Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Assignment - 2: `Lexer for nanoC` *Marks: 100*
Assign Date: *September 18, 2023* Submit Date: *23:55, September 29, 2023*

---

1. *You must submit your assignment using the naming convention* `group_A2.x` *where* `x` *is one of* `tar`, `zip` *or* `rar` *and* `group` *is your group number.*

2. *To receive full credit, your program must be correct and your* `.pdf` *file must explain your program adequately.*

In a series of four assignments we intend to implement a compiler for a C-like language. To keep the problem tractable, we present a very small subset nanoC of C that is easy to manage and yet has most of the key flavours of C. It is framed based on C99 as standardized in International Standard **ISO/IEC 9899:1999 (E)**. We present an overview of this language in Section 1. This gives most notions of its syntax and semantics. The details its lexical and syntactic specifications, framed by stripping down C99 standard, is presented in Section 2. Finally a few example programs in the language are given in Section 3.

The implementation of the compiler for nanoC is split into 4 assignments as follows:

1. **Assignment 2**: Lexical Analyzer for nanoC using Flex. The lexical grammar specification is given here.

2. **Assignment 3**: Parser for nanoC using Bison. The phase structure grammar specification is given in Assignment 2.

3. **Assignment 4**: Machine-Independent Code Generator for nanoC using syntax-directed translation with Bison. Three-Address (intermediate) Code (TAC) used as target of translation is explained here.

4. **Assignment 5**: Target Code Generator for nanoC by simple code generation by table lookup. The target processor is taken to be x86 and a subset of its assembly language is presented here for use.

1 Overview of nanoC

The language is designed after C99 by substantially stripping it, but maintaining the key flavours. The main features include:

1. **Data Types**: The following are allowed:

   (a) Three *built-in types*: `void`, `int` and `char`, and pointers to these types: `void*`, `int*` and `char*`.
   (b) An *implicit boolean* type is used for conditional expressions, though no explicit use of this type is allowed.
   (c) *No conversion* between data types is allowed.
   (d) *No type alias* (`typedef`) is allowed.

2. **Constants / Literals**: The following are allowed:

   (a) *Integer constants* (of type `int`) in decimal notation. These are signed.
   (b) *Character constants* (of type `char`)
   (c) *String constants* (of type (`char *`)
   (d) *Boolean constant*: No explicit literal like `true` or `false` is allowed. However, an integer constant may be interpreted in the context of conditional expressions (`1` ≡ `true` and `0` ≡ `false`).

3. **Identifiers**: Any identifier is the name of a:

   (a) *Variable* (referred to as *Simple Identifier*). The type of a variable may be any built-in type (except `void`) or pointer to a built-in type.
   (b) *1-D array*. The type of the elements of an array may any type applicable for a variable.
   (c) *Function*. The types of the parameters of a function and the type of its return value may be of any type applicable for a variable.

- In addition, `void` is allowed for the parameter of a function not taking any parameter or the return value of a function that does not return any value.
- Note that arrays may not be passed to a function or returned from a function.

4. **Declarations**:

   (a) *Declaration before use*: Every identifier must be declared with its type before it is used.
   (b) *Single declaration*: Every variable / 1-D array is declared individually in separate statements.
   (c) *Initialization*: A variable may be (optionally) initialized by a constant in the declaration. No array may be initialized.
   (d) A function may be *forward declared* by its header / signature before its definition. However, it is not mandatory for a function to be declared by its header / signature before the *function definition*.
   (e) *Formal parameter names* may be skipped in a *forward declaration*, but not in a *function definition*.
   (f) The return type, parameter types and their order must be identical between a forward declaration and the definition.

5. **Operators**: The following are allowed:

   (a) *Arithmetic Operators*: Addition, Subtraction, Multiplication, Division, & Remainder: `+ - * / %`
   (b) *Relational Operators*: Less than, Greater than, Less than or Equal to, Greater than or Equal to, Equal to, and Not Equal to: `< > <= >= == !=`
   (c) *Logical Operators*: AND, OR, and NOT: `&& || !`
   (d) *Conditional Operator*: `?:`
   (e) *Pointer Operators*: Address Of, De-reference, and Indirection: `& * ->`
   (f) *Assignment Operator*: `=`

   Operators follow the arity, precedence and associativity of C.

6. **Comments**: Both single-line (`// ... \n`) and multi-line comments (`/* ... */` are allowed.

