

# **Compiler Construction**

## Compiler Project: Specification Document

### **Team Members:**

Huzefa Saifuddin, 22K-5125  
Baasim Ahmed, 22K-5029

### **Section:**

BCS-7E

## Lexical Rules

letter	::= a   b   ...   z   A   B   ...   Z
digit	::= 0   1   ...   9
id	::= letter { letter   digit   _ }
intcon	::= digit { digit }
realcon	::= intcon.intcon
charcon	::= 'ch'   '\n'   '\0', where ch denotes any printable ASCII character, as specified by the C function isprint(), except for \ (backslash) and ' (apostrophe).
stringcon	::= "{ch}", where ch denotes any printable ASCII character, as specified by the C function isprint(), except for " (quotes) and the newline character.
comment	Comments are like in C, i.e., a sequence of characters preceded by /* and followed by */, which contains no occurrence of */.

# Syntactic Rules

## Grammar Production Rules

```
prog      ::= { decl ';' | func }
decl     ::= type decl_var { ',' decl_var }
           | type id '(' param_types ')' '{' { type decl_var { ';' decl_var } }
           | { cmd } '}'
           | void id '(' param_types ')' '{' { type decl_var { ';' decl_var } }
           | { cmd } '}'
decl_var ::= id [ '[' intcon ']' ]
type      ::= char
           | int
           | float
           | bool
param_types ::= type (id | &id | id '[' ']') { ',' type (id | &id | id '[' ']') }
func      ::= type id '(' param_types ')' '{' { type decl_var { ';' decl_var } }
           | { cmd } '}'
           | void id '(' param_types ')' '{' { type decl_var { ';' decl_var } }
           | { cmd } '}'
cmd       ::= if '(' expr ')' cmd [ else cmd ]
           | while '(' expr ')' cmd
           | for '(' [ atrib ] ';' [ expr ] ';' [ atrib ] ')' cmd
           | return [ expr ] ';'
           | atrib ';'
           | id '(' [ expr { ',' expr } ] ')' ';'
           | '{' { cmd } '}'
           | ';'
atrib    ::= id [ '[' expr ']' ] = expr
expr     ::= expr_simp [ op_rel expr_simp ]
expr_simp ::= [ + | - ] termo { ( + | - || ) termo }
termo   ::= factor { (* | / | &&) factor }
factor  ::= id [ '[' expr ']' ] | intcon | realcon | charcon |
           id '(' [ expr { ',' expr } ] ')' | '(' expr ')' | '!' factor
op_rel  ::= ==
           | !=
           | <=
           | <
           | >=
           | >
```

## Associativity and Operator Precedence

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

# Attributed Grammar

## General Definitions and Helper Functions

Attribute Type	Description
<b>Synthesized (syn)</b>	Passed up the parse tree (e.g., .type, .node).
<b>Inherited (inh)</b>	Passed down the parse tree (e.g., expected return type, current scope).

Helper Function	Purpose
<b>lookup(id)</b>	Retrieves the symbol table entry for an identifier.
<b>insert(id, type, scope)</b>	Adds an identifier (variable/function) to the symbol table.
<b>new Node(...)</b>	Creates a new node in the Abstract Syntax Tree (AST).
<b>check(condition)</b>	If the condition is false, reports a compile-time error.
<b>enter_scope() / exit_scope()</b>	Manages the nesting of symbol table scopes for functions/blocks.

# Core Structure and Declarations

## Type Definitions

Production	Semantic Rules / Actions
type ::= char	type.val = char
type ::= int	type.val = int
type ::= float	type.val = float
type ::= bool	type.val = bool

## Variable Declaration Details

Production	Semantic Rules / Actions
decl_var ::= id	decl_var.name = id.lexval decl_var.isArray = false
decl_var ::= id '[' intcon ']'  decl ::= type decl_var { ',' decl_var2 }	decl_var.name = id.lexval decl_var.isArray = true decl_var.size = intcon.val  For each decl_var in list:  check(!lookup(decl_var.name)) (Check for redefinition)  insert(decl_var.name, type.val, decl_var.size)

<b>Production</b>	<b>Semantic Rules / Actions</b>
	decl.node = new VarDeclNode(type.val, list_of_vars)

## Function Definitions

<b>Production</b>	<b>Semantic Rules / Actions</b>
decl ::= type id '(' param_types ')' '{' ... '}'	<p>insert(id.lexval, type.val, function)</p> <p>enter_scope()</p> <p>Process param_types (add to symbol table)</p> <p>Process cmd (body)</p> <p>exit_scope()</p> <p>check(cmd.hasReturn == true)</p>

