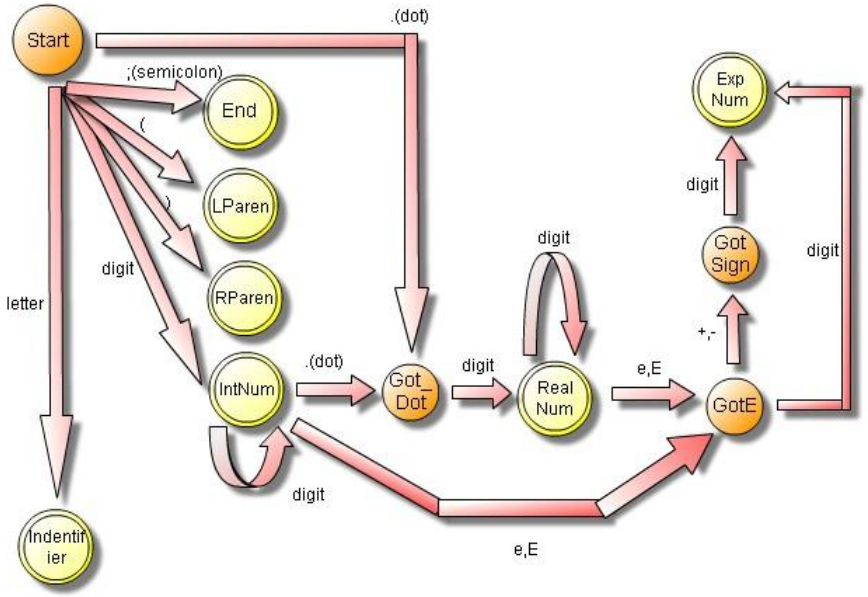
Descsription and Experience of my project

Xiaoqing Tang

September 29, 2008

This program can calculate the results of a sequence of expressions. The expressions can include variables and functions. Generally speaking, the program consists of a scanner, a parser, a variable table, and something miscellaneous.

SCANNER The scanner receives the input string (expression) and split them into tokens. I used a deterministic finite automaton (DFA) to accomplish this task. Here is the automaton:



Since the DFS is implemented by the "switch... case..." technique, the complexity is $\Theta(n)$ for an input of length n.

PARSER The parser analyzes the tokens from the scanner, and decide a parse tree to calculate the expression. Originally, I designed the following CFG

(context free grammar) for parsing:

$$
\begin{array}{rcll}
S & \rightarrow & EXPR0; & \\
EXPR0 & \rightarrow & EXPR1 & \\
 & \rightarrow & var = EXPR0 & //allowing a = b = 0 \\
EXPR1 & \rightarrow & EXPR2 & \\
 & \rightarrow & EXPR1(+|-)EXPR2 & \\
EXPR2 & \rightarrow & EXPR3 & \\
 & \rightarrow & EXPR2(*|/)EXPR3 & \\
EXPR3 & \rightarrow & EXPR4 & \\
 & \rightarrow & EXPR4\hat{}\,EXPR3 & //2\hat{}\ 2\hat{}\ 2 \text{ means } 2\hat{}\ (2\hat{}\ 2) \\
EXPR4 & \rightarrow & var & //variable \\
 & \rightarrow & var(EXPR1) & //function \\
 & \rightarrow & num & //constant \\
 & \rightarrow & (EXPR1) &
\end{array}
$$

So, we have now some left recursions and common prefixes. First we'll remove the common prefixes in EXPR4. The rule will be changed into:

$$
\begin{array}{rcl}
EXPR4 & \rightarrow & (EXPR1) \\
 & \rightarrow & varEXPR4x \\
 & \rightarrow & num \\
EXPR4x & \rightarrow & (EXPR1) \\
 & \rightarrow & \epsilon
\end{array}
$$

$\hat{}$ Then we'll remove the common prefixes in EXPR3. The rule will be changed into:

$$
\begin{array}{rcl}
EXPR3 & \rightarrow & EXPR4EXPR3x \\
EXPR3x & \rightarrow & \hat{}\,EXPR3 \\
 & \rightarrow & \epsilon
\end{array}
$$

Then we'll remove the left recursion of EXPR2. The rule will be changed into:

$$
\begin{array}{rcl}
EXPR2 & \rightarrow & EXPR3EXPR2x \\
EXPR2x & \rightarrow & (*|/)EXPR3EXPR2x \\
 & \rightarrow & \epsilon
\end{array}
$$

It's the similar case for EXPR1:

$$
\begin{array}{rcl}
EXPR1 & \rightarrow & EXPR2EXPR1x \\
EXPR1x & \rightarrow & (+|-)EXPR2EXPR1x \\
 & \rightarrow & \epsilon
\end{array}
$$

