Accessing Values for Variables

* convention is that param 1 of wain is in $1, param 2 of wain is in $2, and the return value is in $3
* when parameters and variables are declared in a procedure, use the stack to store it since the number of registers we have for MIPS is limited
* to keep track of where we have stored each variable, extend the symbol table for a procedure’s variables to include the location of where it’s stored on the stack
  + new convention is that $4 will always contain the number 4
* we could map each variable to an offset wrt to $30 (stack pointer) but that presents 2 issues:
  + offset to variables would depend on number of variables in each function
  + stack is likely going to be updated and used to store other information (e.g. procedure calls)

Diagram

Description automatically generated with medium confidence

* first issue is easy to resolve since all variables must be declared at the top of the function in WLP4 so the code generator can precompute the offsets for each variable
* to fix the second issue, must use a frame pointer that points to a fixed memory location within the stack while executing a procedure
  + we use $29 as frame pointer
* e.g.

Diagram

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

Generating Code for Expressions

* first define shorthand for writing pseudocode for code generator
  + code(a) indicates that generator will generate code to load the value for var a into register $3
    - e.g. assume var a is param 1 and the offset is 0 wrt to $29 so code(a) will be lw $3, 0($29)
    - later, code will be generalized for other rules in grammar

Background pattern

Description automatically generated with low confidence

* + push($3) rep a store of the value in $3 onto the stack and an update to the stack pointer
  + pop($5) rep a load from the top of the stack into $5 and an update to the stack pointer
* e.g.

A picture containing graphical user interface

Description automatically generated

A picture containing text, receipt

Description automatically generated

Table

Description automatically generated with medium confidence

* + above coding strategy uses temp register to compute any arbitrarily complicated expression
* for grammar rule expr1 → expr2 PLUS term, code generation pseudocode would be:

Text, letter

Description automatically generated

* + other arithmetic operations (subtraction, multiplication, division, and modulo) on integers work the same way as addition
* some code generation strategies for trivial rules:
  + S → |- procedure -|: code(S) = code(procedure)
  + expr → term: code(expr) = code(term)
  + factor → LPAREN expr RPAREN: code(factor) = code(expr)
* a specific case for assignment when a variable is assigned a value (i.e. the rule is statement → lvalue BECOMES expr SEMI when lvalue → ID)

A picture containing chart

Description automatically generated

* + offset value is computed by compiler by looking up the offset wrt $29 assigned for the var with this specific ID in the symbol table

Generating Code for Print Statements

* WLP4 provides print statement through the rule: statement → PRINTLN LPAREN expr RPAREN SEMI
  + expr must be an int value
* 3 ways to handle a print statement:
  + compiler can output the full code to convert a binary number to decimal and to ASCII, then output it ever time the println is used
  + compiler can output a print procedure once and then call it every time a println statement is used
  + compiler can import a print procedure and assume this procedure is available in the runtime environment, then it calls the imported statement every time a println statement is used
* a runtime environment is the execution environment provided to an application or software by the operating system to assist programs in their execution
  + may include procedures, libraries, environment variables, etc.
* real compilers rely on runtime environments since this allows the binary they produce to be smaller
* pre-compiled code in the runtime must contain information regarding what the code provides and what it expects
  + compiler outputs an object file, which contains the compiled version of the code and additional information regarding what a piece of code expects and what it provides
* we use MIPS Executable Relocatable Linkable (MERL) format for object files
  + MERL format contains MIPS machine code (not assembly) and additional info needed by both linker and loader
* while generating code, if compiler needs to use the print procedure that’s part of the runtime, it’ll generate an assembler directive to import print
  + have .import print appear once at the top of the generated file
* once compiler is don’t generating output, result is assembly file that may contain import statements so we translate it into a MERL file: cs241.linkasm < output.asm > output.merl
* use linker to take any number of object files, links them, and produces a new object file which contains the combined machine code and announcements for whenever the code expects to be linked: cs241.linker output.merl print.merl > linked.merl
* finally, strip out MERL metadata from object file and produce pure MIPS machine code: cs241.merl 0 < linked.merl > final.mips
* code generation for println statements will look like:

