Syms

a small Symbolic Parser

Pop Dan Ionut, 30434

Purpose

The purpose of the project is to implement a symbolic expression parser using yacc and lex(naturally the host language is C).
A general name for such programs would be “Computer Algebra System”(CAS).
In general such parsers provide functionality such as: perform possible computations, reduce an expresion to some standard form, symbolic operations(derivation, integration) or translation to a language suited for graphical representations (such as LaTex).

Description

The provided implementation handles a subset of the previous enumeration.
It can:

- Handle addition and multiplication, reducing identity operations (+0, *1)
- Group terms wihich differ only by a known(integer) factor
- Express products of sums as sums of products(this is the “standard form” assumed by the parser)
- Perform symbolic derivation with respect to a given symbol of multinomial expressions
- The derivation operation also identifies functions and marks their derivative as f’
- Assign expressions to symbols and treat a symbol either as an associated expression or as a bare symbol
- Recover from errors(inform user about error and discard erroneous input)

Some urging limitations(these are points of further development*)

- It does not handle real numbers
- It does not handle essential operations such as ‘-’, ‘/’ or power raising
- Symbols can not be expanded into expressions under the derivation operator

*Further development is not probable

Implementation

The grammar of the parser is as follows(using the same convention as yacc-nonteminals lowercase, terminals upper case):

start: expr | UNKN ‘=’ exp | ;/*empty expression is also ok*/

expr: term | expr ‘+’ expr | expr ‘*’ expr | ‘(‘ expr ‘)’
| ‘D’ ‘[‘ UNKN ‘,’ der ‘]’ /*there is a dirty trick in the implementation here –see below*/

term: UNKN | NUM | UNKN ‘(‘ UNKN ‘)’ | ‘!’ UNKN /*this last one is for expanding the symbol*/

der: der ‘+’ der | der ‘*’ der | ‘(‘ der ‘)’ | UNKN ‘(‘ der ’)’ | UNKN | NUM | ‘!’ UNKN
/*the last rule will only warn the user that the symbol is not expanded*/
(end of gramar)

It is important to point out that we can see the parser as having two quite distinct gramars: one for algebraic manipulation and one for differentiation.

The value of **der** non-terminals will be a pair of strings representing the derived expression and the initial expression(which is needed for implementation purposes)
The “dirty trick” mentioned above is that after a derivation rule is recognized, having the string representing the derivative, the algebraic part of the parser enters in action.
This is done by:

1. Pointing the input of the lexer to an in-memory file containing the derivatives text
2. The actual rule for the expression(which ends with ‘]’ in the gramar in this doc) actually continues with another **expr** non-terminal.

The 4 shift/reduce conflicts have been studied and, to my best knowledge, the default behaviour is allright.