Identifiers

In CPlus, identifiers are combinations of characters, numbers, and the special character ‘_’; they must begin with a letter, and they are case-sensitive. There are two types of identifiers:

1. **Value and Function Identifiers**: these must begin with a lower-case letter; they represent variable values of all datatypes as well as function names. Defined formally as
   
   \[ ['a'..'z'] ['a'..'z'] ['A'..'Z'] ['0'..'9'] '_'] *

2. **Struct Identifiers**: these must begin with a capital letter; they represent globally-defined structure types, used to group several datatypes into a single record. Defined formally
   
   \[ ['A'..'Z'] ['a'..'z'] ['A'..'Z'] ['0'..'9'] '_'] *

Keywords

The reserved keywords in CPlus include types, control flow indicators, and built-in functions:

<table>
<thead>
<tr>
<th>if</th>
<th>else</th>
<th>for</th>
<th>while</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>char</td>
<td>size_t</td>
<td>string</td>
<td>char</td>
</tr>
<tr>
<td>bool</td>
<td>void</td>
<td>true</td>
<td>false</td>
<td>sizeof</td>
</tr>
<tr>
<td>struct</td>
<td>NULL</td>
<td>printf</td>
<td>atoi</td>
<td>strdup</td>
</tr>
<tr>
<td>printb</td>
<td>print</td>
<td>printbig</td>
<td>malloc</td>
<td>free</td>
</tr>
</tbody>
</table>

Literals

Literals are the primitive values which represent the core data of any language, in this language there are only five

1. **Integer Literals**: Any combination of one or more numerals - ['0'..'9']+
2. **Character Literals**: Any ASCII Character, including all specials - ['\x00'..'\7F']
3. **Boolean Literals**: Either of true or false.
4. **String Literals**: Any combination of characters found between “” marks; with supported escape sequences:
   
   \ - escape character
   \b – backspace character
Separators:

Used to help build the abstract syntax tree and determine semantics, discarded after parsing; in this language we use:

```
‘(’ – LPAREN
‘)’ – RPAREN
 ‘{’ – LBRACE
 ‘}’ – RBRACE
 ‘[’ – LSQUARE
 ‘]’ – RSQUARE
 ‘,’ – SEMI
 ‘,’ – COMMA
 ‘/*’ – open comment
 ‘*/’ – close comment
```

Operators

There are three categories of operators in CPlus, which dictate all the mathematical, memory-access, and other logical functionality of the language:

(1) **Binary Operators**: these operate on two expressions, and the order of which expression is on the left or right is significant

- `+` addition of two signed 32-bit integers – returns 32-bit int
- `-` subtraction of two signed 32-bit integers – returns 32-bit int
- `*` multiplication of two signed 32-bit integers – returns 32-bit int
- `/` division of two signed 32-bit integers – returns 32-bit int
- `%` modulus of two signed 32-bit integers – returns 32-bit int
- `==` physical equality of two primitive literals of same type – returns bool
- `!=` physical inequality of two primitive literals of same type – returns bool
- `<` less than of two 32-bit integers – returns bool
- `<=` less-than-equal-to of two 32-bit integers – returns bool
- `>` greater than of two 32-bit integers – returns bool
- `>=` greater-than-equal-to of two 32-bit integers – returns bool
- `&&` logical AND of two bools – returns bool
- `||` logical OR of two bools – returns bool

(2) **Unary Operators**: these operate on a single expression, and significance is placed on whether the operator comes before (prefix) or after (postfix) the expression

- `-` Applied before integer, makes it negative (cannot be chained together)
- `!` Applied after bool, inverts its value
- `++` Applied before or after integer, increases its value by one
- `--` Applied before or after integer, decrements its value by one
* Applied before an identifier, retrieves its value from memory
& Applied before an identifier, returns its memory address
-> Applied after struct identifier, retrieve struct from mem and get member
. Applied after struct identifier, get member

(3) Assignment Operators: these operate on two expressions, the lefthand of which must evaluate to an lvalue and the righthand of which must evaluate to an rvalue.
We have only implemented two for our simple purposes
= store rvalue at the address lvalue
% = lvalue must point to an integer, performs ‘m[lvalue] = m[lvalue] % rvalue’

Operator Precedence & Associativity

In decreasing order from top to bottom:

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>ASSOCIATIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>* &amp; SIZEOF ()</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>() [] . - &gt; ++ -- (POSTFIX)</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>! ++ -- - (PREFIX)</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>() (CAST)</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>* / %</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>+ -</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>== !=</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>= %=</td>
<td>Right-to-left</td>
</tr>
</tbody>
</table>

Data Types

The data, this is the good stuff. What can we store, manipulate, understand? There are 8 datatypes in CPlus, six of which are primitive, and two of which are complex

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>Size_t</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>Bool</td>
<td>Boolean values true or false</td>
</tr>
<tr>
<td>Char</td>
<td>any of the ascii values described as charlit above</td>
</tr>
<tr>
<td>String</td>
<td>any string between two &quot;&quot; quotes described above</td>
</tr>
<tr>
<td>Void</td>
<td>The type we return when we don’t want to return anything</td>
</tr>
<tr>
<td>Struct(ID)</td>
<td>A struct with the name ID, cannot exist without ID</td>
</tr>
<tr>
<td>Pointer(typ)</td>
<td>A pointer to any of the above datatypes</td>
</tr>
</tbody>
</table>
It is worth noting that we have implemented arrays in CPlus, but have done so using pointers under the hood, instead of defining an explicit array type.

