3. Other tokens are **ID** and **NUM**, defined by the following regular expressions:

```
ID = letter letter*
NUM = digit digit*
letter = a|..|z|A|..|Z
digit = 0|..|9
```

   Lower- and uppercase letters are distinct.
4. White space consists of blanks, newlines, and tabs. White space is ignored except that it must separate **ID**'s, **NUM**'s, and keywords.
5. Comments are surrounded by the usual C notations `/*...*/`. Comments can be placed anywhere white space can appear (that is, comments cannot be placed within tokens) and may include more than one line. Comments may not be nested.

A.2 SYNTAX AND SEMANTICS OF C−

A BNF grammar for C− is as follows:

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID** `;` | *type-specifier* **ID** `[` **NUM** `]` `;`
5. *type-specifier* → `int` | `void`
6. *fun-declaration* → *type-specifier* **ID** `(` *params* `)` *compound-stmt*
7. *params* → *param-list* | `void`
8. *param-list* → *param-list* `,` *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID** `[` `]`
10. *compound-stmt* → `{` *local-declarations statement-list* `}`
11. *local-declarations* → *local-declarations var-declaration* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* `;` | `;`
15. *selection-stmt* → `if` `(` *expression* `)` *statement* | `if` `(` *expression* `)` *statement* `else` *statement*
16. *iteration-stmt* → `while` `(` *expression* `)` *statement*
17. *return-stmt* → `return` `;` | `return` *expression* `;`
18. *expression* → *var* `=` *expression* | *simple-expression*
19. *var* → **ID** | **ID** `[` *expression* `]`
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*
21. *relop* → `<=` | `<` | `>` | `>=` | `==` | `!=`
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → `+` | `-`
24. *term* → *term mulop factor* | *factor*
25. *mulop* → `*` | `/`
26. *factor* → `(` *expression* `)` | *var* | *call* | **NUM**

27. *call* → **ID** (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* , *expression* | *expression*

For each of these grammar rules we give a short explanation of the associated semantics.

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*

A program consists of a list (or sequence) of declarations, which may be function or variable declarations, in any order. There must be at least one declaration. Semantic restrictions are as follows (these do not occur in C). All variables and functions must be declared before they are used (this avoids backpatching references). The last declaration in a program must be a function declaration of the form `void main(void)`. Note that C− lacks prototypes, so that no distinction is made between declarations and definitions (as in C).

4. *var-declaration* → *type-specifier* **ID** ; | *type-specifier* **ID** [**NUM**] ;
5. *type-specifier* → `int` | `void`

A variable declaration declares either a simple variable of integer type or an array variable whose base type is integer, and whose indices range from 0 . . **NUM** − 1. Note that in C− the only basic types are integer and void. In a variable declaration, only the type specifier `int` can be used. `void` is for function declarations (see below). Note, also, that only one variable can be declared per declaration.

6. *fun-declaration* → *type-specifier* **ID** (*params*) *compound-stmt*
7. *params* → *param-list* | `void`
8. *param-list* → *param-list* , *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID** [ ]

A function declaration consists of a return type specifier, an identifier, and a comma-separated list of parameters inside parentheses, followed by a compound statement with the code for the function. If the return type of the function is `void`, then the function returns no value (i.e., is a procedure). Parameters of a function are either `void` (i.e., there are no parameters) or a list representing the function's parameters. Parameters followed by brackets are array parameters whose size can vary. Simple integer parameters are passed by value. Array parameters are passed by reference (i.e., as pointers) and must be matched by an array variable during a call. Note that there are no parameters of type "function." The parameters of a function have scope equal to the compound statement of the function declaration, and each invocation of a function has a separate set of parameters. Functions may be recursive (to the extent that declaration before use allows).

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

A compound statement consists of curly brackets surrounding a set of declarations and statements. A compound statement is executed by executing the statement