Text

Description automatically generated

* some conventions we have chosen so far:

Text

Description automatically generated

* + while evaluating any arbitrary expression, we’ll use $5 to temporarily hold values

Comparisons

* code(test) is a convention we’ll use to generate code that’ll cause the value in $3 to be 1 whenever a comparison is true and 0 otherwise
* for the rule test → expr1 < expr2, we generate:

A picture containing text

Description automatically generated

* generating code for test → expr1 > expr2 can reuse the same expression above but we switch the order of expr1 and expr2
* code generation for test → expr1 != expr2 will look like:

Text

Description automatically generated

* + after putting evaluated values of expr1 and expr2 into $5 and $3 respectively, perform comparisons that check whether if either one is less than the other
  + if either one of these comparisons are true, then $3 will be 1 and we know expr1 isn’t equal to expr2 (i.e. one will always be less than the other)
* to generate code for test → expr1 == expr2, use the above code but with the idea that a == b is the same as !(a != b)
  + flip the value of $3 after by adding the line sub $3 , $11 , $3
* for the other comparisons, we note that a <= b is the same as !(a > b) and similarly a >= b is the same as !(a < b)

Control-Flow Statements

* assume the shorthand code(statements) generates code for any sequence of statements, define pseudocode for statement → IF (test) {stmts1} ELSE {stmts2}:

Text

Description automatically generated with medium confidence

* + to ensure that the labels being used are unique across the entire output (i.e. can’t use exact same labels of else and endif for every if statement generate), append each label with a counter that increments every time a new label is generated
* for the rule statement → WHILE (test) {statements}, code generation pseudocode is:

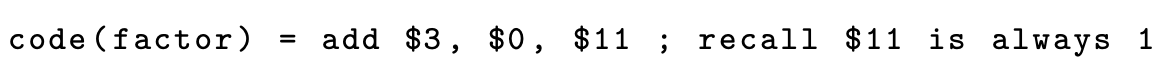
Text

Description automatically generated

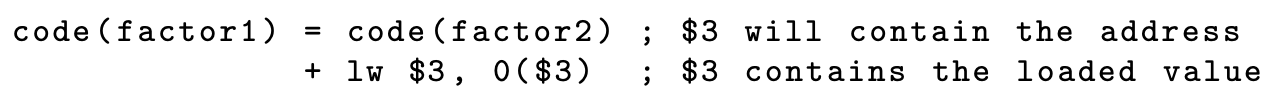
* + once again, ensure labels are unique

Pointers in WLP4

* pointer variables are initialized to NULL
  + in C, address 0x0 (accessing it results in a segmentation fault) is used to rep a NULL value but that doesn’t work in MIPS because 0x00 is a valid address and accessing it wouldn’t result in an error
  + instead, associate NULL with an unaligned address so that the program crashes
* code generation for factor → NULL would be like



* pointer vars are dereferenced through the rule factor1 → STAR factor2
  + factor2 must evaluate to a pointer type
  + dereference the pointer to access the value stored at that address
  + code generation:



* address-of operator is used to obtain a pointer to a mem location
  + we can only take the address of an lvalue, which rep a storage location
* in the grammar rule for the address-of operator, factor → AMP lvalue, the lvalue can derive an ID, STAR factor, or LPAREN lvalue RPAREN
  + for the case lvalue → ID:

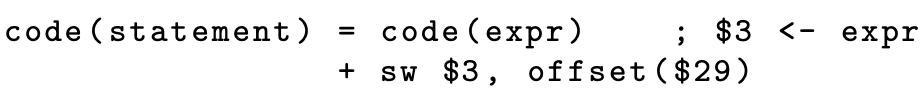
Chart

Description automatically generated with medium confidence

* + - intent is to get address of a var so we use the offset for the var that’s stored in the symbol table
  + for the case lvalue → STAR factor2:



