Methodology:

I preferred to start with the nested class (ScopeTable) then work out to ensure that I have all the functions necessary at each function higher up in the chain.

<u>SymbolTable::ScopeTable::ScopeTable()</u>

The ScopeTable has 3 private class members in mSymbols, mChildren, and mParent. The constructor initializes the mParent of the current ScopeTable, so we just need to ensure that the parent and the current ScopeTable are bidirectionally connected. To safely do this, we first check if the mParent is not NULL, because the global scope's parent = NULL, otherwise we push the new ScopeTable to the list of its parents mChildren.

<u>SymbolTable::ScopeTable::~ScopeTable()</u>

For the deconstructor, we need to clean up dynamically allocated memory in the mChildren and mSymbols data structures. However, we should be careful not to delete mParent, because the child table does not own the parent and deleting them would cause issues later. All other objects will be deleted automatically as they are stack-allocated like the map and strings.

<u>SymbolTable::ScopeTable::addIdentifier</u>

Simply just add the identifier to the map with the key as its name. Allow higher level code to handle errors such as duplicates or NULL.

<u>SymbolTable::ScopeTable::searchInScope</u>

Search through current scope's mSymbols map to see if it contains the given identifier name as a key. If it is found, return the corresponding identifier, otherwise return NULL.

<u>SymbolTable::ScopeTable::search</u>

First check the current scope by using the searchInScope function. If it is found, return the identifier else you check if there exists a mParent and then recursively call search on the parent if it exists, otherwise return NULL.

<u>SymbolTable::enterScope</u>

Create the new scopeTable with mCurrScope as the parent and then we set the mCurrScope to the newScope along with returning the newScope pointer.

<u>SymbolTable::SymbolTable</u>

Use an initializer list to set the mCurrScope of the root SymbolTable as NULL so that we can use the enterScope function to handle creating and entering the global scope for us. Then simply create the identifiers laid out by the document, set their corresponding type, and add to the current scope. We rely on createIdentifer to make the identifiers along with adding them.

<u>SymbolTable::exitScope()</u>

Just make mCurrScope set to the parent scope using the getParent function from ScopeTable.

<u>SymbolTable::~SymbolTable()</u>

First ensure that we are at the root table by walking back to the root table using exitScope. We want to make sure we are at the root table so that when deleting the scopeTables, it will delete the entire tree by recursively calling delete on each of its children as it walks down by calling the destructor in ScopeTable.

<u>SymbolTable::createIdentifier</u>

Check if the name is already declared in scope with isDeclaredInScope and if it is do nothing or how I interpreted It - returning the existing identifier. If it doesn't exist already then create it along with adding it to mCurrScope and return the identifier.

<u>SymbolTable::getIdentifier</u>

Check if the identifier is found at all in the symbolTable, not just the current scope, and if it is return it; otherwise, return nullptr.

<u>SymbolTable::isDeclaredInScope</u>

Search for the identifier name in the current scope using searchInScope, returning true if it is there; otherwise, return false.

<u>Parser::getVariable</u>

Use getIdentifier to see if the variable can be found. If the function returns NULL, then reportSemanterror for 'use of undeclared variable', and return the dummy variable identifier @@variable; otherwise, return the corresponding found variable.

<u>Parser::parseDecl</u>

For redeclaration errors check if it is declared in the same scope after parsing a identifier token. If it is declared in the same scope after parsing, reportSemantError before consuming the token to ensure the columnLine number is correct. Also make sure to set the identifier to @@variable to catch further errors.

<u>Parser::intToChar</u>

There are also the same 3 cases to consider here: 1) It is a char expression already, 2) It is a int constant expression, 3) It is an int expression, but it is not constant. For case 1, simply return the expression as it is. For case 2, simply change the type of the expression to char using the type change functions for ASTConstantExpr. Case 3 is where things get tricky, since char's are aggressively converted to int whenever it is accessed in an expression node. This means that if you have a case where you assign a char variable to a character, it will redundantly convert to int first before reconverting to a char again. The optimization here would just be to check if it is of derived class ASTToIntExpr and returning the child, which would be a char expression. However, if it is not of derived class ASTToIntExpr, we have to use ASTToCharExpr to do the conversion. To check if the derived class types match, I used dynamic_pointer_cast to determine at runtime. It is also important to check for NULL arguments since the parse functions that use this could pass in a NULL.

<u>Parser::charToInt</u>

There are 3 cases to consider: 1) It is an int expression already, 2) It is a char constant expression, 3) It is a char expression, but it is not constant. Cases 1, 2, and 3 for this are practically the same as intToChar; however, it doesn't require the optimization as ASTToCharExpr are less frequent in appearance than ASTToIntExpr as we do not aggressively convert to char. The only case where this slight optimization could help is if we are reassigning a char to an int after assigning an int to a char, but this rarely occurs. It is also important to check for NULL arguments since the parse functions that use this could pass in a NULL.

<u>Parser::parseIdentFactor(), Parser::parseIncFactor(), parseDecFactor()</u>

Since they all return identifier-based expressions and we know to aggressively convert charToInt, simply call charToInt at the end whenever we are returning retVal.

<u>Parser::parseAssignStmt(), Parser::parseDecl, Parser::parseReturnStmt()</u>

For these functions, I followed the existing code that showed a blueprint before each of the respective sections for PA2. This consisted of checking for the mcheckSemant flag then checking if the identifier is of type char and expression is of type int, where intToChar is called if they matched, or in the case of pair mismatched types, reportSemantError.

finalizeOp

This was just a matter of following the instructions and setting mType to int, returning true if lhs and rhs are integers and false otherwise.

The parsing functions I hooked the finalizeOp up to are parseExprPrime, parseAndTermPrime, parseRelExprPrime, parseNumExprPrime, parseTermPrime. In those functions, I just reportSemantError if finalizeOp returned false. It was also important to save the column and line numbers before consuming the tokens for the reportSemantError.

Parser::parseCompoundStmt

First if we are not isFuncBody we have to enter new scope and exit as it is not handled yet. The other addition is for the return statement if we isFuncBody. To ensure that we have a return statement, we have 2 cases: if the function is void and it doesn't have a return statement, in which case we create the void return node and add it, and if the return type isn't void with no return statement, in which case we output an error message.

parseReturnStmt

Ensure that the function has a return statement if it should have one and reportSemant otherwise. This is done by checking if the return type is not void and there is no expression.

parseAssignStmt

Do not allow reassignment of arrays by checking if the identifier is an array if they attempt to do so and if so reportSemantError otherwise just proceed as normal.

ReportSemantError Usage

A lot of the parse functions utilized consuming tokens, so it is important to save the respective column and line numbers for the caret when using ReportSemantError. So wherever the caret was supposed to be for the test cases is where I saved the corresponding numbers before allowing the parse functions to consume the tokens.