CS3300: Compiler Design	August 2020

Assignment #1

Scanning & Parsing	Deadline: 06/09/2020, 11:55PM

Note: corrections are in red.

Task

Write input files for lex and yacc to generate lexical analyzer and parser for the following language description.

This language specification conforms to the ANSI C-2011 standard with the following requirements/modifications:

• The language constructs:
- ✔ global variables, both normal as well as extern.
- ✔ function declaration, function definition, and function call (including recursion).
- ✔ for loop, while loop, do-while.
- ✔ if, if-else, if-else-if (nested/ladder) {if condition can be any expression including a literals/constants}.
- ✔ switch/case with case labeled with ':', must support fall-through, must support the optional default clause.
- ✔ variable declaration, definition, initialization locally/globally {multiple variables comma-separated}.
- ✔ strings (of the forms "..." (single), "..." "..." (multiple)), character, integer and floating point literals.
- ✔ single (//) and multi-line (/*...*/) comments {multi-line comments should be properly closed lexically}.
- ✔ break, continue, and return statements.
- ✔ a statement must end in a semi-colon ';' like the usual standard of C.

• The supported data types are:
- ✔ primary: int, long, short, float, double, void, char and their pointers.
- ✔ user-defined: structures (struct) {pointer dereferences must support arbitrary level of indirection.}
- ✔ derived: functions, arrays, and pointers: all supported data types. {no need to support function pointers.}

• Supported operators (precedence and associativity same as in C):
- ✔ *relational operators*: <, >, ==, <=, >=, !=
- ✔ *unary operators*: +, −, !, ∗, &, ~
- ✔ *binary operators*: +, −, ∗, &, |, /, %, ^
- ✔ *logical operator*: &&, ||
- ✔ *assignment operator*: =
- ✔ *suffix/postfix increment and decrement*: ++, --
- ✔ *structure field access operators*: ->, .
- ✔ *pointers*: *, & {pointer dereferences must support arbitrary/multiple level of indirection, i.e., int *****p;.}
- ✔ *function call*: () {any valid expression inside including blank, constants, and literals}
- ✔ *array subscripting*: [] {any valid expression inside including constants and literals}
- ✔ sizeof() {operands: <typename> or unary expression}
- ✔ *typecast*: (<typename>)

• Constructs **NOT** supported by this language:
- ✗ *preprocessors*: none allowed. {no #includes, #defines needed anywhere}
- ✗ ... (ellipsis), ?: (ternary operator) {function vararg list specification not required}
- ✗ *operators*: >>=, <<=, +=, -=, *=, /=, %=, &=, ^=, |=, <<, >>
- ✗ *constructs/keywords*: auto, const, goto, inline, register, restrict, signed, static, typedef, unsigned, enum, union,volatile, _Alignas, _Alignof, _Atomic, _Bool, _Complex, _Generic, _Imaginary, _Noreturn, _Static_assert, _Thread_local, __func__

• The following operator is *introduced*:

Operator	Description
<=>	*Three way comparison.* a<=>b returns the values −1, 0, or 1 depending on whether a < b, a == b, or a > b. Precedence and associativity same as: <, >, <=, >=

Note:
1. The ANSI C-2011 keywords that are not a part of this language can be used as identifiers.
2. No need to handle:
   - Semantics and expression evaluations.
   - Track whether a variable, structure, function is declared/defined before being used.
   - Type of a variable, structure, function, type checking.

Input

The input to the parser will be a program which may or may not be valid according to the above language description.
Sample execution format:

```
$ ./a.out [filename.c]
```

Note: input file `[filename.c]` should be passed as command line argument to `a.out`

Output

[1] *If the program is parsed successfully, then the following should be printed*:

```
***parsing successful***
#global_declarations = @
#function_definitions = @
#integer_constants = @
#pointers_declarations = @
#ifs_without_else = @
if-else max-depth = @
```

Note: Replace @ with corresponding counts. There must be one white space before and after '='. See sample testcases.

