CS325 Compiler Design: Report

1. Grammar Transformation:

The original MiniC grammar required three transformations. Left recursion in productions like “extern_list ::= extern_list extern” causes infinite recursion because the function calls itself before consuming input. This was eliminated using the standard transformation to right recursion; the resulting tail-recursive A’ productions were implemented as while loops rather than recursive calls for efficiency. Ambiguity in rval (where “2+3*4” had multiple valid parse trees) was resolved by introducing a precedence hierarchy: or_expr calls and_expr, which calls eq_expr, and so on down to primary_expr, ensuring higher-precedence operators bind more tightly. A FIRST-FIRST conflict between var_decl and fun_decl was resolved through left factoring: the common prefix (type identifier) is parsed first, then lookahead on the next token distinguishes alternatives – semicolon for variables, opening brackets for arrays, opening parenthesis for functions.

2. FIRST and FOLLOW Sets:

FIRST sets determine which production to apply: in ParseStmt(), seeing “if” triggers if-statement parsing, “while” triggers while-loop parsing, and tokens in FIRST(expr) trigger expression-statement parsing. FOLLOW sets handle ε-productions: when parsing or_expr’ ::= “||” and_expr or_expr’ | ε, if the current token is not “||” (not in FIRST minus ε), it must be in FOLLOW(or_expr’), so we return without consuming input. The LL(1) verification in the Appendix confirms FIRST and FOLLOW sets are disjoint for all ε-productions, guaranteeing single token lookahead suffices, making the parser linear-time and deterministic.

3. Parser Design Decisions:

C defines most binary operators as left associative, meaning “a – b – c “ must parse as “(a – b) – c”. The transformed grammar’s prime productions achieve this when implemented as while loop as each iteration combines the accumulated left-hand side with a new right operand, building the tree leftward. Assignment is right-associative meaning “a = b = c” evaluates as “a = (b = c)” which enabling chained assignments to work. This requires direct recursion rather than iteration: ParseExpr() calls itself for the right-hand side, building the rightmost assignment first. Three error functions (LogError, LogErrorP, LogErrorV) were necessary because C++ requires matching return types. Parsing functions return different pointer types (ASTnode*, FunctionPrototypeAST*, Value*), so each error function prints an error message with location information and terminates compilation immediately (as mentioned in the coursework brief, error propagation is not required).

4. AST Storage and Display:

Initially, AST nodes were printed immediately and then discarded, making them unavailable for code generation. The solution stores nodes in global vectors (ProgramAST and ExternAST) using std::move(), with PrintAST() traversing them after parsing completes. During development, fprintf debugging statements (e.g. “Parsed a function declaration”) were inserted at key points to verify each grammar production was being recognised correctly; these were kept ensuring changes to other parts of the code didn’t affect the parsing for different tests. The to_string() methods evolved from flat strings to tree-style output using the inherited attributes “prefix” and “isLast”, with the latter determining “|--” or “`--“, and the children extending the prefix accordingly, producing clear hierarchy matching clang’s “ast-dump” format.

2241543: CS325 Compiler Design: Report

5. Code Generation:

5.1 Symbol Tables and Scope:

Two symbol tables are necessary because locals are stack-allocated (AllocaInst*) whilst globals reside in static memory (GlobalVariable*). Lookup checks local scope first to implement shadowing, matching C semantics. For nested blocks, BlockAST::codegen() saves bindings to OldBindings before processing declarations and restores them afterwards, ensuring declarations don't persist beyond their scope.

5.2 Type Checking:

Only widening conversions (bool -> int -> float) are permitted because narrowing loses information and must be flagged as an error. Two helper functions implement this: promoteType() performs conversions unconditionally and is used in binary expressions where operands must match types. promoteTypeWithCheck() first calls isNarrowingConversion() to detect float -> int, float -> bool or int -> bool conversions. If any of these are the case, it reports an error via LogErrorV with context (e.g: "Narrowing conversion from float to int in variable assignment"); otherwise, it delegates to promoteType(). This checked version is used for assignments, return statements and function arguments – the three contexts where MiniC mandates narrowing should be rejected.

