**PROJECT Part 3 – Parser**

In Part 1 of the project, you designed a high-level programming language, the Green World Language (GWL) for the Green World application, whose goal is to allow a programmer to construct and change a world constructed of blocks, tubs and plants planted in the tubs. In Part 2, you worked on the Lexical Analyzer (Lexer), which transforms the source program that is its input into a token stream. The latter becomes the input to the Parser, whose construction is the next step in your project.

Diagram

Description automatically generated

My responsibility

Note that you will need a working Lexer to be able test the Parser, unless you want to construct token streams by hand, which is a lot of work! If you fix or make improvements to your Lexer while working on the Parser, please resubmit it with the Parser. If, similarly, you find yourself needing to make adjustments to the Lexical Description and Syntactic Description (Part 1), resubmit those too. Make sure you signal clearly what has been changed since the previous submissions.

**The Parser**

The Parser checks the incoming token stream against the grammar to determine if the input is syntactically correct and, if it is, generates a Concrete Syntax Tree (CST).

You will be coding a Recursive Descent implementation of a LL(1) parser. This parser works by following closely the grammar. Starting from the start symbol of the grammar, it predicts based on one token of input that it has access to, which RHS of a production it needs to use to rewrite the nonterminal on it LHS. Therefore, the **first thing** to do in this part of the project is to check that the grammar you have written, in addition to accurately describing your GWL, is suitable for such a parser implementation. Specifically:

* It must be free of left recursion.
* Each production must have RHSs that are pairwise disjoint on the first terminal.
* It must be unambiguous (of course).

The **second step** will be for you to create an acceptor. A Parser is really a combination of an acceptor and a tree constructor. The acceptor determines if the input is syntactically correct according to the grammar. If it isn’t then there is no point in building any output structures: anything that is built will be thrown away because execution will stop at the parsing stage.

The **third step** will be for you to add into your code snippets that, as the code is being checked, will construct the relevant pieces of the Concrete Syntax Tree. Hint: While with a recursive descent parser you check the grammar from the top down, you will be building the tree from the bottom up, i.e., from the leaves to the root.

The **fourth step** includes tasks that are normally performed in the Static Semantics phase, but we won’t have a full Static Semantics phase for this project. Instead:

* We will assume that your input programs don’t have any type issues (i.e., that everything will type check).
* You will need to check that all identifiers (variables, subprogram names) that are referenced appear in the code with their definitions before they are referenced.
* You will need to check that the actual parameters used in a function call match the formal parameters of a function definition (excluding type – again, we’ll assume no type-checking issues).

The **fifth** **step** will consist of transforming the CST into an Abstract Syntax Tree (AST), which involves removing all those nodes that are not immediately useful in generating code: rewrite chains of non-terminals, punctuation tokens that served to structure the input and the like. E.g., for the expression **id \* id + id**:

|  |  |
| --- | --- |
| **concrete syntax tree** | **ABSTRACT syntax tree** |
|  |  |

**Very Strong Recommendation**

Work bottom up: get the easy pieces implemented first (expressions, statements, etc.) and work your way up to the top of the grammar. It is better to get some things working from source input to AST than to try to do everything and nothing ends up working.

**Parser Input and Output**

Parser input is the token stream output by the Lexer. The Parser can either “call” the Lexer as a subprogram or call a “Lex” function that gets the next token. In order to be able to show where syntax errors occur, it is a good idea to put line numbers of tokens in the token stream. This may mean that you will need to modify slightly your Lexer output.

Parser output is either error messages (if syntactic errors are found) or parse trees. You will probably find it easier to show the parse tree by printing it sideways, i.e., with the root at the left edge of the screen and the children of a node indented by a constant amount, e.g., for the above trees a simple representation is shown below. You may also be able to find tree printing algorithms that do a more elegant job.

|  |  |
| --- | --- |
| **concrete syntax tree** | **ABSTRACT syntax tree** |
| E --- E --- T --- T --- F --- id  --- \*  --- F --- id  --- +  --- T --- F --- id | + --- \* --- id  --- id  --- id |

**Testing**

As you work through your parsing steps, build a suite of test programs that allow you to test all the productions of your GWL individually. The write a small number of programs that combine some of them. Finally try to run your original sample programs (given in Part 1), appropriately modified in light of any changes you made to the lexical and syntactic design. Show the results of each of the tests by printing out the CST and AST in the same file as the test input.

**Work Distribution**

In this part of the project, you are probably better off working together on the design of each component but splitting the implementation work.

**Team Captains & Team Captain Report**

Team captains for Part 3 are indicated with C3 in [this file](https://alakhawayn365.sharepoint.com/:x:/s/FA21SSECSC331501/ERydhtiYJLdKsPN86y5N5xQBiPuLJbsUr26H0M6PPwhKWg?e=aURq1D) on Teams. The report should indicate the team members and contain a paragraph (or more, if necessary) reflecting on team dynamics (successes, challenges, issues, etc.) for this delivery. Only the team captain submits the report.

**Submission & Grading**

The Team Captain submits to Jenzabar for the Team with a separate report on teamwork. Use the following filename: *TeamName*\_ProjectPart3 and *TeamName*\_ProjectPart3\_CapRep.

The entire project counts for 15% of your final grade. This delivery counts for 4% of your final grade, or 4/15of the project grade.

The contents of the submissions include documentation, grammar, code, and tests. If you are resubmitting revisions of Part 1 and/or Part 2, put those in a separate folder from Part 3 components.

The table below shows how you will be graded.

**The team captain should submit to Jenzabar, on or before the due date, a zip or rar file.**

Late penalties, as specified in the Syllabus, apply.

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **Documentation (Including Team Members)** | **5** |
| How to run the Parser (alone or along with the Lexer) | 1 |
| Input and output files used by the parser. | 1 |
| Description of interaction between parser and Lexer | 1 |
| Description of parser output format (tree) with examples | 2 |
| **Grammar Check (Include in Documentation)** | **7** |
| No left recursion | 2 |
| Pairwise disjoint check (show FIRST set for each production's RHS) | 3 |
| No ambiguity (explain why grammar not ambiguous) | 2 |
| **Parser** | **15** |
| Grammar coverage of acceptor | 6 |
| Concrete Syntax Tree construction | 6 |
| Error handling and reporting | 3 |
| **Static Semantics** | **7** |
| Check: Identifiers defined before used | 1 |
| Function call checks (actuals match formals) | 1 |
| Abstract Syntax Tree construction | 5 |
| **Testing** | **6** |
| Test suite coverage | 3 |
| Test suite clarity (test, expected output) | 3 |
| **TOTAL** | **40** |