* + - first dereferencing factor2 then taking the address of the location means the two ops cancel each other out
  + for the case lvalue1 → LPAREN lvalue2 RPAREN, code(lvalue1) = code(lvalue2)
    - parentheses don’t affect the code
* to do assignment through pointer dereference, the relevant rule is statement → lvalue BECOMES expr SEMI
  + case when lvalue → ID:



* + - doesn’t deal with assignment through pointer deference and it simply needs to look up the offset of the var wrt to the frame pointer $29 in the symbol table
  + case when lvalue → STAR factor:

A picture containing text

Description automatically generated

* + - since program has passed through type-checking, we know that code(factor) will generate an address in $3
* when doing comparisons with pointers, the code generation is identical except we used sltu instead of slt because pointers can’t be negative
  + so far, we’ve never had to look up a type because we’ve been able to rely on rules stored within the parse tree nodes
  + for comparisons, must look up type of one of the operands to determine whether to use slt or sltu
* for the rule expr1 → expr2 + term, where type(expr2) == int\* and type(term) == int (i.e. adding an int value to an address), we must add sizeof(int) times the int value to the address:

A picture containing text

Description automatically generated

* + i.e. we’re computing expr2 + (4 × term)
  + e.g. if we had address of 1st element of array and added int 3 to it, the above code would give us the address of the 4th element
* for the rule expr1 → expr2 + term, where type(expr2) == int and type(term) == int\*, just use the above code but switch the order of the operands (i.e. compute (expr2 × 4) + term)
* for the rule expr1 → expr2 − term where type(expr2) == int∗ and type(term) == int (i.e. subtracting an int value from an address), we use the same code generation pseudocode as above but we would subtract instead of add
  + i.e. we compute expr2 − (4 × term)
* finally, for the rule expr1 → expr2 – term, with both type(expr2) == int\* and type(term) == int\*, we need to compute the number of elements between the 2 elements

Text

Description automatically generated with medium confidence

* + we compute (expr2−term)/4

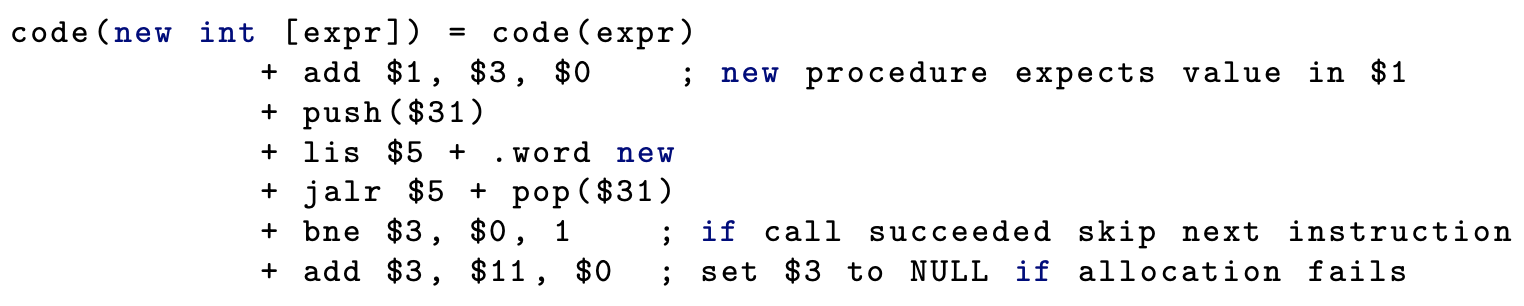
Heap Memory Allocation

* WLP4 supports heap mem and we will rely on the runtime environment for allocation and deallocation of mem
* to use new and delete, must import 3 new labels from the alloc.merl module

Text

Description automatically generated

* + init label is procedure that must be called once before any calls to new or delete
* if output program will be run using mips.array, $2 is length of the array
  + check if WLP4 program will be run using mips.array by checking first param of wain (i.e. if it’s a pointer, then program is run by mips.array)
  + otherwise, $2 must be set to 0
* new procedure exported by allocator module expects number of words of mem being requested in $1 and returns starting address of allocated mem in $3
  + code generation pseudocode:



* delete procedure requires that $1 contains mem address to be deallocated
  + code generation pseudocode:

