# Introduction to exercise 5b

CS-460: Programming Languages

Robert Bruce

# Return of the Shunting Yard algorithm: Evaluating Boolean expressions

- Previously, we discussed using Dijkstra's Shunting Yard algorithm to convert numerical mathematical expressions from infix notation into postfix notation while preserving operator precedence.
- In this lecture, we'll convert Boolean expressions from infix notation to postfix notation using Dijkstra's Shunting Yard algorithm.
- Operator precedence will apply.

# What is a Boolean expression?

- A Boolean expression is any expression that evaluates to either TRUE or FALSE.
- Boolean expressions can be comprised of numerical expressions (as long as those expressions have some sort of comparator operator to evaluate to TRUE or FALSE).
- Boolean expressions can be mixture of numerical expressions and Boolean logic too!

# What is a Boolean expression?
## (continued)

- A Boolean expression can contain a mixture operand datatypes (e.g. *bool*, *char*, *int*) as long as the overall expression evaluates to a TRUE or FALSE.
- Examples:

```
(byte_index < 255) && (byte != '\0')

((i >= 0) && (i <= 99))

((valid_input == TRUE) && (number_input_bytes < MAX_BUFFER_SIZE))

((byte >= 'A') && (byte <= 'Z') || (byte >= 'a') && (byte <= 'z'))
```

# What makes Boolean expressions difficult to evaluate?

- In addition to observing Boolean operator precedence, we must also observe mathematical operator precedence!
- Example:

```
((foo - bar * 2) > 128)
```

*We must first evaluate* `foo - bar * 2` *before determining if that resultant is greater than* `128` *because parenthesis have higher precedence than the greater-than sign. Within the numerical expression,* `foo - bar * 2`*, we must observe operator precedence: multiplication has higher precedence than subtraction.*

- In the files folder under Canvas, I've written an implementation of the Shunting Yard algorithm specifically for Boolean expressions in a C-like programing language.
- The file is titled, "boolean_expression_postfix_algorithm.txt".

# Why convert and evaluate Boolean expressions in postfix?

- Evaluating a Boolean expression in infix notation can be difficult while observing operator precedence.
- Converting the expression into postfix notation while observing operator precedence makes evaluating the Boolean expression much easier.
- The conversion process from infix to postfix will require a temporary working stack.
- The evaluation of the postfix expression will also require a temporary working stack.
- The stack is a Last-In-First-Out (LIFO) data structure.

# Why convert and evaluate Boolean expressions in postfix?

- Ultimately, you will be evaluating Boolean expressions on-the-fly when running programs in your interpreter:
  - ✓ You will need to determine if the Boolean result of some complex expression is either TRUE or FALSE.
  - ✓ IF statements, WHILE statements, and FOR statements all contain Boolean expressions. Examples:

    ```
    if ((hex_digit >= '0') && (hex_digit <= '9') || (hex_digit >= 'A') && (hex_digit <= 'F') || (hex_digit >= 'a') && (hex_digit <= 'f'))
    ```

    ```
    while ((num_digits > 0) && (num_digits <= 2))
    ```

    ```
    for (counter = 0; counter < 100; counter = counter + 1)
    ```

  - ✓ An assignment statement could also contain a Boolean expression. Example:

    ```
    foo = (num_digits > 0) && (current_byte != '\0')
    ```

# Converting from infix to postfix using the Shunting Yard algorithm

Example of a Boolean expression in infix notation:

```
(byte >= '0') && (byte <= '9')
```

# Converting the following Boolean expression from infix to postfix using the Shunting Yard algorithm:

```
(byte >= '0') && (byte <= '9')
```

