A Grammar for the C- Programming Language (Version F20)
September 3, 2020

1 Introduction

This is a grammar for the Fall 2020 semester's C- programming language. This language is very similar to C and has a lot of features in common with a real-world programming language. There are also some real differences between C and C-. For instance the declaration of procedure arguments, the loops that are available, what constitutes the body of a procedure, assignment is not a simple expression, array operators, etc. Also because of time limitations this language unfortunately does not have any heap related structures. It would be great to do a lot more, but we'll save for a second semester of compilers ☺. NOTE: this grammar is not a Bison grammar! You'll have to fix that as part of your assignment.

For the grammar that follows here are the types of the various elements by type font or symbol:

- **Keywords are in this type font.**
- **TOKEN CLASSES ARE IN THIS TYPE FONT.**
- *Nonterminals are in this type font.*
- The symbol ϵ means the empty string in a CS grammar sense.

1.1 Some Token Definitions

- letter = a | ... | z | A | ... | Z | _
- digit = 0 | ... | 9
- letdig = digit | letter
- **ID** = letter letdig*
- **NUMCONST** = digit+
- **CHARCONST** = is a representation for a single character by placing that character in **single quotes**. A backslash is an escape character. Any character preceded by a backslash is interpreted as that character. For example \x is the letter x, \' is a single quote, \\ is a single backslash. There are **only two exceptions** to this rule: \n is a newline character and \0 is the null character.
- **STRINGCONST** = any series of zero or more characters enclosed by **double quotes**. A backslash is an escape character. Any character preceded by a backslash is interpreted as that character without meaning to the string syntax. For example \x is the letter x, \" is a double quote, \' is a single quote, \\ is a single backslash. There are **only two exceptions** to this

rule: `\n` is a newline character and `\0` is the null character. The string constant can be an empty string: a string of length 0. All string constants are terminated by the first unescaped double quote. String constants **must be entirely contained on a single line**, that is, they contain no unescaped newlines!

- **White space** (a sequence of blanks and tabs) is ignored. Whitespace may be required to separate some tokens in order to get the scanner not to collapse them into one token. For example: "intx" is a single **ID** while "int x" is the type **int** followed by the **ID** x. The scanner, by its nature, is a greedy matcher.
- **Comments** are ignored by the scanner. Comments begin with `//` and run to the end of the line.
- All **keywords** are in lowercase. You need not worry about being case independent since not all lex/flex programs make that easy.

2 The Grammar

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funDeclaration*

---

4. *varDeclaration* → *typeSpecifier varDeclList* **;**
5. *scopedVarDeclaration* → **static** *typeSpecifier varDeclList* **;** | *typeSpecifier varDeclList* **;**
6. *varDeclList* → *varDeclList* **,** *varDeclInitialize* | *varDeclInitialize*
7. *varDeclInitialize* → *varDeclId* | *varDeclId* **:** *simpleExpression*
8. *varDeclId* → **ID** | **ID [NUMCONST]**
9. *typeSpecifier* → **int** | **bool** | **char**

---

10. *funDeclaration* → *typeSpecifier* **ID (** *params* **)** *statement* | **ID (** *params* **)** *statement*
11. *params* → *paramList* | ϵ
12. *paramList* → *paramList* **;** *paramTypeList* | *paramTypeList*
13. *paramTypeList* → *typeSpecifier paramIdList*
14. *paramIdList* → *paramIdList* **,** *paramId* | *paramId*

15. *paramId* → **ID** | **ID** [ ]

---

16. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*
17. *expressionStmt* → *expression* **;** | **;**
18. *compoundStmt* → **{** *localDeclarations statementList* **}**
19. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ
20. *statementList* → *statementList statement* | ϵ
21. *selectionStmt* → **if** (*simpleExpression*) *statement* | **if** (*simpleExpression*) *statement* **else** *statement*
22. *iterationStmt* → **while** (*simpleExpression*) *statement* | **for** (**ID in ID**) *statement*
23. *returnStmt* → **return ;** | **return** *expression* **;**
24. *breakStmt* → **break ;**

---

25. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* ∗= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*
26. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*
27. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*
28. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*
29. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*
30. *relop* → <= | < | > | >= | == | **!** =
31. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*
32. *sumop* → + | −
33. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*
34. *mulop* → ∗ | / | %
35. *unaryExpression* → *unaryop unaryExpression* | *factor*
36. *unaryop* → − | ∗ | **?**

37. *factor* → *immutable* | *mutable*
38. *mutable* → **ID** | *mutable* [*expression*]
39. *immutable* → (*expression*) | *call* | *constant*
40. *call* → **ID** (*args*)
41. *args* → *argList* | ϵ
42. *argList* → *argList* , *expression* | *expression*
43. *constant* → **NUMCONST** | **CHARCONST** | **STRINGCONST** | **true** | **false**

3 Semantic Notes

- The only numbers are **int**s.
- There is no conversion or coercion between types such as between **int**s and **bool**s or **bool**s and **int**s.
- There can only be one function with a given name. There is no function overloading. The function name space is the same as the variable name space so a function and a variable cannot have the same name in the same scope.
- The unary asterisk takes an array as an argument and returns the size of the array.
- The **STRINGCONST** token translates to a fixed size **char** array.
- The logical operators **and** and **or** are NOT short cutting. Although it is easy to do, we have plenty of other stuff to implement.
- In if statements the **else** is associated with the most recent **if**. The above grammar allows for ambiguous associations between **else** and **if**. In your assignment you will have to fix this.
- Expressions are evaluated in order consistent with operator associativity and precedence found in mathematics. Also, no reordering of operands is allowed.
- A char occupies the same space as an integer or bool. This is an artifact of the virtual machine.
- A string is a constant char array.
- Initialization of variables can only be done with expressions that are constant, that is, they are able to be evaluated to a constant at compile time. For this class, it is not necessary that you actually evaluate the constant expression at compile time. But you will have to keep track of whether the expression is constant. Type of variable and expression must match (see exception for char arrays below).

