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 $\epsilon$ 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. $program \rightarrow declarationList$

2. $declarationList \rightarrow declarationList\ declaration \mid declaration$

3. $declaration \rightarrow varDeclaration \mid funDeclaration \mid recDeclaration$

4. $recDeclaration \rightarrow record\ ID\ \{\ localDeclarations\ \}$

5. $varDeclaration \rightarrow typeSpecifier\ varDeclList\ ;$

6. $scopedVarDeclaration \rightarrow scopedTypeSpecifier\ varDeclList\ ;$

7. $varDeclList \rightarrow varDeclList\ ,\ varDeclInitialize\mid varDeclInitialize$

8. $varDeclInitialize \rightarrow varDeclId\mid varDeclId:\ simpleExpression$

9. $varDeclId \rightarrow ID \mid ID\ [\ NUMCONST\ ]$

10. $scopedTypeSpecifier \rightarrow static\ typeSpecifier \mid typeSpecifier$

11. $typeSpecifier \rightarrow returnTypeSpecifier \mid RECTYPE$

12. $returnTypeSpecifier \rightarrow int \mid bool \mid char$

13. $funDeclaration \rightarrow [typeSpecifier] ID\ (\ params\ )\ statement \mid ID\ (\ params\ )\ statement$

14. $params \rightarrow paramList \mid \epsilon$

15. $paramList \rightarrow paramList\ ;\ paramTypeList \mid paramTypeList$

16. $paramTypeList \rightarrow typeSpecifier\ paramIdList$

17. $paramIdList \rightarrow paramIdList\ ,\ paramId \mid paramId$

18. $paramId \rightarrow ID \mid ID\ [\ ]$
19. \(\text{statement} \rightarrow \text{expressionStmt} \mid \text{compoundStmt} \mid \text{selectionStmt} \mid \text{iterationStmt} \mid \text{returnStmt} \mid \text{breakStmt}\)

20. \(\text{compoundStmt} \rightarrow \{ \text{localDeclarations} \ \text{statementList} \}\)

21. \(\text{localDeclarations} \rightarrow \text{localDeclarations} \ \text{scopedVarDeclaration} \mid \epsilon\)

22. \(\text{statementList} \rightarrow \text{statementList} \ \text{statement} \mid \epsilon\)

23. \(\text{expressionStmt} \rightarrow \text{expression} ; \mid ;\)

24. \(\text{selectionStmt} \rightarrow \text{if} (\ \text{simpleExpression} ) \ \text{statement} \mid \text{if} (\ \text{simpleExpression} ) \ \text{statement} \ \text{else} \ \text{statement}\)

25. \(\text{iterationStmt} \rightarrow \text{while} (\ \text{simpleExpression} ) \ \text{statement}\)

26. \(\text{returnStmt} \rightarrow \text{return} \ ; \mid \text{return} \ \text{expression} ;\)

27. \(\text{breakStmt} \rightarrow \text{break} ;\)

28. \(\text{expression} \rightarrow \text{mutable} = \text{expression} \mid \text{mutable} += \text{expression} \mid \text{mutable} -= \text{expression} \mid \text{mutable} *= \text{expression} \mid \text{mutable} /= \text{expression} \mid \text{mutable} ++ \mid \text{mutable} -- \mid \text{simpleExpression}\)

29. \(\text{simpleExpression} \rightarrow \text{simpleExpression} \ \text{or} \ \text{andExpression} \mid \text{andExpression}\)

30. \(\text{andExpression} \rightarrow \text{andExpression} \ \text{and} \ \text{unaryRelExpression} \mid \text{unaryRelExpression}\)

31. \(\text{unaryRelExpression} \rightarrow \text{not} \ \text{unaryRelExpression} \mid \text{relExpression}\)

32. \(\text{relExpression} \rightarrow \text{sumExpression} \ \text{relop} \ \text{sumExpression} \mid \text{sumExpression}\)

33. \(\text{relop} \rightarrow <\mid <=\mid >\mid >=\mid ==\mid !=\)

34. \(\text{sumExpression} \rightarrow \text{sumExpression} \ \text{sumop} \ \text{term} \mid \text{term}\)

35. \(\text{sumop} \rightarrow +\mid -\)

36. \(\text{term} \rightarrow \text{term} \ \text{mulop} \ \text{unaryExpression} \mid \text{unaryExpression}\)

37. \(\text{mulop} \rightarrow *\mid /\mid %\)

38. \(\text{unaryExpression} \rightarrow \text{unaryop} \ \text{unaryExpression} \mid \text{factor}\)

39. \(\text{unaryop} \rightarrow -\mid *\mid ?\)

40. \(\text{factor} \rightarrow \text{immutable} \mid \text{mutable}\)

41. \(\text{mutable} \rightarrow \text{ID} \mid \text{mutable} [\ \text{expression} ] \mid \text{mutable} . \text{ID}\)
42. immutable → ( expression ) | call | constant
43. call → ID ( args )
44. args → argList | €
45. argList → argList , expression | expression
46. constant → NUMCONST | CHARCONST | true | false

3 Semantic Notes

- The only numbers are ints.
- There is no conversion or coercion between types such as between ints and bools or bools and ints.
- 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arguments</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialization</td>
<td>equal</td>
<td>N/A</td>
</tr>
<tr>
<td>not and or</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>== !=</td>
<td>equal types</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;= &lt; &gt;</td>
<td>int,int</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;= &lt; &gt;</td>
<td>int,int</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;= &lt; &gt;</td>
<td>char,char</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;= &lt; &gt;</td>
<td>char,char</td>
<td>bool</td>
</tr>
<tr>
<td>==</td>
<td>equal types</td>
<td>type of lhs</td>
</tr>
<tr>
<td>+= -= *= /=</td>
<td>int,int</td>
<td>int</td>
</tr>
<tr>
<td>-= ++</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>*</td>
<td>any array</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>[]</td>
<td>array,int</td>
<td>type of lhs</td>
</tr>
</tbody>
</table>