Declarations

CPlus requires that – inside of a function – all declarations (even with initializers) for any variable used later on in that function must take place first, before any operations. Declarations are explicitly defined in the Context Free Grammar section, and are summarized below:

1. **Literal Variables**: Datatype followed by Value Identifier. Cannot directly declare a void variable. Legal variable declarations look like this:

   ```
   int a; bool b; char c; string d; Node n; Edge e;
   ```

2. **Functions**: Datatype followed by Function Identifier followed by LBRACE function-body RBRACE. Datatype can be either primitive or complex. Legal function declarations look like this:

   ```
   int x { ... }
   bool* { ... }
   Node** { ... } – where Node was previously defined as a struct
   ```

3. **Structures**: `struct` keyword followed by a Struct Identifier then LBRACE struct-body RBRACE SEMI. Note that the struct body can only contain un-initialized Literal Variable declarations. Legal struct declarations look like this:

   ```
   struct Node {
       int x;
       char* y;
       Edge z; /* previously declared struct Edge */
   };
   ```

4. **Pointers**: Datatype followed by * followed by Value Identifier. Can chain multiple * together in a single declaration. Legal pointer declarations look like this:

   ```
   int* x;
   bool** y;
   Node* np;
   ```
(5) Arrays: Arrays are not explicit datatypes, and they resolve to pointers when referenced. They do, however, have their own declarations as syntactic sugar. Datatype – possibly complex – followed by Value Identifier the RSQUARE integer-literal LSQUARE. The integer-literal must be greater than 0 and is not optional. Variable length arrays are not supported. Legal array declarations look like this:

```c
int a[10];
bool* b[10];
Node* nodeIndices[100];
```

(6) Initialization: With the exception of structs and arrays, all of the above declarations can optionally come paired with initializers. Furthermore, declarations of the same type can be chained together in a list through commas. Memory allocation is also possible in the initialization of a declaration. Legal initializations for declarations look like this:

```c
int x = 5;
int y = (300 * 21 + 4 ) / 2;
int*x = (int*) malloc(20*sizeof(int)); /* defacto 20-int array */
Node* nodes = (Node*) malloc(sizeof(Node));
int a = 5 , b = 6, q = x + 1;
```

Scope Rules

Standard static scoping rules apply, i.e. it is determined spatially based on position within the code at compile time.

Context-Free Grammar

**TERMINALS**: in uppercase bold

```
EOF Epsilon ID LPAREN RPAREN
LBRACE RBRACE COMMA STRUCT STRUCT_ID
SEMI INT SIZE_T BOOL STRING
CHAR VOID TIMES LSQUARE RSQUARE
ASSIGN RETURN IF ELSE FOR
WHILE MOD_ASSIGN OR AND EQ
NEQ LT GT GEQ LEQ
NOT MINUS_ASSIGN INC DEC DOT
ARROW PRINTF PRINT PRINTB PRINTBIG
ALLOC FREE STRDUP ATOI PLUS
TIMES DIVIDE MOD LITERAL STRINGLIT
CHARLIT TRUE FALSE AMP SIZEOF
NULL
```
nonterminals: in lowercase italics

<table>
<thead>
<tr>
<th>program</th>
<th>decls</th>
<th>declaration</th>
<th>func_decl</th>
<th>struct_decl</th>
</tr>
</thead>
<tbody>
<tr>
<td>typ</td>
<td>formals_opt</td>
<td>formal_list</td>
<td>declaration_list</td>
<td>stmt_list</td>
</tr>
<tr>
<td>init_declarator_list</td>
<td>expr</td>
<td>add_expr</td>
<td>init_declarator</td>
<td>stmt</td>
</tr>
<tr>
<td>selection_statement</td>
<td>iteration_statement</td>
<td>expr_opt</td>
<td>assignment_expression</td>
<td>logical_or_expr</td>
</tr>
<tr>
<td>postfix_expr</td>
<td>logical_and_expr</td>
<td>equality_expr</td>
<td>relational_expr</td>
<td>add_expr</td>
</tr>
<tr>
<td>unary_expr</td>
<td>cast_expr</td>
<td>unary_operator</td>
<td>postfix_expr</td>
<td>built_in_expr</td>
</tr>
<tr>
<td>actuals_list_opt</td>
<td>primary_expr</td>
<td>actuals_list</td>
<td>mult_expr</td>
<td></td>
</tr>
</tbody>
</table>