- Array assignment works. The source array is copied to the target array. If the target array is smaller the source array is trimmed. If the target array is larger all the source elements are copied and the remainder of the target is zero filled (this is false in bool, and end of string in char). There is hardware support for this.
- To be clear, assignment of a string (char array) to a char array works as if it is any other array assignment. It will not overrun the end of the lhs array. If it is too short it will pad the lhs array with null characters which are equivalent to zeroes.
- Passing of arrays is done by reference implemented as pointers. Functions cannot return an array, but they can modify the content of an array passed in.
- Array comparison works. There is hardware support for this.
- Assignments in expressions happen at the time the assignment operator is encountered in the order of evaluation. The value returned is value of the rhs of the assignment. Assignments include the $++$ and $--$ operator. That is, the $++$ and $--$ operator do NOT behave as in C or C++. NOTE: assignment does NOT occur in a *simpleExpression* without enclosing parens.
- The initializing a char array to a string behaves like an array assignment to the whole array.
- Initializing an array to a scalar constant fills the array with that constant.
- Function return type is specified in the function declaration, however, if no type is given to the function in the declaration then it is assumed the function does not return a value. To aid discussion of this case, the type of the return value is said to be void, even though there is no **void** keyword for the type specifier.
- Code that exits a procedure without a **return** returns a 0 for an function returning **int** and **false** for a function returning **bool** and a blank for a function returning **char**.
- All variables, functions must be declared before use.
- $?n$ generates a uniform random integer in the range 0 to $|n|-1$ with the sign of n attached to the result. ?5 is a random number in the range 0 to 4. ? –5 is a random number in the range 0 to -4. ?0 is undefined.
- $?x$ for array x gives a random element from the array x.
- The **for** loop is a simple loop for going through all the elements of an array. It takes two **ID**'s of the same type, the first of which is a scalar and the second is an array. It successively assigns each element of the array to the scalar. It opens a new scope at the keyword **for** and the first ID is declared in the new scope to be of the type of the array on the right of the **in**. Note the array is not an expression it is simply an array.

4 An Example of C- Code

```
// C-F20
char zev[10]:"corgis";
char yurt[20];
int x:42, y:666;

int ant(int bat, cat[]; bool dog, elk; int fox; char gnu)
{
    int goat, hog[100];

    gnu = 'W';
    goat = hog[2] = 3**cat;    // hog is 3 times the size of array passed to cat
    if (dog & elk | bat > cat[3] | !dog) dog = ! dog;
    else fox++;
    if (bat <= fox) {
        while (dog) {
            static int hog;          // hog in new scope

            hog = fox;
            dog = fred(fox++, cat)>666;
            if (hog>bat) break;
            else if (fox!=0) fox += 7;
        }
     }

    for (c in zev) outputc(c);
    if (zev > "dog") outputs("bark");
    yurt = zev;
    yurt[3] = ?zev;

    return (fox+bat*cat[bat])/-fox;
}

// note that functions are defined using a statement
int max(int a, b) if (a>b) return a; else return b;
```

Table 1: A table of all operators in the language to help with type checking and showing what operations are allowed. Note that C- supports = for all types of arrays. Initialization is listed as an operator for type checking purposes and could have been implemented as an operator but I chose not to for no good reason. But the type checking needs to work.

Operator	Arguments	Return Type
initialization	equal,string	N/A
initialization	equal	N/A
not	bool	bool
and	bool,bool	bool
or	bool,bool	bool
==	equal types + arrays	bool
! =	equal types + arrays	bool
<=	equal (int/char) + arrays	bool
<	equal (int/char) + arrays	bool
>=	equal (int/char) + arrays	bool
>	equal (int/char) + arrays	bool
=	equal types + arrays	type of lhs
+=	int,int	int
−=	int,int	int
*=	int,int	int
/=	int,int	int
−−	int	int
++	int	int
*	array	int
−	int	int
?	int	int
*	int,int	int
+	int,int	int
−	int,int	int
/	int,int	int
%	int,int	int
[ ]	array,int	type of lhs

Table 2: A table of just the array operators.

Operator	Arguments	Return Type
==	equal types + arrays	bool
! =	equal types + arrays	bool
<=	equal (int/char) + arrays	bool
<	equal (int/char) + arrays	bool
>=	equal (int/char) + arrays	bool
>	equal (int/char) + arrays	bool
=	equal types + arrays	type of lhs
*	any array	int
?	any array	type of array
[ ]	array,int	type of lhs