- `#global_declarations:` Total number of global declarations (enitites) including functions, global variables, extern variables at the global scope. Note that declaration and definition for the same function are counted separately.
-
- `#function_definitions:` Total number functions having bodies, i.e. defined.
- `#integer_constants:` Total number of integer constants that appear *anywhere* throughout the program. Should support hexadecimal constants like `0x1234`. Upper-case ‘`X`’ for hex need not be supported.
- `#pointers_declarations:` Total count of pointers that are declared throught the file. Note that if a function returning a pointer, has both prototype (declaration) and definition, they are counted separately. Pointer casts should not be counted.
- `#ifs_without_else:` Total count `if` statements that have no associated `else` clause.
- `if-else max-depth:` The max height of `if-else-if` ladder. The height is recursively defined as follows.

  height(Ladder)
  =0, if there is no `if` statement at all.
  =0, if there is `if` but no accompanying `else`.
  =1 (for the accompanying `else`) + *height*(Ladder from this `else`)

  There will be many such ladders, the last line must print the *height of the longest ladder* present in the program. You should keep track of ladders across functions also. Note that there can be ties also.

  `if-else max-depth` = $max\{height(\text{Ladder}_1), ..., height(\text{Ladder}_n)\}$when there are n ladders in the input file.

[2] *On rejection by the parser, the following should be printed*:

```
***parsing terminated*** [syntax error]
```

[3] *On rejection by the lexer when there are unclosed comments, the following should be printed*:

```
***lexing terminated*** [lexer error]: ill-formed comment
```

[4] *On rejection by your a.out file it must handle these cases*:

Case 1: (invalid number of command-line arguments)-

```
***process terminated*** [input error]: invalid number of command-line arguments
```

Case 2: (no such file <filename> exists)-

```
***process terminated*** [input error]: no such file <filename> exists
```

Submission

Submit a tar.gz file with filename as <ROLLNO>_A1.tar.gz (eg. CS16D004_A1.tar.gz) having the following structure:

- CS16D004_A1<directory>
  - *.l
  - *.y
  - Makefile

Note: The Makefile should run `lex`, `yacc`, compile the generated code and generate an executable `a.out` file. Please be careful about the naming conventions and structure of the directory.

Sample testcase #1

Input

```
int *var=6;
struct mystruct *ms=&var;

int auto, static, inline;

extern void *k;
int p;
int p;
int p;

int *hh(char *p);

int main(int b)
{
        int auto=2,b=3,c;
        c=auto+b;
        printf("%d",c);

        struct player
        {
                int a;
                double c;
        };

        int *jj=&auto;

        System.out.print("java here");
        char echo[3]="bash here";
        myprintf("CS3300 here");
        printf(echo);

        /*
                mixing things here a bit
        */

        if(a==9)
        {
                //NO-OP
        }

        if(a==1)
                hh(++jj);
        else if(a==2)
                hh(jj++);
        else if(a==3)
                hh(*jj++);
        else
        {
                //NO-OP
        }

        struct player *p;
        p=(struct player *)malloc(sizeof(struct player));

        p->a=1;
        p->b=2.4;
        //#pragma omp parallel for
        for(a=1;a<=5;a++)
                static(a);

        int *p;
        float *j;
        p=&auto;
        j=0x1234;
        printf("*j=%p",(char *)j);

        void *static=&c;

        int a=5,b=8,c;
        c=a<=>b;

        hh(p);

        return(0);
}

int *hh(char *p)
{
        int n=7;

        scanf("%d",&n);

        if(n==0)
                printf("%d\n",n+1);
        else if(n==1)
                printf("%d\n",n+2);

        return(NULL);
}
```

Expected output

```
***parsing successful***
#global_declarations = 10
#function_definitions = 2
#integer_constants = 20
#pointers_declarations = 12
#ifs_without_else = 2
if-else max-depth = 3
```

Sample testcase #2

Input

```
int a, b, c;

int main(int *argc, char* argv[])
{
        int a, i;
        int b[10];
        for(i=1; i<=10; ++i) {
                b[i-1] = i*i;             // square
        }
        int *c = b;
        i = 0;
        int d= 0;
        while(i<10) {
                d += *(c + i);
        }
        printf("%d\n", d);        // print sum of values
        return 0;
}
```

Expected output

```
***parsing terminated*** [syntax error]
```