Last, we need to remove common prefixes in EXPR0 since $EXPR0 \rightarrow^* EXPR4 \rightarrow var$, and $EXPR0 \rightarrow var = EXPR0$, thus we have common prefixes to remove. First, we'll convert $EXPR0 \rightarrow EXPR1$ into a terminal-starting rule

$$
\begin{array}{rcl}
EXPR0 & \rightarrow & (EXPR1)EXPR3xEXPR2xEXPR1x \\
 & \rightarrow & varEXPR4xEXPR3xEXPR2xEXPR1x \\
 & \rightarrow & numEXPR3xEXPR2xEXPR1x
\end{array}
$$

Thus, we have common prefixes between the second rule and the assignment rule. So we'll modify it as:

$$
\begin{array}{rcll}
EXPR0 & \rightarrow & (EXPR1)EXPR3xEXPR2xEXPR1x & \\
 & \rightarrow & numEXPR3xEXPR2xEXPR1x & //thesearethesame \\
 & \rightarrow & varEXPR0x & \\
EXPR0x & \rightarrow & =EXPR0 & \\
 & \rightarrow & EXPR4xEXPR3xEXPR2xEXPR1x &
\end{array}
$$

So the grammar is done. The overall grammer would be like:

$$
\begin{array}{rcl}
S & \rightarrow & EXPR0; \\
EXPR0 & \rightarrow & (EXPR1)EXPR3xEXPR2xEXPR1x \\
 & \rightarrow & numEXPR3xEXPR2xEXPR1x \\
 & \rightarrow & varEXPR0x \\
EXPR0x & \rightarrow & =EXPR0 \\
 & \rightarrow & EXPR4xEXPR3xEXPR2xEXPR1x \\
EXPR1 & \rightarrow & EXPR2EXPR1x \\
EXPR1x & \rightarrow & (+|-)EXPR2EXPR1x \\
 & \rightarrow & \epsilon \\
EXPR2 & \rightarrow & EXPR3EXPR2x \\
EXPR2x & \rightarrow & (*|/)EXPR3EXPR2x \\
 & \rightarrow & \epsilon \\
EXPR3 & \rightarrow & EXPR4EXPR3x \\
EXPR3x & \rightarrow & \hat{} EXPR3 \\
 & \rightarrow & \epsilon \\
EXPR4 & \rightarrow & (EXPR1) \\
 & \rightarrow & varEXPR4x \\
 & \rightarrow & num \\
EXPR4x & \rightarrow & (EXPR1) \\
 & \rightarrow & \epsilon
\end{array}
$$

The $FIRST$ and $FOLLOW$ set would be:

	$FIRST$	$FOLLOW$
S	$\{(, var, num\}$	$\{\epsilon\}$
$EXPR0$	$\{(, var, num\}$	$\{;\}$
$EXPR0x$	$\{=, \epsilon\}$	$\{;\}$
$EXPR1$	$\{(, var, num\}$	$\{;\}$
$EXPR1x$	$\{+, -, \epsilon\}$	$\{;\}$
$EXPR2$	$\{(, var, num\}$	$\{+, -, ;\}$
$EXPR2x$	$\{*, /, \epsilon\}$	$\{+, -, ;\}$
$EXPR3$	$\{(, var, num\}$	$\{+, -, *, /, \hat{}, ;\}$
$EXPR3x$	$\{\hat{}, \epsilon\}$	$\{(, var, num\}$
$EXPR4$	$\{(, var, num\}$	$\{+, -, *, /, \hat{}, ;\}$
$EXPR4x$	$\{(, \epsilon\}$	$\{+, -, *, /, \hat{}, ;\}$

Thus, since we got a LL(1) grammar, we can implement it directly. Actually, in my program, there are functions called parse_xxx(), where xxx stands for one

of those non-terminals. Since it's a LL(1), the implementation will be in $\Theta(n)$.

After we build the parse tree, we can perform calculation within the tree. Actually, it's very easy to calculate using the original CFG, storing each intermediate value for each non-terminal. However, when we convert it into a LL(1) grammar, it's not so easy to store the intermediate results.

So I passed a parameter for the non-terminals corresponding to the left-associative operations, i.e. EXPR1x, EXPR2x, as the value of the previous terminal. Then, I can carry out calculations with some rules.

VARIABLE TABLE The variable table is used to store the names of variables and their values. I just use an dynamic allocated string array and double array to implement this.

MISCELLANEOUS Since I can't modify the reader.*, I have to read the whole input and make the results after the input. However, I want a program that can run on multiple cases without calling it from the prompt again and again. So I implement a shell which calls the core program again and again. So you can just call the main program once and carry out the calculation.