Clayton Green (230089109)

kgreen1@unbc.ca

October 20, 2013

CPSC 425 - Compiler Project Phase III

Introduction

Included is Phase III. It consists of classes for the creation of abstract syntax trees and an abstract syntax tree visitor. The print visitor is fully implemented and prints out AST from their root. Error recovery is broken, as in AST's are malformed or incomplete after an error has occured.

Participation

Clayton Green - All

Project Status

Scanner - Completed
Parser Basic - Complete
Parser Full - Unfinished
+ Add more sophisticated error recovery
+ Current error recovery causes problems with AST

TODO
+ Fix --err/-e appending to file rather than overwriting
+ Add functionality for --out/-o
+ Strip filename from file path to append to line trace
+ Parser needs more comments
+ Use makefile instead of shell script

Architecture and Design

The Parser class parses incoming Tokens from the Scanner class of a C*13 source file. The parsing is started when Parser.parse() is executed. This method recursively parses the source file until an ENDFILE token is encountered or until no more error recovery can happen. When parsing an abstract syntax tree is created and returned. This is done recursively and each LL1 grammar production returns an ASTNode.

The ASTNodePrintVisitor class traverses an AST in a depth first fashion. All of the traversal and printing of the nodes is removed from the actual nodes into this class. The root node accepts a PrintVisitor and that node depending on what children it has sends this printer down to them as well through the node's visit method.

The abstract syntax tree is made up of 5 main node types: DeclarationNode, ExpressionNode, StatementNode, ParameterNode, and ProgramNode. DeclarationNodes are the nodes that describe declarations in the C*13 language. These include function and variable declarations. ExpressionNodes are the expressions of the C*13 langauge, like literals and variables. StatementNodes are the statements of the C*13 language, like if-statements and return statements. ParamterNodes are special variable nodes for function declarations. And finally the ProgramNode is the root of the AST and the C*13 program.

Implementation

Limitations:

Error recovery breaks AST as it returns null pointers and unfinished nodes.

Error recovery doesn't take advantage of first or follow sets. Only moves on to the next semicolon when an error occurs.

Building and Use

Extract:

```
$ tar zxf green-parser-basic.tar.gz     -- Compressed tar
$ tar xf green-parser-basic.tar         -- Standard tar
```

This extracts the project into a directory named green-parser-basic.

Build:

```
$ cd green-parser-basic
$ ./build.sh      -- make sure build.sh can be ran as an executable
```

This will build the compiler and put the executable in the folder

```
$ /green-parser-basic/executables/Compiler
```

Usage:

```
$ ./Compiler {-l | -p | -s | -t | -c} {-q | -v} -I <filepath> { -I <filepath> }* [-o]
```

Option Listing:

-h | --help	*Display usage info and exits*
-v | --version	*Display version and exit*
-l | --lex	Process up to the Lexer phase
-p | --parse	Process up to the Parser phase
-s | --sem	Process up to the Semantic Analyser phase
-t | --tup	Process up to the Tuple phase

-c | --compile	Process all phases and compile (default)
-q | --quite	Only display error messages (default)
-v | --verbose	Display all trace messages
-I | --input	File to be compiled
-e | --err	Error file (defaults to STDOUT)
-o | --out	Output file (defaults to STDOUT)

Examples:

$ ./Compiler -p -v -I ./../Tests/input/parser/correct.cs13
-I ./../Tests/input/parser/incorrect.cs13 -e ./../Tests/results/parser/out.txt

This will run the files *correct.cs13* and *incorrect.cs13* through the compiler up to the parser phase (*-p*) with trace messages (-v). Then output the trace messages and error messages to *out.txt*. (Note: this is when running Compiler from the executables directory).

Code

Parser Class

You can create a Parser object, p, by,

Parser p(filepath, filename);

where *filepath* is the path to the source file to be parsed and *filename* is the name of the file for error and trace purposes. Parsing on the source file begins when *p.parse()* is called. This parses the source file, using the productions from the LL1 grammar, until the endfile token is received or when too many syntax errors have occurred where the arser cannot move on. The user is responsible for checking if the parser is good to parse (i.e. the source file was successfully opened) by calling *p.canParse()*.

The trace method was created to reduce duplicate trace code. It is used to enter production methods and takes three arguments: a filename, a reference to the calling object and a pointer to the function that object is to call. Because each production method has the same arguments and return value, only one trace method was needed. A trace message ENTERING is printed when the function is called and LEAVING is printed when the pointer function returns. To make it more readable a define macro was used to call the function.

Every production of the C*13 LL1 grammar produces a ASTNode pointer, and recursively builds the AST.

ASTNodeVistor class

The AST visitor class follows the visitor pattern, in which it has an accept method for every ASTNode. This class is purely abstract (an interface).

ASTNodePrintVisitor class

This class can be used on the root node of an AST (program node),

```
ASTNode *root = parser.Parse();
root->Accept(new ASTPrintVisitor(&Messenger));
```

The PrintVisitor takes a messenger to print output to stdout or a file along with trace output, and error messages. The print visitor takes care of the traversal of the AST. There is an accept method for every subclass of ASTNode, that prints the node's information and prints its children.

ASTNode Class + Subclasses

```
+ASTNode
        +DeclarationNode
                +FunctionDeclarationNode
                +VariableDeclarationNode
        +ExpressionNode
                +BinaryNode
                +LiteralNode
                +UnaryNode
                +VariableNode
        +StatementNode
                +AssignmentNode
                +BranchNode
                +CallNode**
                +CaseNode
                +CompoundNode
                +ContinueNode
                +ExitNode
                +IfNode
                +LoopNode
                +NullNode
                +ReturnNode
        +ParameterNode
        +ProgramNode
```

*CallNode subclass of Statement and Expression
** Each ASTNode implements a Visit method, to allow traversal by the ASTNodeVisitor.

Tests and Observations

Tests are correct.cs13 and incorrect.cs13. Correct is a syntactically correct C*13 program and has every construct of the C*13 language in it. No errors are encountered and the AST is printed after the trace output if trace is enabled. The other file, incorrect.cs13, gives problems with the creation of the ASTNodes. Because of the error recovery, nodes can be returned unfinished.