Project 5: Add Function support to project 4

Please read the following guideline and look at the example I posted on the web page.
Then you would have a clear idea of how to extend your compiler for this project.

Stage0:

Make sure you have a separate Symbol Table and IR list per Function.
Also temp variables are local to a Function so every function will have its own set of temp variables.
To add function support out IR is extended to include 5 new instructions
JSR “label” /* used in stage 1: */
PUSH “Var name” /* used in stage 1: */
POP “Var name” /* used in stage 1: */
RET /* used in stage 2: */
LINK /* used in stage 3: */
For JSR “label” is the name of a LABEL used to mark the start of a function body (see stage 3)
For PUSH and POP “Var name” is optional and it could be a name of a global or local or temp variable

Stage1: Translation of Call Expression (Function call)

Function call consists of the following 7 steps
1. Push onto the stack and provide space for return value.
2. Push parameter onto the stack
3. Push register r0 – r3 on to the stack /* see stage4 for details */
4. Add JSR call
5. Pop registers
6. Pop Parameters
7. Pop return value

To understand what is going on you have to first understand how the stack looks like when you have a function call. Sometimes it is called the activation record.
Let’s see what is stored on the stack.
First, Function parameters and return value are passed through the stack
Second, register values are stored on the stack. This is because each function invocation should have a fresh set of register. That means the caller of a function should backup it’s registers. Otherwise after returning from the function call, registers could have different values than what it had before the function call.
Third, local variables are on the stack. It might not be too obvious why local variables are stored on the stack but those have to be stored on the stack. Every invocation of a function should have it’s own set of local variable and the way of the doing it is by allocating variables on the stack. In that way local variables could be allocated when a function is called and removed when the function returns.
And finally previous stack pointer and return address is stored on the stack. This is automatically handled by “Tiny” so you don’t have to handle these.
The following figure show how the stack looks like when a function is called

IR Node	Tiny Code
$Tk (Local Temp)	$-(m+k)
…	…
$T1 (Local Temp)	$-(m+1)
$Lm (Local Var)	$-m
…	…
$L1 (Local Var)	$-1
Return Address	$0
Previous Stack Pointer	$1
R3	$2

R2 $3
R1 $4
R0 $5
$Pn (Parameter) $(6+n-n)
… …
$P1 (Parameter) $(6+n-1)
$R (Return Value) $(6+n)
The left side of the figure shows how IR nodes would be placed on the stack. This is only conceptual since IR nodes are not executable.
The right side shows the actual activation record (stack) by executing tiny code.
The most important thing in this figure is getting used to stack relative addressing.
$# is a stack relative address (here # is an integer number)
The figure shows how $Lx, $Py, and $R should be mapped to a stack relative address (x, y are integer numbers)
Temp variables $Tx will also be mapped to stack relative address if it need to be stored.
Translation for Temp variables is not needed for this step.
Stage 2: Translation of Return expression
Return expression should return a return value.
Let’s assume we have an integer value in Temp8 and want to return that value
Then the sequence will be
STOREI Temp8 $R
RET
Here, $R is the place on the stack that store the return value.
RET is the command to return from this function. And it will be translated into a sequence of tiny instruction
unlnk
ret
Stage 3: Translation of Function Declaration (body)
Now it is time to actually handle the function definition
Let’s say we have a function called “foo”. Then,
LABEL foo /* set function entry point */
LINK # of local variables or some large number /* allocate space for local vars */
…..
/* code for function body */
……
/* part of return statement */
RET /* free up space used for local vars and return */
Here. LINK is the command to reserve space for local variable on the stack and it would be nice to get the exact # of local variables needed and only allocate that much but when you have register spills and have to introduce a local temp variable (You will not encounter this situation until you do register allocation) you will eventually have to get the number of local + temp variables.
For this step you can just allocate space for local variables.
Some functions may not have a return statement at the end. In that case, add a “RET” instruction at the end of the IR list. This can be done by first looking at the last IR node in the list and see if it a “RET” instruction. If it is not a “RET” add a new IR node “RET” at the end of the list.
Here is one suggestion for a cleaner implementation.
Number of parameters and number of local variables are properties of individual functions and every function already has it own IR list and symbol table. So you can create a “Function” class/structure that
has some properties like number of local variables, number of parameters and an IR list and symbol table as its member.
Symbol table entry will need a slight modification to help address translation for tiny code.

As you can see from the previous figure you are not using variable names anymore for local variable and function parameter.
So you symbol table for function should be slightly different from your previous normal symbol table.
What I did is
1. Implemented a symbol table as a hashtable/map
2. Used normal variable name as a key
3. Field “name” in a symbol table entry now uses the $Lx , $Py naming scheme

Stage 4: Limitations and trade off

Step 5 does not implement register allocation and that means the number of active registers are unlimited.
When there is a function call any register with a live value has to be saved on the stack. But if you read
stage 1, it tells you to save only 4 registers(the number of registers allowed in step6). A perfect implementation will save all active registers but that requires keeping track of all the register and could
make a long list of tiny code and further complicate stack addressing explained in stage 2. So instead we
put some restrictions on the test cases for this step. In step 6, this restriction is removed.
The key is not to make a register have live value that is defined before a function call and used after a function call. To do this, if a function call is needed the function call has to be the only expression that appears on the right hand side of an assignment.
For example,
A := foo(3);
Is allowed but
A := 45+foo(4);
Or
A := foo(3) + foo(4);
Is not allowed in step 5(but will be allowed in step 6)

Other helpful data structures :

Although you are not implementing register allocation, you are still doing some form of register mapping. So it would be useful to have a hashtable/map that keeps track of the mapping during code generation.

Some final words :

Function calls will be explained in more detail in class. So please make sure you understand what is covered in lecture. And don’t forget to check for updates for this document.