A Grammar for the C- Programming Language (Version F16)
September 20, 2016

1 Introduction

This is a grammar for the Fall 2016 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 that, I guess we'll save for a second semester of compilers.

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 any 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.
- **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* | *recDeclaration*

---

4. *recDeclaration* → **record ID** { *localDeclarations* }

---

5. *varDeclaration* → *typeSpecifier varDeclList* ;
6. *scopedVarDeclaration* → *scopedTypeSpecifier varDeclList* ;
7. *varDeclList* → *varDeclList* , *varDeclInitialize* | *varDeclInitialize*
8. *varDeclInitialize* → *varDeclId* | *varDeclId* : *simpleExpression*
9. *varDeclId* → **ID** | **ID** [**NUMCONST**]
10. *scopedTypeSpecifier* → **static** *typeSpecifier* | *typeSpecifier*
11. *typeSpecifier* → *returnTypeSpecifier* | **RECTYPE**
12. *returnTypeSpecifier* → **int** | **bool** | **char**

---

13. *funDeclaration* → $\boxed{\textit{typeSpecifier}}$ **ID** (*params*) *statement* | **ID** (*params*) *statement*
14. *params* → *paramList* | ϵ
15. *paramList* → *paramList* ; *paramTypeList* | *paramTypeList*
16. *paramTypeList* → *typeSpecifier paramIdList*
17. *paramIdList* → *paramIdList* , *paramId* | *paramId*
18. *paramId* → **ID** | **ID** [ ]

---

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

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

21. *localDeclarations* → *localDeclarations scopedVarDeclaration* | ϵ

22. *statementList* → *statementList statement* | ϵ

23. *expressionStmt* → *expression* **;** | **;**

24. *selectionStmt* → **if** (*simpleExpression*) *statement* | **if** (*simpleExpression*) *statement* **else** *statement*

25. *iterationStmt* → **while** (*simpleExpression*) *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 term* | *term*

35. *sumop* → + | −

36. *term* → *term mulop unaryExpression* | *unaryExpression*

37. *mulop* → * | / | %

38. *unaryExpression* → *unaryop unaryExpression* | *factor*

39. *unaryop* → − | * | **?**

40. *factor* → *immutable* | *mutable*

41. *mutable* → **ID** | $\boxed{\textit{mutable}}$ [*expression*] | *mutable* **. ID**

42. *immutable* → (*expression*) | *call* | *constant*

43. *call* → **ID** (*args*)

44. *args* → *argList* | ϵ

45. *argList* → *argList* , *expression* | *expression*

46. *constant* → **NUMCONST** | **CHARCONST** | **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 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**.
- 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.
- 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 const. Type of variable and expression must match (see exception for char arrays below).
- array and record assignment works. Array and record comparison don't. We just don't have time. Passing of arrays and records are done by reference. Functions cannot return an array or record.
- 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++.

- 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, and record types must be declared before use.
- Record types are stored like arrays except indexing is done with an **ID**.
- Record types can contain items that are of record type but recursive record definition is not allowed.
- $?n$ 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

```
record point {
   int x, y;
}

record line {
   point x, y;
}

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

    point aPoint;
    line aLine;

    line two[2];

    aPoint.x = 666;
    aPoint.y = 667;

    aLine.x.x = 1;
    aLine.x.y = 2;
    aLine.y.x = 3;
    aLine.y.y = 4;

    two[0].x.x = 42;
    two[1].y.x = 43;

    gnu = hog[2] = 3**cat;    // hog is 3 times the size of array passed to cat
    if (dog and elk or bat > cat[3]) 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;
        }
     }
     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. Note that C- supports = for all types of arrays and records. It does not support relative testing: $\geq, \leq, >, <$ for any arrays or records. Array initialization can not happen for any arrays or records. Record access is done with the dot operator and an **ID** from the set of ids in the record definition. That is the record type has its own symbol table.

Operator	Arguments	Return Type
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	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