Project 2

Due: Tuesday, April 23, 2024 by 11:59 PM

A. Overview

In this project, you will work with one or two other people to design a language of your own, describe its grammar, implement a translator for it, and provide some sample programs. You will also look at another group's language and write a couple programs in that language.

B. General Guidelines

- You may not work individually on this project. You must work in groups of 2, 3, or 4 people, and the quality of your final product should reflect the size of your group.
- The language you develop must be different from any other existing languages, but you can borrow ideas from languages you have seen. In particular, you should be careful to make your language differ substantially from your target language. That is, if you translate to Java, your language should differ significantly from Java.
- If you have done a similar project in another class (such as CSc 453), make sure that you do not reuse that project. I am sufficiently familiar with that project that it will definitely raise some red flags. This project is simpler.

C. Specific Expectations

I. The Grammar

Design a language and describe its syntax as a grammar in BNF. Your grammar must be **unambiguous** and include all the required language features.

II. The Translator

Using your grammar, implement a translator that parses your language and produces a runnable program in an existing language. You may choose that language. You can also choose the language you use to write the translator, as it can be different from your target language. **It is difficult to design a simple language that is substantially different from all other existing languages. The rule of thumb here will be that your language should be substantially different from your target language. For example, if you translate to Java, your language should differ substantially from Java. If you translate to Python, your language should differ substantially from Python. Note that "differ substantially from" does not refer entirely to how the code looks. Python and Java, for example, have some similar syntax but they differ quite a bit in overall design.**

III. The Programs

You are required to submit programs written in your language that show that your grammar is complete and that your translator works. You will need to provide correct programs to show that your translator processes them correctly, and you will need to provide incorrect programs to show that your translator catches errors. There is no specific limit to the size of these programs, but *every feature you want credit for needs to be exemplified in a sample program.*

IV. The Presentation
In this part, you will make a presentation (preferably a video) that will be your chance to showcase all your work and prove that it meets all the requirements. I strongly recommend you do this in video format, but a document may also be acceptable. Please be sure to follow all the guidelines carefully as this is one of the main things we look at to make sure your project works and is complete.

V. Stuff Related to Group Work
Working in groups can be challenging, but it is also a necessary skill for you to learn as much of CS-related work is done on teams. Some of the items you will need to submit are meant to help you with this process, as you will be required to submit project plans and evidence that you have communicated expectations among yourselves. You will also be asked to provide updates, which will include updates on the work distribution and how the team is working together.

VI. More Programs
In this part, your group will become bigger as you will be paired with another group. Each group will need to "teach" their language to the other group, who will then write a couple small programs. The details of these programs will be released a week before the deadline, but they will be simple enough that they can be done with the basic features required by the project.

D. Submission & Deadlines.

There are various smaller deadlines before the final deadline as noted below.
Note: I feel like I shouldn't have to say this, but just in case: All these deadlines are in 2024.
Note that for each item, there should just be one submission per group. Once I know who your groups are, I can set them up on D2L so you can submit once per group.

- **Friday, March 15, by 11:59 PM:** Email me at lotz@cs.arizona.edu to let me know the members of your group. I only need one email per group, but I need to know this soon so that I can set up the groups on D2L before the first item needs to be submitted there. If you need help finding a group, you may post on Piazza to try to find one. Anyone who does not have a group by March 15 at 11:59 PM will be automatically put in a group.
- **Friday, March 22, by 11:59 PM:** Meet with your group and complete the "Collaboration Plan". Submit it to D2L.
- **Friday, March 29, by 11:59 PM:** Submit "Initial Language Design & Grammar" to D2L.
- **Friday, April 5, by 11:59 PM:** Submit "Translator Update 1 & Additional Features Plan" to D2L.
- **Friday, April 12, by 11:59 PM:** Submit "Translator Update 2 & Presentation Plan" to D2L.
- **Friday, April 19, by 11:59 PM:** Submit "Additional Programs Check-in" to D2L.
- **Tuesday, April 23, by 11:59 PM:** Submit all project materials to D2L.

Early/Late Submissions.

- Except for the final submission, all submission deadlines above are final. If you do not submit the required item on time, you will forfeit those points.
- For the final submission, we will follow the policy in the syllabus:

- **Late Policy for Project:** The project can be submitted up to three days late. Any part of a day counts as a full day. Each day will result in a 5-point deduction from your project grade.
- **Early Submission Policy for Project:** You can also turn in the project early and earn a few extra points. You can earn one extra point per day, up to 5 days. This only applies to your final submission (i.e. if you decide to make changes and submit again, that is fine, but we will consider that your official submission). If you do submit your final submission early, email lotz@cs.arizona.edu so we know we can start grading. If you don't email, then you will not get the early points as the whole point is to allow us to get a headstart on the grading.

