**PROJECT Part 3 – Ready to Generate Code?**

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, which was Part 3 of your project. In Part 4, you get ready to generate code using as target language a simple assembly-like language that I have designed and you get to critique!

Diagram

Description automatically generated

My responsibility

**Before Generation**

Before you actually begin to generate code, you will need to make sure you have gone through the two steps that were labeled as “fourth” and “fifth” steps in Par 3 of the project. These steps are included in the Static Semantics phase, which we will do only partially. If you haven’t done that yet, this is the time to do it, so I repeat below what I wrote there a little differently.

The output of the parser is a Concrete Syntax Tree (CST). This kind of tree which follows the structure of the grammar and is useful for doing type checking, because the semantic rules follow the syntactic rules closely, allowing you to label the CST with type information and check for type conflicts. As mentioned earlier, we don’t need to do type checking (we have a simple type system and we’ll assume that your programs are written without any type issues). However, you still need to do a few things that cannot be expressed by a CFG and cannot be done by a parser.

1. 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 do this by traversing the parse tree and with the help of the symbol table. This is also the time deal with identifiers that may have the same name but occur in different scopes.

Now that you have a parse tree, you know what the structure of your program is and therefore also which scope(s) an identifier is defined in, and with which type and other characteristics in each scope. You also know if the use of an identifier occurs in the same scope in which it is defined or not. Therefore, the symbol table will need to have information about the scope an identifier occurs in. For you, this is simple: it’s either local or global, because we are not using nested scopes. But I have posted a set of supplemental slides under Unit 8 that shows what things would look like in the symbol table if you had multiple nested scopes. (Slide set:

08b\_Subprograms & Runtime in BSL Details&Variations\_SUPPLEMENTAL.pptx)

1. 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—you do the type checking manually).

Before trying to generate code, you will want to simplify 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** |
|  |  |

NOTE: You may find that the two operations described above (making sure an identifier is defined before it is used and checking that in a function call the actual parameters match the formal parameters) is easier to do on the simplified AST, rather than on the more verbose CST, but since they both carry similar information either is an option.

**TASK 1.** **Language Evaluation** (1/4 of Project Part 4)

The language you will generate to is described in the document “GW Assembly Language.docx”, posted with this project. It is a very simple assembly language that, apart from a little more structure and being symbolic, resembles quite a bit the Pseudo-Code Interpreter language we looked at the start of the course.

I think that the language design is fairly complete, but, as you try to generate your GWL programs in GWAL, you may find that there are some aspects of the language that are missing that would make your life significantly easier when it comes to translating programs for the GW task. I also made some design decisions that you may not approve of.

Referring back to your textbook’s section 1.3 on Language Evaluation Criteria, evaluate GWAL as a language for expressing GW problems. Yes, it is very low level, but other than that, how does it rate?

**TASK 2. Generation into GWAL** (3/4 of Project Part 4)

Write a small set of simple programs that jointly use all aspects of your GWL (if you have already written them for Project Part 3, so much the better; if not, here is your chance again) and generate programs use as many of the instructions in GWAL’s instruction set as possible. Specifically, I want to make sure that you develop programs which jointly, or individually:

* **Exercise the GWAL instruction set by**
  + Using some arithmetic operations – this should be easy to do.
  + Using conditionals (branching instructions) – this means your program(s) should check at least one of the constraints that must hold between the objects.
  + Using looping (looping) – this means that some operation will get repeated a certain number of times which is not fixed but depends, for example, on one or more of the dimensions of the world
  + Using function call to a built-in function (use stack manipulation instructions to set up an activation record and return from it)

For the function(s), you can either define your own function (e.g., to operate with fractions or perform an operation repeatedly based on the value of one of its input parameters), or write the definition of one (or more) of the primitive operations (e.g. *makeBlock*/*unmakeBlock* and the rest) and call it.

* + Input/output instructions
  + Correct/errorful
* **Express and use the object and types used by GWL using GWAL simple types**. These include:
  + The fractional type
  + Block, tub, earth, plant
  + The 3D world, if you represent it explicitly to place objects in it

P.S. I’m working on the GWAL interpreter but it’s a big job, so it is likely that I will not be able to complete it so that you can actually run the programs you write (but I’ll try!). I should at least be able to check syntax before you turn in your Part 4 and maybe even run before you submit your Part 5.

**Deliverables**

1. GWAL evaluation
2. GWL programs and their translation into GWAL
3. Your generation code (from CST or AST to GWAL)
4. Lexer and parser that work with the code in 3.

**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 4 are indicated with C4 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 should submit to Jenzabar, on or before the due date, a zip or rar file.**

The Team Captain submits to Jenzabar for the Team with a separate report on teamwork. Use the following filename: *TeamName*\_ProjectPart4 and *TeamName*\_ProjectPart4\_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 deliverables are specified above. Part 1, Part 2, and Part 3 submissions should go in a separate folder from Part 4 components. If these are being revised, please indicate what has been revised, but I will regrde revisions when you turn in everything with Part 5.

The table below shows how you will be graded.

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **GWAL Evaluation** | **10** |
| Criteria coverage | 4 |
| Well argued | 3 |
| Well written | 3 |
| **GWL & GWAL Programs** | **14** |
| Arithmetic | 2 |
| Branching | 2 |
| Looping | 2 |
| Function call, execution, and return | 5 |
| Input/Ouput instructions | 1.5 |
| Stopping with and without error | 1.5 |
| **Complex Type Representations** | **10** |
| Fractional | 2 |
| Builtins (block, tub, earth, plant) | 6 |
| Handling of world representation (implicit/explicit) | 2 |
| **Code** | **6** |
| Clear and well-organized | 2 |
| Works correctly | 4 |
| **TOTAL** | **40** |