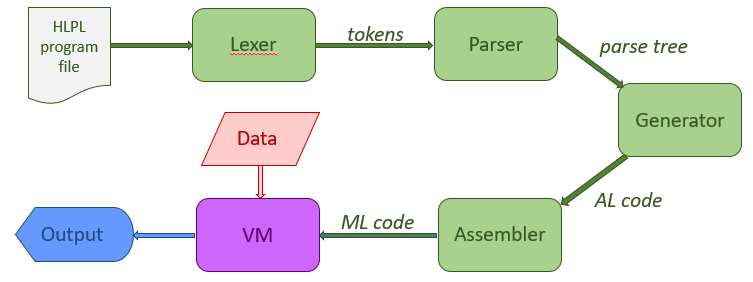
**PROJECT – Part 2 – due Friday, April 9 at 11:55 pm  
High-Level Language Design**

* **Introduction**

In this first part of this project, you started from the back end of your language translator. You designed an assembly language (AL), developed an assembler to translate it to a kind of machine language (ML), and constructed in interpreter to execute the ML. The step from ML to Assembly Language was one of the first in the development of higher-level languages, as it permitted the use of symbols and a syntax for instruction that was a little closer to how a programmer might think about telling the machine to compute. You worked on the back end, enclosed in the green rectangle below.



In this second part of the project, we will start from the other end. You will design a relatively high-level programming language (HLPL), which you will then need to perform lexical and syntactic analysis (and maybe some static semantic analysis) before going on to generate the AL code. The Lexer implementation is actually Part 3 of the project, but you should be thinking about it as you design the HLPL(which you can name as you wish, or just call HLPL). Its characteristics are described below.

**Note that, if you did not get the first part of your program fully working, this is your opportunity to do so. You can resubmit a working Assembler-Interpreter together with this part of the project. If you do, make sure you say so and describe what has changed.** Even if your previous submission was working fine, it is possible that you might find that you need to modify a little in order to be able to properly translate the HLPL to it.

* **HLPL Requirements**

**Code Format**

Unlike the AL, which had a fixed format for instructions in which operators/opcodes were separated from their operands and started in fixed columns, the HLPL is freer in its use of elements and their placement on the page. In that sense, it resembles more in spirit a subset of a language like C, even though its reserved words, operators, and punctuation may be different. You will get to choose the lexemes of the language and they should not be identical to C or Java or Python. You can be a little creative, use languages other than English as a base, etc. Whatever you choose, it should make sense.

**Language Elements**

The language uses an imperative computational model with some limitations.

* **Constants**: The language should have a mechanism for defining constants, i.e., named quantities that cannot change. Constants can be of any of the simple types or arrays of characters.
* **Types**: The language supports the following types:
  + **Simple**: addresses, integers, characters
  + **Structured**: arrays of integers or characters whose size can be set when a variable or constant of this type is defined.
  + No other types can be defined by the user.
* **Variables**:
  + Variables must be defined and can be of any of the types above.
  + Variables cannot change type (within the same scope). I.e., if you define variable **a**to be an integer in a function, within that function it remains an integer throughout.
  + Variables can be global or local.
* **Functions**: Units of code that perform a specific task, are named, have a clear interface for input and out put with the rest of the program.
  + Input will be done through parameters, but only parameters passed by value are supported.
  + Output will be done by returning a single result. Parameters and return result can be any of the simple types.
* **Operators**:
  + The assignment operator (do not use ‘=’; it is one of C’s bad design decisions)..
  + The standard arithmetic operators (**addition**, **subtraction**, **multiplication**, **division**) used for integers; division truncates.
  + The standard comparison operators (**equal**, **not equal**, **greater** **than**, **less than**) – up to you if you want to include **greater/less than or equal to**: they can be derived from the simpler ones.
  + The standard logical operators: **not**, **and**, **or**.
* **Expressions**: The type of an expression is determined by the operators and operands used.
* **Statements**:
  + The assignment statement.
  + A selection statement: you can have a 1-way selection and a 2-way selection statement, or just use a 2-way selection for both, depending on how you design it.
  + A repetition statement: a while-style loop is the most general and can take the place of other looping constructs.
* **Control Structures:** It is assumed that, like all imperative languages, unless the flow of control is modified by a conditional or repetition statement, the computation will follow the basic sequence control structure: one statement, then the next one, then the next one, etc.
* **Program Structure:** 
  + There should be a main program and a clear indication of where the main program starts and ends (note that C, Algol, Pascal, have used different ways of doing this).
  + The code should be block-structured, meaning that groups of related statements should be enclosed in delimiters (like **{ }** in C or **begin** **end** in Pascal).
  + It should be possible to nest a function inside another function.
  + It should be possible for a compound statement to contain variable declarations.
* **Comments**: You should provide a syntax for comments. You only need to provide comments that are one-line long, no need to bother with multiple line comments.

