Parser Design

My parser design is quite unique amongst others, and results in clean, concise, bug-free and resilient code.

Generating First and Follow sets programmatically

I parse the grammar from a string form (`grammarString.hpp`), create a data structure to hold the productions for each nonterminal, and then compute the first and follow sets. The sets are then used to make the parser predictive - I converted the grammar to only require a lookahead of 1 (in almost every production), and so could use the sets to check the next production to take or whether to take an epsilon production.

Doing this programmatically avoided the inevitable errors of manual computation and ensures correctness and robustness of the compiler.

The `nonterminal` function

I abstracted out the basic framework for a production into a function `nonTerminalInfo nonterminal(const std::string& name)`, which takes the name of the production to parse, and returns a map of strings to vectors of `ASTNodes`, for each nonterminal (and terminal) on the production's RHS. A `functionMap` is maintained so that the corresponding production function for a given nonterminal can be called dynamically. This removes the common and confusing spaghetti code involving `getNextToken()` and `putBackToken()` that would otherwise be present, resulting in very short, readable production functions. In fact `putBackToken()` was never even needed (and has been removed). As an example, here is the function for the `param` production:

```
ptrVec<ASTNode> param() {
    nonTerminalInfo info = nonterminal("param");
    ptrVec<ASTNode> temp;
    ptrVec<ASTNode> var_type = std::move(info["var_type"]);
    ptrVec<ASTNode> ident = std::move(info["IDENT"]);
    temp.push_back(std::make_unique<ParamAST>(var_type, ident));
    return temp;
}
```

Everything is moved around as a general `ptrVec<ASTNode> = std::vector<std::unique_ptr<ASTNode>>` to facilitate this approach, until moved into an `ASTNodes` constructor, where nodes are cast and extracted as needed. Multiple overloaded templates, `castToDerived<Base, Derived>` were created to facilitate clean and easy casting of nodes.

Error handling

Yet another benefit of my abstracted `nonterminal` function was that it allowed for a sublime way of implementing comprehensive error handling for the parser. I build up a `expected` set of terminals (using the first sets) as I check for productions to use - if no production can be found matching the current token, I pass this expected set to `throwParserError(const std::unordered_set<std::string>& expected, std::string found)`, which prints a comprehensive error:

```
Encountered an error on line 2 column 23.
Expected one of: void int bool float extern
Found: eextern
```

and then exits the program.
Similarly, `throwCodegenError` makes use of tokens stored with each `ASTNode` to print the exact line, column and lexeme where an error occurred.

IR Generation

Casting and Type Checking

Following the spirit of delightful code, multiple helper functions were created to make this area clean and robust.

- `stringToPtrType` and `ptrToStringType` to convert between `std::string` and `Type*` representations of types
- `Value* widenLtoR(Value* leftVal, Type* rightType)` which widens leftVal to rightType and returns the new corresponding `Value*`
- Dually, `Value* narrowLtoR(Value* leftVal, Type* rightType)`

Widening and narrowing were implemented using IRBuilder instructions `CreateSIToFP`, `CreateZExt`, `CreateFPToSI`, and `CreateICmpNE`. Following the coursework guide, as well as the C99 specification, the following are the rules that I created for type checking and casting:

- Types are widened as much as needed during arithmetic and logical operations
- Types are only widened when passed into functions

- Types are only widened when returned from functions
- Types are widened or narrowed during variable assignment

Given the inverse of a rule, a descriptive error is thrown if needed.

Scope

```cpp
using SymbolTable = std::unordered_map<std::string, AllocaInst*>;
using GlobalTable = std::unordered_map<std::string, GlobalVariable*>;
using Tables = std::vector<SymbolTable>;
Tables tables;
GlobalTable globalTable;
```

Using a vector of symbol tables for scope led to a fairly intuitive implementation - a new symbol table is pushed every time a new block is being code generated. A few tricks were required to completely make this approach work - for example, a global flag `isFunctionBlock`, so that we don't double push a symbol table when declaring a new function body.

