PA1: Recursive Descent Parser

Introduction

In this assignment, you will write large portions of a recursive descent parser for the USC programming language. Before you begin working on this assignment, you should read the USC language reference. Specifically, you will need to refer to the grammar definitions while working on this assignment.

The scanner for USCC has already been provided to you. It is implemented via flex, and the input file is `parse/usc.l`. The list of tokens are in `scan/Tokens.def`, and they are defined in an `enum`. Provided in `namespace uscc::scan`, the tokens can be accessed via a `Token::` prefix. For example, `Token::Key_while` accesses the `while` keyword token. The `Parser` class, which uses the flex-generated scanner, is declared in `parse/Parse.h`, and implemented in `parse/Parse.cpp`, `parse/ParseExpr.cpp`, and `parse/ParseStmt.cpp`.

The `Parser` class has many functions prefixed with `parse`, such as `parseProgram`, `parseCompoundStmt`, and so on. These functions are the recursive descent parsing functions, which you will implement in this assignment. Each of these functions returns a pointer to a specific type of ***abstract syntax tree*** (AST) node. The AST is used as the initial representation of the source program. Each of these nodes is declared in `parse/ASTNodes.h`. For this assignment, you do not need to worry about the implementation of any of the nodes. You simply need to ensure that you are constructing the correct types of nodes with the correct parameters during the parse.

The parse functions are *speculative* when invoked. For example, if `parseWhileStmt` is called, it does not mean that there definitely is a while statement upcoming. Rather, `parseWhileStmt` will first determine whether it thinks the upcoming token matches a `while` statement – specifically, if the current token is the `while` keyword. If the current token is not a while token, then `parseWhileStmt` will simply return an empty (null) `shared_ptr` to denote that a while statement is not a match for the current token stream.

However, suppose that `parseWhileStmt` encounters a `while` keyword. That means that the token stream can only be valid if the rest of the `while` statement is correct. However, if something in the `while` statement is malformed, `parseWhileStmt` should throw an exception to denote there was an error in the parse. The most common exception you will manually throw is a `ParseExceptMsg`, though there are other types of exceptions as defined in `parse/ParseExcept.h`. These exceptions then must be caught somewhere, ideally as deep in the call stack as possible. This is because the parser should catch as many errors as possible in one pass. It would be very annoying for end users if it simply stopped on the first error in a source file that had ten errors.

In this example, if you look at the partial implementation of `parseStmt` (in `ParseStmt.cpp`), you will see that there is a `try`/`catch` that catches any exceptions of type `ParseExcept` and then uses `reportError` to actually report the error. Errors that are reported are then written to stderr upon conclusion of the parse.

There are also several helper functions that aid with the parsing. The `peekToken` function simply returns the current token in the token stream. If there are multiple possibilities you want to peek for, you can use `peekIsOneOf`, which takes in an (braces formatted) `initializer_list` of tokens to choose from. Similarly, `getTokenTxt` returns the actual C-style string for the current token, which is important for tokens that could be a variety of strings, such as identifiers. The `consumeToken` function “eats” the current token and then moves on to the next token that’s not a new line or comment. Since a very common operation is to peek for a specific token and then consume it if it is that token, this combined functionality is in the `peekAndConsume` function.

For cases where you absolutely require a specific token, or sequence of tokens, you can use `matchToken` or `matchTokenSeq`. These functions will match and consume either one or a sequence of tokens. The difference between these functions and `peekAndConsume` is that `matchToken` and `matchTokenSeq` will throw a `TokenMismatch` exception in the event of a mismatch. Returning to the `while` statement example, if you `peekAndConsume` a `while` token, you know for a fact that the next token must be an open parenthesis in a valid program. So you can use `matchToken` to process the parenthesis.

The last set of helper functions that have some use in error correction is the two `consumeUntil` functions. These functions mostly are used by the catch blocks. If you look at the code for `parseStmt` again, you will see `consumeUntil` is used to skip all tokens until the next semi-colon. This way, if there’s a major error in a statement, the parser can attempt to continue onto the subsequent statement.

In this assignment, you should only have to edit `ParseStmt.cpp` and `ParseExpr.cpp`. A handful of the functions in these files are already implemented, but you will have to implement the rest. Specifically, you should not modify `parseDecl`, `parseAssignStmt`, `parseExpr`, `parseExprPrime`, and `parseIdentFactor`.

