Assignment 2: Syntax Error Recovery

COMP 3010 – Organization of Programming Languages
February 2020

[**Note:** *This assignment is adapted from the original version authored by Prof. Michael Scott.*]

Problem description

During Assignment 1, you probably encountered a wide variety of error messages. The nature of these messages depends on both the language definition and the compiler or interpreter. You may have noticed that across languages and implementations, these messages differ greatly in their usefulness and specificity.

One feature common to all of the languages you used is syntax error recovery. In the simplest sense, syntax error recovery is the mechanism by which a compiler or interpreter continues to parse a program (and find more syntax errors) after it encounters an instance of invalid syntax.

Your task in this assignment is to implement syntax error recovery for an extended version of the calculator language discussed in the required textbook and in class. We are providing a basic ad-hoc scanner and recursive-descent parser (written in C). Given this initial code base, in order to receive full credit for this assignment you must:

1. Translate the code we provide into C++. Obviously, you must make any changes needed for the code to compile without errors under `g++`. In addition, you must replace any calls to C libraries (e.g., for I/O) with the standard C++ equivalents (no `printf`!).
2. Extend the calculator language with `if` and `do`/`check` statements, as shown in the grammar below. Note that you must augment both the scanner code and the parser code with the new tokens and nonterminals.
3. Implement exception-based syntax error recovery, as described in Section 2.3.5 on the textbook's companion site. At the least, you should attach handlers to statements, relations, and expressions.
4. Output an abstract syntax tree (AST) with the structure suggested (for a slightly different language) in Example 4.15 and Figure 4.12. Your output should be in linear form, where every subtree is represented (recursively) by a bracketed list, or a parenthesized tuple in which the first element (immediately inside the parentheses) is the root, and the remaining elements are its children, in order. More on this below.

When run, your program should read a calculator program from standard input, and then output either syntax error messages or a correct syntax tree.

The initial source code for this assignment is available on Blackboard. As currently written, it prints a trace of predictions and matches. You should disable that.

Extended Calculator Language

Here is an LL(1) grammar for the calculator language (in a format that we will be reusing in Assignment 3), extended with if and do/check statements (tokens/terminal symbols are indicated in red):

P → SL $$
SL → S SL | ε
S → id := R | read id | write R | if R SL fi | do SL od | check R
R → E ET
E → T TT
T → F FT
F → (R) | id | lit
ET → ro E | ε
TT → ao T TT | ε
FT → mo F FT | ε
ro → == | <> | < | > | <= | >=
ao → + | -
mo → * | /

Here, the new nonterminal R is meant to suggest a "relation." As in C, a value of 0 is taken to be false; anything else is true. The relational operators (==, <> [not equal], <, >, <=, and >=) produce either 0 or 1 when evaluated. A do loop is intended to iterate until some check-ed relation inside it evaluates to false— "check R" is analogous to "if (!R) break;" in C.

The FIRST, FOLLOW, and PREDICT sets for this extended grammar are shown on page 6 of this document. You'll need these sets in order to augment the recursive-descent parser code to recognize the extended calculator language.

As it turns out, if we assume that integers are unbounded, the extensions make the calculator language Turing complete (if still quite impractical). As an illustration, here is a program that calculates the first *n* primes:

```
read n
  cp := 2
  do check n > 0
      found := 0
      cf1 := 2
      cf1s := cf1 * cf1
      do check cf1s <= cp
          cf2 := 2
          pr := cf1 * cf2
          do check pr <= cp
              if pr == cp
                  found := 1
              fi
              cf2 := cf2 + 1
              pr := cf1 * cf2
          od
          cf1 := cf1 + 1
          cf1s := cf1 * cf1
      od
      if found == 0
          write cp
          n := n - 1
      fi
      cp := cp + 1
  od
```

Note that while every `check` statement in this program is immediately inside the `do`, that need not in general be the case.