| STEP | INPUT TOKEN | POSTFIX OUTPUT | STACK |
|---|---|---|---|
| 1 | ( | | |
| 2 | | | ( |
| 3 | byte | byte | ( |
| 4 | >= | | ( >= |
| 5 | ‘ | byte | ( >= |
| 6 | | byte ‘ | ( >= |
| 7 | 0 | byte ‘ | ( >= |
| 8 | | byte ‘0 | ( >= |
| 9 | ’ | byte ‘0 | ( >= |
| 10 | | byte ‘0’ | ( >= |
| 11 | ) | byte ‘0’ | |
| 12 | | byte ‘0’ >= | |
| 13 | && | byte ‘0’ >= | |
| 14 | | byte ‘0’ >= | && |
| 15 | ( | byte ‘0’ >= | && |
| 16 | | byte ‘0’ >= | && ( |
| 17 | byte | byte ‘0’ >= | && ( |
| 18 | | byte ‘0’ >= byte | && ( |
| 19 | <= | byte ‘0’ >= byte | && ( |
| 20 | | byte ‘0’ >= byte | && ( <= |
| 21 | ‘ | byte ‘0’ >= byte | && ( <= |
| 22 | | byte ‘0’ >= byte ‘ | && ( <= |
| 23 | 9 | byte ‘0’ >= byte ‘ | && ( <= |
| 24 | | byte ‘0’ >= byte ‘9 | && ( <= |
| 25 | ’ | byte ‘0’ >= byte ‘9 | && ( <= |
| 26 | | byte ‘0’ >= byte ‘9’ | && ( <= |
| 27 | ) | byte ‘0’ >= byte ‘9’ | && ( <= |
| 28 | | byte ‘0’ >= byte ‘9’ <= | && |
| 29 | | byte ‘0’ >= byte ‘9’ <= && | |

# Converting from infix to postfix using the Shunting Yard algorithm

Example of a Boolean expression in infix notation:

```
(byte >= '0') && (byte <= '9')
```

Same Boolean expression in postfix notation:

```
byte '0' >= byte '9' <= &&
```

# Evaluating a Boolean expression in infix notation

Example: evaluating a Boolean expression in infix notation while observing operator precedence. *Note: for this example, byte equals 'A'.*

```
EVALUATE: (byte >= '0') && (byte <= '9')
EVALUATE: ('A'  >= '0') && ('A'  <= '9')
EVALUATE: (    TRUE   ) && (   FALSE   )
EVALUATE:      TRUE     &&     FALSE
EVALUATE: FALSE
```

When *byte* equals 'A', the Boolean expression above evaluates to **FALSE**.

# Evaluating a postfix Boolean expression using a stack

Example: evaluating in postfix notation for a Boolean expression. *Note: for this example, byte equals 'A'.*

```
byte ‘0’ >= byte ‘9’ <= &&
```

# Evaluating the following postfix Boolean expression using a stack:

```
byte '0' >= byte '9' <= &&
```

| STEP | INPUT TOKEN | STACK | NOTES |
|---|---|---|---|
| 1 | byte | | |
| 2 | | byte | Push (byte) |
| 3 | '0' | byte | |
| 4 | | byte '0' | Push ('0') |
| 5 | >= | byte '0' | |
| 6 | | byte '0' >= | Push (>=) |
| 7 | | | Pop (>=) |
| 8 | | | Pop ('0') |
| 9 | | | Pop (byte) |
| 10 | | | Evaluate: byte >= '0'. Since byte is 'A', evaluation results in TRUE. |
| 11 | | TRUE | Push (TRUE) |
| 12 | byte | TRUE | |
| 13 | | TRUE byte | Push (byte) |
| 14 | '9' | TRUE byte | |
| 15 | | TRUE byte '9' | Push ('9') |
| 16 | <= | TRUE byte '9' | |
| 17 | | TRUE byte '9' <= | Push (<=) |
| 18 | | TRUE byte '9' | Pop (<=) |
| 19 | | TRUE byte | Pop ('9') |
| 20 | | TRUE | Pop (byte) |
| 21 | | TRUE | Evaluate: byte <= '9'. Since byte is 'A', evaluation results in FALSE. |
| 22 | | TRUE FALSE | Push (FALSE) |
| 23 | && | TRUE FALSE | |
| 24 | | TRUE FALSE && | Push (&&) |
| 25 | | TRUE FALSE | Pop (&&) |
| 26 | | TRUE | Pop (FALSE) |
| 27 | | | Pop (TRUE) |
| 28 | | | Evaluate: TRUE && FALSE. Evaluation results is FALSE. |
| 29 | | FALSE | Push (FALSE) |
| 30 | | | Pop (FALSE) |
| 31 | | | Evaluation complete: Boolean expression is FALSE. |