COL216 Computer Architecture

Lab Assignment 2 : Evaluation of Expressions With Parentheses

Extend the program of assignment 1 to accept expressions with parentheses. Allow parentheses to be nested. There is still no operator precedence and evaluation is from left to right. Syntax for these expressions can be described as follows.

expression	→	term expTail
expTail	→	OP term expTail | empty
term	→	(expression) | constant
constant	→	DIGIT consTail
consTail	→	DIGIT consTail | empty

The first two rules say that an expression is a series of one or more terms, separated by OPs. The third rule says that a term is either an expression in parentheses or a constant. The last two rules say that a constant is a series of one or more DIGITs. OP and DIGIT are defined as follows.

OP	→	+ | – | *
DIGIT	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The above rules are written in such a way that we can straight away write a recursive program to process the expressions, simply by looking at the rules. Such a program in C is shown in Annexure I. This program has no loops! It can be seen in the program that for each rule, there is a function with name same as the left hand side of the rule and this function calls other function(s) whose name(s) appear on the right hand side of the rule, in the left to right order.

Convert this C program into ARM assembly language and test using #ARMSim. The arguments/results for each function can fit into registers and need not be transferred through stack. Use stack for saving return address and for any registers that require saving. For this assignment, do not bother about input/output. Use .asciz to define an expression to be processed and leave the result in register R0.

If the fully recursive program appears too cryptic, you may use an alternative version shown in Annexure II. This has reduced number of functions and uses loops in addition to recursion. Both program assume correct input. No error checking is done.

Annexure I

```
#include <stdio.h>
char expString [64];    /* array to hold the expression string */
char* p;    /* pointer to expr string, global to all functions */
int expression ( );
int consTail (int x) {
    if (*p >= '0' && *p <= '9') return(consTail(x * 10 + *p++ -'0'));
    else return (x);
};
int constant ( ) {
    return (consTail (*p++ - '0'));
};
int term ( ) {
    int x;
    if (*p == '(') {p++; x = expression ( ); p++; return (x);}
    else return (constant ( ));
};                              /* second p++ is to go past ')' */
int expTail (int x) {
    if (*p == '+') {p++; return (expTail (x + term ( )));}
    else if (*p == '-') {p++; return (expTail (x - term ( )));}
    else if (*p == '*') {p++; return (expTail (x * term ( )));}
    else return (x);
};
int expression ( ) {
    return (expTail (term ( )));
};
int main ( ) {
    p = &expString [0];
    scanf ("%s", p);
    printf ("%i\n", expression ( ));
    return (0);
};
```

Annexure II

```
#include <stdio.h>
char expString [64];    /* array to hold the expression string */
char* p;    /* pointer to expr string, global to all functions */
int expression ( );
int constant ( ) {
    int x = 0;
    while (*p >= '0' && *p <= '9')
        x = x * 10 + *p++ -'0';
    return (x);
};
int term ( ) {
    int x;
    if (*p == '(') {p++; x = expression ( ); p++; return (x);}
    else return (constant ( ));
};                            /* second p++ is to go past ')' */
int expression ( ) {
    int x;
    x = term ( );
    while (*p == '+' || *p == '-' || *p == '*')
        if (*p == '+') {p++; x = x + term ( );}
        else if (*p == '-') {p++; x = x - term ( );}
        else if (*p == '*') {p++; x = x * term ( );}
    return (x);
};
int main ( ) {
    p = &expString [0];
    scanf ("%s", p);
    printf ("%i\n", expression ( ));
    return (0);
};
```