In order to validate that your parser works correctly, there is a parsing test suite. To run the test suite run the following command from inside the `tests` directory:

```
python3 testParse.py
```

When you start this assignment, none of the 22 tests will pass. The tests are broken down into two types. There are some tests which intentionally cause parse errors, and these tests confirm that your parser identifies the same errors. The other set of tests should parse and generate an AST – the expected AST is compared against the AST your program generated. Note that several of these parse tests are actually semantically invalid USC programs, but in PA1 we aren’t checking for semantic validity.

Upon conclusion of this assignment, all the tests in the test `testParse.py` test suite will pass.

You also can run USCC directly from the command line to look at a specific test case. In order to output the AST of a program, you can run the command with the `-a` switch. It’s best to test this through the Visual Studio Code debugger. To do this, you will need to edit the `launch.json` file’s `args` array.

For example, the default `args` are:

```
"args": ["-a", "test002.usc"]
```

This means it will output the AST for `test002.usc`. To test the output for a different program, just change the file name in the `args` array.

Once you’ve implemented the first three items in the implementation section, the AST output for test002.usc should be:

```
Program:
---Function: int main
------CompoundStmt:
---------CompoundStmt:
------------ReturnStmt:
---------------ConstantExpr: 0
------------CompoundStmt:
---------------ReturnStmt:
------------------ConstantExpr: 1
---------------ReturnStmt: (empty)
```

Alternatively, if there are errors you will see a list of errors. For example, *after implementing the first three items in the implementation section*, if you try to compile `parse02e.usc`, you should get this output:

```
parse02e.usc:11:6: error: Function name 123 is invalid
void 123()
     ^
parse02e.usc:19:17: error: Additional function argument must follow a comma.
int func2(int a,)
                ^
parse02e.usc:24:1: error: Missing argument declaration for function func3
{
^
parse02e.usc:27:14: error: Unnamed function parameters are not allowed
int func4(int)
             ^
parse02e.usc:31:14: error: Unnamed function parameters are not allowed
int func5(int[)
             ^
parse02e.usc:37:1: error: Expected: ) but saw: {
{
^
6 Error(s)
```

Implementation

You should implement the parsing functions in the order outlined in these instructions. This will allow you to verify the functions work as you go along.

parseCompoundStmt

The interior of a compound statement can begin with zero or more declarations and is followed by zero or more statements. So, you’ll want to use `parseDecl` and `parseStmt` to find the declarations and statements. The `ASTCompoundStmt` node has `addDecl` and `addStmt` to append the declaration/statement nodes as children. Remember that because nodes handled as shared pointers, you have to `use std::make_shared` to dynamically create instances of this (and any other) node.

parseReturnStmt

Next, implement `parseReturnStmt`. In USCC, return statements can either be `void` or return a value. When you implement this, you need to update `parseStmt` so that it calls `parseReturnStmt`, as well.

parseConstantFactor and parseStringFactor

If you look at the current implementation of `parseAndTerm` in `ParseExpr.cpp`, you will see that it directly calls `parseFactor`. This is actually not correct as per the grammar, but for the moment we just want to skip all the other grammar rules and jump directly to $Factor$.

You should implement `parseConstantFactor` (which is for constant numeric expressions) and `parseStringFactor` (which is for string expressions). Then, you should update `parseFactor` so that after checking for an `ident` factor, it checks for constants and strings, as well.

You should now have three additional tests pass: `test002`, `parse01e`, and `parse02e`.

> **NOTE**
> Since the parse error test cases do a text match on your output, you will need to use the same error message format and syntax as the test cases.

parseParenFactor

Next, implement `parseParenFactor` and hook it up to the `parseFactor` function. Now the `test003` test should also pass.

parseIncFactor and parseDecFactor

Now implement `parseIncFactor` and `parseDecFactor`, and hook these up to `parseFactor`, as well. You should now also pass the `test004` test case.

parseValue

You should now implement `parseValue`. Once you’ve implemented `parseValue`, you should change `parseAndTerm` so it now calls `parseValue` instead of `parseFactor`. Now the `test005` test should also pass.

