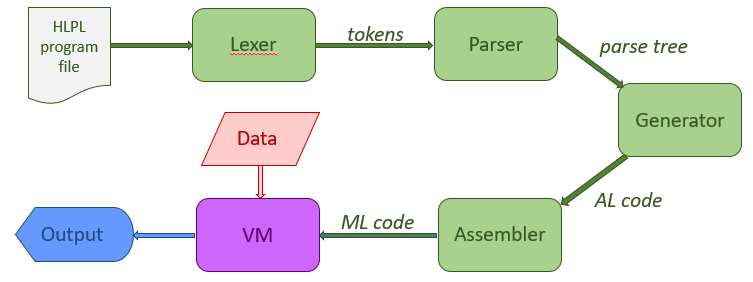
**PROJECT – Part 4 & 5 – due Monday, May 17 at 11:55 pm  
Parser, Parse Tree, Code Generator and Subprograms**

* **Introduction**

In Part 1 of the project worked on the back end of your language translator, designing an assembly language (AL), developing an assembler to translate it to a kind of machine language (ML), and constructing and interpreter to execute the ML. This was the back end, enclosed in the green rectangle below. In Part 2 of the project, you looked at the other end, designing a (relatively) high-level programming language (HLPL) according to certain specifications, in which to write your program file (the orange rectangle in the figure below. The next step was to work on the Lexical Analyzer (Lexer or Scanner) for your HLPL. In Part 3 of the project, you concentrated on building the Lexer to work with your HLPL. We now need to bridge the gap between the different parts by building the Parser and Generator, at least for a subset of the language. These are the parts framed in blue below.



**Note that, if you did not get the first part of your program fully working, or you want to improve your grade, this is an opportunity to do so. You can resubmit an improved version of Part 1 with this submission. 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.

**You may also want to resubmit Part 2 of your project with it, to fix issues in the grammar, although you do not yet have to worry about scoping and nested blocks, because we won’t address them in this part of the project.**

* **Part 4: CODING (5%)**

**The First Step: An LL(1) Recursive Descent Program Acceptor**

The first thing you want to do in this part of the project is work on the parser, but before you worry about building the parse tree, you should just write the code that determines whether a given input is syntactically correct according to your grammar.

The acceptor you write should use recursive descent and obviously must work with a grammar that is not left recursive and passes the pairwise disjointness test for every rule. This is why I gave you the exercise in Assignment 3. If you cannot get that working correctly, you will have to do the checks by hand.

You also have a choice of writing the acceptor completely by hand or trying to generate it (semi-) automatically using Racket or another PL from a computer readable specification of the CFG. The code is so strictly tied to the CFG rules, that (mostly) automatic generation of it is possible.

**The Second Step: Design your Concrete Parse Tree**

Before going on, take the time to design what should go in your parse tree. For each rule and RHS of each rule in the grammar, there should be a corresponding piece in the parse tree design.

Remember that the output of the parser is a concrete parse tree, which follows the CFG very closely. You will have a chance to clean up the parse tree in a later step.

**The Third Step: Modify your Acceptor to Construct Pieces**

Knowing what your parse tree should look like, you can now go back into your acceptor code and insert calls to functions that will construct the parse tree and return relevant pieces of it so that they can be inserted into larger pieces as the subprograms that parse the different non-terminals return control to their callers.

**The Fourth Step: Static Semantics (sort of)**

In this step, you will perform the three major functions of the static semantics phase by going through your program parse tree with the help of other data structures, e.g., the literal table and the symbol table:

1. Check that identifiers are declared and defined if they are used:
   1. A variable or parameter must be defined before it’s used.
   2. A named constant must be defined before it’s used.
   3. A function/procedure must be declared (the prototype given, if you use prototypes) or defined (the whole function is provided) before it’s used. If not, you may need to do a second pass through the parse tree, depending on what you stored in the symbol table.
2. Check that your expressions and assignments contain the correct types of operands. You will do this in a somewhat home-brewed style, not using attribute grammars.
3. Transform the concrete syntax tree into an abstract syntax tree by removing all unnecessary nodes:
   1. Removing chains of non-terminal rewrites   
      (e.g. the E => T => F => id gets replaced by id)
   2. Moving meaningful operators/functions to subtree root position.

I.e., instead of the at the left, you should get the abstract syntax tree at the right

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

**The Fifth Step: Code Generation**

In this last coding step, you will generate Assembly Language (AL) code using your AL design. You will need to generate:

1. Instructions to allocate and initialize global memory
2. Instructions for code in blocks (main, function/procedure)
3. Instructions for subprogram call and return (in a restricted fashion)

It is strongly suggested that you approach this part incrementally, not trying to address all of the HLPL constructs at one but adopt a “divide and conquer” approach.

We begin by simplifying and assuming that you will only generate code for programs where functions are not nested within functions (like C) and where there are no nested nameless blocks (unlike C), i.e., where compound statements do not contain variables. We leave some of these complications to Part 5 (below), where you only have to design for them but are not required to achieve a working program.