E. More Specifics

To design a full language is a lot of work. Your language is going to be very simple and cover only a small subset of a typical language. However, you will have the option to go above the basic requirements for a higher score if you want.

I. The Grammar

The grammar needs to be well-organized, in BNF, and should include all the required elements of the language, as well as any additional features you choose to add. It should also be unambiguous. You may simplify some parts of the grammar using regular expression notation, but only within reason. (For example, if you have variables that can use any combination of lowercase characters, you can express that more simply with a regular expression.)

II. The Translator

- How you write your translator depends on several choices you need to consider.
  (1) *What is your translation language* (i.e. the language you are using to write the translator)*?* The most straightforward approach here is probably to choose a language with good support for regular expressions, which you can use for the parsing process. Both Python and Java are good choices. However, there are other options. For example, if you feel comfortable in Prolog, it has some features that work well with this kind of program.
  (2) *How is your language designed?* Most of you will probably design a simple imperative-type language, which is acceptable. But you may want to consider a different kind of language that may actually make the parsing process easier. An example of this would be to design a stack-based language, which may make the initial design of the syntax and the use of the language more difficult but would significantly decrease the complexity of the parsing process.
  (3) *Will you write a compiler-type translator or an interpreter-type translator?* In a compiler-type, the input to the translator would be a text file containing a program written in your language, and the output would be a file containing a working program in your translation language, which can then be compiled/run. In an interpreter-type, the input is the text file, but instead of outputting a translated file, the translator would translate and run the program line by line. From what I've seen in past semesters, these are mostly equivalent difficulty-wise *unless you add specific features.* For example, if you include functions and function calls in

your language (which is not required but a possible feature you can add), this tends to be more difficult to implement with an interpreter-style translator than with a compiler-style translator. Keep in mind that “more difficult” translates into “more points” for this project, so don’t let this necessarily stop you from attempting it.

- A very simple example of what I want to see is as follows:
    - Assume my translator is written in Java and translates my language to Java.
    - It is also a compiler-style version.
    - Assume, as well, that my program is in a text file called `program1.txt` and my translator is in a file called `Translator.java.`
    - I should be able to do the following:
        - **`javac Translator.java`** `//This compiles the Translator code.`
        - **`java Translator.java program1.txt >> Program1.java`** `//This runs the Translator on the input program and pipes the resulting java program into the file called Program1.java.`
        - **`javac Program1.java`** `//This compiles the Program1 code.`
        - **`java Program1.java`** `//This runs the program code.`
- This is an example, and you may decide to follow a different procedure. Just make sure that is clear in your instructions, so we can run things. But overall, it’s easier for us if you keep it pretty simple.

III. Minimum Language Requirements

These are the features your language must support. It’s important not to leave any of these out as they may be necessary for later when another group will write programs in your language.

- integers, Strings, and booleans
- variables that can be integers, Strings, or booleans
- variable assignment
- basic integer expressions with basic operators: addition, subtraction, multiplication, division, modulus
- boolean expressions with basic operators: and, not, or
- comparison operators: greater than, less than, equal to
- conditionals: at least “if-then-else” and should allow for nesting and blocks
- loops: at least one kind of loop; should allow for nesting; should allow for blocks
- printing to output
- command line arguments or reading input from user–this is important for the programs the other group will write–basically you either need to allow the user to input something from the command line or as part of the program, and it should allow for both Strings and integers (note that the easiest way to do this is to just designate certain variables as command-line arguments, which then get translated to the appropriate syntax in your target language).

If you meet these requirements but do not add any additional features, your grade will likely be in the B-C range (depending on the overall quality of the work). If you want to earn an A, you will need to add some complexity to the project. The next section gives some ideas of what you can do.