Productions:

\[
\text{program} \rightarrow \\
\quad \text{decls EOF}
\]

\[
\text{decls} \rightarrow \\
\quad \text{decls declaration} \\
\quad | \text{decls func}_{\text{decl}} \\
\quad | \text{decls struct}_{\text{decl}} \\
\quad | \text{Epsilon}
\]

\[
\text{func}_{\text{decl}} \rightarrow \\
\quad \text{typ ID LPAREN formals}\_\text{opt RPAREN LBRACE declaration_list stmt_list RBRACE}
\]

\[
\text{formals}\_\text{opt} \rightarrow \\
\quad \text{Epsilon} \\
\quad | \text{formal_list}
\]

\[
\text{formal_list} \rightarrow \\
\quad \text{typ ID} \\
\quad | \text{formal_list COMMA typ ID}
\]

\[
\text{struct}_{\text{decl}} \rightarrow \\
\quad \text{STRUCT STRUCT}\_\text{ID LBRACE declaration_list RBRACE SEMI}
\]

\[
\text{typ} \rightarrow \\
\quad \text{INT} \\
\quad | \text{SIZE}_{\text{T}} \\
\quad | \text{STRING} \\
\quad | \text{BOOL} \\
\quad | \text{CHAR}
\]
VOID
| STRUCT_ID
| typ TIMES

declaration_list →
  Epsilon
  | declaration_list declaration

declaration →
  typ init_declarator_list SEMI

init_declarator →
  ID
  | ID LSQUARE add_expr RSQUARE
  | ID ASSIGN expr

init_declarator_list →
  init_declarator
  | init_declarator_list COMMA init_declarator

stmt_list →
  Epsilon
  | stmt_list stmt

stmt →
  expr SEMI
  | selection_stmt
  | iteration_stmt
  | RETURN SEMI
  | RETURN expr SEMI
  | LBRACE stmt_list RBRACE

selection_stmt →
  IF LPAREN expr RPAREN stmt (%prec NOELSE)
  | IF LPAREN expr RPAREN stmt ELSE stmt

iteration_stmt →
  FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  | WHILE LPAREN expr RPAREN stmt

expr_opt →
  Epsilon
  | expr

expr →
  assignment_expr

assignment_operator →
  ASSIGN
  | MOD_ASSIGN

assignment_expr →
logical_or_expr | postfix_expr assignment_operator expr

logical_or_expr ->
    logical_and_expr
    | logical_or_expr OR logical_and_expr

logical_and_expr ->
    equality_expr
    | logical_and_expr AND equality_expr

equality_expr ->
    relational_expr
    | equality_expr EQ relational_expr
    | equality_expr NEQ relational_expr

relational_expr ->
    add_expr
    | relational_expr LT add_expr
    | relational_expr GT add_expr
    | relational_expr LEQ add_expr
    | relational_expr GEQ add_expr

cast_expr ->
    unary_expr
    | LPAREN typ RPAREN cast_expr

unary_operator ->
    NOT
    | MINUS

unary_expr ->
    postfix_expr
    | unary_operator postfix_expr
    | INC postfix_expr
    | DEC postfix_expr

postfix_expr ->
    built_in_expr
    | postfix_expr INC
    | postfix_expr DEC
    | postfix_expr LPAREN actuals_list_opt RPAREN
    | postfix_expr LSQUARE postfix_expr RSQUARE
    | postfix_expr DOT ID
    | postfix_expr ARROW ID

built_in_expr ->
    primary_expr
    |PRINTF LPAREN actuals_list_opt RPAREN
    |PRINT LPAREN actuals_list_opt RPAREN
    |PRINTB LPAREN actuals_list_opt RPAREN
    |PRINTBIG LPAREN actuals_list_opt RPAREN
    |MALLOC LPAREN actuals_list_opt RPAREN
act\als_list_opt \rightarrow 
\quad \text{Epsilon} \\
\quad \text{act}\text{uals_list}

act\text{uals_list} \rightarrow 
\quad \text{expr} \\
\quad \text{act}\text{uals_list COMMA expr}

add\_expr \rightarrow 
\quad \text{mult\_expr} \\
\quad \text{add\_expr PLUS mult\_expr} \\
\quad \text{add\_expr MINUS mult\_expr}

mult\_expr \rightarrow 
\quad \text{cast\_expr} \\
\quad \text{mult\_expr TIMES cast\_expr} \\
\quad \text{mult\_expr DIVIDE cast\_expr} \\
\quad \text{mult\_expr MOD cast\_expr}

primary\_expr \rightarrow 
\quad \text{LPAREN expr RPAREN} \\
\quad \text{LITERAL} \\
\quad \text{STRINGLIT} \\
\quad \text{CHARLIT} \\
\quad \text{TRUE} \\
\quad \text{FALSE} \\
\quad \text{ID} \\
\quad \text{AMP primary\_expr} \\
\quad \text{TIMES primary\_expr} \\
\quad \text{SIZEOF LPAREN typ RPAREN} \\
\quad \text{NULL}