Ensuring valid IR

This involved:

1. Ensuring each basic block is terminated (ends in either a `ret` or `br` statement). Almost every block will naturally either branch or return. The only block which may not do either is the last block in our function body. If no return statement is present in this block, `verifyFunction()` returns true, and we either:

   (a) Add a void return for a void function

   (b) Return a default value for a typed function

   This also results in a more intuitive experience for the end user.

2. That no other instructions follow a `ret`. This is handled by stopping codegen of a block's statement list when a node of type `ReturnAST` is encountered. For cases where there may be a return statement within an if/else/while block, `hasCreatedReturn` is used to check if a branch should avoid being added to these blocks.

Lazy Boolean Evaluation

Final Grammar

```
program ::= extern_list decl_list
program ::= decl_list
extern_list ::= extern extern_list’
extern_list’ ::= extern extern_list’
extern_list’ ::= ’’
extern ::= "extern" type_spec IDENT "(" params ")" ";"
decl_list ::= decl decl_list’
decl_list’ ::= decl decl_list’
decl_list’ ::= ’’
decl ::= var_type IDENT decl’
decl ::= "void" IDENT "(" params ")" block
decl’ ::= "(" params ")" block
decl’ ::= ";"
type_spec ::= "void"
type_spec ::= var_type
var_type  ::= "int"
var_type ::= "float"
var_type ::= "bool"
params ::= param_list
params ::= "void"
params ::= ’’
param_list ::= param param_list’
param_list’ ::= "," param param_list’
param_list’ ::= ’’
param ::= var_type IDENT
block ::= "{" local_decls stmt_list "}"
local_decls ::= local_decl local_decls
local_decls ::= ’’
local_decl ::= var_type IDENT ";"
stmt_list ::= stmt stmt_list’
stmt_list ::= ’’
stmt_list’ ::= stmt stmt_list’
stmt_list’ ::= ’’
stmt ::= expr_stmt
stmt ::= block
stmt ::= if_stmt
stmt ::= while_stmt
stmt ::= return_stmt
expr_stmt ::= expr ";"
expr_stmt ::= ";"
while_stmt ::= "while" "(" expr ")" stmt
if_stmt ::= "if" "(" expr ")" block else_stmt
else_stmt ::= "else" block
else_stmt ::= ’’
return_stmt ::= "return" return_stmt’
return_stmt’ ::= ";"
return_stmt’ ::= expr ";"
expr ::= IDENT "=" expr
expr ::= rval_or
rval_or ::= and_exp rval_or’
rval_or’ ::= "||" and_exp rval_or’
rval_or’ ::= ’’
and_exp ::= equality_exp and_exp’
and_exp’ ::= "&&" equality_exp and_exp’
and_exp’ ::= ’’
equality_exp ::= relational_exp equality_exp’
equality_exp’ ::= "==" relational_exp equality_exp’
equality_exp’ ::= "!=" relational_exp equality_exp’
equality_exp’ ::= ’’
relational_exp ::= additive_exp relational_exp’
relational_exp’ ::= "<=" additive_exp relational_exp’
relational_exp’ ::= "<" additive_exp relational_exp’
relational_exp’ ::= ">=" additive_exp relational_exp’
relational_exp’ ::= ">" additive_exp relational_exp’
relational_exp’ ::= ’’
additive_exp ::= multiplicative_exp additive_exp’
additive_exp’ ::= "+" multiplicative_exp additive_exp’
additive_exp’ ::= "-" multiplicative_exp additive_exp’
additive_exp’ ::= ’’
multiplicative_exp ::= factor multiplicative_exp’
multiplicative_exp’ ::= "*" factor multiplicative_exp’
```

```
multiplicative_exp' ::= "/" factor multiplicative_exp'
multiplicative_exp' ::= "%" factor multiplicative_exp'
multiplicative_exp' ::= ''
factor ::= "-" factor
factor ::= "!" factor
factor ::= primary
primary ::= "(" expr ")"
primary ::= IDENT primary'
primary ::= INT_LIT
primary ::= FLOAT_LIT
primary ::= BOOL_LIT
primary' ::= "(" args ")"
primary' ::= ''
args ::= arg_list
args ::= ''
arg_list ::= expr arg_list'
arg_list' ::= "," expr arg_list'
arg_list' ::= ''
```