7. **Statements**: The following are allowed:

   (a) *Compound Statement*: Zero or more statements within a lexical / block scope: `{ ... }`
   (b) *Expression Statement*: Any expression terminated by semicolon (`;`)
   (c) *Selection Statement*: `if-else` or `if` statements which also models `switch` by nesting
   (d) *Loop Statement*: For loop (`for(...; ...; ...)`) which can model `do-while` and `while` as well
   (e) *Jump Statement*: Return from a function with or without a value: `return`

8. **Functions**: The following are allowed:

   (a) A function takes zero or more *parameters*. A function not taking any parameter may be coded as `<return_type> <function_name>()` or as `<return_type> <function_name>(void)`.
   (b) A function may *return* zero or one value. A function not returning any value is coded as `void <function_name>(<parameters>)`. For such a function, `return` statement is optional.
   (c) A function may be *recursive* or *non-recursive*. *Co-recursive* functions are allowed.
   (d) *Function Declaration / Definition*:
   - A function may be forward declared by its header / signature before its definition.
   - Formal parameter name may be skipped in such a declaration.
   - The return type, parameter types and their order must be identical between a forward declaration and the definition.

   (e) Computation starts with `int main()` function that takes no parameter and returns an integer.
   (f) *Pointers to functions* are not allowed.
   (g) *Variadic functions* are not allowed.
   (h) `inline` *functions* are not allowed.

9. **Scopes**: The following are allowed:

   (a) *Scoping Rule*: Static or Lexical.
   (b) *Block Scope*: Nested block scopes are allowed.
   (c) *Function Scope*: Block scope associated with a function definition.
   (d) *Global Scope*: Any declaration outside of any function scope is global. It is available from the point of declaration to the end of the file.

10. **Files**: The following are allowed:

    (a) *Source file*: A single file containing `int main()` function, other functions and global declarations. Multiple source files are not allowed.

(b) *Header file*: No header file is allowed.
(c) *File Extension*: The source file has extension `.nc`

11. **Pre-Processor**: *No pre-Processor* or its directives is allowed.

12. **Standard Library**: *No standard library* may be included.

13. **Input / Output**: The following are allowed:

    (a) In the absence of standard library, `scanf` / `printf` may not be used.
    (b) Following I/O functions will be provided in x86 assembly with a C wrapper in Assignment 5. The respective function headers may be implicitly included as forward declaration for semantic actions.

    - `int printStr(char *)`: Prints a string of characters. The parameter is terminated by `'\0'`. The return value is the number of characters printed.
    - `int readInt(int *n)`: Reads a signed integer in `'%d'` format. Caller gets the value through the pointer parameter. The return value is `1` (on success) or `0` (on failure).
    - `int printInt(int n)`: Prints a signed integer (`n`) with left-alignment. The sign for a negative number is printed while for a positive number it is skipped. On success, function will return the number of characters printed and on failure it will return `0`.

    (c) *No FILE I/O* to be supported.

14. **Major Features Omitted**:

    (a) *Data Types, Specifiers, and Qualifiers*:

    - Variants of Integer: `short`, `long`; Floating point: `float`, `double`; `_Bool`; `_Complex`; `_Imaginary`; `signed`; `unsigned`
    - Storage-class Specifiers: `typedef`, `extern`, `static`, `auto`, `register`
    - Type Qualifier: `const`, `restrict`, `volatile`
    - Structure / Union: `struct`, `union`
    - Enumerated Type: `enum`
    - Type Alias: `typedef`

    (b) *Constants / Literals*: Non-decimal integer constants (oct / hex), Floating point constants, Integer & Floating point suffixes
    (c) *Array and Pointers*:

    - Multi-D array
    - Multi-level interactions
    - Pointers to functions

    (d) *Declarations*:

    - Multiple declarations: Multiple identifiers may be declared in a statement.
    - Initialization: Arrays may be initialized.
    - A function may be *forward declared* by its header / signature before its definition.
    - A function may be not declared by its header / signature before the *function definition*.
    - *Formal parameter names* may be skipped in a *forward declaration*.

    (e) *Operators*:

    - Post / Pre Increment / Decrement Operators: `++`, `--`
    - Bit-wise Operators: `&`, `^`, `|`
    - Assignment Operators: `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
    - Comma Operator: `,`
    - `sizeof` Operator: `sizeof`

    (f) *Statements*:

    - Labeled Statement: *identifier*, `case`, `default`
    - Selection Statement: `switch`
    - Loop Statement: `while`, `do-while`, `for` with local loop control
    - Jump Statement: `goto`, `continue`, `break`

    (g) *Functions*:

    - Parameters and return value of different data types
    - Pointers to functions
    - Variadic functions: `...`
    - Inline functions: `inline`

    (h) *Scopes*:

    - External scope: `extern`

- Static file scope: `static`

(i) *Files*:

- Source files: Any number of files with one file containing `int main()` function
- Header files: Any number of header files
- File Extensions: A source file has extension `.c` and a header file has extension `.h`

(j) *Pre-Processor*: All directives: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`, `#include`, `#define`, `#undef`, `#line`, `#error`, `#pragma`

(k) *Standard Library*: To be included.