First, assuming that you did not use a C-like design where even the main program is a function, get the generator working for a program that does not use any function calls. This program does not have local variables, only references to global variables. This will allow you to concentrate first on initializing and changing global memory and on generating code for your different control structures, assignment statements, and expressions.

Second, get the generator working for subprogram calls and returns, but only for subprograms that do not use any parameters or local variables, so you can focus on generating AL code for the basics of stack management. You may find yourself needing to change your assumptions about how you manage data memory and needing to add some instructions in the AL and ML as a result. This would then be an opportunity to improve your design.

Third, add parameters and local variables and assume no non-local references (functional style): all information the subprogram needs is passed in through parameters. Information is returned via parameters that are addresses and the value returned by the subprogram (if it returns a value).

Finally, add non-local references which, in your case, will only by references to variables defined in the global scope. Since there are no nested functions and there are no nested nameless blocks, there can only be global variables and, in functions, local variables and parameters.

**Very Strong Recommendation**

If you do not manage to do everything, you will get a much better grade by having a subset of the HLPL working all the way across (i.e. from parsing through code generation), than by trying to do a complete job of the parser and then having the other components not working. So you should be thinking in terms of applying an Iterative Model to this development: get some functionality working all across the 5 steps (and, of course, generate correct AL). Then go back and add things, once again making sure they work all the way across the 5 steps, and so on.

* **Implementation Language**

You can choose between, but it should be Java, C, C++, Python, Common Lisp, or Racket.

* **Part 5**

In this step, you will read Chapter 6 of Bruce Mac Lennan’s Principles of Programming Languages: Design, Evaluation, and Implementation 3rd Edition. **Please note that some of the material from this chapter (Sections 6.3, 6.4) may not be directly something you want to use for this part, but you will still need to understand it for the final exam.**

You are asked to design but are not required to code:

1. How you would implement static scoping in your language.
2. How you would implement dynamic scoping in your language.

In both cases, what is being asked is to think about what AL code would need to be generated for local and non-local references for variables only. You do need to consider nested nameless blocks, but you do not need to consider functions defined within other functions. All function definitions will be placed at the file level and their scope will be global.

* **Work Distribution**

In this part of the project, you should be working together on the entire project, at least on the design, before each one of you goes off and does their part.

**Team Captains**

To ease the pressure of too many deliverables before the end of the semester, Parts 4 and 5 of the project have been merged and both are due on May 17. Part 5, instead of requiring coding, will now only require thinking and, a result, more weight has been shifted to Part 4. Nonetheless, to encourage you to not procrastinate about the project, I will ask for two Team Captain interim reports.

For this delivery, the team captain will be the person indicated 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 in Jenzabar by 11:55pm of
  + Wednesday, May 5
  + Wednesday, May 12
* 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 reports on time 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, Leila Farah |
| **SIFO** | EL AMRANI, Omar |
| **OMA** | BEN SEDDIK, Ismail |
| **DACY** | CHAOUNI, Mohamed |
| **MEBB** | MORTAJI, MOHAMED |
| **TRNWRCK** | JABALI, Iliass |
| **Cup\_of\_Java** | SAKOUT, Hamza |
| **2As 2Bs** | ATANANE, Othmane |
| **ABN** | ZAHER, Hamza |
| **MQ** | QASSIBI, Khaoula |
| **AB** | BENKHALIFA, AIS |

* **Submission and Grading**

The entire project counts for 15% of your final grade. This delivery counts for 7% of your final grade, or 7/15of 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** | **4** |
| Team members | 0.5 |
| Team captain for this delivery | 0.5 |
| Paragraph reflecting on team dynamics (successes, challenged, issues, etc.) for this delivery | 1 |
| Team captain mid-term report (by Monday April 5) | 2 |
| **Part 4: Parser (Acceptor plus Parse Tree)** | **15** |
| Brief documentation, including parse tree, external to code and in the code | 3 |
| Completeness: degree to which it covers grammar | 6 |
| Correctness: the parts that are completed work correctly | 6 |
| **Part 4: Static Semantics** | **13** |
| Brief documentation external to code and in the code | 3 |
| Type checking | 3 |
| Definition checking | 3 |
| Abstract Syntax Tree Design | 4 |
| **Part 4: AL Code Generation** | **23** |
| Brief documentation external to code and in the code | 3 |
| Main program and global variables only | 5 |
| Subprogram call and return (no parameters or local variables) | 5 |
| Subprogram call and return with parameters & local variables but no non-local references | 5 |
| Non-local references (to globally defined variables). | 5 |
| **Part 4: Testing** | **7** |
| Show output of each for your three programs and/or other programs, to demonstrate that the functionality you have implemented works. | 7 |
| **Part 5: Design for Nesting** | **8** |
| Static scoping | 5 |
| Dynamic scoping | 3 |
| **TOTAL** | **70** |