

The University of Newcastle
School of Electrical Engineering and Computing
COMP3290 Compiler Design
Semester 2, 2020

Project Pt 2 A CD20 Recursive Descent Parser Due September 27th

This is the next are phase of your *CD20 compiler* for the *SM20 machine*.

You were previously given the grammar for CD20, 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 *CD20 Syntax Tree Rules and Node*, in the *CD20 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 dated August 1, and has been released on Blackboard.

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 CD20, 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).

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 CD20 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 close to 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*).

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 – but this is NOT required.

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

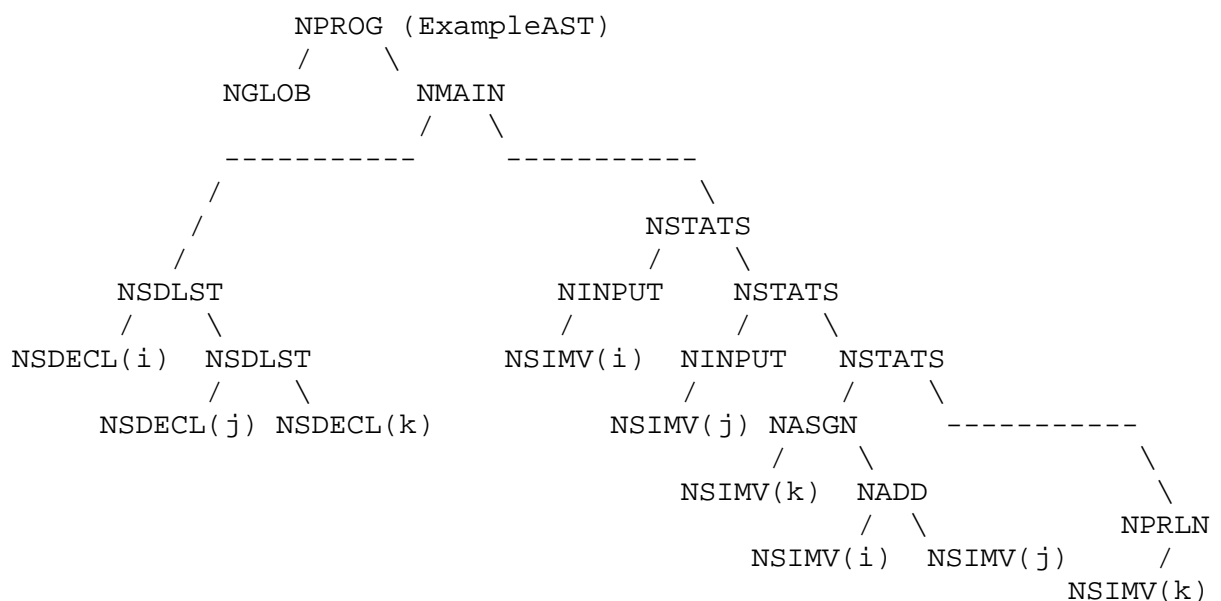
Output Example:

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.

Code Example:

```
CD20 ExampleAST
main
  i:int, j:int, k:int
begin
  input i;
  input j;
  k = i + j;
  println k;
end CD20 ExampleAST
```

Syntax Tree Visualisation:



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 CD20 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 2 (*Parser*), is due on **Friday September 27 at 23:59pm**. Please zip up all your files and submit them via the Part 2 submission point within the assessments tab on Blackboard. Use a file name that contains your student number (e.g. **c7090832.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 A2.java**, and executed similarly with the command **java A2 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**.