2 Specification of nanoC

We now present the formal specification of nanoC following the C99 standard: International Standard **ISO/IEC 9899:1999 (E)**. For this we strip down the Lexical Grammar and the Phase Structure Grammar from the Standard. The specifications quoted here are written using a precise yet compact notation typically used for writing language specifications.

2.1 Notation

In the syntax notation used here, syntactic categories (non-terminals) are indicated by *italic type*, and literal words and character set members (terminals) by `text type`. A colon (:) following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript $_{opt}$ so that the following indicates an optional expression enclosed in braces.

`{` *expression*$_{opt}$ `}`

2.2 Lexical Grammar of nanoC

1. **Lexical Elements**

   token:

   keyword
   identifier
   constant
   string-literal
   punctuator

2. **Keywords**

   keyword: one of

   ```
   char else for if int return void
   ```

3. **Identifiers**

   identifier:

   identifier-nondigit
   identifier identifier-nondigit
   identifier digit

   identifier-nondigit: one of

   ```
   _  a  b  c  d  e  f  g  h  i  j  k  l  m
      n  o  p  q  r  s  t  u  v  w  x  y  z
      A  B  C  D  E  F  G  H  I  J  K  L  M
      N  O  P  Q  R  S  T  U  V  W  X  Y  Z
   ```

   digit: one of

   ```
   0  1  2  3  4  5  6  7  8  9
   ```

4. **Constants**

   constant:

   integer-constant
   character-constant

   integer-constant:

   `0`
   sign$_{opt}$ *nonzero-digit*
   integer-constant digit

   nonzero-digit: one of

`1 2 3 4 5 6 7 8 9`

sign: one of

`+ -`

character-constant:

`'`*c-char-sequence*`'`

c-char-sequence:

c-char

c-char-sequence c-char

c-char:

any member of the source character set except

the single-quote `'`, backslash `\`, or new-line character

escape-sequence

escape-sequence: one of

`\'  \''  \?  \\  \a  \b  \f  \n  \r  \t  \v`

5. **String literals**

string-literal: *// Terminated by null =* `'\0'`

`"`*s-char-sequence*$_{opt}$`"`

s-char-sequence:

s-char

s-char-sequence s-char

s-char:

any member of the source character set except

the double-quote `''`, backslash `\`, or new-line character

escape-sequence

6. **Punctuators**

punctuator: one of

`[ ] ( ) { } -> & * + - / % !  ? < > <= >= == != && || = :  ; ,`

7. **Comments**

(a) *Multi-line Comment*

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it. Thus, `/* ...  */` comments do not nest.

(b) *Single-line Comment*

Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

2.3 Phrase Structure Grammar of nanoC

1. **Expressions**:

/ The grammar is structured in a hierarchical way with precedences resolved. Associativity is handled by left or right recursion as appropriate.*/*

primary-expression:

identifier *// Simple identifier*

constant *// Integer or character constant*

string-literal

`(` *expression* `)`

postfix-expression: *// Expressions with postfix operators. Left assoc. in C; non-assoc. here*

primary-expression

postfix-expression `[` *expression* `]` *// 1-D array access*

postfix-expression `(` *argument-expression-list*$_{opt}$ `)` *// Function invocation*

postfix-expression `->` *identifier* *// Pointer indirection. Only one level*

// Only a single postfix op is allowed in an expression here

argument-expression-list:

assignment-expression

```
        argument-expression-list , assignment-expression

unary-expression:
        postfix-expression
        unary-operator unary-expression    // Expr. with prefix ops. Right assoc. in C; non-assoc. here
        // Only a single prefix op is allowed in an expression here

unary-operator: one of
        & * + - !    // address op, de-reference op, sign ops, boolean negation op

multiplicative-expression:    // Left associative operators
        unary-expression
        multiplicative-expression * unary-expression
        multiplicative-expression / unary-expression
        multiplicative-expression % unary-expression

additive-expression:    // Left associative operators
        multiplicative-expression
        additive-expression + multiplicative-expression
        additive-expression - multiplicative-expression

relational-expression:    // Left associative operators
        additive-expression
        relational-expression < additive-expression
        relational-expression > additive-expression
        relational-expression <= additive-expression
        relational-expression >= additive-expression

equality-expression:    // Left associative operators
        relational-expression
        equality-expression == relational-expression
        equality-expression != relational-expression

logical-AND-expression:    // Left associative operators
        equality-expression
        logical-AND-expression && equality-expression

logical-OR-expression:    // Left associative operators
        logical-AND-expression
        logical-OR-expression || logical-AND-expression

conditional-expression:    // Right associative operator
        logical-OR-expression
        logical-OR-expression ? expression : conditional-expression

assignment-expression:    // Right associative operator
        conditional-expression
        unary-expression = assignment-expression    // unary-expression must have lvalue

expression:
        assignment-expression
```

