USCC
Programming Assignment 1 – Recursive Descent Parsing

By: Sanjay Madhav
University of Southern California

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, which is in the same directory as this document. You will need to repeatedly refer to the grammar definitions while working on this assignment, so you may want to print out those pages from the language reference.

The scanner for USCC has already been provided to you. It is implemented via flex, and the input file is `scan/usc.l`. The list of tokens are in `parse/Tokens.def`, and they are defined in an enum. Provided your code is using namespace `uscc::scan`, the tokens can be accessed via a `Token::` prefix. For example, `Token::Key_while` would access 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`.

You will notice that the `Parser` class has a large number of functions prefixed with `parse`, such as `parseProgram`, `parseStmt`, `parseCompoundStmt`, and so on. These functions are the recursive descent parsing functions, most of which you will implement in this assignment. Each of these functions returns a `std::shared_ptr` to a specific type of *abstract syntax tree* (or AST) node. The AST is used as the initial intermediate 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 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 or not 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 (or 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 should then be caught somewhere, ideally as deep in the call stack as possible. This is because ideally, 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 particular 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 are provided to 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 `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 these functions 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 a 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 will be 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 on the subsequent statement.

In this assignment, you should only have to edit `ParseStmt.cpp` and `ParseExpr.cpp`. Furthermore, a handful of the functions in these files have already been implemented for you, but you will have to implement the rest. Specifically, `parseDecl`, `parseAssignStmt`, `parseExpr`, `parseExprPrime`, and `parseIdentFactor` should not need to be modified.

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:

```
$ python testParse.py
```

When you start this assignment, only one of the 23 tests (`Err_Parse03`) will pass. The tests are broken down into two types. There are some tests which are intended to 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 of the tests in the test suite should 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.

For example, if you wanted to output the AST of `test002.usc`, you could run the following command (again, from the tests directory):

```
$ ../bin/uscc -a test002.usc
```

Once you've implemented the first three items in the implementation section, this command will output:

```
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, if you try to compile parse03e.usc, with the following command:

```
$ ../bin/uscc -a parse03e.usc
```

You will get the following 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

It is recommended that you 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 a 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: `AST_002`, `Err_Parse01`, and `Err_Parse02`. Note that the error tests will fail if your error messages are not identical to the expected ones, so you may have to edit your error messages so that they match the test cases.

parseParenFactor

Next, implement `parseParenFactor` and hook it up to the `parseFactor` function. Now the `AST_003` 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 `AST_004` 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 `AST_005` test should also pass.

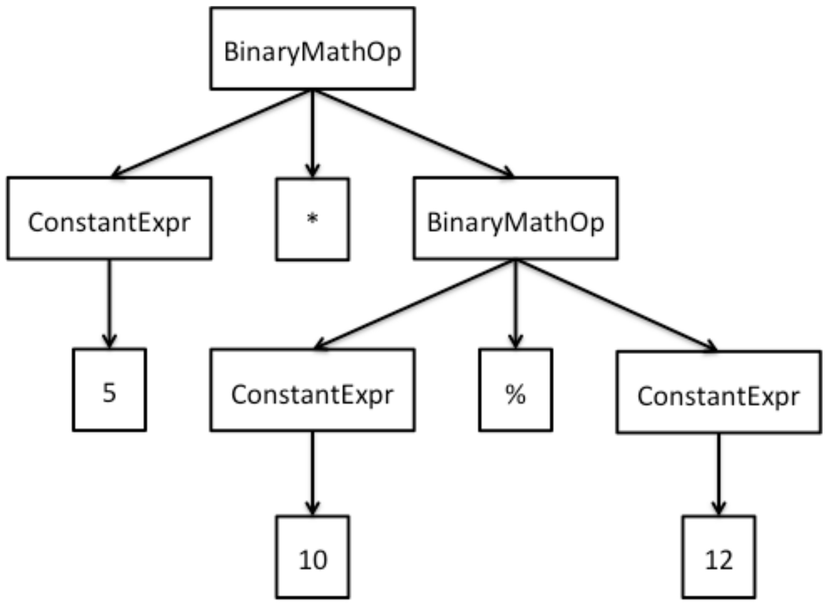
parseTerm and parseTermPrime

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, 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 should look like this:



You need this *leftmost derivation* to ensure that binary operators are evaluated left to right. To solve this problem, after grabbing the RHS value you need to call `parseTermPrime` again, and 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 leftmost derivation.

Next, you should update `parseAndTerm` so that it calls `parseTerm` instead of `parseValue`. You should now pass `AST_006`.

parseNumExpr/Prime

You should now implement the rules for `parseNumExpr/Prime`. 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 `AST_007` and `AST_013`.

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 `AST_008`.

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 `AST_009`.

parseWhileStmt

Now back in `ParseStmt.cpp`, implement `parseWhileStmt` and update `parseStmt` so that it also checks for while loops. You should now also pass `AST_010`.

parseExprStmt and parseNullStmt

Now implement these two types of statements, and hook them up to `parseStmt`. You should then also pass `AST_011`, `AST_015`, and `AST_016`.

parseIfStmt

The last statement type to parse is if statements. Remember that ifs may or may not have an else associated with them. Once hooked up to `parseStmt`, you should now pass every test other than `Err_06`.

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.

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 or not 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 of the rules of the language.