C-- Language Specification

Extended BNF Notation

In the lexical and syntax rules given below, BNF notation characters are written in green.

- Alternatives are separated by vertical bars: i.e., *'a | b'* stands for *"**a** **or** **b**"*.
- Square brackets indicate optionality: *'[a]'* stands for an optional *a*, i.e., *"a | epsilon"* (here, *epsilon* refers to the empty sequence).
- Curly braces indicate repetition: *'{ a }'* stands for *"epsilon | a | aa | aaa | ..."*

---

1. Lexical Rules

letter ::= a | b | ... | z | A | B | ... | Z

digit ::= 0 | 1 | ... | 9

id ::= *letter* { *letter* | *digit* | _ }

intcon ::= *digit* { *digit* }

charcon ::= '*ch*' | '\n' | '\0', where *ch* denotes any printable ASCII character, as specified by **isprint()**, other than \ (backslash) and ' (single quote).

stringcon ::= "{*ch*}", where *ch* denotes any printable ASCII character (as specified by **isprint()**) other than **"** (double quotes) and the newline character.

Comments Comments are as in C, i.e. a sequence of characters preceded by /* and followed by */, and not containing any occurrence of */.

---

2. Syntax Rules

Nonterminals are shown in italics; terminals are shown in boldface, and sometimes enclosed within quotes for clarity.

2.1 Grammar Productions

prog : { *dcl* **';'** | *func* }

dcl : *type var_decl* { **','** *var_decl* }
| [**extern**] *type* **id** **'('** *parm_types* **')'** { **','** **id** **'('** *parm_types* **')'** }
| [**extern**] **void** **id** **'('** *parm_types* **')'** { **','** **id** **'('** *parm_types* **')'** }

var_decl : **id** [**'['** **intcon** **']'**]

type : **char**
| **int**

parm_types : **void**
| *type* **id** [**'['** **']'**] { **','** *type* **id** [**'['** **']'**] }

func : *type* **id** **'('** *parm_types* **')'** **'{'** { *type var_decl* { **','** *var_decl* } **';'** } { *stmt* } **'}'**
| **void** **id** **'('** *parm_types* **')'** **'{'** { *type var_decl* { **','** *var_decl* } **';'** } { *stmt* } **'}'**

stmt : **if** **'('** *expr* **')'** *stmt* [**else** *stmt*]
| **while** **'('** *expr* **')'** *stmt*
| **for** **'('** [*assg*] **';'** [*expr*] **';'** [*assg*] **')'** *stmt*
| **return** [*expr*] **';'**
| *assg* **';'**
| **id** **'('** [*expr* { **','** *expr* }] **')'** **';'**
| **'{'** { *stmt* } **'}'**
| **';'**

assg : **id** [**'['** *expr* **']'**] = *expr*

expr : **'-'** *expr*
| **'!'** *expr*
| *expr binop expr*
| *expr relop expr*
| *expr logical_op expr*
| **id** [**'('** [*expr* { **','** *expr* }] **')'** | **'['** *expr* **']'**]
| **'('** *expr* **')'**
| **intcon**
| **charcon**
| **stringcon**

binop : **+**
| **–**
| *****
| **/**

relop : **==**
| **!=**
| **<=**
| **<**
| **>=**
| **>**

logical_op : **&&**
| **||**

2.2. Operator Associativities and Precedences

The following table gives the associativities of various operators and their relative precedences. An operator with a higher precedence binds "tighter" than one with lower precedence. Precedences decrease as we go down the table.

Operator	Associativity
!, – (unary)	right to left
*, /	left to right
+, – (binary)	left to right
<, <=, >, >=	left to right
==, !=	left to right
&&	left to right
||	left to right

---

3. Typing Rules

3.1. Declarations

The following rules guide the processing of declarations. Here, the *definition* of a function refers to the specification of its formals, locals, and its body.

1. An array must have non-negative size.
2. An identifier may be declared at most once as a global, and at most once as a local in any particular function; however, an identifier may appear as a local in many different functions.
3. A function may have at most one prototype; a function may be defined at most once.
4. If a function has a prototype, then the types of the formals at its definition must match (i.e., be the *same*), in number and order, the types of the argument in its prototype; and the type of the return value at its definition must match the type of the return value at its prototype.

   The prototype, if present, must precede the definition of the function.

5. An identifier can occur at most once in the list of formal parameters in a function definition.
6. The formal parameters of a function have scope local to that function.
7. If a function takes no parameters, its prototype must indicate this by using the keyword **void** in place of the formal parameters.
8. A function whose prototype is preceded by the keyword **extern** must not be defined in the program being processed.

3.2. Type Consistency Requirements

Variables must be declared before they are used. Functions must have their argument types and return value specified (either via a prototype or via a definition) before they are called. If an identifier is declared to have scope local to a function, then all uses of that identifier within that function refer to this local entity; if an identifier is not declared as local to a function, but is declared as a global, then any use of that identifier within that function refers to the entity with global scope. The following rules guide the checking of type consistency. The notion of two types being *compatible* is defined as follows:

1. **int** is compatible with **int**, and **char** is compatible with **char**;
2. **int** is compatible with **char**, and vice versa;
3. an array of **int** is compatible with an array of **int**, and an array of **char** is compatible with an array of **char**; and
4. any pair of types not covered by one of the rules given above is not compatible.