As you design your language, think about how you will need to describe it formally, given that you will need to parse it using a top-down recursive descent parser.

* **Language Description**

You need to provide a formal description of your language at two levels:

1. **The LEXICAL level:** This should use regular expressions.
   * What lexemes will you need to extract from an input file?
   * How do you define a legal user-defined identifier?
   * What numeric, character and string literals look like?
   * What operators and punctuation do you need to describe?

See also the grading table specific things we will be looking for in grading.

1. **The SYNTACTIC level:** This should use a context-free grammar in BNF or EBNF.
   * Your grammar should be able to generate any program in your language (i.e., it should not undergenerate).
   * Ideally your grammar should not overgenerate either. If it does, explain how you are going to catch input that should not be accepted.
   * Make your grammar pairwise disjoint. Think about this as you are designing your language!

* **Team Captains**

For this delivery, the team captain will be the person highlighted in yellow below. He/she has the following tasks:

* To make sure the team members are communicating and the project is advancing.
* To provide a brief report of advancement of the project by Monday April 5 at midnight (11:55pm) in Jenzabar.
* To provide a summary of what was done by whom when.
* To submit by the deadline.

**Captains**: You will personally get up to 2 points off your score (not your team members’ score) if you do not submit the report by April 5 and/or are not monitoring progress advancement. It’s your responsibility to keep the project advancing and on time for this phase.

|  |  |
| --- | --- |
| **TEAM** | **CAPTAIN** |
| **MMA** | MOUSSA, Hanane Nour |
| **SIFO** | DRIOUECH, Saad |
| **OMA** | KARMOUCHE, Mehdi |
| **DACY** | JAIDI, Driss |
| **MEBB** | BELFARSI, El Arbi |
| **TRNWRCK** | BOUKHARI, Nada |
| **Cup\_of\_Java** | BENCHEKROUN, Amine |
| **2As 2Bs** | BENABDALLAH, Yassir |
| **ABN** | ABOULRHIT, Mohamed |
| **MQ** | QASSIBI, Khaoula |
| **AB** | AZEDINE, Soufiane |
| **HISH** | SAKOUT, Hamza |

* **Submission and Grading**

The entire project counts for 15% of your final grade. This delivery counts for 2% of your final grade, or 2/15of the project grade.

The contents of the submission are specified in the table below.

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

|  |  |
| --- | --- |
| **CONTENTS** | **POINTS** |
| **Cover document** | **5** |
| Team members | 0.5 |
| Team captain for this delivery | 0.5 |
| Paragraph reflecting on team dynamics (successes, challenged, issues, etc.) for this delivery | 2 |
| Team captain mid-term report (by Monday April 5) | 2 |
| **General Language Design and Requirement Satisfaction** | **10** |
| Language Design overview and justification | 5 |
| Requirements Check | 5 |
| **Lexical Description - using Regular Expressions** | **9** |
| Handling of reserved words | 1 |
| User-defined identifiers | 1 |
| Numeric literal syntax | 1 |
| Character literal syntax | 1 |
| String literal syntax | 1 |
| Operators | 1 |
| Punctuation | 1 |
| Comments | 1 |
| Whitespace | 1 |
| **Syntactic Description - CFG in BNF or EBNF** | **16** |
| Grammar correctensss | 7 |
| Small example program (a few lines) including comments/what it computes | 2 |
| Medium example program (1/2 page) including comments/what it computes | 3 |
| Longer example program (1 page at least) including comments/what it computes | 4 |
| **TOTAL** | **40** |