2. **Declarations** *declaration:* // Simple identifier, 1-D array or function declaration of built-in type

```
        type-specifier init-declarator ;    // Only one element in a declaration
init-declarator:
        declarator    // Simple identifier, 1-D array or function declaration
        declarator = initializer    // Simple id with init. initializer for array / fn/ is semantically skipped
type-specifier:    // Built-in types
        void
        char
```

`int`

declarator:

pointer$_{opt}$ *direct-declarator* *// Optional injection of pointer*

direct-declarator:

identifier *// Simple identifier*
identifier `[` *integer-constant* `]` *// 1-D array of a built-in type or ptr to it. Only +ve constant*
identifier `(` *parameter-list*$_{opt}$ `)` *// Fn. header with params of built-in type or ptr to them*

pointer:

`*`

parameter-list:

parameter-declaration
parameter-list `,` *parameter-declaration*

parameter-declaration:

type-specifier pointer$_{opt}$ *identifier*$_{opt}$ *// Only simple ids of a built-in type or ptr to it as params*

initializer:

assignment-expression

3. **Statements**

statement:

compound-statement *// Multiple statements and / or nest block/s*
expression-statement *// Any expression or null statements*
selection-statement *//* `if` *or* `if-else`
iteration-statement *//* `for`
jump-statement *//* `return`

compound-statement:

`{` *block-item-list*$_{opt}$ `}`

block-item-list:

block-item
block-item-list block-item

block-item: *// Block scope - declarations followed by statements*

declaration
statement

expression-statement:

expression$_{opt}$ `;`

selection-statement:

`if` `(` *expression* `)` *statement*
`if` `(` *expression* `)` *statement* `else` *statement*

iteration-statement:

`for` `(` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

jump-statement:

`return` *expression*$_{opt}$ `;`

4. **Translation Unit**

translation-unit: *// Single source file containing* `main()`

function-definition
declaration

function-definition:

type-specifier declarator `(` *declaration-list*$_{opt}$ `)` *compound-statement*

declaration-list:

declaration
declaration-list declaration

3 Example Programs in nanoC

We present a set of programs in nanoC based on the lexical, syntactic, and semantic specification of the language. These example programs would help in understanding nanoC better. Also, these can be used as testcases for the compiler.

We assume the following functions for IO in the examples.

```
#include <stdio.h>

int printInt(int x){
    return printf("%d", x);
}

int readInt(int *n){
    return scanf("%d", n);
}

int printStr(char *s){
    return printf("%s", s);
}
```

3.1 Program 1: Add

```
// Add two numbers
int main() {
    int x = 2;
    int y = 3;
    int z;
    z = x + y;
    printInt(x);
    printStr("+");
    printInt(y);
    printStr(" = ");
    printInt(z);
    return 0;
}
```

3.2 Program 2: Max of 3

```
// Find max of three numbers
int main() {
    int x = 2;
    int y = 3;
    int z = 1;
    int m;
    m = x > y? x: y;
    m = m > z? m: z;
    printStr("max(");
    printInt(x); printStr(", ");
    printInt(y); printStr(", ");
    printInt(z); printStr(") = ");
    printInt(m);
    return 0;
}
```

3.3 Program 3: Add + IO

```
// Add two numbers from input
int main() {
    int x;
    int y;
    int z;
    readInt(&x);
    readInt(&y);
    z = x + y;
    printInt(x);
    printStr("+");
    printInt(y);
    printStr(" = ");
    printInt(z);
    return 0;
}
```

3.4 Program 4: Swap

```
// Swap two numbers
void swap(int*, int*);
int main() {
    int x;
    int y;
    readInt(&x);
    readInt(&y);
    printStr("Before swap:\n");
    printStr("x = "); printInt(x);
    printStr(" y = "); printInt(y);
    swap(&x, &y);
    printStr("\nAfter swap:\n");
    printStr("x = "); printInt(x);
    printStr(" y = "); printInt(y);
    return 0;
}

void swap(int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t;
    return;
}
```