First Sets

```
FIRST(stmt_list') = { '(' 'while' ';' '{' 'if' '!' 'FLOAT_LIT' 'BOOL_LIT' 'IDENT' '''' '-' 'return' 'INT_LIT' }
FIRST(relational_exp) = { '-' 'IDENT' 'INT_LIT' 'BOOL_LIT' '(' 'FLOAT_LIT' '!' }
FIRST(type_spec) = { 'float' 'void' 'int' 'bool' }
FIRST("else") = { 'else' }
FIRST(primary') = { '(' '''' }
FIRST("-") = { '-' }
FIRST(decl_list) = { 'bool' 'int' 'float' 'void' }
FIRST("%") = { '%' }
FIRST(multiplicative_exp') = { '/' '%' '''' '*' }
FIRST(relational_exp') = { '''' '<=' '<' '>=' '>' }
FIRST(stmt_list) = { ';' 'while' '{' '(' '!' 'if' 'FLOAT_LIT' 'BOOL_LIT' '''' 'return' '-' 'IDENT' 'INT_LIT' }
FIRST("!") = { '!' }
FIRST(else_stmt) = { '''' 'else' }
FIRST(INT_LIT) = { 'INT_LIT' }
FIRST(BOOL_LIT) = { 'BOOL_LIT' }
FIRST(factor) = { '(' 'FLOAT_LIT' '!' 'INT_LIT' 'IDENT' '-' 'BOOL_LIT' }
FIRST("<") = { '<' }
FIRST('') = { '''' }
FIRST(FLOAT_LIT) = { 'FLOAT_LIT' }
FIRST("/") = { '/' }
FIRST(arg_list') = { ',' '''' }
FIRST(multiplicative_exp) = { '-' 'IDENT' 'INT_LIT' 'BOOL_LIT' '(' 'FLOAT_LIT' '!' }
FIRST(return_stmt') = { 'INT_LIT' '-' 'IDENT' ';' 'BOOL_LIT' '!' 'FLOAT_LIT' '(' }
FIRST(extern_list') = { 'extern' '''' }
FIRST(additive_exp) = { '(' 'FLOAT_LIT' '!' 'INT_LIT' 'IDENT' '-' 'BOOL_LIT' }
FIRST(return_stmt) = { 'return' }
FIRST(equality_exp) = { '(' '!' 'FLOAT_LIT' 'INT_LIT' 'IDENT' '-' 'BOOL_LIT' }
FIRST("(") = { '(' }
FIRST(and_exp) = { '-' 'INT_LIT' 'IDENT' 'BOOL_LIT' '(' 'FLOAT_LIT' '!' }
FIRST(extern_list) = { 'extern' }
FIRST(rval_or') = { '||' '''' }
FIRST("*") = { '*' }
FIRST(";") = { ';' }
FIRST("+") = { '+' }
FIRST(expr) = { '-' 'IDENT' 'INT_LIT' 'BOOL_LIT' '(' 'FLOAT_LIT' '!' }
