A Grammar for the Compiler course project

Fall 2016

*Caution: First read this document completely, and then start the project!

1 Introduction

This is the grammar for the Fall 2016 semester's Compiler course project. 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 this language. For instance, the declaration of function arguments, the loops that are available, what constitutes the body of a function etc. Also because of time limitations this language unfortunately does not have any heap related structures.

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.

1.1 Some Token Definitions

- letter = a | ... | z | A | ... | Z
- digit = 0 | ... | 9
- letdig = digit | letter
- **ID** = start with a '#', then just two letter characters, and followed by just two digits. e.g. "#ab12", "#zz99". So its length must be exactly 5 characters, neither more nor less than five.
- **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** | **real** | **bool** | **char**
13. *funDeclaration* → *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* | **switch (** *simpleExpression* **)** *caseElement defaultElement* **end**

25. *caseElement* → **case NUMCONST :** *statement* **;** | *caseElement* **case NUMCONST :** *statement* **;**

26. *defaultElement* → **default :** *statement* **;** | ε

27. *iterationStmt* → **while (** *simpleExpression* **)** *statement*

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

29. *breakStmt* → **break ;**

30. *expression* → *mutable* **=** *expression* | *mutable* **+=** *expression* | *mutable* **-=** *expression* | *mutable* ***=** *expression* | *mutable* **/=** *expression* | *mutable* **++** | *mutable***--** | *simpleExpression*

31. *simpleExpression* → *simpleExpression* **or** *simpleExpression* | *simpleExpression* **and** *simpleExpression* | *simpleExpression* **or else** *simpleExpression* | *simpleExpression* **and then** *simpleExpression* | **not** *simpleExpression* | *relExpression*

32. *relExpression* → *mathlogicExpression relop mathlogicExpression* | *mathlogicExpression*

33. *relop* → **.le** | **.lt** | **.gt** | **.ge** | **.eq** | **.ne**

34. *mathlogicExpression* → *mathlogicExpression mathop mathlogicExpression* | *unaryExpression*

35. *mathop* → **+** | **-** | ***** | **/** | **%**

36. *unaryExpression* → *unaryop unaryExpression* | *factor*

37. *unaryop* → **-** | ***** | **?**

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

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

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

41. call → **ID (** *args* **)**

42. args → *argList* | ε

43. argList → *argList* **,** *expression* | *expression*

44. constant → **NUMCONST** | **REALCONST** | **CHARCONST** | **true** | **false**

3 Semantic Notes

- The numbers are **int**s or **real**s. Be sure to eliminate leading or trailing zeros.
- 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.
- 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.
- Note that **and then** and **or else** evaluate as short-circuit.
- Array and record assignment works. Array and record comparison do not. Passing of arrays and records are done by reference. Functions cannot return an array or record.
- 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.

- Note that this language supports = for all types of arrays and records. It does not support relative testing: .ge, .le, .gt, .lt for any arrays or records. Array initialization cannot 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.

4 An Example

```
record #po11 {
    int #xx11, #yy11;
}

record #li11 {
    #po11 #xx11, #yy11;
}

int #at11 (int #ba12, #ca23[]; bool #do43, #el32; int #fo12)
{
     int #gn11, #ho12[100];

     #po11 #aP11; #li11 #aL11;

     #li11 #tw33[2];

     #aP11.#xx11 = 666; #aP11.#yy11 = 667;

     #aL11.#xx11.#xx11 = 1; #aL11.#xx11.#yy11 = 2; #aL11.#yy11.#xx11 = 3;
     #aL11.#yy11.#yy11 = 4;

     #tw33[0].#xx11.#xx11 = 42;
     #tw33[1].#yy11.#xx11 = 43;

     #gn11 = #ho12[2] = 3** #ca23;    // hog is 3 times the size of array passed to cat
     if (#do43 and  #el32 or  #ba12 .gt  #ca23[3])  #do43  =  not #do43;
     else #fo12++;
     if (#ba12 .le  #fo12) {
          while (#do43) {
               static int #ho12;  // hog in new scope
                #ho12 = #fo12;
                #do43 = #fr77(#fo12++, #ca23) .lt 666;
                if (#ho12 .gt #ba12) break;
                else if (#fo12 .ne 0) #fo12 += 7;
          }
      }
      return (#fo12+#ba12 *#ca23 [#ba12])/- #fo12;
}
// note that functions are defined using a statement
int #ma11(int #aa11, #bb11)
if (#aa11>#bb11) return #aa11; else return #bb11;
```