## Commands (Statements)

<b>Production</b>	<b>Semantic Rules / Actions</b>
cmd ::= attrib ;'	cmd.node = attrib.node
cmd ::= if '(' expr ')' cmd1 [ else cmd2 ]	<p>check(expr.type == bool)</p> <p>cmd.node = new IfNode(expr.node, cmd1.node, cmd2.node)</p>
cmd ::= while '(' expr ')' cmd1	check(expr.type == bool)

Production	Semantic Rules / Actions
	cmd.node = new WhileNode(expr.node, cmd1.node)
cmd ::= return [ expr ] ;'	<p>check(expr.type == current_function.returnType)</p> <p>cmd.node = new ReturnNode(expr.node)</p>
attrib ::= id [ '[' expr ']' ] = expr2	<p><b>Scalar Assignment:</b> (id = expr2)</p> <p>var = lookup(id.lexval)</p> <p>check(var.isArray == false)</p> <p>check(var.type == expr2.type)</p> <p>attrib.node = new AssignNode(id, expr2.node)</p> <p><b>Array Assignment:</b> (id[expr] = expr2)</p> <p>var = lookup(id.lexval)</p> <p>check(var.isArray == true)</p> <p>check(expr.type == int) (Index must be int)</p> <p>check(var.type == expr2.type)</p> <p>attrib.node = new ArrayAssignNode(id, expr.node, expr2.node)</p>

# Expressions

## Relational Operators

Production	Semantic Rules / Actions
expr ::= expr_simp	expr.type = expr_simp.type expr.node = expr_simp.node
expr ::= expr_simp1 op_rel expr_simp2	check(expr_simp1.type == expr_simp2.type)  expr.type = bool  expr.node = new OpNode(op_rel.op, expr_simp1.node, expr_simp2.node)

## Additive & Logical OR Operations (expr\_simp)

Production	Semantic Rules / Actions
expr_simp ::= [+   -] termo	(Unary plus/minus)  check(termo.type == int    termo.type == float)  expr_simp.type = termo.type  expr_simp.node = new UnaryOpNode(op, termo.node)
expr_simp ::= expr_simp1 (+   -) termo	check(expr_simp1.type == termo.type)  check(type is numeric)

Production	Semantic Rules / Actions
	expr_simp.type = expr_simp1.type  expr_simp.node = new OpNode(op, expr_simp1.node, termo.node)
`expr_simp ::= expr_simp1 '	

### Multiplicative & Logical AND Operations (termo)

Production	Semantic Rules / Actions
termo ::= factor	termo.type = factor.type  termo.node = factor.node
termo ::= termo1 (*   /) factor	check(termo1.type == factor.type)  check(type is numeric)  termo.type = termo1.type  termo.node = new OpNode(op, termo1.node, factor.node)
termo ::= termo1 '&&' factor	check(termo1.type == bool \text{ \&\& } factor.type == bool)  termo.type = bool  termo.node = new OpNode('&&', termo1.node, factor.node)

## Base Values (factor)

Production	Semantic Rules / Actions
factor ::= id	<pre>var = lookup(id.lexval) factor.type = var.type factor.node = new IdNode(var)</pre>
factor ::= id '[' expr ']'	<pre>var = lookup(id.lexval) check(var.isArray == true) check(expr.type == int) factor.type = var.type factor.node = new ArrayAccessNode(var, expr.node)</pre>
factor ::= intcon	<pre>factor.type = int factor.node = new IntNode(intcon.val)</pre>
factor ::= realcon	<pre>factor.type = float factor.node = new FloatNode(realcon.val)</pre>
factor ::= charcon	<pre>factor.type = char factor.node = new CharNode(charcon.val)</pre>

<b>Production</b>	<b>Semantic Rules / Actions</b>
factor ::= '(' expr ')'	<p>factor.type = expr.type</p> <p>factor.node = expr.node</p>
factor ::= '!' factor1	<p>check(factor1.type == bool)</p> <p>factor.type = bool</p> <p>factor.node = new OpNode('!', factor1.node)</p>
factor ::= id '(' [args] ')'	<p>(Function Call)</p> <p>func = lookup(id.lexval)</p> <p>check_args_match(func.params, args)</p> <p>factor.type = func.returnType</p> <p>factor.node = new CallNode(func, args.list)</p>

## Operators (op\_rel)

<b>Production</b>	<b>Semantic Rules / Actions</b>
op_rel ::= '=='	op_rel.op = EQ
op_rel ::= '!='	op_rel.op = NEQ
op_rel ::= '<'	op_rel.op = LT
op_rel ::= '>'	op_rel.op = GT
op_rel ::= '<='	op_rel.op = LTE

<b>Production</b>	<b>Semantic Rules / Actions</b>
op_rel ::= '>='	op_rel.op = GTE