IV. Added complexity. In this part, I will describe variations and additions you can include to earn more points.

- Implement an interactive system that allows the user to type in commands in your language and see the results. Please note that this would not replace the requirement that the translator can translate a full program from a file.
- Add additional features to the grammar, such as functions and function calls. There are other features you might add as well, and the points associated would depend on the complexity. (For example, adding a few more arithmetic operators will not earn you many points, as that is pretty easy.)
- Add a type-checking system.
- Add additional or more explicit error handling–i.e. more informative error messages, additional types of errors, etc. You have to make sure that it is your translator that is catching these errors and not the compiler/interpreter of the target language.
- Implement your own tokenization and pattern matching algorithm rather than using regular expressions. This is pretty difficult, so I don't recommend it unless you feel really comfortable with the tokenization and parsing processes, and you are familiar with pattern matching algorithms.
- Add an optional feature that explicitly shows the parsing process. The better this is, the more points you could get.
- Add a scoping system. This can be challenging but is an interesting exercise in problem solving and use of data structures.
- If you have ideas for other additions or variations, you may discuss them with me.
- **NOTE: Anything you add to the language must be illustrated with program example(s).**

V. Sample Programs

You need to illustrate that your translator works and includes all the features by providing program samples. There is no specification as to the size or complexity of these programs, but *every feature you want included in your grade must be shown in a sample program.* So all the required features of the language need to be shown. Anything you add needs to be shown. Any errors you want to show need to be shown. This means that you need to provide valid programs that work and invalid programs that produce errors. At the very least you must provide:

- Enough valid programs to illustrate all the features of your language.
- At least three invalid programs that produce *syntax errors* that are caught by your translator. Please keep in mind that not all errors are syntax errors. These programs must be errors that are due to bad syntax, which means that they should be caught

during the parsing process. There is no specific requirement for the output messages, but in general, more informative messages earn more points.
- Anything you add to the basic requirements must be illustrated in a program as well, including additional error handling (e.g. type errors).

VI. Presentation

The final presentation needs to be well-organized and uploadable. I recommend putting all your materials together in a well-organized folder and making a video to present all of the required aspects. You may also do a written document if you prefer, but it needs to be very clear and well-organized. The required parts of the presentation are listed below.

- Explain your language design choices. This needs to be detailed and show that you clearly thought about the choices you made. You should reference comparisons between your language and other existing languages, and you should reference language design details that we've discussed in class. I want to know *why* you made the choices you made.
- Explain your process. This needs to be detailed and include a discussion of any unexpected challenges you faced and how you solved them.
- Provide a tutorial of your language. This is key because another group will need to understand your language well enough to write some simple programs, so this should be complete and clear enough that someone can go through it and be able to write some simple programs in your language.
- Explain fully all of the artifacts you are providing to prove that your code works–this includes explaining what each of the sample programs is showcasing.
- If you are providing a video (which is recommended), I suggest you also run all the programs to show that they work properly. If you are not, make sure you provide clear instructions on how to run them–and keep it simple!
- Explain any additional features you want factored into the grading. This presentation is very important for the grading process, so use it to really showcase what you've done.
- Discuss additional features or any changes you would make to your language if you were to continue developing it. The purpose of this is for you to think about all the aspects of language design and reflect upon those choices and challenges.

VII. Additional Programs

As mentioned previously, you will be paired with another group later and expected to write programs in each other's languages. The grade for this section will factor in the effort from both groups, so I suggest you communicate with each other, provide good resources for helping them with your language and give feedback to each other to help improve the final product.

VIII. Additional Documents

There are some additional documents you will need to submit as indicated above in the intermediary deadlines. These are detailed in the other documents. Additionally, in your final submission, you need to include the "Final Work Distribution and Reflection" document.

F. Grading & Summary of Deadlines

Please note that the actual awarded points here will depend on the quality and complexity of your work. This is to give you an idea of how the points will be broken down and what the maximum number of points is in each category.

Item	Points	General Criteria	Deadline (by 11:59 PM)
Email with group members	5	On time	March 15
Collaboration Plan	5	On time, complete, and well-developed	March 22
Initial Language Design and Grammar	5	On time, complete, and well-developed	March 29
Translator Update 1 and Additional Features Plan	5	On time, complete, and well-developed	April 5
Translator Update 2 & Presentation Plan	5	On time, complete, and well-developed	April 12
Additional Programs Check-in	5	On time, complete, and well-developed	April 19
Final Submission			**April 23**
Grammar	10	Complete, Correct, Clear, Unambiguous	
Translator & Sample Programs	60	Complete, Compiles and runs, Sample programs are thorough and work correctly	
Extra Programs	15	Correctly written programs for other group, the programs they write work.	
Presentation	30	Complete, correct, and well-organized.	
Final Work Distribution and Reflection	5	Complete and well-thought-out **Note: This part will be submitted and assessed individually.**	