5.3 Variable Storage and Control Flow:

All local variables use alloca instructions rather than SSA with phi nodes. A helper CreateEntryBlockAlloca() inserts allocas at function entry point; variable reads use CreateLoad and writes use CreateStore. This approach, recommended in the LLVM tutorial, simplifies control flow; when an if-else modifies a variable in both branches, we store the same alloca from each branch and load afterwards. Therefore, no phi node is needed to merge divergent values at the join point. LLVM's mem2reg pass optimises to SSA automatically, so there is no runtime penalty. Control flow constructs generate multiple basic blocks: If-else creates then, else, and merge with a conditional branch selecting the path and unconditional branches rejoining at merge. While loops create condition, body, and exit blocks, with the body branching back to the condition block for re-evaluation.

6. Array Implementation:

The grammar extends with array_dims for declarations and "[" expr "]"in postfix expressions for access, maintaining LL(1). Three maps (LocalArrayInfo, GlobalArrayInfo, ParamArrayInfo) are necessary because array parameters decay to pointers in C, and so the function receives only a base address. Local/global arrays use nested LLVM array types with multi-index GEP. For parameters, ArrayAccessAST::codegenPtr() calculates linear offsets manually (arr[i][j] with dimension N: offset = (i*N) + j) using stored dimension sizes.

7. Known Limitations:

Short-circuit evaluation is a known limitation as both operands of && and || are always evaluated. This prevents idiomatic C patterns from working safely, as the second operand would be evaluated even when the guard condition fails. Implementing this would require conditional branching to skip the second operand and phi nodes to merge the Boolean result, which would increase the complexity. Arrays are limited to three dimensions with zero initialisation only.

8. Sources:

- LLVM Tutorial (Chapters 2, 3, 5, 7), LLVM API Documentation, CS325 Lecture Materials

Appendices:

FIRST Sets for Original Grammar:

Non-terminal	FIRST Set
program	extern, void, int, float, bool
extern_list	extern
extern	extern
decl_list	void, int, float, bool
decl	void, int, float, bool
var_decl	int, float, bool
fun_decl	void, int, float, bool
type_spec	void, int, float, bool
var_type	int, float, bool
params	int, float, bool, void, ε
param_list	int, float, bool
param	int, float, bool
block	{
local_decls	int, float, bool, ε
local_decl	int, float, bool
stmt_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, ε
stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return
expr_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;
if_stmt	if
else_stmt	else, ε
while_stmt	while
return_stmt	return
expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
rval	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
args	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ε
arg_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !

FOLLOW Sets for Original Grammar:

Non-terminal	FOLLOW Set
program	$
extern_list	void, int, float, bool
extern	extern, void, int, float, bool
decl_list	$
decl	void, int, float, bool, $
var_decl	void, int, float, bool, $
fun_decl	void, int, float, bool, $
type_spec	IDENT
var_type	IDENT
params	)
param_list	)
param	), ,
block	void, int, float, bool, $, else, IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
local_decls	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
local_decl	int, float, bool, IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
stmt_list	}
stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
expr_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
if_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
else_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
while_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
return_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
expr	;,), ,
rval	;,), ,, ||, &&, ==, !=, <, <=, >, >=, +, -, *, /, %
args	)
arg_list	)

Transformed Grammar (LL(k) Form):

- *Left recursion eliminated, operator precedence enforced, arrays supported.*

Non-terminal	Production
program	::= extern_list decl_list | decl_list
extern_list	::= extern extern_list'
extern_list'	::= extern extern_list' | ε
extern	::= "extern" type_spec IDENT "(" params ")" ";"
decl_list	::= decl decl_list'
decl_list'	::= decl decl_list' | ε
decl	::= type_spec IDENT ";"
	| type_spec IDENT array_dims ";"
	| type_spec IDENT "(" params ")" block
type_spec	::= "void" | var_type
var_type	::= "int" | "float" | "bool"
array_dims	::= "[" INT_LIT "]" array_dims'
array_dims'	::= "[" INT_LIT "]" array_dims' | ε
params	::= param_list | "void" | ε
param_list	::= param param_list'
param_list'	::= "," param param_list' | ε
param	::= var_type IDENT
	| var_type IDENT array_dims
block	::= "{" local_decls stmt_list "}"
local_decls	::= local_decl local_decls | ε
local_decl	::= var_type IDENT ";"
	| var_type IDENT array_dims ";"
stmt_list	::= stmt stmt_list | ε
stmt	::= expr_stmt | block | if_stmt | while_stmt | return_stmt
expr_stmt	::= expr ";" | ";"
while_stmt	::= "while" "(" expr ")" stmt
if_stmt	::= "if" "(" expr ")" block else_stmt
else_stmt	::= "else" block | ε
return_stmt	::= "return" ";" | "return" expr ";"
expr	::= or_expr "=" expr | or_expr
or_expr	::= and_expr or_expr'
or_expr'	::= "||" and_expr or_expr' | ε
and_expr	::= eq_expr and_expr'
and_expr'	::= "&&" eq_expr and_expr' | ε
eq_expr	::= rel_expr eq_expr'
eq_expr'	::= ("==" | "!=") rel_expr eq_expr' | ε
rel_expr	::= add_expr rel_expr'
rel_expr'	::= ("<" | "<=" | ">" | ">=") add_expr rel_expr' | ε
add_expr	::= mul_expr add_expr'
add_expr'	::= ("+" | "-") mul_expr add_expr' | ε
mul_expr	::= unary_expr mul_expr'
mul_expr'	::= ("*" | "/" | "%") unary_expr mul_expr' | ε
unary_expr	::= ("-" | "!") unary_expr | postfix_expr
postfix_expr	::= primary_expr postfix_expr'
postfix_expr'	::= "(" args ")" | "[" expr "]" postfix_expr' | ε
primary_expr	::= IDENT | INT_LIT | FLOAT_LIT | BOOL_LIT | "(" expr ")"
args	::= arg_list | ε
arg_list	::= expr arg_list'
arg_list'	::= "," expr arg_list' | ε

