# Tokenizer

Sunny Chen and Michael Zhao

## Program Description:

This program implements a tokenizer with support for:

- C operators
- hex/float/decimal/octal numbers
- /*...*/ and //… style comments
- Single and double quotes (strings)
- C reserved keywords (int, while, etc.)

Included files are:

- tokenizer.c
- testcases.txt
- readme.pdf

## Usage:

All the code is contained within the tokenizer.c file. The program can be built using:

```
> gcc tokenizer.c -o tokenizer
```

Tokenizer accepts inputs as a single command-line argument. It can be run as the following:

```
> ./tokenizer “test123 123.3”
word "test123"
float "123.3"
```

Alternatively, you can input the contents of a file. For example, if we want to tokenize a file called “input.txt” containing the string “filetest 123”, we would run:

```
> ./tokenizer “`cat input.txt`”
word "filetest"
decimal "123"
```

## How Token Recognition Works:

Tokenizer scans through an input string iteratively within the main scan() function, moving across the string character by character. For each character, we run a series of checks to determine what class of token it belongs to. Our program is greedy, meaning once it determines the type of a token, it will try to make it as long as possible. After length, specificity takes priority, so in order of decreasing priority the numerical token classes are hexadecimal, octal, decimal, float. (ex: ‘07’ can be either octal or decimal, and the tokens are the same length but since octal is more specific than decimal, ‘07’ is octal. However, ‘079’ would be decimal since the token is longer than the alternative ‘07’ octal ‘9’ decimal). We demonstrate this priority by having our scan_type functions check there are no other longer token types possible before categorizing a token, and by ordering octal before decimal in the scan function.

The order of priority is as follows:

1. Whitespace (‘\n’, ‘\t’, spaces, etc.)
2. Comments
3. Quotes (strings)
4. Operators
5. Hex
6. Octal
7. Decimal
8. Float
9. Words/Reserved keywords
10. Unknown characters

Note that while operator precedence is higher than those of the number token types, should certain operators (‘.’, ‘+’, ‘-’) be used in a number token, it will be considered part of the number token and not an operator. The same is true for comments that appear within quotes. (ex: 3.14e-10 is considered a float despite containing the minus operator, and “//hi” is considered a string despite containing the higher priority single line comment symbol)

We’ll describe how our program deals with operators, numbers, and reserved C keywords, which are more complicated. The other tokens are easier to deal with, and detection is handled within scan().

## Numbers

Our program has 4 functions to recognize a number token, one for each type. We run them in the order of their priority (hex -> oct -> dec -> float). Once a function detects a match, we don’t check any of the other number types and restart the main loop in scan(). Since differentiating between the number class can be difficult and fraught with corner cases, we explain how each function works in detail below.

## Hexadecimal

scan_hex(constant char* arg, int index)

**Description:**

Scans the string arg parameter, starting at the given index parameter for a hexadecimal token and prints it it found.

**Return Values:**

Returns the index of the next available character not part of a hexadecimal token. (returns the parameter index if no hexadecimal is found)

**Operation:**

This function first compares the characters stored at the index and index+1 in the string to check for the hexadecimal code “0x” or “0X” (both are accepted). If no hexadecimal code is found, the original index is returned. Otherwise, prints the hexadecimal label, the first two characters, and loops through the following characters checking via the is_hex function if one does not belong in a hexadecimal token. Should the character be a valid hexadecimal digit, it is printed. The function will return the index of the first character is_hex finds to not be a hexadecimal digit.

**Special Cases:**

Note the edge case where just the hexadecimal code is found is still accepted as a hex token (see first two examples). Also note (as seen in example 3) uppercase characters A-F and lowercase characters a-f are both considered valid hexadecimal digits. In example 6 see that white space is not considered a valid hexadecimal digit.

**Assumptions:**

Assumes index will be less than the length of the string

**Examples:**

scan_hex(“0x”,0); returns 2 and prints hex “0x”
scan_hex(“0X”,0); returns 2 and prints hex “0X”
scan_hex(“0xA34f3s4”,0); returns 7 and prints hex “0xA34f3s4”
scan_hex("g*30XFa8.fg8",2); returns 2 and prints nothing
scan_hex("g*30XFa8.fg8",3); returns 8 and prints hex “0xFa8”
scan_hex(“0    x292”,0); returns 0 and prints nothing

## Octal

scan_oct(constant char* arg, int index)

**Description:**

Scans the string arg parameter, starting at the given index parameter for an octal token and prints if it is found.

**Return Values:**

Returns the index of the next available character not part of a octal token (returns the parameter index if no octal is found)

**Operation:**

First compares the character stored at the index to check for the octal char “0”. If no octal char is found, the original index is returned. If an octal code is found, loop through the following characters WITHOUT printing, using is_oct to check until a non-octal character is found. The non-octal character must be checked for edge cases. If the character is a non-octal digit (8 or 9) then the token is a decimal and the original index parameter should be returned. If the following digit is a decimal point “.” and the character after that is a decimal digit (according to is_dec) then the original index should be returned. If neither of these two edge cases are true, then the preceding characters we checked will be looped through and printed as an octal token.

**Special Cases:**

“0” is accepted as an octal (see example 1).

If a decimal, non-octal digit is present in the token, the entire token is considered decimal (examples 3 and 4).

If a decimal point is found and followed by a decimal digit, the token is considered a float (examples 5, 6, 8).

