The University of Newcastle
School of Electrical Engineering and Computing
COMP3290/6290 Compiler Design
Semester 2, 2018

Project Part 3   A Recursive Descent Parser   Due: September 21ˢᵗ

This is the next are phase of your *CD18 compiler* for the *SM18 machine*.

You were previously given the grammar for CD18, as a set of BNF rules. This will not necessarily be in a form that allows you to write the parser, and certainly is not LL(1).

This has now been extended to include the Syntax Tree Nodes (see *CD18 Node Rules*, in the *CD18 Language Specifications folder*) that need to be built as sections of the grammar are recognized. At the time of writing these specifications, the latest version of this document is **v1.0 2018-08-02**, and has been released on Blackboard.

A *TreeNode* class has also been released on Blackboard which you may refer to, or use, as you write your parser.

Transforming the Grammar for Recursive Descent:

You must first transform the grammar so that it is LL(1), and this will allow the recursive descent technique to be used. Most of this will be *simple left-factoring*, but there will be some left-recursive rules in the grammar which need to be transformed. While it is possible to code left-factoring *"into the parser"* rather than transforming those parts of the grammar, you will find it easier to do the full transformation to LL(1).

Note that the left-recursive rules must be transformed or top-down parsing cannot work.

The recursive descent technique is very useful in rapid prototyping a parser. It can be a little more difficult to respond to and recover from syntactic errors, as well as production-level parsers might be able to do, but the time saved in prototyping a parser that will correctly parse syntactically correct programs is well worth the effort.

The recursive descent technique simply requires a method for each non-terminal in the grammar with the initial call being to the start symbol of the grammar. As you process tokens from the scanner and organize them into the various syntactic structures of CD18, you will need to build these structures as a syntax tree, which will be used in subsequent phases for semantic analysis and code generation.

The leaves of the syntax tree will be the identifiers and constants that appear in the program (probably in the form of *references/pointers* into the respective *symbol table records*).

Yes, you will need a Symbol Table.

Error Detection and Reporting:

Error detection is straight forward and this technique is guaranteed to find a syntactic error at the earliest possible time within the parsing process, given a left to right reading of the input.

Error reporting is a *required feature of your parser*. These error messages will appear beneath your pre-order listing of the Syntax Tree (or Partial Listing, if you were unable to recover).

You should now begin to produce a program listing (if you didn't do so as part of part 1). You may report errors within the listing (but not in the middle of a line of source code; report errors on a separate line below the error code line), or you may produce a complete error listing at the end of the program listing (this must carry line numbers to refer back to the appropriate place in the source listing).

The error messages should be informative to the CD18 programmer (and the marker) and show that you understand the type of error that you have discovered.

Error Recovery:

Error Recovery is easier to do with other forms of parsing, but it is still possible to do this at quite a powerful level in a recursive descent parser. You should aim at least at recovering from a syntax error at the *statement level*, but it will also be possible to recover from syntax errors within expressions. The best measure of error recovery is *"how soon does the parser recover to the point where it can discover another unrelated syntax error?"*

Another good measure for effective error recovery is the amount of additional error messages that are generated where there are no real errors, however, remember that you cannot overcome every problem, and errors are mistakes, so in attempting error recovery, you are trying to be of assistance to a programmer who is error-prone.

In the final analysis the minimum that is required for a parser is that it builds correct syntax trees for correct programs and that it finds *at least* one error (*obviously the first*) in a program that contains errors.

If you cannot do this, then *semantic analysis* and *code-generation* are *impossible*.

Error Correction:

Error Correction is ***not required*** for this project, but may be attempted by those who wish. The key rule here is that the programmer has made a mistake, so don't try to correct everything. One possibility is to check for simple errors and let the programmer know that you've altered these to make the program "correct". There are other measures of this kind that can be taken (such as the insertion of the colon operator before a function return type, or inside a variable declaration).

For *COMP6290 students*, see section below.

For COMP6290 Students:

COMP6290 students will need to correct implement *Error Recovery* for the following instances:

- Syntactically incorrect/missing period;
- Syntactically incorrect/missing colon;
- Syntactically incorrect/missing comma, and;
- Syntactically incorrect/missing semicolon.

```
eg: constants PI=3.14159, E=2.71828. FIGENALPHA=2.5029
eg: x:integer. y;real, z:boolean
eg: x=2+7: y=PI-E;
eg: myArr[x],myValue;
…etc…
```

You must also output a *Warning*, for each corrected error at the end of your output, informing the user that the Compiler has made Corrections to the code, such as:

```
Warning – Error Correction (line 34) : corrected invalid
use of comma in expression: myArr[x],myValue;
```

Please ensure you clearly identify that you're a *COMP6290* student in your submission (such as in your readme file and on your coversheet), and where you have implemented this functionality.