3.2.1. Function Definitions

1. Any function called from within an expression must not have return type **void**. Any function call that is a statement must have return type **void**.
2. A function whose return type is **void** cannot return a value, i.e., it cannot contain a statement of the form "**return** *expr*;"

   A function whose return type is not **void** cannot contain a statement of the form "**return**;" Such functions must contain at least one statement of the form "**return** *expr*;" (Note that it is still possible for such functions to fail to return a value by "falling off the end".)

3.2.2. Expressions

The type of an an expression *e* is given by the following:

1. If *e* is an integer constant, then its type is **int**.
2. If *e* is an identifier, then the type of *e* is the type of that identifier; if *e* is an array element, then the type of *e* is the type of the elements of that array.
3. If *e* is a function call, then the type of *e* is the return type for that function.
4. If *e* is an expression of the form *e1 + e2*, *e1 - e2*, *e1 * e2*, *e1 / e2*, or *-e1*, then the type of *e* is **int**.
5. If *e* is an expression of the form *e1 >= e2*, *e1 <= e2*, *e1 > e2*, *e1 < e2*, *e1 == e2*, or *e1 != e2* then the type of *e* is **bool**.
6. If *e* is an expression of the form *e1* **&&** *e2*, *e1* || *e2*, or *!e1*, then the type of *e* is **bool**.
7. A string constant has the type "array of **char**".

The rules for type checking expressions are given by the following:

1. The type of the index in an array reference must be compatible with **int**.
2. Each actual parameter of a function call must be compatible with the corresponding formal parameter.
3. The subexpressions associated with the operators +, -, *, /, <=, >=, <, >, ==, and != must be compatible with **int**.
4. The subexpressions associated with the operators **&&**, ||, and **!** must be of type **bool**.

3.3.3. Statements

1. Only variables of type **char** or **int**, or elements of arrays, can be assigned to; the type of the right hand side of an assignment must be compatible with the type of the left hand side of that assignment.
2. The type of the expression in a **return** statement in a function must be compatible with the return type of that function.
3. The type of the conditional in **if**, **for**, and **while** statements must have type **bool**.
4. Each actual parameter of a function call must be compatible with the corresponding formal parameter.

---

4. Operational Characteristics

The C-- language has the execution characteristics expected of a C-like block-structured language. The description below mentions only a few specific points that are likely to be of interest. For points not mentioned explicitly, you should consider the behavior of C-- to be as for C.

4.1. Data

4.1.1. Scalars

An object of type **int** occupies 32 bits; an object of type **char** occupies 8 bits.

Values of type **char** are considered to be signed quantities, and widening a **char** to an **int** requires sign extension.

String Constants

A string constant **"***a_1 ... a_n***"** is an array of characters containing *n*+1 elements, whose first *n* elements are the corresponding characters in the string, and whose last element is the NUL character **\0**.

Arrays

An arrays of size *n* consists of *n* elements, each occupying an amount of storage equal to that required for the type of the array element, laid out contiguously in memory.

4.2. Expressions

4.2.1. Order of Evaluation

- ***Arithmetic Expressions*** : Obviously, the operands of an expression have to be evaluated before the expression can be evaluated. When there is more than one operand, however, the order in which they are evaluated is left unspecified.
- ***Boolean Expressions*** : The order of evaluation of the operands of comparison operators (<=, >, <=, <, ==, !=) is left unspecified.

  Expressions involving the logical operators **&&** and || must be evaluated using short circuit evaluation.

4.2.2. Type Conversion

If an object of type **char** is part of an expression, its value is converted (sign extended) to a value of type **int** before the expression is evaluated.

Array Indexing

Arrays are zero-based, i.e., the elements of an array of *n* elements are indexed from 0 to *n*-1.

The result of indexing into an array with an out-of-range index is left unspecified.

4.3. Assignment Statements

Order of Evaluation

The order in which the left and right hand sides of an assignment are evaluated is left unspecified.

Type Conversion

A value of type **char** is converted (sign extended) to a 32-bit quantity before it is assigned to an object of type **int**.

A value of type **int** is converted (truncated) to an 8-bit quantity, by discarding the top 24 bits, before it is assigned to an object of type **char**.

4.4. Functions

4.4.1. Evaluation of Actuals

The order in which the actual parameters in a function call are evaluated is unspecified.

4.4.2. Parameter Passing

Scalar values are passed by value, while arrays (and string constants, which are represented as arrays of characters) are passed by reference.

An object of type **char** is converted (sign extended) to a 32-bit quantity before it is passed as an actual parameter to a function.

Since a function that has a formal parameter of type **char** will, in any case, be passed a 32-bit quantity as an actual, it must convert (truncate) the actual to an 8-bit quantity before using it.

4.4.3. Return from a Function

Execution returns from a function if either an explicit **return** statement is executed, or if execution "falls off" the end of the function body. In the latter case, no value is returned.

4.5. Program Execution

Execution begins at a procedure named **main()**.