FIRST Sets for Transformed Grammar:

Non-terminal	FIRST Set
program	extern, void, int, float, bool
extern_list	extern
extern_list'	extern, ε
extern	extern
decl_list	void, int, float, bool
decl_list'	void, int, float, bool, ε
decl	void, int, float, bool
type_spec	void, int, float, bool
var_type	int, float, bool
array_dims	[
array_dims'	[, ε
params	int, float, bool, void, ε
param_list	int, float, bool
param_list'	,, ε
param	int, float, bool
block	{
local_decls	int, float, bool, ε
local_decl	int, float, bool
stmt_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, ε
stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return
expr_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;
if_stmt	if
else_stmt	else, ε
while_stmt	while
return_stmt	return
expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
or_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
or_expr'	||, ε
and_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
and_expr'	&&, ε
eq_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
eq_expr'	==, !=, ε
rel_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
rel_expr'	<, <=, >, >=, ε
add_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
add_expr'	+, -, ε
mul_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
mul_expr'	*, /, %, ε
unary_expr	-, !, IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (
postfix_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (
postfix_expr'	(, [, ε
primary_expr	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (
args	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ε
arg_list	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !
arg_list'	,, ε

FOLLOW Sets for Transformed Grammar:

Non-terminal	FOLLOW Set
program	$
extern_list	void, int, float, bool
extern_list'	void, int, float, bool
extern	extern, void, int, float, bool
decl_list	$
decl_list'	$
decl	void, int, float, bool, $
type_spec	IDENT
var_type	IDENT
array_dims	;,)
array_dims'	;,)
params	)
param_list	)
param_list'	)
param	,,)
block	void, int, float, bool, $, else, IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
local_decls	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
local_decl	int, float, bool, IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
stmt_list	}
stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
expr_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
if_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
else_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
while_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
return_stmt	IDENT, INT_LIT, FLOAT_LIT, BOOL_LIT, (, -, !, ;, {, if, while, return, }
expr	],), ;, ,
or_expr	], =,), ;, ,
or_expr'	], =,), ;, ,
and_expr	], ||, =,), ;, ,
and_expr'	], ||, =,), ;, ,
eq_expr	], &&, ||, =,), ;, ,
eq_expr'	], &&, ||, =,), ;, ,
rel_expr	], ==, !=, &&, ||, =,), ;, ,
rel_expr'	], ==, !=, &&, ||, =,), ;, ,
add_expr	], <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
add_expr'	], <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
mul_expr	], +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
mul_expr'	], +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
unary_expr	], *, /, %, +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
postfix_expr	], *, /, %, +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
postfix_expr'	], *, /, %, +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
primary_expr	], (, [, *, /, %, +, -, <, <=, >, >=, ==, !=, &&, ||, =,), ;, ,
args	)
arg_list	)
arg_list'	)

Operator Precedence:

Level	Operators	Associativity
1	= (assignment)	Right-to-left
2	||	Left-to-right
3	&&	Left-to-right
4	== !=	Left-to-right
5	< <= > >=	Left-to-right
6	+ -	Left-to-right
7	* / %	Left-to-right
8	! - (unary)	Right-to-left
9	() function call, [] array access	Left-to-right
10	literals, identifiers, (expr)	N/A