CPS2000 - Compiler Theory and Practice
Course Assignment (Part II) 2019/2020

Department of Computer Science, University of Malta
Sandro Spina

July 30, 2020

Instructions

- This is the description for Part II of the **resit** assignment. Please note that the description has not changed from the original. Also, the instructions outlined on this first page remain unchanged except for the submission deadline date.
- This is **an individual** assignment and carries **50%** of the total mark for the unit.
- The submission deadline is the **15^{th} September 2020**. A soft-copy of the report and all related files (including code) must be uploaded to the VLE by midnight of the indicated deadline. Hard copies **are not** required to be handed in. Source and executable files must be archived into a single .zip file before uploading to the VLE. It is the student's responsibility to ensure that the uploaded .zip file is valid. The PDF report (separately from that for Part I) must be submitted through the Turnitin submission system on the VLE.
- A report describing how you designed and/or implemented the different tasks of the assignment **is required**. Tasks (1-4) for which no such information is provided in the report will **not be** assessed.
- You are welcome to share ideas and suggestions. However, under no circumstances should code be shared among students. Please remember that plagiarism will not be tolerated; the final submission must be entirely your work.
- You are to allocate approximately **40** hours for this assignment.

Description

In this assignment you will be extending *SmallLang* (the specification of *SmallLang* can be found in Part I of the resit assignment) into *SmallLangV2* to include a few additional language features. Moreover, you will be carrying out research on compiler-compiler tools (e.g. ANTLR, JavaCC, Bison, etc) and use any one of them to create additional parsers for both versions of the language. This assignment (Part II) is composed of four major components: i) modifications to the original *SmallLang* language, ii) development of additional Lexer/Parser programs for *SmallLang* and *SmallLangV2* using compiler compiler tools, iii) the development of tools which transform the AST generated by the compiler compiler parser for *SmallLang* into an AST generated by your compiler from Part I and iv) a report detailing how you designed and implemented the different tasks. These components are further subdivided in tasks and explained in detail in the Task Breakdown section.

The following is a syntactically correct *SmallLangV2* program:

```
ff XGreaterThanY(int toCompare[]) : auto {
 let ans:bool = false;
 if (toCompare[0] > toCompare[1])
    {
       ans = true;
    }
 return ans;
}

ff Average(float toAverage[], int count) : float {
  let total:float = 0.0;

  for (int i = 0; i<count; i = i+1)
  {
    total = total + toAverage[i];
  }

  return total / count;
}

let arr1[2]:float;
let arr2[4]:float = { 2.4, 2.8, 10.4, 12.1 };

arr1[0] = 2.4;
arr1[1] = 6.25;
print arr1[1];                          //6.25
print XGreaterThanY(arr1);              //true
print Average(arr2);                    //6.92
```

Task Breakdown

Task 1 - Extending *SmallLang*

In this first task you are to extend *SmallLang* into *SmallLangV2* by adding another primitive type, 'char', to represent individual characters and also include support for arrays. An array in *SmallLangV2* consists of a series of elements of the **same** type in contiguous memory that can be individually accessed using the indexing operator [ ] over the array identifier. By default arrays are left uninitialised when declared, i.e. element values are not individually set. The programmer has the option of initialising the elements in the array upon declaration. Check out the code fragments above for examples of how this can be done. Deviations from the syntax used in this code fragment are allowed as long as you support the required functionality. These extensions are to be added as new (or modified) rules in the EBNF specification of *SmallLang* provided in Part I.

[Marks: 15%]

Task 2 - *SmallLangV2* Lexer and Parser

In this task you are to extend the table-driven lexer and hand-crafted predictive parser you have developed for the *SmallLang* language in Assignment (Part I) in order to handle the new primitive 'char' and array functionality. A successful parse of the input should produce an abstract syntax tree (AST) describing the structure of a valid *SmallLangV2* program. Array identifiers (e.g. counts[]) can be represented using an AST node similar to the one used to represent identifiers (e.g. count) for primitive types. **Note:** This task can be carried out entirely even if not all of the productions of *SmallLang* are implemented in tasks 1 and 2 of Assignment (Part I). Also, I recommend you create a copy of the sources (Lexer and Parser from Part I) and work on these copies (fork) for this task.