3.5 Program 5: Factorial: Iteration

```
// Find factorial by iteration
int main() {
    int n;
    int i = 0;
    int r = 1;
    readInt(&n);
    for(i = 1; i <= n; i = i + 1)
        r = r * i;
    printInt(n);
    printStr("! = ");
    printInt(r);
    return 0;
}
```

3.6 Program 6: Max + Array

```
// Find max of n numbers using array
int main() {
    int n;
    int a[10];
    int m;
    int i;
    readInt(&n);
    for(i = 0; i < n; i = i + 1) {
        readInt(&m);
        a[i] = m;
    }
    m = a[0];
    for(i = 1; i < n; i = i + 1) {
        if (a[i] > m)
            m = a[i];
    }
    printStr("Max of: ");
    printInt(a[0]);
    for(i = 1; i < n; i = i + 1) {
        printStr(", "); printInt(a[i]);
    }
    printStr(": = ");
    printInt(m);
    return 0;
}
```

3.7 Program 7: Factorial: Recursion

```
// Find factorial by recursion
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
int main() {
    int n = 5;
    int r;
    r = factorial(n);
    printInt(n);
    printStr("! = ");
    printInt(r);
    return 0;
}
```

3.8 Program 8: Fibonacci: Co-Recursion

```
// Find fibonacci by co-recursion
int f_odd(int);
int f_even(int);

int fibonacci(int n) {
    return (n % 2 == 0)? f_even(n): f_odd(n);
}

int f_odd(int n) {
    return (n == 1)? 1: f_even(n-1) + f_odd(n-2);
}

int f_even(int n) {
    return (n == 0)? 0: f_odd(n-1) + f_even(n-2);
}

int main() {
    int n = 10;
    int r;
    r = fibonacci(n);
    printStr("fibo(");
    printInt(n);
    printStr(") = ");
    printInt(r);
    return 0;
}
```

3.9 Program 9: Bubble Sort

```
// Forward declarations
void swap(int *p, int *q);
void readArray(int size);
void printArray(int size);
void bubbleSort(int n);
```

```
int arr[20]; // Global array

// Driver program to test above functions
int main() {
    int n;
    printStr("Input array size: \n");
    readInt(&n);
    printStr("Input array elements: \n");
    readArray(n);
    printStr("Input array: \n");
    printArray(n);
    bubbleSort(n);
    printStr("Sorted array: \n");
    printArray(n);
    return 0;
}

void swap(int *p, int *q) { /* Swap two numbers */
    int t = *p;
    *p = *q;
    *q = t;
}
void readArray(int size) { /* Function to read an array */
    int i;
    for (i = 0; i < size; i = i + 1) {
        printStr("Input next element\n");
        readInt(&arr[i]);
    }
}
void printArray(int size) { /* Function to print an array */
    int i;
    for (i = 0; i < size; i = i + 1) {
        printInt(arr[i]); printStr(" ");
    }
    printStr("\n");
}
void bubbleSort(int n) { /* A function to implement bubble sort */
    int i;
    int j;
    for (i = 0; i < n - 1; i = i + 1)
        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j = j + 1)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```

This code is lifted and edited from Bubble Sort Algorithm.

3.10 Program 10: Binary Search

```
int arr[10]; // Sorted array to search

// A recursive binary search function. It returns location of x
// in given array arr[l..r] is present, otherwise -1
int binarySearch(int l, int r, int x) {
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > x)
```

```
            return binarySearch(l, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearch(mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

int main() {
    int n = 5; // Number of elements
    arr[0] = 2;
    arr[1] = 3;
    arr[2] = 4;
    arr[3] = 10;
    arr[4] = 40;

    int x = 10; // Key to search
    int result = binarySearch(0, n - 1, x);
    if (result == -1)
        printStr("Element is not present in array");
    else {
        printStr("Element is present at index ");
        printInt(result);
    }
    return 0;
}
```

This code is lifted and edited from Binary Search.

4 The Assignment

1. Write a flex specification for the language of nanoC using the lexical grammar. Name of your file should be *group*_A2.l, where group is your group number. *The* *group*_A2.l *should not contain the function* main().
2. Write your main() (in a separate file *group*_A2.c to test your lexer.
3. Prepare a Makefile to compile the specifications and generate the lexer.
4. Prepare a test input file *group*_A2.nc that will test all the lexical rules that you have coded.
5. Prepare a *group*_A2.pdf file explaining the working of your lexer and your design choices.
6. Prepare a compressed-archive with the name *group*_A2.x, where x is one of zip, tar or rar, containing all the above files and upload it to Classroom.

5 Credits

1. Flex Specifications: **60**
2. Main function and Makefile: $\mathbf{15 + 5 = 20}$
3. Test file: **20**