Text

Description automatically generated

* + - we see in the explicit check on not calling delete on NULL that deleting NULL is not an error in WLP4 (it just does nothing)
* ensure that the generated labels are all unique
* to assemble and link the generated output:

Text

Description automatically generated

* + allocator must appear last in the link command
* finally, run generated program using mips.{twoints, array} exec.mips

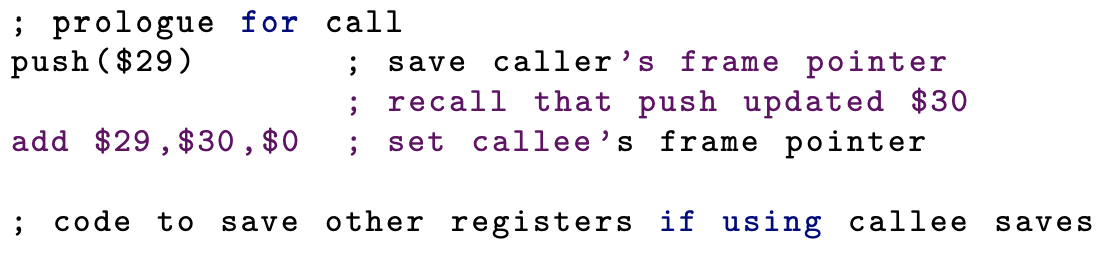
Generating and Calling Procedures

* since WLP4 require all programs to begin with executing the wain function, we can organize our generated output so that the output for wain appears first
  + any other defined procedures would appear after the end of code for wain
  + to ensure code doesn’t fall through to code for next procedure, end each procedure with a jr $31

Table

Description automatically generated

* prologue for wain is different than prologue for other procedures since at start of program, must import any external procedures (e.g. print, new, etc.), initialize any conventions (e.g. $4, $11, etc.), call init to set up the heap mem allocator, and store $1 and $2 on the stack before overwriting them with input values to the program
* epilogue for wain is to reset the stack and then exit using jr $31
* other procedures need to set up frame pointer and save any registers they use in their prologue, then restore the registers and the stack before exiting with jr $31 in their epilogue
* we know that $29, $30, and $31 are changed and these registers must be appropriately saved and restored
  + $30 is updated as needed and if we ensure the callee leaves $30 where it was when it was called, the caller can proceed
  + for $31, we use the caller-save approach since the caller knows which address it must return to and calling the callee using jalr will overwrite this value
  + 2 diff approaches to save $29:
    - callee-save method:



* + - caller-save method:

Text

Description automatically generated

* assuming caller-save semantics for the frame pointer, pseudocode for a procedure call (i.e. the rule factor → ID(expr1, ..., exprn)):

Text

Description automatically generated

* to generate code for a procedure (i.e. the rule procedure → int ID(params){dcls stmts RETURN expr;}), pseudocode is:

Text

Description automatically generated with medium confidence

* to see dilemma that is posed by above code generation scheme, consider this example:

A picture containing text

Description automatically generated

Diagram

Description automatically generated

* + foo will need to access the params a and b, but also local vars c and d
    - these vars are no longer contiguous in mem so the offsets wrt to the frame pointer have changed
  + following table shows foo and all available vars:

Table

Description automatically generated

* + - param i has an offset of 4(n – i + 1) where n is number of params
    - local var j has offset of -4r – 4(j – 1) where r is number of registers to preserve
* another way to lay out the stack is to switch the order so that code generation for declarations are first and then preserving registers follow
  + this makes sure that all params and local vars occur contiguously
  + for same example procedure above, it would look like this:

Diagram

Description automatically generated

Table

Description automatically generated

* + - param i has an offset of 4(n – i + 1) where n is number of params
    - local var j has offset -4(j – 1)

A Note on Duplicate Labels

* even though counters can distinguish labels for the most part, functions can end up having the same name
  + e.g. we have labels else1 and endif1 for an if statement but the input WLP4 program has a function labelled else1
* solution is to attach a prefix to all WLP4 function label names
  + e.g. putting “F” in front of each label being generated for a function name