A Grammar for the C- Programming Language (Version S20)
January 21, 2020

1 Introduction

This is a grammar for the Spring 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 structured 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 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.

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

---

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

15. *paramIdList* → *paramIdList* , *paramId* | *paramId*

16. *paramId* → **ID** | **ID** [ ]

---

17. *statement* → *expressionStmt* | *compoundStmt* | *selectionStmt* | *iterationStmt* | *returnStmt* | *breakStmt*

18. *expressionStmt* → *expression* ; | ;

19. *compoundStmt* → { *localDeclarations statementList* }

20. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ

21. *statementList* → *statementList statement* | ϵ

22. *elsifList* → *elsifList* **elsif** *simpleExpression* **then** *statement* | ϵ

23. *selectionStmt* → **if** *simpleExpression* **then** *statement elsifList* | **if** *simpleExpression* **then** *statement elsifList* **else** *statement*

24. *iterationRange* → **ID** = *simpleExpression* .. *simpleExpression* | **ID** = *simpleExpression* .. *simpleExpression* : *simpleExpression*

25. *iterationStmt* → **while** *simpleExpression* **do** *statement* | **loop forever** *statement* | **loop** *iterationRange* **do** *statement*

26. *returnStmt* → **return ;** | **return** *expression* ;

27. *breakStmt* → **break ;**

---

28. *expression* → *mutable* = *expression* | *mutable* += *expression* | *mutable* −= *expression* | *mutable* *= *expression* | *mutable* /= *expression* | *mutable* ++ | *mutable* −− | *simpleExpression*

29. *simpleExpression* → *simpleExpression* **or** *andExpression* | *andExpression*

30. *andExpression* → *andExpression* **and** *unaryRelExpression* | *unaryRelExpression*

31. *unaryRelExpression* → **not** *unaryRelExpression* | *relExpression*

32. *relExpression* → *sumExpression relop sumExpression* | *sumExpression*

33. *relop* → <= | < | > | >= | == | !=

34. *sumExpression* → *sumExpression sumop mulExpression* | *mulExpression*

35. *sumop* → + | −

36. *mulExpression* → *mulExpression mulop unaryExpression* | *unaryExpression*
37. *mulop* → * | / | %
38. *unaryExpression* → *unaryop unaryExpression* | *factor*
39. *unaryop* → − | * | **?**
40. *factor* → *immutable* | *mutable*
41. *mutable* → **ID** | *mutable* [*expression*]
42. *immutable* → (*expression*) | *call* | *constant*
43. *call* → **ID** (*args*)
44. *args* → *argList* | **ϵ**
45. *argList* → *argList* , *expression* | *expression*
46. *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 unary asterisk is the only unary operator that takes an array as an argument. It takes an array 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**.
- **elsif** is treated as if it is an **else** containing the **if** test and all the immediately following **elsif**'s. The rule of matching the **else** is associated with the most recent **if** applies here as a result.
- **loop** with a range creates a new scope with the **ID** declared as a variable in that scope. The from, to, and by values for range are computed once before the loop begins and stored in non-visible variables related to the loop and stored in the scope of the loop.

- 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.
- A string is a constant char array.
- Initialization of variables can only be 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 only the elements in the target corresponding to the source elements change. Array comparison doesn't work natively. We just don't have time. Passing of arrays is done by reference. Functions cannot return an array, but they can modify the content of an array passed in.
- 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++.
- Assignment of a string (char array) to a char array. This simply assigns all of the chars in the rhs array into the lhs array. 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.
- The initializing a char array to a string behaves like an array assignment to the whole array.

  ~~The second initializing case for a char array is to initialize it to a char (not a char array). This will fill the array with copies of the given character. By the way, this is an illegal assignment.~~
- 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.

4 An Example of C- Code

```
char zev[10]:"corgis";
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 and elk or bat > cat[3] then dog = !dog;
    else fox++;
    if bat <= fox then {
        while dog do {
            static int hog;          // hog in new scope

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

     loop i=1..10:3 do {  // i is an int local to the loop
         if x==1 then cat[i] = bat;
         elsif (x==2) then cat[i] = bat%17;
         elsif (x==3) then cat[i] = 78;
         else x++;
     }

     loop forever if x>333 then break; else x++;

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

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

Table 1: A table of all operators in the language. Note that C- supports = for all types of arrays. It does not support relative testing: $\geq, \leq, >, <$ for any arrays.

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	bool
! =	equal types	bool
<=	int,int	bool
<	int,int	bool
>=	int,int	bool
>	int,int	bool
<=	char,char	bool
<	char,char	bool
>=	char,char	bool
>	char,char	bool
=	equal types incl. arrays	type of lhs
+=	int,int	int
−=	int,int	int
*=	int,int	int
/=	int,int	int
−−	int	int
++	int	int
*	any 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