CS143 Handout #8
Autumn 2000

PS1: Top-down parsing

Due 11:59pm Tue Oct 17th
(can be submitted at most 3 days late)

The goal

This week's assignment is an exploration into the techniques and details of top-down parsing. It consists of two parts: the first is a small program that implements a recursive-descent parser and the second is the derivation of the table for a predictive parser. Both parts will be submitted together as one electronic submission. Although you can discuss ideas with others, you are expected to submit your own independent work for both parts. Please refer to web site for details of the collaboration policy as it applies to programs and problem sets to be sure you understand the ground rules.

Part a: recursive-descent top-down parser for SQL select statements

For part a, you are to design and implement a recursive-descent top-down parser that recognizes valid SQL select statements. SQL is the Standard Query Language used by many popular databases and the "select" command is used to query the database to find entries matching various criteria. For example, SELECT people FROM employees WHERE height > IQ queries the database to return the "people" column from the "employees" table where each row must meet the criteria that its "height" field is greater than the "IQ" field. It is possible to select multiple columns by separating the identifiers by commas or using * to mean all columns. The WHERE clause is optional, if present, it contains a condition that is a simple relational test between a field and an integer or another field. Here is a grammar that describes the (simplified) version of the select command that you are to recognize:

```
S         ← SELECT cols FROM table $ | SELECT cols FROM table WHERE criteria $
cols      ← * | col_list
col_list  ← col | col_list, col
col       ← id
table     ← id
criteria  ← field relop int | int relop field | field relop field
relop     ← > | < | =
field     ← id
```

Your program needs to accept syntactically correct queries and reject and halt on any other input.

The starting project directory /usr/class/cs143/assignments/ps1 on leland contains the following files (the boldface entries are the ones you will need to modify):

`Makefile`	builds project
`main.cc`	place your solution here
`scanner.h`	defines prototypes for scanner functions
`scanner.l`	skeleton file for your scanner
`tokens.h`	defines token constants
`samples/`	directory of test input files

Here's a suggested order of attack for building your parser:

1) Use lex to create your scanner. Given last week's excursion into lex for pp1, this should be a breeze for you, since there are only few token types and the patterns are not complex. A few sample input strings (the middle one is invalid, by the way):

```
SELECT books FROM library $
select * from Fruit where Is_Ripe$
SELECT lecturers, professors FROM cs WHERE fan_club_membership > 100$
```

You can assume that identifiers (e.g. column, table, and field names) start with an upper or lowercase letter and can be followed by any sequence of upper, lower letters and underscores. The keywords SELECT, FROM, and WHERE can be in upper or lower case, or a mix. Keywords may not be used as identifier names. Integers are sequences of digits, leading zeros are allowed. You can assume all input strings (valid or invalid) will have a $ to mark the end of input. Whitespace is ignored. Any characters that don't fit one of the token patterns should halt scanning with an error about "Unexpected char @ on line 4". Once you have defined your scanner, test it out by itself to ensure you are getting the correct tokens. The given main has a function for testing the scanner that might be useful here.

2) The grammar as given, is not in a form suitable for LL parsing. Before you start on parsing, you must convert this grammar to a suitable, equivalent grammar by eliminating all ambiguity, common left-factors, and left recursion.

3) Using your equivalent grammar, compute the first and follow sets. Place your new grammar and the first and follow sets in comments in your main program. Using these sets as a guide, implementing the individual parsing function for each of your non-terminals. Remember to get the next token, all you have to do is call `yylex()`. Notice we are asking for recursive-descent functions, not a table-driven parser.

4) Modify the main function to call the start symbol's parsing function, and to output whether it was valid. If the input is not syntactically valid, your program should print an error message at the point of failure and halt. The error message should be of the form ("Expecting SELECT, found SLECT instead"). During debugging, you may find it useful to print the chosen production each time the program chooses to expand a non-terminal. The output of your program would then be a leftmost derivation of the input string. You should remove such debugging code in the version that you submit. Our testing will expect only the output "Valid" or a single error message at the point of failure.

5) Test, test, and test on lots of various inputs. We provide some sample input files, but as usual, they won't cover everything, so you'll have to challenge yourself to handle all the possibilities.

Part b: constructing a predictive parsing table

Part b will give you some more practice with first and follow sets, and then going on from there to build the table for a table-driven predictive parser. The grammar you are to use is that of boolean expressions:

```
B       ← B and B | B or B | not B | (B) | term
term    ← true | false | id
```

1) Convert the grammar so it is LL(1). The precedence should be such that unary `not` has the highest precedence, followed by the binary `and`, and lastly the binary `or`.
2) Compute first and follow sets for all non-terminals in your converted grammar. It might be helpful to compute the nullable non-terminals first.
3) Create a parse table.
4) Trace through a parse of the input **`false or (not fun)$`.** Show the parse stack, remaining input, and parser action taken at each step like the example in the top-down parsing handout.

You do not have to do any programming for this problem, just build the table from which a top-down predictive parser could be implemented and use it to trace through a sample parse. You should submit an ASCII text file with the converted grammar, the first and follow sets, the completed parse table, and your trace of the parse.

Those cells in the parse table that represent errors should be left blank. (As an exercise for yourself, you might want to consider what an appropriate message would be for each error condition. It's also interesting to think about what action the parser might take so it can continue the parse: inserting a token, popping the parse stack, skipping input, etc.) In the table you submit, though, we want the error cells to be empty.

The text file needs to look reasonably neat both on-screen and printed so that we will be able to fairly evaluate your work. Please name this file "part_b" and submit it with the project directory containing the program for part a.

Grading

This problem set is worth 80 points, part a is worth 40 points and part b is worth 40. We will test part a by running it against a variety of tests and diff-ing its output versus ours. Part b will be graded using an ordinary problem-set-type criteria.

Electronic submission

You should electronically submit a project directory containing everything needed to build and run your program for part a, along with a text file containing your answer to part b. The submit instructions are detailed on web site at **http://www.stanford.edu/class/cs143/policies/**.