addition operation, the result may be larger than can be held in the word size of the system. This condition is called *overflow*.
- When an overflow occurs, the arithmetic logic unit (ALU) must signal the control unit (within the CPU) that an overflow condition exists and no attempt be made to use the invalid result.

Why Two's Complement? (cont.)

- To detect overflow, the following rule must be observed:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign of the operands to the addition. Note that overflow can occur whether or not there is a carry out of the MSB position.

Why Two's Complement? (cont.)

- Example using 2's complement:

Suppose that our problem (in decimal) is: 7 + (-3).

Representing these numbers in 4 bits we have:

$7_{10} = 0111_2$ $3_{10} = 0011_2$ 2's comp form = 1101_2

$$\begin{array}{r} 0111 \\ +\ 1101 \\ \hline 10100 \end{array}$$

ignoring the overflow (extra bit) we have our answer = $0100_2 = 4_{10}$

Why Two's Complement? (cont.)

- Example using 2's complement:

Suppose that our problem (in decimal) is: 27 - 13.

Representing these numbers in 5 bits we have:

$27_{10} = 11011_2$ $13_{10} = 01101_2$ 2's comp form $= 10011_2$

```
   11011
+  10011
 -------
  101110
```

101110 ignoring the overflow (extra bit) we have our answer

$01110_2 = 14_{10}$

(Note we know overflow has occurred since the MSB of the result is different than that of the operands.)

Solutions: Practice Constructing Recursive Functions

```
/* recursively computes the sum of the first n integers */
int sum (int n)
{    if (n == 1)
          return n;
     else return (n + rsum(n-1));
}
```

A solution to practice problem #1

Solutions: Practice Constructing Recursive Functions

```
/* recursively counts the number of occurrences of a */
/* specific character in a given string.             */
int rcount (char ch, const char *string)
{    int answer;
     if (string[0] == '\0')       /*simple case of empty string */
          answer = 0;
     else if (ch == string[0])    /* first character is a match */
               answer = 1 + rcount(ch, &string[1]);
          else
               answer = rcount(ch, &string[1]);
     return (answer);
}
```

A solution to practice problem #2