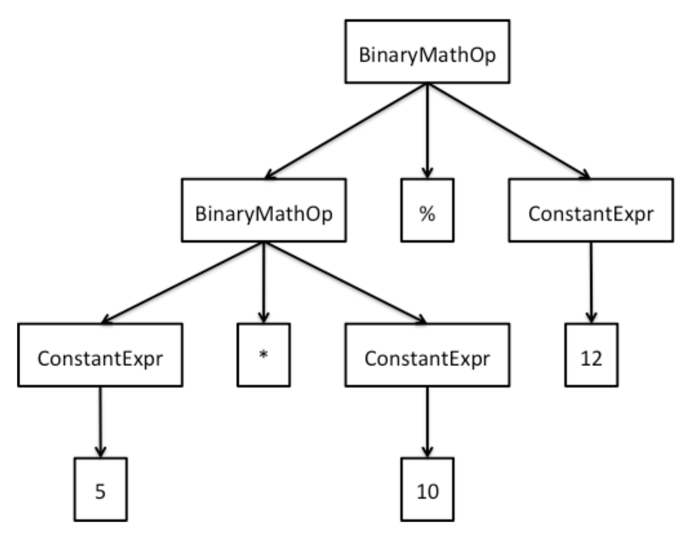
parseTerm and parseTermPrime

Now implement `parseTerm` and `parseTermPrime`. The reason there is a function for term prime is because the term grammar rule is left-recursive. So, you will need to transform the grammar so it is right-recursive instead (as discussed in class), and then implement these two rules.

There’s one aspect of `parseTermPrime` that may be a little confusing. Suppose `parseTermPrime` sees a `*` token. In this case, it means that a valid program must have an RHS value. So you can parse the value for the RHS operand. However, it’s possible that after you get the RHS operand, it is followed by another term prime, since it’s possible the expression might be along the lines of:

5 * 10 % 12

In the above case, this branch of the AST needs to have the modulus node at the top:



You need this *rightmost derivation* to ensure that binary operators with the same precedence are evaluated left to right in a post-order traversal of the AST. However, *by default a recursive parser will produce a leftmost derivation*. To solve this issue, after grabbing the RHS value you need to call `parseTermPrime` again to see if there’s another term prime. If there is another term prime, you should actually return that `ASTBinaryMathOp` and not the one you originally found. This guarantees a rightmost derivation as above.

Next, you should update `parseAndTerm` so that it calls `parseTerm` instead of `parseValue`. You should now pass `test006`.

parseNumExpr/Prime

You should now implement the rules for `parseNumExpr` and `parseNumExprPrime`. Since these are also binary operators, you need to handle the leftmost derivation as with `parseTermPrime`. Then update `parseAndTerm` so that it calls `parseNumExpr` instead. You should now also pass `test007` and `test013`.

parseRelExpr/Prime

These are very similar to `parseNumExpr`/`Prime`, except they are relational operators. Once you update `parseAndTerm` so that it calls `parseRelExpr`, you should additionally pass `test008`.

parseAndTerm/Prime

One of the last expression rules to implement is `parseAndTerm`/`Prime`. Same rules as all other binary operators apply. Once you’ve implemented this, you will also pass `test009`.

parseWhileStmt

Now back in `ParseStmt.cpp`, implement `parseWhileStmt` and update `parseStmt` so that it also checks for `while` loops. You should now also pass `test010`.

parseExprStmt and parseNullStmt

Now implement these two types of statements and hook them up to `parseStmt`. You should then also pass `test011`, `test015`, and `test016`.

parseIfStmt

The last statement type to parse is `if` statements. Remember that `if`s may or may not have an `else` associated with them. Once hooked up to parseStmt, you should now pass every test other than `parse06e`.

parseAddrOfArrayFactor

The last thing to implement is back in `ParseExpr.cpp`. Implement `parseAddrOfArrayFactor` and hook up `parseFactor` to check for this as well. You should now pass all the tests.

Testing on GitHub

Once you pass all the tests locally, you should push your code to GitHub and manually run the PA1 unit tests. If they don’t pass or your code compiles, you should make the necessary fixes and try again.

For this assignment, there are a total of 22 test cases. Each test failure is -5 points.

Submitting Your Assignment

Once you have a GitHub Actions run you are happy with, to submit the assignment you need to submit this form on Gradescope. You will have to provide the *full URL* for the actions run you want us to grade, and also tell us if you are using any of your four slip days.

> **DANGER**
> Make sure you submit this form, as otherwise we will not know that you have a submission you want us to grade. We only grade based on your GitHub Actions run and code on GitHub, and we will not download everyone’s repo. If your code does not compile or you do not submit the form, you will get a 0 on the assignment.

Making a Git Tag

Once you’ve submitted your assignment, you should make a tag so it’s easier to go back to the specific pa1 submission later, should you need to:

```
git tag pa1
git push origin --tags
```

Conclusion

If you pass all the tests, you now have a working parser for the USC language. However, if you look at the test programs, you will see that the majority of them are actually not valid USC code. This is because the parser only checks whether the file conforms to the context-free grammar for the language. In the next assignment, you will add semantic checks which will ensure that the program conforms to all the rules of the language.