

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
Synthesized (syn)	Passed up the parse tree (e.g., .type, .node).
Inherited (inh)	Passed down the parse tree (e.g., expected return type, current scope).

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

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

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

Function Definitions

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

Commands (Statements)

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

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

Expressions

Relational Operators

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

Additive & Logical OR Operations (expr_simp)

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

Production	Semantic Rules / Actions
	$\text{expr_simp.type} = \text{expr_simp1.type}$ $\text{expr_simp.node} = \text{new OpNode}(\text{op}, \text{expr_simp1.node}, \text{termo.node})$
$\text{expr_simp} ::= \text{expr_simp1}$	

Multiplicative & Logical AND Operations (termo)

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

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>

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

Operators (op_rel)

Production	Semantic Rules / Actions
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

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