# CS325 Compiler Design Report u2022016

**Final Grammar:**
start -> program EOF
program -> extern_list decl_list | decl_list
extern_list -> extern extern_list1
extern_list1 -> extern extern_list1 | epsilon
extern -> 'extern' type_spec IDENT '(' params ')' ';'
decl_list -> decl decl_list1
decl_list1 -> decl decl_list1 | epsilon
decl -> var_decl | fun_decl
var_decl -> var_type IDENT ';'
type_spec -> 'void' | var_type
var_type -> 'int' | 'float' | 'bool'
fun_decl -> type_spec IDENT '(' params ')' block
params -> param_list | 'void' | epsilon
param_list -> param param_list1
param_list1 -> ',' param param_list1 | epsilon
param -> var_type IDENT
block -> '{' local_decls stmt_list '}'
local_decls -> local_decl local_decls | epsilon
local_decl -> var_type IDENT ';'
stmt_list -> stmt stmt_list | epsilon
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 | epsilon
return_stmt -> 'return' ';' | 'return' expr ';'
expr -> IDENT '=' expr | rval
rval -> rval1 rvalprime
rvalprime -> '||' rval1 rvalprime | epsilon
rval1 -> rval2 rval1prime
rval1prime -> '&&' rval2 rval1prime | epsilon
rval2 -> rval3 rval2prime
rval2prime -> '==' rval3 rval2prime | '!=' rval3 rval2prime | epsilon
rval3 -> rval4 rval3prime
rval3prime -> '<=' rval4 rval3prime | '<' rval4 rval3prime | '>=' rval4 rval3prime | '>' rval4 rval3prime | epsilon
rval4 -> rval5 rval4prime
rval4prime -> '+' rval5 rval4prime | '-' rval5 rval4prime | epsilon
rval5 -> rval6 rval5prime
rval5prime -> '*' rval6 rval5prime | '/' rval6 rval5prime | '%' rval6 rval5prime | epsilon
rval6 -> '-' rval6 | '!' rval6 | '+' rval6 | rval7
rval7 -> '(' expr ')' | IDENT | IDENT '(' args ')' | INT_LIT | FLOAT_LIT | BOOL_LIT
args -> arg_list | epsilon
arg_list -> expr arg_list1
arg_list1 -> ',' expr arg_list1 | epsilon

**First and Follow Sets:**
Attached in firstfollowg11.csv

**Computing First and Follow sets:**
We programmatically generated first and follow sets using the algorithms provided in the course notes. We used Python3 for this. We first found the null sets for each item. It is possible to recursively generate each first set by visiting each item only once which we did next. To determine follow sets we must do a parse of the entire grammar and build a directed graph of the dependencies of follow sets. A cycle in this graph implies every item in the cycle has the same follow set (cyclic set inclusion). We then contracted cycles until there were no cycles left. We then topologically sorted the vertices (in ascending order) and then computed follow sets in that order. This guarantees that required follow sets for the current follow set are already computed.

We also then constructed a parser table and found all instances where there is ambiguity when only looking at one token. We then manually inspected the small number of these cases and implemented lookahead by hand in each case. The maximum we had to lookahead was 2 tokens so this is an LL(3) parser.

**Lexing:**
Already implemented. We lazily lex as we parse, only lexing text when we need to get the next token from the parser.

**Parser and Building AST:**
We transformed the grammar to remove left recursion and to enforce operation precedence. We implement one function that parses each line of the grammar. This is needed as opposed to a general algorithm (which was a wrong turn made) because of the very specific things that need to be done to build the AST for each case – lookahead, moving around parameters to create AST node. We create AST nodes at the parsing functions that correspond to our programming constructs (which has less nodes than the number of nonterminals in the grammar). More specifically we reduced the number of node classes by having a general Binary operation node, Unary operation node, using the same Decl node for global and local declarations, using the same VarDecl node (with an isGlobal flag) for both global and local variables, implementing all lists except ParamLists as just a vector of pointers instead of a node and using the same FunProto node for externs and locally declared functions.

