Department of Computer Science
Ashoka University

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

Assignment - 3: `Parser for nanoC` *Marks: 100*
Assign Date: *October 12, 2023* Submit Date: *23:55, October 26, 2023*

---

1. *You must submit your assignment using the naming convention* `group_A3.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.*

---

1 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*

   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 `(` $parameter\text{-}list_{opt}$ `)` *// 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 compound-statement

2 The Assignment

1. Write a Bison specification for defining the tokens and phase structure grammar of nanoC and generate the required `y.tab.h` file.
2. Write a Bison specification for the language of nanoC using the phase structure grammar given in Assignment 2. Use the Flex specification that you had developed for Assignment 2 (if required, you may fix your Flex specification).
   While writing the Bison specification, you may need to make some changes to the grammar. For example, some non-terminals like

   argument-expression-list

   are shown as optional on the right-hand-side as:

   postfix-expression:

   postfix-expression (*argument-expression-list*$_{opt}$)

   One way to handle them would be to introduce a new non-terminal, *argument-expression-list-opt*, and a pair of new productions:

   argument-expression-list-opt:

   argument-expression-list

   ϵ

   and change the above rule as:

   postfix-expression:

   postfix-expression (*argument-expression-list-opt*)

3. Names of your `.l` and `.y` files should be ***group***`_A3.l` and ***group***`_A3.y` respectively. *The* `.y` *or the* `.l` *file should not contain the function* `main()`. Write your `main()` (in a separate file ***group***`_A3.c`) to test your lexer and parser.
4. Prepare a Makefile to compile the specifications and generate the lexer and the parser. Your Makefile must have a build rule such that when we run `make build`, the output is an executable named `parser`. Your `build` rule should have these commands. The `gcc` command without the `-Werror` will count as the Makefile being incorrect.
5. Prepare a test input file ***group***`_A3.nc` that will test all the rules that you have coded.
6. Prepare a tar-archive with the name ***group***`_A3.tar` containing all the files and upload to Classroom.

3 Credits

1. Correctness of Implementation: **70**
2. Main file: **5**
3. Makefile[1]: **5**
4. Test file: **5**
5. Explanation of Program: **15**

4 Autograding Specifics

1. The autograder will first try to execute `make build`. If this succeeds **and** if after running there is an executable file named `parser` in the directory, it goes to 3, else 2.

2. It executes the following commands[2][3] in order,

   ```
   bison -d group_A3.y
   flex -o lex.yy.c group_A3.l
   gcc -o parser lex.yy.c group_A3.tab.c group_A3.c -lfl -Werror
   ```

   If this succeeds **and** if after running there is an executable file named `parser` in the directory, it goes to 3 else reports `Compilation Error` and halts.

3. It runs your parser on each test case and matches the output of your program with the correct output. If there is a `Runtime Error` error on any test-case, it throws an exception which is handled by flagging your program and marking that case as failed.

4. We will release a document on the expected output format of the parser at a later date.

[1] If upon make build your Makefile overwrites the wrong file, it will be considered incorrect and if this causes a compilation error, it will be graded as such.

[2] The -Werror flag treats all warnings as errors. Thus, if your program compiles with a warning on your system, it will fail on ours. So please test your programs with this flag enabled.

[3] The -lfl flag can be substituted with -ll