FIRST("while") = { 'while' }
FIRST(primary) = { 'FLOAT_LIT' '(' 'IDENT' 'BOOL_LIT' 'INT_LIT' }
FIRST(arg_list) = { '(' '!' 'FLOAT_LIT' 'INT_LIT' 'IDENT' '-' 'BOOL_LIT' }
FIRST(">=") = { '>=' }
FIRST("void") = { 'void' }
FIRST("{") = { '{' }
FIRST("int") = { 'int' }
FIRST(local_decl) = { 'float' 'bool' 'int' }
FIRST(while_stmt) = { 'while' }
FIRST(block) = { '{' }
FIRST(rval_or) = { 'IDENT' 'INT_LIT' '-' 'FLOAT_LIT' '!' '(' 'BOOL_LIT' }
FIRST(">") = { '>' }
FIRST(expr_stmt) = { '(' 'FLOAT_LIT' '!' 'INT_LIT' 'IDENT' '-' ';' 'BOOL_LIT' }
FIRST("if") = { 'if' }
FIRST(args) = { '-' 'IDENT' 'INT_LIT' '''' 'BOOL_LIT' '!' '(' 'FLOAT_LIT' }
FIRST(params) = { 'void' 'float' 'int' '''' 'bool' }
FIRST(local_decls) = { ’int’ ’’’’ ’bool’ ’float’ }
FIRST("==") = { ’==’ }
FIRST(if_stmt) = { ’if’ }
FIRST(decl’) = { ’;’ ’(’ }
FIRST("bool") = { ’bool’ }
FIRST("extern") = { ’extern’ }
FIRST(param) = { ’float’ ’int’ ’bool’ }
FIRST("float") = { ’float’ }
FIRST(IDENT) = { ’IDENT’ }
FIRST(var_type) = { ’bool’ ’int’ ’float’ }
FIRST(",") = { ’,’ }
FIRST("||") = { ’||’ }
FIRST(program) = { ’bool’ ’int’ ’void’ ’float’ ’extern’ }
FIRST(equality_exp’) = { ’!=’ ’==’ ’’’’ }
FIRST("!=") = { ’!=’ }
FIRST(decl_list’) = { ’int’ ’’’’ ’bool’ ’void’ ’float’ }
FIRST(and_exp’) = { ’’’’ ’&&’ }
FIRST(decl) = { ’void’ ’float’ ’int’ ’bool’ }
FIRST(stmt) = { ’BOOL_LIT’ ’!’ ’if’ ’FLOAT_LIT’ ’;’ ’IDENT’ ’INT_LIT’ ’return’ ’-’ ’(’ ’while’ ’{’ }
FIRST("<=") = { ’<=’ }
FIRST("return") = { ’return’ }
FIRST(param_list’) = { ’,’ ’’’’ }
FIRST("&&") = { ’&&’ }
FIRST(extern) = { ’extern’ }
FIRST(param_list) = { ’int’ ’bool’ ’float’ }
FIRST(additive_exp’) = { ’+’ ’’’’ ’-’ }
```

