A Grammar for the C- Programming Language (Version F15) October 20, 2015

1 Introduction

This is a grammar for this 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$^+$
- **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 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!
- **CHARCONST** = is any representation for a single character in a string constant. All the rules for a STRINGCONST apply to a CHARCONST with the substitution of enclosing single quotes rather that double quotes and the length of the string is exactly one character.

- **White space** (a sequence of blanks and tabs) is ignored except that it must separate tokens such as **ID**'s, **NUMCONST**'s, and keywords.
- **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* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*

---

4. *var-declaration* → *type-specifier var-decl-list* **;**
5. *scoped-var-declaration* → *scoped-type-specifier var-decl-list* **;**
6. *var-decl-list* → *var-decl-list* **,** *var-decl-initialize* | *var-decl-initialize*
7. *var-decl-initialize* → *var-decl-id* | *var-decl-id* **:** *simple-expression*
8. *var-decl-id* → **ID** | **ID [NUMCONST]**
9. *scoped-type-specifier* → **static** *type-specifier* | *type-specifier*
10. *type-specifier* → **int** | **bool** | **char**

---

11. *fun-declaration* → *type-specifier* **ID (** *params* **)** *statement* | **ID (** *params* **)** *statement*
12. *params* → *param-list* | ϵ
13. *param-list* → *param-list* **;** *param-type-list* | *param-type-list*
14. *param-type-list* → *type-specifier param-id-list*
15. *param-id-list* → *param-id-list* **,** *param-id* | *param-id*
16. *param-id* → **ID** | **ID [ ]**

---

17. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt* | *break-stmt*

18. *compound-stmt* → **{** *local-declarations statement-list* **}**

19. *local-declarations* → *local-declarations scoped-var-declaration* | ϵ

20. *statement-list* → *statement-list statement* | ϵ

21. *expression-stmt* → *expression* **;** | **;**

22. *selection-stmt* → **if (** *simple-expression* **)** *statement* | **if (** *simple-expression* **)** *statement* **else** *statement*

23. *iteration-stmt* → **while (** *simple-expression* **)** *statement* | **foreach (** *mutable* **in** *simple-expression* **)** *statement*

24. *return-stmt* → **return ;** | **return** *expression* **;**

25. *break-stmt* → **break ;**

---

26. *expression* → *mutable* **=** *expression* | *mutable* **+=** *expression* | *mutable* **−=** *expression* | *mutable* **∗=** *expression* | *mutable* **/=** *expression* | *mutable* **++** | *mutable* **−−** | *simple-expression*

27. *simple-expression* → *simple-expression* **|** *and-expression* | *and-expression*

28. *and-expression* → *and-expression* **&** *unary-rel-expression* | *unary-rel-expression*

29. *unary-rel-expression* → **!** *unary-rel-expression* | *rel-expression*

30. *rel-expression* → *sum-expression relop sum-expression* | *sum-expression*

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

32. *sum-expression* → *sum-expression sumop term* | *term*

33. *sumop* → **+** | **−**

34. *term* → *term mulop unary-expression* | *unary-expression*

35. *mulop* → **∗** | **/** | **%**

36. *unary-expression* → *unaryop unary-expression* | *factor*

37. *unaryop* → **−** | **∗** | **?**

38. *factor* → *immutable* | *mutable*

39. *mutable* → **ID** | **ID [** *expression* **]**

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

41. *call* → **ID** **(** *args* **)**

42. *args* → *arg-list* | $\boldsymbol{\epsilon}$

43. *arg-list* → *arg-list* **,** *expression* | *expression*

44. *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 **|** are NOT short cutting[1].
- 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.
- 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 const. Type of variable and expression must match (see exception for char arrays below).
- 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 ++ operator. That is, the ++ operator does NOT behave as it does in C or C++.

[1]Although it is easy to do, we have plenty of other stuff to implement.

- 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.
- 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 must be declared before use and functions must be defined 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. $x[? * x]$ give a random element from the array x.

- The **foreach** loop construct has a *mutable* and a *simple-expression* in the parentheses. If the type of *simple-expression* is an array then the *mutable* must be the same type but a non-array and the foreach will loop through all the elements of the array putting each in turn into the mutable. If *simple-expression* is an not an array then then then the mutable must be of type int and it will be assigned the values from 0 to the size of the array minus one i.e. it goes through all the indexes of the array. Here is the meaning of the two types of foreach. Note the use of a new scope to hide local variables and compute the loop bounds once if y is an expression.

```
// THE MEANING OF: foreach (x in y) stuff;
// if y is a scalar means
{
    int p, limit;

    p = 0;
    limit = y;
    while (p<limit) {
        x = p;
        stuff;
        p++;
    }
}


//  THE MEANING OF: foreach (x in ya) stuff;
// if ya is an array it means
{
    int p, limit;

    p = 0;
    limit = *ya;          // this changes from above by *ya
    while (p<limit) {
        x = ya[p];        // this changes from above by indexing the array
        stuff;
        p++;
    }
}
```

4 An Example of C- Code

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

    gnu = hog[2] = 3**cat;    // hog is 3 times the size of array passed to cat
    if (dog & elk | 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 $==$, $! =$ and $=$ for all types of arrays. It does not support relative testing: $\geq, \leq, >, <$ for any arrays. Array initialization can only happen for the char type since the only array constant available in C- is char (a string).

Operator	Arguments	Return Type
initialization	equal,string	N/A
!	bool	bool
&	bool,bool	bool
|	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	same as lhs operand
$+=$	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