This task will comprise *20%* of your total mark for *Project Part 3*.

Output for Syntactically Correct Programs:

You will display a *Pre-Order Traversal of the Syntax Tree* (to show the marker that you have produced it correctly). The output *node values* and *id's/literals* will be padded to multiples of *7 characters*, with the final character required to be a space – *for readability*.

The output will produce *10 columns* and begin a new line *after you complete the output for the 10th column* (see example in next section).

If you wish, you may also provide a *"pretty"* output using display tools such as XML.

You should produce a *program listing* as a prelude to this output (with errors for incorrect programs) as was stated above.
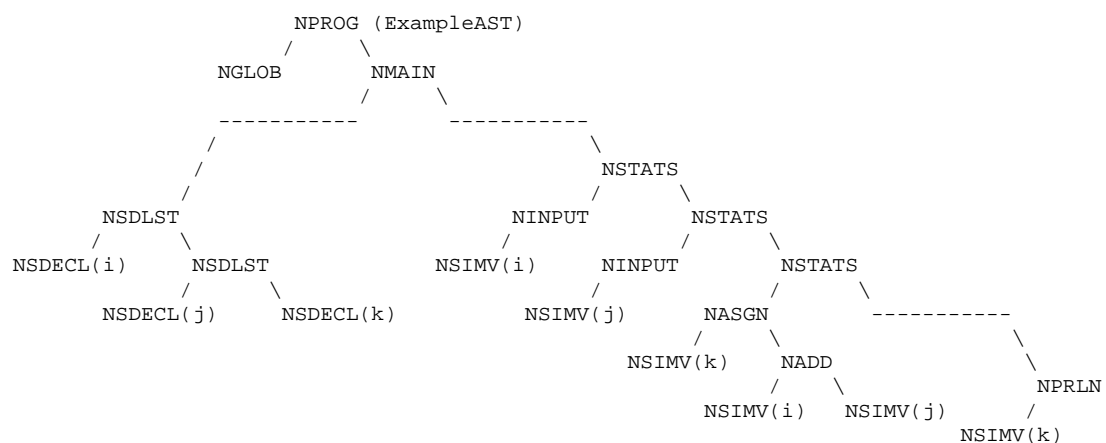
Output Example:

As an example, consider the following ***Code Example → Syntax Tree Visualisation → Pre-Order Traversal***. *You are not required to produce the Tree Visualisation – this is for the benefit of seeing a code-to-output example.*

(Note: I did this in my head; so if you find any errors, let me know! DB)

Code Example:

```
CD18 ExampleAST
main
i:integer, j:integer, k:integer
begin
input i;
input j;
k = i + j;
printline k;
end CD ExampleAST
```

Syntax Tree Visualisation:

```
                    NPROG (ExampleAST)
                    /        \
              NGLOB            NMAIN
                    /        \
         -----------        -----------
        /                              \
       /                                NSTATS
      /                              /        \
   NSDLST                      NINPUT          NSTATS
   /     \                    /         /        \
NSDECL(i)   NSDLST       NSIMV(i)    NINPUT        NSTATS
           /     \                  /        /        \
     NSDECL(j)   NSDECL(k)     NSIMV(j)    NASGN     -----------
                                          /     \                \
                                    NSIMV(k)   NADD               \
                                              /    \               NPRLN
                                         NSIMV(i)  NSIMV(j)       /
                                                            NSIMV(k)
```

Pre-Order Traversal:

```
NPROG   ExampleAST    NGLOB   NMAIN   NSDLST NSDECL i        NSDLIST NSDECL j
NSDECL k        NSTATS NINPUT NSIMV  i       NSTATS NINPUT  NSIMV    j
NSTATS NASGN   NSIMV   k       NADD    NSIMV  i       NSIMV    j       NPRLN
NSIMV  k
```

Testing Your Parser:

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 Part 1, 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.

Project Part 3 (*Parser*), is due on **Friday September 21 at 23:59pm**. Please zip up all your files and submit them via the **Part 3 submission point** within the assessments tab on **Blackboard**. Use a file name that contains your student number and "pt3" (e.g. `c7090832_pt3.zip`), and put your name into the associated comment field for the submission.

Remember to incorporate an *assignment cover sheet* into your submission.

Please ensure that your project can be compiled on the standard *University Lab Java environment*, using the command `javac A3.java`, and executed similarly with the command `java A3 source.txt`, where source.txt will be specified by the end user (note also, it *may or may not be a txt file*); **do not hardcode this filename**.

DB   **v1.02 2018-09-18** (originally issued : 2018-08-23)

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 v1.02
- Added more in-depth description of Output Standards (v1.01)
- Added code to output example (v1.01)
- Further clarified Error Reporting (v1.02)