CS 540 Program #4: UnCool to IA32

Due: December 5, by the end of the day (midnight)

For this last programming assignment you are going to generate IA32 code, extending your parser from the previous assignment. I talked about IA32 in class and there are examples online; however, in this handout I will summarize some of the most important information. For this assignment, we are going to make lots of simplifying assumptions - be sure to read this document carefully! I strongly suggest you deal with the different issues in the order I discuss them below. For each, test completely, including running under SPIM to see if what you generate is correct (both syntatically and in terms of the underlying computation). Examples of most of the functionality are available online.

I've decided to give you two options with respect to implementation. The base assignment is worth 95 points overall (out of 100). The advanced assignment, for a larger language set, is worth 110 (out of 100). Note that completing the base assignment is part of completing the advanced assignment.

The Base Assignment:

For the base, you will only be implementing a subset of UnCool – a single main object. This results in an imperative language much like C with a 'global' scope and local scopes. The grammar for this subset is given below.

```
class           :       CLASS_T TYPE '{' feature_list '}'
                ;
feature_list    :       feature_list feature ';'
                |
                ;
feature         :       ID '(' formal_list ')' ':' typename '{' expr_list  '}'
                |       ID '( ' ')' ':' typename '{' expr_list  '}'
                |       ID ':' typename
                |       ID ':' typename ASSIGN simple_constant
                |       ID ':' STRING_T ASSIGN STRING_CONST
                |       ID ':' INT_T '[' ']'
                ;
typename        :       INT_T    |       BOOL_T    |       STRING_T
                ;
simple_constant :       INT_CONST   |       TRUE_T     |       FALSE_T
                ;
formal_list     :       formal_list ',' formal
                |       formal
                ;
formal          :       ID ':' typename
                |       ID ':' INT_T '[' ']'
                ;
expr            :       ID ASSIGN  expr
                |       ID '[' expr ']'  ASSIGN  expr
                |       ID '(' ')'
                |       IN_INT '(' ')'
                |       OUT_STRING '(' expr ')'
                |       OUT_INT '(' expr ')'
                |       ID '(' actual_list ')'
```

```
                |       ID
                |       ID '[' expr ']'
                |       IF_T expr THEN_T expr ELSE_T expr FI_T
                |       WHILE_T expr  LOOP_T expr POOL_T
                |       '{'    expr_list '}'
                |       LET_T  formal_list    IN_T expr TEL_T
                |       NEW_T INT_T '[' expr ']'
                |       expr  '+' expr
                |       expr  '-' expr
                |       expr  '*' expr
                |       '~' expr
                |       NOT_T expr
                |       expr LT expr
                |       expr LE expr
                |       expr EQ expr
                |       expr GT expr
                |       expr GE expr
                |       expr NE expr
                |       '(' expr ')'
                |       TRUE_T
                |       FALSE_T
                |       INT_CONST
                |       STRING_CONST
                ;

actual_list     :       actual_list ',' expr   |       expr
                ;

expr_list       :       expr_list ';'  expr  |       expr
                ;
```

I strongly suggest you work on this assignment incrementally, testing each added part thoroughly before moving to the next. I'm going to walk through the parts in the order that I implemented them - this order is not required but might make your life easier.

Getting Started

Implementing I/O first will make it easier to test your code by running it and will have you generating the setup/finish code that will be used by every program you generate. We are going to implement `out_int` (and `out_string`) statements using `printf`. In C, the format of a call for an integer is: `printf("%d\n",the_int);`
To call printf correctly, you need to put the parameters onto the stack. This series of IA32 instructions will print 32.

```
        movl    $32, %edx
        movl    %edx, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
```

.LC0 is the address of the "%d\n" string. The simplest UnCool program to print this is:

```
class Main {
    main () : Int {
      out_int(32)
    };
}
```

The complete IA32 code to implement this is shown below. Note that there is a fair amount of code you have to generate just to set things up correctly.

```
        .section        .rodata.str1.1,"aMS",@progbits,1
        .data
.LC0:
        .string "%d"
.LC1:
        .string "%d "
.LC2:
        .string "%s "
        .align 4
        .type _uncool_input,@object
        .size _uncool_input,4
_uncool_input:
        .long 0
        .text
        .globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl    %ecx
        subl  $128,%esp
        movl %ebx,-16(%ebp)
        movl %esi,-20(%ebp)
        movl %edi,-24(%ebp)

        movl $32, %ebx
                        # Line 3 - print
        movl %edx,-12(%ebp)
        movl %ecx,-8(%ebp)
        movl %ebx, 4(%esp)
        movl $.LC1, (%esp)
        call printf
        movl -12(%ebp),%edx
        movl -8(%ebp),%ecx

        movl -16(%ebp),%ebx
        movl -20(%ebp),%esi
        movl -24(%ebp),%edi
        addl        $128,%esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
```

The print statement itself is implemented in the four lines in bold which move the values onto the stack and then call printf. All printing will be done this way.

The rest is just code to initialize the stack for the main program. The only difference from the standard way of setting up is that fact that I've allocated more than usual on the stack to use for any variables or registers you want to store there. Two global strings (.LC0 and .LC1) are defined to be used throughout the program. You will want to generate similar code for the main program of every UnCool program.

