Computer Science and Engineering Department
Indian Institute of Technology Kharagpur

Compilers Laboratory: CS39003
3rd Year CSE: 5th Semester

Assignment: *Parser for* tiny**C** Marks: *100*
Date posted: *Sep 5, 2024* Submission deadline: *Sep 29, 2024, 23:59*

1 Preamble – tinyC

This assignment follows the phase structure grammar specification of `C` language from the International Standard **ISO/IEC 9899:1999 (E)**. To keep the assignment within our required scope, we have chosen a subset of the specification as given below. We shall refer to this language as tiny**C**.

The lexical specification of tiny**C**, also taken and abridged from the Standard, has already been discussed in an earlier assignment. The phase structure grammar specification is written using the common notation of language specifications as discussed in that assignment.

2 Phrase Structure Grammar of tinyC

1. **Expressions**

   primary-expression:
      identifier
      constant
      string-literal
      (*expression*)
   postfix-expression:
      primary-expression
      postfix-expression **[** *expression* **]**
      postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
      postfix-expression **.** *identifier*
      postfix-expression $- >$ *identifier*
      postfix-expression $++$
      postfix-expression $--$
      (*type-name* **)** **{** *initializer-list* **}**
      (*type-name* **)** **{** *initializer-list* **, }**
   argument-expression-list:
      assignment-expression
      argument-expression-list **,** *assignment-expression*
   unary-expression:
      postfix-expression
      $++$ *unary-expression*
      $--$ *unary-expression*
      unary-operator cast-expression
      sizeof *unary-expression*
      sizeof **(** *type-name* **)**
   unary-operator: one of
      `&  *  +  -  ~  !`
   cast-expression:
      unary-expression
      (*type-name* **)** *cast-expression*
   multiplicative-expression:
      cast-expression
      multiplicative-expression $*$ *cast-expression*
      multiplicative-expression $/$ *cast-expression*
      multiplicative-expression $\%$ *cast-expression*

additive-expression:

multiplicative-expression

additive-expression + *multiplicative-expression*

additive-expression − *multiplicative-expression*

shift-expression:

additive-expression

shift-expression << *additive-expression*

shift-expression >> *additive-expression*

relational-expression:

shift-expression

relational-expression < *shift-expression*

relational-expression > *shift-expression*

relational-expression <= *shift-expression*

relational-expression >= *shift-expression*

equality-expression:

relational-expression

equality-expression == *relational-expression*

equality-expression ! = *relational-expression*

AND-expression:

equality-expression

AND-expression & *equality-expression*

exclusive-OR-expression:

AND-expression

*exclusive-OR-expression*ˆ*AND-expression*

inclusive-OR-expression:

exclusive-OR-expression

inclusive-OR-expression | *exclusive-OR-expression*

logical-AND-expression:

inclusive-OR-expression

logical-AND-expression && *inclusive-OR-expression*

logical-OR-expression:

logical-AND-expression

logical-OR-expression || *logical-AND-expression*

conditional-expression:

logical-OR-expression

logical-OR-expression ? *expression* : *conditional-expression*

assignment-expression:

conditional-expression

unary-expression assignment-operator assignment-expression

assignment-operator: one of

`=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=`

expression:

assignment-expression

expression , assignment-expression

constant-expression:

conditional-expression

2. Declarations

declaration:

declaration-specifiers init-declarator-list$_{opt}$ *;*

declaration-specifiers:

storage-class-specifier declaration-specifiers$_{opt}$

type-specifier declaration-specifiers$_{opt}$

type-qualifier declaration-specifiers$_{opt}$

function-specifier declaration-specifiers$_{opt}$

init-declarator-list:

init-declarator

init-declarator-list **,** *init-declarator*

init-declarator:

declarator

declarator = *initializer*

storage-class-specifier:

extern

static

auto

register

type-specifier:

void

char

short

int

long

float

double

signed

unsigned

_Bool

_Complex

_Imaginary

specifier-qualifier-list:

type-specifier specifier-qualifier-list$_{opt}$

type-qualifier specifier-qualifier-list$_{opt}$

type-qualifier:

const

restrict

volatile

function-specifier:

inline

declarator:

pointer$_{opt}$ *direct-declarator*

direct-declarator:

identifier

(*declarator* **)**

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**

direct-declarator

[static *type-qualifier-list*$_{opt}$ *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**

direct-declarator **[** *type-qualifier-list*$_{opt}$ ***]**

direct-declarator **(** *parameter-type-list* **)**

direct-declarator **(** *identifier-list*$_{opt}$ **)**

pointer:

***** *type-qualifier-list*$_{opt}$

***** *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:

type-qualifier

type-qualifier-list type-qualifier

parameter-type-list:

parameter-list

parameter-list **, ...**

parameter-list:

parameter-declaration

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

parameter-declaration:

declaration-specifiers declarator

declaration-specifiers

identifier-list:

identifier

identifier-list **,** *identifier*

type-name:

specifier-qualifier-list

initializer:

assignment-expression

{ *initializer-list* **}**

{ *initializer-list* **, }**

initializer-list:
designation$_{opt}$ *initializer*
initializer-list **,** *designation*$_{opt}$ *initializer*

designation:
designator-list **=**

designator-list:
designator
designator-list designator

designator:
[*constant-expression* **]**
. *identifier*

3. Statements

statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

labeled-statement:
identifier **:** *statement*
case *constant-expression* **:** *statement*
default : *statement*

compound-statement:
{ *block-item-list*$_{opt}$ **}**

block-item-list:
block-item
block-item-list block-item

block-item:
declaration
statement

expression-statement:
expression$_{opt}$ **;**

selection-statement:
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **)** *statement*

iteration-statement:
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for (*expression*$_{opt}$ **;** *expression*$_{opt}$ **;** *expression*$_{opt}$ **)** *statement*
for (*declaration expression*$_{opt}$ **;** *expression*$_{opt}$ *) statement*

jump-statement:
goto *identifier* **;**
continue ;
break ;
return *expression*$_{opt}$ **;**

4. External definitions

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

function-definition:
declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

3 The Assignment

1. Write a bison specification for defining the tokens of tiny**C**, and generate the required y.tab.h file.

2. Write a bison specification for the language of tiny**C**, using the above phase structure grammar. Use the flex specification that you had developed for the liny**C** lex assignment (if required, you may fix your flex specification). Construct the **parse tree** that comes as an output of your sample input program, and store the parse tree in a human-readable format in the output file `output_`$roll1$`_`$roll2$`.txt`.

3. While writing the bison specification, you may need to make some changes to the grammar. For example, some non-terminals like

   argument-expression-list$_{opt}$

   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* **)**

4. The names of your lex and bison files should be `tinyC2_`$roll1$`_`$roll2$`.l` and `tinyC2_`$roll1$`_`$roll2$`.y`, respectively. ***Neither the*** `.y` ***nor the*** `.l` ***file should contain the function*** `main()`. Write a separate file `tinyC2_`$roll1$`_`$roll2$`.c` with the `main()` function to test your lexer and parser.

5. Prepare a Makefile to compile the specifications and generate the lexer and the parser. Also write a clean target to remove all the new files generated by make.

6. Prepare a test input file `input_`$roll1$`_`$roll2$`.c` that will be used for testing all the rules that you have coded.

7. Prepare a tar-archive with the name `tinyC2_`$roll1$`_`$roll2$`.tar` containing all the files (after cleaning), and upload to Moodle.

4 Credits

1. Specifications and testing: **70**

2. Main file: **10**
   No marks for Makefile, but a penalty of 20 marks for not including Makefile.

3. Test file: **20**