We had to implement operator precedence in grammar. Originally we made a mistake where we did not have operations of the same precedence be left to right associative and thus had to refactor. An interesting part of building the AST is for expressions. Since for rval to rval5 we do not know how many of their operations there will be and we have to evaluate from left to right, the root node will not be that of the first called parsing function. We pass a pointer to the root node of the LHS of an expression to the next parsing function so that it will either return it as the root node or make it a child of its own node and return its own node to be the overall parent.

We print the AST by recursively calling to_string() on each of the child nodes of a node with some minor beautifying string additions in between. We include the indentation level as a parameter to each node.

**Code Generation and Semantic Checks:**
General structure is a traversal of the AST. We need to do special actions at each point such as mutable variable declaration/assignment, type checking and scope checking which makes it more complex than just recursively calling codegen() on each child.

We check to see if a variable is declared before we allow it to be defined or referenced.

We implement mutable variables using LLVM allocas.
We keep track of declared local variables in a
vector<unique_ptr<std::map<string, AllocaInst*>>> NamedValuesVector
For every new local scope (a new block which is bounded by curly braces) we push_back to this vector. We allow existing local variables to be redeclared in an inner scope but not in the same scope. We find which AllocaInst to load from by searching the vector in reverse order and choosing the AllocaInst we find first. When we exit a local scope we pop_back the last element in the vector.

For global variables we have
std::map<std::string, GlobalVariable*>  GlobalNamedValues
which keeps track of globals that have already been declared. When a global variable is declared we register it with the Module with CommonLinkage.

Functions are externally linked. We also allow extern function declarations. (codegen for this  is identical to a locally declared function's prototype codegen).

We have 4 functions which are used for casting: force_cast, bool_cast, int_cast, widening_cast_or_err
During assignment, we forcibly cast the RHS to be the type of the LHS.
We only allow widening casts for function arguments and for function return value.
We check types for binary operations and widen them to the widest of the two for the computation. The exceptions to this are the AND and OR binary operations. In this case we forcibly cast both operand to Boolean values.
We do not allow variable argument length functions and this is denoted in function definition by empty brackets: function() or the void keyword: function(void).
We check that a function returning void does not return a value.
We stop generating code after the first return statement in a given block.

We allow assignment and computation to be done at the same with the expression (a = b) having the value of b.

We have implemented short circuiting AND and OR binary operations that do not evaluate the RHS of the operation if the LHS will already give us the answer of the operation. Other binary operations evaluate both the LHS and the RHS. There exist 3 unary operations PLUS, MINUS and NOT with their expected, simple behaviour.

We have implemented if statements using the end of the current BasicBlock, a "then" BasicBlock, an optional "else" BasicBlock and an "end" BasicBlock which is where the IRBuilder continues to write code from.

We have implemented while statements by breaking from the current BasicBlock to a "whilecondition" BasicBlock where a condition is checked and then going to a "whilebody" BasicBlock or an "end" BasicBlock where IRBuilder can continue. We unconditionally go from "whilebody" back to "whilecondition".

We fill empty BasicBlocks with one command assigning a temporary constant. This is because LLVM will raise a syntax error if there is a BasicBlock with no instructions.

**Limitations:**
Global variables are set to only be defined onc. This however judges the first time it is defined by reading the source code top to bottom. Thus we have also included a setting where global variables can be modified an arbitrary number of times in line 573 of astnodes.cpp. (Note: we still only allow global variables to be declared once).

An optimisation pass could be done which eliminates (what would be) empty BasicBlocks and appropriately modifies other BasicBlocks that reference it.

**Other Sources Consulted:**
We consulted the LLVM documentation, Prof. Gihan Mudalige's CS325 lecture slides and used stackoverflow.com extensively. ChatGPT3.5 was used to generate the code for 2 functions used for parsing a csv file (and nowhere else). ChatGPT3.5 was often asked C++ syntax or LLVM C++ API questions which the answers were then verified by reading the respective documentation.