Follow Sets

```
FOLLOW(relational_exp) = { ’;’ ’,’ ’||’ ’==’ ’&&’ ’!=’ ’)’ }
FOLLOW(args) = { ’)’ }
FOLLOW(expr_stmt) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’if’ ’!’ ’(’ ’while’ ’;’ }
FOLLOW(else_stmt) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’if’ ’!’ ’(’ ’while’ ’;’ }
FOLLOW(arg_list’) = { ’)’ }
FOLLOW(decl’) = { ’bool’ ’0’ ’float’ ’void’ ’int’ }
FOLLOW(rval_or) = { ’)’ ’;’ ’,’ }
FOLLOW(equality_exp) = { ’)’ ’;’ ’,’ ’||’ ’&&’ }
FOLLOW(rval_or’) = { ’)’ ’;’ ’,’ }
FOLLOW(and_exp) = { ’)’ ’;’ ’,’ ’||’ }
FOLLOW(multiplicative_exp) = { ’,’ ’!=’ ’==’ ’>=’ ’>’ ’&&’ ’+’ ’<=’ ’||’ ’-’ ’;’ ’)’ ’<’ }
FOLLOW(return_stmt’) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’if’ ’!’ ’(’ ’while’ ’;’ }
FOLLOW(equality_exp’) = { ’)’ ’;’ ’,’ ’||’ ’&&’ }
FOLLOW(program) = { ’0’ }
FOLLOW(stmt_list’) = { ’}’ }
FOLLOW(relational_exp’) = { ’;’ ’,’ ’||’ ’&&’ ’==’ ’!=’ ’)’ }
FOLLOW(additive_exp’) = { ’;’ ’,’ ’||’ ’!=’ ’<=’ ’>’ ’==’ ’&&’ ’)’ ’>=’ ’<’ }
FOLLOW(arg_list) = { ’)’ }
FOLLOW(type_spec) = { ’IDENT’ }
FOLLOW(multiplicative_exp’) = { ’,’ ’!=’ ’==’ ’>=’ ’>’ ’&&’ ’<=’ ’+’ ’)’ ’||’ ’-’ ’;’ ’<’ }
FOLLOW(stmt) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’!’ ’if’ ’;’ ’(’ ’while’ }
FOLLOW(stmt_list) = { ’}’ }
FOLLOW(expr) = { ’)’ ’;’ ’,’ }
FOLLOW(return_stmt) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’if’ ’!’ ’(’ ’while’ ’;’ }
FOLLOW(additive_exp) = { ’;’ ’,’ ’||’ ’>=’ ’<=’ ’!=’ ’>’ ’==’ ’&&’ ’<’ ’)’ }
FOLLOW(local_decls) = { ’BOOL_LIT’ ’INT_LIT’ ’return’ ’-’ ’IDENT’ ’FLOAT_LIT’ ’}’ ’while’ ’{’ ’(’ ’!’ ’if’ ’;’ }
FOLLOW(block) = { ’INT_LIT’ ’-’ ’return’ ’IDENT’ ’BOOL_LIT’ ’float’ ’int’ ’else’ ’FLOAT_LIT’ ’}’ ’bool’ ’{’ ’while’ ’void’
FOLLOW(params) = { ’)’ }
FOLLOW(decl) = { ’bool’ ’float’ ’void’ ’int’ }
FOLLOW(primary) = { ’,’ ’&&’ ’>’ ’>=’ ’==’ ’<=’ ’+’ ’)’ ’||’ ’-’ ’<’ ’%’ ’*’ ’/’ ’;’ ’!=’ }
FOLLOW(extern_list) = { ’void’ ’float’ ’bool’ ’int’ }
FOLLOW(decl_list) = { ’0’ }
FOLLOW(if_stmt) = { ’BOOL_LIT’ ’INT_LIT’ ’IDENT’ ’-’ ’return’ ’FLOAT_LIT’ ’}’ ’{’ ’if’ ’!’ ’(’ ’while’ ’;’ }
FOLLOW(decl_list’) = { ’0’ }
FOLLOW(and_exp’) = { ’)’ ’;’ ’,’ ’||’ }
FOLLOW(factor) = { ’,’ ’&&’ ’>’ ’>=’ ’==’ ’<=’ ’+’ ’)’ ’||’ ’-’ ’<’ ’;’ ’/’ ’!=’ ’%’ ’*’ }
FOLLOW(param_list) = { ’)’ }
FOLLOW(param_list’) = { ’)’ }
FOLLOW(primary’) = { ’,’ ’&&’ ’>’ ’>=’ ’==’ ’<=’ ’+’ ’)’ ’||’ ’-’ ’<’ ’%’ ’*’ ’/’ ’;’ ’!=’ }
FOLLOW(param) = { ’)’ ’,’ }
FOLLOW(extern_list’) = { ’void’ ’float’ ’bool’ ’int’ }
FOLLOW(var_type) = { ’IDENT’ }
FOLLOW(local_decl) = { ’INT_LIT’ ’IDENT’ ’return’ ’-’ ’int’ ’}’ ’FLOAT_LIT’ ’bool’ ’!’ ’if’ ’(’ ’;’ ’float’ ’BOOL_LIT’
'while' '{' }
FOLLOW(while_stmt) = { 'BOOL_LIT' 'INT_LIT' 'IDENT' '-' 'return' 'FLOAT_LIT' '}' '{' 'if' '!' '(' 'while' ';' }
FOLLOW(extern) = { 'void' 'int' 'bool' 'float' 'extern' }
```