Parser Milestone - Due 02/14/14 (11:59 pm)

Objective

Objective 1 of this milestone is to produce the parser and continue the symbol table for the project.

Objective 2 is to use the data structure developed for the tokens.

Objective 3 to give you experience using the production system concept as a design tool.

Objective 4 is to give you experience in designing a program using a categorical style design.

Objective 5 is to give you practice in designing test files based on the formal definitions.

Professional Methods and Values

Design and testing. Use of formal method. Parsing has perhaps the strongest theoretical grounding of any subject in computer science. Here are two pages to help you review this material: Grammar and Production Systems and Chomsky Hierarchy review.

Assignment

The purpose of this milestone is to develop a working parser utilizing the scanner and symbol table.

Performance Objectives

1. Develop a formal definition of the recursive descent parsing algorithm from the grammatical structure for IBTL.
2. Develop a parser program from a formal definition.
3. Develop a data structure for the abstract syntax graphs.
4. Develop a test driver for the developed program .
5. Test the resulting program for correctness based on the formal definition.

The milestone report should concentrate on the development of the program. It must include a copy of your formal design docments.

Detailed Information

You are to develop the following.

- A scanner reads characters from a file or standard input. The file names are taken from the command line. The task of the scanner is to classify the terminal symbols as specified by the parser's grammar.
- A parser is a program that gets its input from the scanner. The parser has two tasks:

The parser verifies that the input meets the rules of the grammar.
The parser produces a list encoding the input.

- The main program sets up the input files by reading the command line flags. You should consider using flags to communicate debugging options. Any flag that does not start with the customary hyphen ('-') is taken as a file to be processed. The process loop is as follows:
  Set up the file to be processed so the scanner can read it.
  Call the parser to parse the input and to receive the output list produced by the parser.
  Echo print the list to verify the input is correct.

This subsystem is the input user interface and could have multiple input files. **Note: a file can have many expressions.** The loop outlined above must be continued until all files are processed.

Testing Requirements

The lists will be used eventually to fulfill future projects. For this project, you are to demonstrate that your parser can (1) properly process correct inputs and (2) reject incorrect inputs. To demonstrate that your parser works, develop a throw-away main program that prints out the tokens. You might want to keep the print subroutine as a debugging aid for later in the project. Here is a link to a few assignments to help with testing.

The Project Grammar

A grammar is a four-tuple (V,T,P,F), where V is a set of non-terminal symbols which are capitalized, T is a set of terminal symbols, P is a set of productions (seen below), and F is the start symbol. The convention is that the first set of productions is for the start symbol.

The tentative productions for our grammar are: IBTL-Grammar.pdf. The reason these are tentative productions because the grammar may be change based on factoring, eliminating left recursion, and adding/changing any current productions to make it easier for your parser.

Example Input.

Below are three "files". Show the derivation of each file.

file1:

[1 x [1 5]]

file2:

[[+ 1 [* [+ 2 3] 7]]]

file3

[[while [= 5 x] [:= x [- x 1]]]]

Specific Grading Points.

Constraints

You are to implement the parser as a recursive descent parser.

The scanner must be a subroutine called by the parser.

You must be able to read any number of input expressions - properly bracketed lists - made up of arbitrary number of sub-lists and properly constructed lexical elements.

You may not use counting of brackets as the parser algorithm.

Conditions

Recovery from input syntax errors is very hard. You can throw an error and exit (puke and die) or you may try to implement an error recovery strategy.

Testing Requirement

The program will be accompanied by test files. These test files are to be developed by you and should clearly demonstrate that the scanner and parser implement the complete specification. They should clearly show both correct processing of correct inputs and rejection of incorrect inputs.