If the decimal point is followed by a non decimal character, it is a structure member and the previously read digits are considered an octal token

**Examples:**

scan_oct(“0”,0); returns 1 and prints octal “0”

scan_oct(“01”,0); returns 2 and prints octal “01”

scan_oct(“09”,0); returns 0 and prints nothing

scan_oct("07.9",0); returns 0 and prints nothing

scan_oct(“07.”,0); returns 2 and prints octal “07”

scan_oct(“e07/9”,1); returns 3 and prints octal 07

## Decimal

scan_dec(constant char* arg, int index)

**Description:**

Scans the string arg parameter, starting at the given integer index parameter for a decimal token and prints it it found.

**Return Values:**

Returns the index of the next available character not part of a decimal token (returns the parameter index if no decimal is found)

**Operation:**

Checks if the first character is a decimal digit, if not, returns the index parameter. If so, continue looping and checking using is_dec WITHOUT printing until a non decimal digit is found. If this character is the decimal point and the character following is a decimal digit, then the token is a float and the function returns the original index parameter before exiting. Otherwise the decimal token has concluded, and we loop through the previous indices that have been checked to print out the decimal token.

**Special Cases:**

If the decimal point is followed by a non decimal character, it is a structure member and therefore the previously read digits are considered a decimal token

**Examples:**

scan_dec(“0”,0); returns 1 and prints decimal “0” (Note: since scan_oct is run before scan_dec, scan_dec would never encounter “0”)

scan_dec(“123”,0); returns 2 and prints decimal “01”

scan_dec(“09”,0); returns 2 and prints decimal “09”

scan_dec(“5.3”,0); returns 0 and prints nothing

scan_dec(“15.”,0); returns 2 and prints decimal “15”

## Floating Point

scan_float(constant char* arg, int index)

**Description:**

Scans the string arg parameter, starting at the given integer index parameter for a float token and prints it it found.

**Return Values:**

Returns the index of the next available character not part of a float token (returns the parameter index if no float is found)

**Operation:**

Checks to ensure the token starts with a decimal number, if not this is not a float. Loops, checking with is_dec and printing until a non decimal digit is reached. If not a decimal point, not a float (this is an error that should not occur so -1 is returned). If it is a decimal point, print it and continue looping with is_dec until a non decimal digit is reached. Then we check if this character is part of scientific notation, by checking for “e”, “e+” or “e-” and looping a third time for the exponent. Should the character not belong to scientific notation, return this last index, the float token is complete. In addition, should no digit follow the “e”, “e+”, or “e-” label, we have found the end of the float and the index should start at e.

**Special Cases:**

e can be followed by a number, a “+” and then a decimal number, or “-” and then a decimal number.

**Examples:**

scan_float(“0.8”,0); returns 3 and prints float “0.8”
scan_float(“12.214.52”,0); returns 6 and prints float “12.214”
scan_float(“1.0e32”,0); returns 5 and prints float “1.0e32”
scan_float(“1.2e+”,0); returns 0 and prints nothing
scan_float(“1.0e-2.3”,0); returns 6 and prints float “1.0e-2”

## Operators

Since we have 43 operators, comparing our input could become time consuming. To facilitate time efficient access, operators are stored in a modified trie such that each individual character in an operator is stored as a node in a linked list, and the same starting character can link to different characters that also begin with the same sequence of characters (and thus different operators). For example, the less than “<”, the less than or equal test “<=”, and the shift left equals “<<=” operators all share the same first character “<”.

When recognizing tokens, we will scan through the trie looking for the longest-length operator that matches our input string. For example, “<<” will take precedence over two “<”s. We start by using op_search to test if our first character is a possible starting character for an operator. Then we increment, attempting to find characters afterward that match the children of the first operator structure matched. We continue doing so until no children match the next character whereupon we print out the operator. If the first character did not match with any children of op_main, then no operator was found.

## C Reserved Words

Our tokenizer supports recognition of C reserved keywords. To quickly check if a work token is a keyword or not, we use a hashtable and insert all 32 keywords into it. Once we have an alphanumeric character which is not part of a number token, we begin incrementing along the string until a non alphanumeric number is reached. Here we check if the word reached is a reserved keyword by formatting as a string and using our ht_lookup function to search the hash table for said word. If it is a keyword we will label appropriately, otherwise we label simply as a word. We handle the error condition where our word is NULL by printing an error message with exiting the program.

## Tokenizer Assumptions:

No `, \, “ characters used during testing, so we assume these characters will not appear in the input string.

## Additional Comments:

Note that our is_hex, is_oct, and is_dec functions are similar, however we chose to write them separately to improve code clarity, function reusability, and time efficiency.

Within some of our scan_type functions, specifically scan_oct and scan_decimal, we encounter other token types but refuse to categorize them. We made this design decision to improve code clarity and modularity of our scan functions, despite a slight reduction in time efficiency (not big

O significant). This makes our program more adaptable to future changes and simplifies our scan function.

Within some of our scan_type functions, specifically scan_oct and scan_decimal, we choose to not print our results until the end of the function by looping through the values again. Here, we chose to prioritize space efficiency over time efficiency because this simplified code (by avoiding the use of dynamic memory allocation) for a negligible change in time efficiency (time complexity remained linear)

We can unrecognized characters that don’t match any of our token classes at the end of our scan function, after all other tokens have been checked and print them as “unknown char”.