Arithmetic Expressions

Next, work on generating IA32 code for expressions that use constant values. At the leaves of tree, you need to put the constant values into registers (movl instructions). You need to have an attribute for some of the non-terminal symbols (like expr) that keeps track of what register is holding the relevant value. At the internal nodes, you need to generate code that would perform the computation given the registers holding intermediate values. The example below shows some expression computation and output.

```
UnCool:                          IA32:
out_int (3 - 9*-2)               movl    $3, %edx
                                 movl    $9, %ecx
                                 movl    $2, %ebx
                                 negl    %ebx
                                 imull   %ebx, %ecx
                                 subl    %ecx, %edx
                                 movl    %edx, 4(%esp)
                                 movl    $.LC0, (%esp)
                                 call    printf
```

Don't worry about generating efficient code. That is **much** more difficult and doesn't fix anything a good optimizer wouldn't fix. Also, don't worry about generating code identical to the examples online or in this document. As long as the computation will produce the same result, your answer is fine. You will notice that the code above uses registers to hold results temporarily. Because the number of registers is limited, you need to keep track of which ones you are actually using at any given point in time. You have 6 registers available (%edx, %ecx, %ebx, %esi, %edi, %eax) but it is safer to only use the first five, saving %eax for special use. Assuming you ``allocate" a register when you need one and ``free" a register when you are done with it, this is plenty for any example that I will test you on. You need a data structure that tells you what registers are available (initially, all registers are available). Write two functions: one that gets an available register and another that returns a register that you are no longer using. Returning to the simple example above, if we start out assuming all registers are available, we can keep track of the following:

Statement	Available after	In Use After
INITIALLY	{ %edx, %ecx, %ebx, %esi, %edi}	{ }
movl $3, %edx	{%ecx, %ebx, %esi, %edi}	{ %edx}
movl $9, %ecx	{%ebx, %esi, %edi }	{%edx, %ecx }
movl $2, %ebx	{%esi, %edi }	{ %edx, %ecx, %ebx }
negl %ebx	{%esi, %edi}	{%edx, %ecx, %ebx}
imull %ebx, %ecx	{%ebx, %esi, %edi }	{%edx, %ecx}
subl %ecx, %edx	{%ecx, %ebx, %esi, %edi}	{%edx}
movl %edx, 4(%esp)	{%edx, %ecx, %ebx, %esi, %edi }	{ }

Hint: You need to look for rules where information in multiple registers is used to compute something where the result is in one register. For example: exp : exp + exp you need to be sure to free a register because your generated code will use the contents of two registers , leaving the result in one of them. The other register is no longer needed. Control flow statements sometimes use several registers. These need to be freed when no longer in use. The above example uses constants. Local variables on the stack will be addressed differently, as described below.

Procedure Call/Return

Implementing the control flow for procedure call return is relatively straightforward. To test, I typically move my code from constant expression evaluation into a function and then call it. Use 'call' and 'ret' to manage control flow. Don't forget about saving registers. You can save all registers at the appropriate points or only the ones in use at the time.

The slides associated with this handout have a drawing of what the stack frame can look like. You should also look at class slides and some of my sample code online to see what you need to generate. In particular, you need to modify the ebp and esp in both the setup and finish. In my code, I always allocate 128 bytes for each stack frame. I use 20 of these bytes to save the five registers I will be using when needed. The rest is for local variables. You can assume that there will be no more than 10 local variables in a function, so the 128 is probably too large but better safe than sorry.

Parameters passed will only be integers and arrays (as pointers).

Variables

You can save global variables in a global storate area but all local variables should be saved on the runtime stack. You will need to use a symbol table just as with assignment 3; however, the important information is not the type, but information that will let you find the variable at runtime. Once you have decided how you want to save variable information, don't forget to add data flow to your procedure call/return. Actual parameters need to be put on the stack at some offset from \%esp. In the procedure body, formal parameters are accessed at an offset from \%ebp. All variables can be found in a global location (if that is what you choose) or can at some offset from \%ebp. Local variables should be stored at a negative offset from \%ebp.

Input

Use system call for scanf to get input from the user. There are several examples of this online.

Control Flow

You need to generate the appropriate labels and branching for the control flow expressions of UnCool. One way to deal with the fact that control flow statements can be nested is to use the YACC stack to hold onto the labels associated with each loop or if statement by associating them

with one of the terminal symbols of the production. An alternative implementation is to use a stack onto which you push active labels. Once you finish with the particular statement, pop the associated labels. Remember that control flow statements use registers and they need to be freed once you are done with them.

Arrays

We are going to implement arrays of integers. Space for the arrays will be explicitly allocated (based on a 'new') using malloc. Array access will require special implementation since you will need to compute the address (base address plus offset) and then load (or store) from (to) that address. Again, I have examples of this online to give you some ideas about how you want to implement this. Do not store the array directly in the activation.

The Advanced Assignment:

Implementing Cool as an object-based language is more challenging. In addition to engineering the constructs above, you need to deal with creation of new objects with making method calls correctly for the given objects. I will talk about this in class (and on the assignment slides). **Do not attempt this unless you complete the base assignment.**

Submitting

Submit on blackboard. Be sure to check that you are submitting what you intend! If you would like your assignment graded with the advanced option, put a note in the comments.