AST output for the primes program might look like this:

```
(program
  [ (read "n")
    (:= "cp" (num "2"))
    (do
      [ (check > (id "n") (num "0"))
        (:= "found" (num "0"))
        (:= "cf1" (num "2"))
        (:= "cf1s" (* (id "cf1") (id "cf1")))
        (do
          [ (check <= (id "cf1s") (id "cp"))
            (:= "cf2" (num "2"))
            (:= "pr" (* (id "cf1") (id "cf2")))
            (do
              [ (check <= (id "pr") (id "cp"))
                (if
                  (== (id "pr") (id "cp"))
                  [ (:= "found" (num "1"))
                  ]
                )
                (:= "cf2" (+ (id "cf2") (num "1")))
                (:= "pr" (* (id "cf1") (id "cf2")))
              ]
            )
            (:= "cf1" (+ (id "cf1") (num "1")))
            (:= "cf1s" (* (id "cf1") (id "cf1")))
          ]
        )
        (if
          (== (id "found") (num "0"))
          [ (write (id "cp"))
            (:= "n" (- (id "n") (num "1")))
          ]
        )
        (:= "cp" (+ (id "cp") (num "1")))
      ]
    )
  ]
)
```

Indentation is shown here for clarity, and need not be generated by your code. The rest of the syntax is meant to mirror the likely internal structure of an AST in C++, and should be generated by your code. The square brackets delimit lists, which have an arbitrary number of elements. The parentheses delimit tuples (structs), which have a fixed number of fields. So, for example, an `if` node has two children: A relation and a body. The relation is a tuple with exactly three children: An operator and two operands. The body of the if is a list, whose elements are the statements that should be executed when the relation is true. Program and do nodes have only one child each—a list.

Suggestions

You do not have to build the syntax tree as an explicit data structure in your program in order to generate the correct output. You are welcome to build it if you want to, though.

We've given you a trivial Makefile. You should add to it a target test that causes make to pipe sample calculator programs (of your choosing) into your parser. This will make it easier for the TA to reproduce your tests.

When match sees a token other than the one it expects, it could simply throw a syntax_error exception. The resulting algorithm would recover by deletion only: The exception handler will delete tokens until it finds something in either the FIRST set or the FOLLOW set of the nonterminal corresponding to the current recursive descent routine. An attractive alternative is to mirror Wirth's recovery algorithm, and have match insert what it expects and continue (presumably after printing an error message). You may implement either strategy.

Division of labor and submission procedure

You may work alone on this project, or in teams of two or three. Be sure your write-up (README file) lists the names of your team members (if any), and describes any features of your code that the TA might not immediately notice.

To turn in your code, use the following procedure, which will be the same for all assignments this semester:

1. Your code should be in a directory called "<YourName>_OPL_A2" (for example, "TomWilkes_OPL_A2"). Put your write-up in a README.txt or README.pdf file in the same directory as your code.
2. In the parent directory of your A2 directory, create a .tar.gz file (so-called "tarball"), or a .zip file, that contains your A2 directory (e.g., "TomWilkes_OPL_A2.tar.gz" or "TomWilkes_OPL_A2.zip").
3. In the page for Assignment 2 on Blackboard, use the Submit button to upload your .tar.gz or .zip file. When you submit, give the name of your partner(s) (if any) in the submission comments field.

PREDICT sets for the extended calculator language

The following table shows the PREDICT sets for the extended calculator language. Compare this table to PLP Figure 2.23 (Chapter 2 Slide 49), which shows the PREDICT sets for the original (unextended) calculator language. The process for developing the FIRST, FOLLOW, and PREDICT sets is described in PLP Chapter 2 Example 2.35 (pp. 84-89).

Production	**PREDICT Set** (Equivalent to FIRST set, unless indicated as FOLLOW set)
P → SL $$	{ id, read, write, if, do, check, $$ }
SL → S SL	{ id, read, write, if, do, check }
SL → ε	{ $$, fi, od } [FOLLOW set]
S → id := R	{ id }
S → read id	{ read }
S → write R	{ write }
S → if R SL fi	{ if }
S → do SL od	{ do }
S → check R	{ check }
R → E ET	{ (, id, lit }
E → T TT	{ (, id, lit }
T → F FT	{ (, id, lit }
F → (R)	{ (}
F → id	{ id } (i.e., identifier)
F → lit	{ lit } (i.e., literal number constant)
ET → ro E	{ <, <=, <>, ==, >, >= } (relational operators)
ET → ε	{ id, read, write, if, do, check, fi, od }
TT → ao T TT	{ +, - } (add-ops)
TT → ε	{ id, read, write, if, do, check, fi, od, <, <=, <>, ==, >, >=,), $$ } [FOLLOW set]
FT → mo F FT	{ *, / } (mult-ops)
FT → ε	{ id, read, write, if, do, check, fi, od, <, <=, <>, ==, >, >=,), $$ } [FOLLOW set]
[ro, ao, & mo nonterminal productions are trivial, thus are not shown here for brevity]	