The University of Newcastle

School of Electrical Engineering and Computing

COMP3290/6290 Compiler Design

Semester 2, 2018

Note: Parts 4 and 5 are submitted together for grading as ONE assignment:

Project Part 4 Semantic Analysis for CD18 Due: November 2nd

While logically a different phase, the simplest approach is for most of your semantic analysis to be done as you are building the Syntax Tree, and so will likely be done by adding functionality to your parser.

This phase determines which syntactically correct CD18 programs can be passed on to the code generator for translation into SM18 object code.

Project Part 5 A CD18 Code Generator for the SM18 Stack Machine Due: November 2nd

Only those CD18 programs that survive previous phases (*Parts 1, 2 and 4*) with no errors should be passed on to Part 5 for Code Generation.

Please note that the due date is the end of **Week 12** so you should plan your workload with your examination timetable in mind; absolute deadline is end of **Week 13**. If you need additional time beyond this, *email Dan as soon as possible*.

1. Part 4 (weight : 40%)

Output:

The only output for Part 4 is a program listing (to file – in the same folder as the compiler) and any errors for CD18 programs with semantic errors (to the terminal). It should be able to report both lexical errors and syntax errors without crashing, that specify the *type of error* and *line number* as a minimum.

Additionally you will output the number of errors found. Suggested formats for these errors are:

```
Syntax Error : line 27 – invalid if-statement (missing Boolean experession)
Semantic Error : line 54 – no return statement for function (int myFunction (int, int))

Found 2 errors
```

If no errors are found, you will simply report:

```
No errors found
```

…and the program should continue to **Code Generation**.

Semantic Checks Required:

The following *Semantic Checking* is to be implemented:

- `<id>` names (arrays and variables) must be declared before they are used;
- array size – must be known at compile time;
- strong typing exists for real variables, real arrays, boolean expressions, and arithmetic operations (such as `numeric ^ INTEGER`);
- valid assignment operations;
- actual parameters in a procedure or function call must match the type of their respective formal parameter in the procedure definition;
- the number of actual parameters in a procedure call must be equal to the number of formal parameters in the procedure definition;
- a function must have at least one return statement.

Additionally:

- <id> names must be unique at their particular block level (scoping)

Suggested target completion date is **October 12**, in order to give yourself plenty of time to complete Part 5.

2. Part 5 (weight : 55%):

Output:

Output for Part 5 will produce a valid SM Module File (using the *same filename as the source*, with the extension **`.mod`**) – assuming a valid source file is provided – in the same directory as the compiler, as well as a listing (with the extension **`.lst`**) to the same location.

For example, the source file `test.cd` will compile to a Module File called `test.mod`

Additionally, the contents of the compilation will be displayed in the terminal window. The output (both to file and on screen) will be formatted as per section ***The Structure of the Module File*** from the ***SM User Guide*** document.

The contents of the Module File will look similar to the form:

```
6
41 03 52 91 00 00 00 00
61 43 91 00 00 00 08 61
43 91 00 00 00 16 81 00
00 00 00 81 00 00 00 08
11 43 00 00 00 00 16 64
67 72 00 00 00 00 00 00
.
.
.
```

If compilation is successful, you will report a short message to the terminal stating that compilation has been successful; such as:

```
module-name compiled successfully
```

…where **`module-name`** is replaced with the name of your module file (sans **`.mod`** extension). Any compilation errors should also be reported to the Terminal, along with a message that informs the user that compilation failed.

3. Report (weight : 5%):

Accompanying your submission (regardless of whether or not you attempted only Part 4, or both Part 4 and Part 5) will be a Report. Your report will contain the following sections:

a) Any *changes*, *refactoring*, or *refinement* of the language that have been made (fixing *Left Recursion issues*, attempts to bring the grammar closer to *LL(1)*, etc.) through the project.

b) A *Semantic Analysis* Overview detailing:
   - what has been successfully implemented, and the approach used;
   - what Semantic Analysis has been attempted;
   - what Semantic Analysis is not complete or unsuccessful;
   - any assumptions made, with respect to Semantic Analysis.

c) Details of what aspects of *Code Generation* have been:
   - implemented successfully;
   - implemented partially (*please attach sample code to demonstrate*);
   - not implemented;
   - any *Machine-Independent Optimisations* you have implemented (such as *constant folding*).

d) A *½ to 1 page overview* on what extensions you feel would be useful to both the Language (CD) and the Machine Environment (SM).

Your report should be *formatted as a report*! This means you will include your *name* and *student number*, appropriate *headings*, and *address the above criteria*.

Your report should be between two and four A4 pages in length.

Marks will also be awarded in part for how closely your assessment of your implementation matches what has been accomplished – ie: tell the truth, get better marks in both the report, and the project parts. **Obviously the inverse also applies**.

Compiler:

Compilation and Execution:

Please ensure that your project can be compiled on the standard *University Lab Java environment*.

Part 4 and Part 5 will be compiled using the command line: `javac CD.java`
Part 4 and Part 5 will be executed using the command line: `java CD source.txt`
where `source.txt` will be the filename of a CD18 source file specified by the end user (note also, it *may or may not be a txt file*); **do not hardcode this filename**.

Testing Your Compiler:

You are responsible for making up sufficient data files which will adequately test your parser. You are encouraged to exchange CD18 source files, in order to thoroughly test your programs.

Like previous parts, it is recommended that you plan out your attack on this project; don't write the whole thing and then go looking for bugs – you will finish up with a

mess, impossible to read, understand and extend later. A short while writing (*henceforth useless*) debug routines will probably save you lots of time later.

Submission:

Project Part 4 and Part 5 (*Semantic Checking* and *Code Generation*), is due on **Friday November 2 at 23:59pm**. Please **zip** up all your files and submit them via the **Part 4 & Part 5 submission point** within the assessments tab on **Blackboard**.

Use a file name that contains your student number and "pt45" (for example: **`c7090832_pt45.zip`**), and put your name into the associated comment field for the submission.

Remember to include your *report*, and incorporate an *assignment cover sheet* into your submission.

Read Me File:

Your submission for parts 4 and 5 should include a Read Me file which contains those aspects of the Semantic Analysis and Code Generation that you believe that you have implemented, and *specifically it should list those aspects which you know that you have not implemented*. This can simply be extracted from your formal report.

If I do not know something is not working, I cannot effectively test other aspects, and your final mark will suffer – conversely if I know what to work around, I know what to expect and you'll bet better marks.

You may include CD18 source files that specifically demonstrate certain aspects of your compiler that are working (both for Part 4 and for Part 5).

DB **v1.01 2018-09-19** (originally issued : 2018-09-19)

Change Notes: any further additions will be highlighted, while redundant sections of text are ~~struck out~~. Changes will result in a change of version number, as seen above, as in the name of the file.

At this stage, we are at v1.0