[Marks: 35%]

Task 3 - Tool generated parsers

Compiler-compiler tools such as ANTLR, JavaCC and Bison take a grammar specification and generate program sources, the parser, for this grammar. For this task you are to carry out research on ANTLR (or another such tool if you so prefer) in order to create ANTLR specific grammar specifications for both *SmallLang* and *SmallLangV2*. The ANTLR specification for the latter case will be an extension of the former.

In order to determine the correctness of the generated parsers you are to parse a few example programs which utilise features from the languages and check whether the ASTs (or parse trees) generated are valid. In both cases you should try to compare them with the ASTs generated by your hand-crafted parsers. You are to come up with these example programs. You can use any tools (e.g. GRUN) which come with ANTLR (or other compiler-compiler tools) in order to help you visualise the AST (parse tree).

[Marks: 25%]

Task 4 - Hybrid Parser

For this task, you are to transform the ASTs generated by the *SmallLang* parser from Task 3 above into AST structures which can be input to the XML generation, semantic analysis and interpreter visitors from Assignment (Part I). In this way you will be replacing your hand-crafted parser for *SmallLang* with the automatically generated one. You can use any software engineering process deemed appropriate to carry out this transform - one option I'd recommend is writing a visitor class (ANTLR supports the visitor design pattern) to visit the AST produced by the compiler compiler tool to generate the required AST.

In order to evaluate this transformation between ASTs, you are to parse the programs used to evaluate Assignment (Part I) using this hybrid compilation pipeline as illustrated in Figure 1.

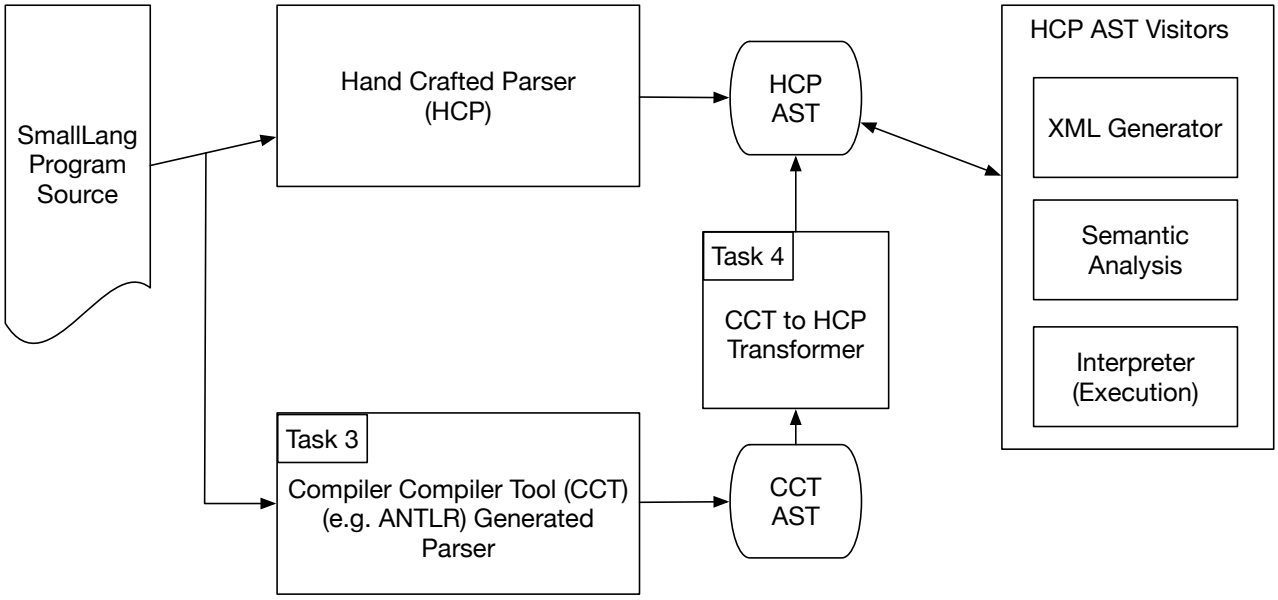

Figure 1: Hybrid parsing pipeline

[Marks: 25%]

Report

In addition to the source and class files, you are to write and submit a report (separate from the Assignment (Part 1) report). Remember that Tasks 1 to 4 for which no information is provided in the report will not be assessed. In your report you should include any sample *SmallLang* and *SmallLangV2* programs you developed for testing the outcome of your compiler. State what you are testing for, insert the program AST (pictorial or otherwise) and the outcome of your test.