Management of R1 When Parsing Expressions
A Summary Sheet for Project 6

I. Based on past experience I have found that it is helpful to provide a sheet that summarizes the management of R1 during Project 6. In terms of your expression parser code generation it is best to think of generating code in three distinct places (a), (b), and (c) as described below.

a) When you first enter the expression parser you will generate code to mark the current top of the stack with R1. This code would be:

```
movw  SP  R1
```

b) Each time you do a reduction that does arithmetic, such as E+E → E you would generate code to perform that operation. For example if you are doing an addition you would generate:

```
movw  +xxx@r0   +yyy@r0
addw  +zzz@r0   +yyy@r0
```

where xxx and zzz are the offsets of the things being added, and yyy is the offset of a new temp.

c) Just before you return from the expression parser you would generate cleanup code to copy the answer that was computed so that it is where the SP originally was when the expression parser was first called (this is marked by r1). So you would generate:

```
movw  +zzz@r0   @r1
```

where zzz is the offset of the variable that is pointed to by the E that is currently only top of the operator precedence stack.

Immediately after the line above you would generate two more lines to pull the SP down so that it is just above the answer. These lines would be:

```
movw   r1  sp
addw   #4  sp
```

At this point you would call a function to delete all the temps (all identifiers that start with $) from the top level of your symbol table. Right after calling this function you can return from the expression parser.

(over)

II. Lets say you just got back from your expression parser after computing the expression that is on the right hand side of the assignment statement:

```
x = a + b + c;
```

As soon as you return from the expression parser you would generate code to take the answer from the expression parser (this should be at -4@sp) and copy this answer into x. So you would generate:

```
movw  -4@sp  +ccc@r0
```

where ccc is the offset of x.

Immediately after this you would generate code to clean up the answer from the stack. Since you have assigned the answer to x you no longer need it on the stack. The code to clean this up would be:

```
subw  #4   sp
```

III. General suggestions:

After completing pre-project 6, one reasonable way to get started with project 6 would be to just try to handle code generation for the addition of two integer variables. A goal might be to handle a YASL program that consists of:

```
cin >> x;
cin >> y;
sum = x + y;
cout << sum;
```

Remember that you can hand-edit your PAL code if it is not working. In other words, suppose you run your compiler and you get 20 lines of PAL code. You run the PAL code and it crashes. In order to figure out where it crashes, insert some statements of the form:

```
outb  ^A
```

which prints an A. By running the program again and watching the output you can determine where it crashes. Now, suppose you are interested in knowing what was in a certain place on the stack when the program crashed. You could insert a line such as:

```
outw +0@R0   : print the value of the first global variable to help you debug
```

or

```
outw  -4@sp   :  print the value just below SP to help you debug.
```