Project step 1 – a compiler for a simple language. v0.16

Change log:
v0.16, changes from 0.15 Make all push types in compiler actions explicit. Simplified and better documentation of call and callr instruction compiler actions. Let the print statement print characters and numbers. Added a printv statement to print variables. Changed compiler actions for retr to push a variable value, not a literal value. Changes are shown in orange.
v0.15, changes from 0.14 Change compiler actions for ret, retr, jmp. Change the description and compiler actions for poke. Change the description for swp. Change the compiler actions for call and callr. Changes shown in green.
v0.14, changes from 0.13 Add peek, poke and swp instructions. Change popm compiler actions. Change callr compiler actions. Other small changes to wording. Changes are shown in blue.
v0.13, changes from 0.12 Add a count field to subr, call and callr to simplify code generation. Changes are shown in red.
v0.12 Changes from 0.11. Added a callr statement that takes a return type. Fix the generated code for this and for call to allow arguments to be pushed by the call. Add a retr that returns a value and update the reg.
v0.11: changes from 0.10. Put typing into push operators. Put opcodes for compare operators. fix actions for call. Make declarations reserve a stack location. Remove redundant store instruction (popv does the same thing.)
v0.10: changes from 0.0. Comparison operators (cmpe, cmplt, cmpgt) added. jump conditional (jmpc) added. bytecode values added. Font changed to Times New Roman.

This project builds a compiler for a small language. The input language is described below. The output language is a bytecode that is interpreted. You will be supplied a binary for the interpreter, and will write an interpreter as a second project.

High level organization of the language:

The first non-comment statement in the language is the start of the *main* routine. After the end of the main routine, other functions are declared and defined.

Within each function the first non-comment statements are variable declarations. All variable declarations must be at the top of the function.

The first non-declaration statement defines the start of the executable and label statements. The various statements are defined below.

At the end of the function there will be a **ret** statement. The next statement, if it exists, should be a new function.

Four kinds of values can be operated on by a program, in addition to labels.

integer: 32 bit integer values. Specified as a string of decimal digits, i.e., 56.

short: 16 bit integer values. specified as a string of decimal digits with an s appended, i.e., 56s.

float: 32 bit floating point values, specified as a string of decimal digits, a decimal point, and a fractional part, i.e., 56.04.

char: these are only present in print statements and are always literals, e.g., ‘c’.

Native representations can be used for integers, shorts and floats. Operations, described below, can be applied to mixed values, i.e., a float and a short can be added and stored in an integer.

Details of different kinds of statements in programs are given in the section **Input language** below.

High level actions of the compiler:

The compiler will read each statement in turn from the source program file, and determine the operation the statement performs and the operands. Each statement has the structures

    operation op, op, …

i.e., and operation plus zero or more operands.

If the statement is a variable declaration, the compiler will create an entry in the ***symbol table***. There is one symbol table for the entire program. The symbol table is a map whose key is one of two kinds. The first is a variable name, and the data for the variable name is the offset on the runtime stack (of the function the variable is declared in) for the variable. Clearly, the compiler needs to have a counter that keeps track of the number of variables declared in a function so that its runtime stack location is known. The second is a label number, where the data is the offset in the byte code of the label. Clearly, the compiler needs to have a counter that keeps track of the offset in the generated bytecode that the label is found.

For symbol table entries, *(x)* indicates the contents of the symbol table for the entry for key *x*.

After processing dec statements that only create entries in the symbol table and do not cause any code to be generated, the executable part of the function begins. The executable part of the function consists of *lab*, *subr*, *ret*, *print*, *jmp*, *jmpc, cmpe, cmplt, cmpgt, call*, *push*, *popm*, *popv*, *st*, *add*, *sub*, *mul* and *div*.

Actions for many statements are straightforward, and are described in the section **Input language**, below. Because variables have to be declared first before executable statements, every variable needed will be present in the symbol table. When we encounter a *jmp* or a *call* statement, however, we may not have visited the corresponding *lab* or *subr* statement, and the location jumped to, or the location of the function being called, may not be in the symbol table, and th compiler will no know how to generate code to jump to that location. In this case the compiler should keep track of the location of all code generated that requires the value of a jmp target or the start of a function. When the entire program has been traversed all of these values are known, and the compiler can go to the locations that need to be updated and update them.

At this time the program can be written to a file with an extension of .smp as a stream of bytes.

Input language.

The input language processes arithmetic expressions with numeric operations of +, -, *, and /. The bytecode representation of an operation is given by bc.op. For simplicity, the runtime stack contains one value in each position, i.e., a short and an int take the same amount of space. The runtime stack can be implemented as a vector whose elements point to objects containing the actual stack value.

Language statements:

The descriptions below give the semantics of the statement when the program is executed (which will be done by another program) and the code generated, or other actions taken, by this program. For all statements that generate byte codes an internal compiler counter giving the bytecode offset should be incremented appropriately. *(x)* indicates the contents of the symbol table for the entry for *(x)*. Statements should all be parseable by a state machine/DFA.

All language statements fit on a single line.

// *string*: A comment that can be ignored
compiler actions: throw away the line of the program that is the comment.

decl *var type*: declares a variable whose name is given by *var*. The name will be 8 alpha characters or less. *type* is one of the three numeric types above, and are specified as *int*, *short* or *float*. All variables are local. There are no global variables.

compiler actions: create an entry in the symbol table (see below). The entry will be of the form <key, value>, where the key will be the concatenation of the function label of the function being compiled and *var*, and the value will be an object containing the stack offset within the stack frame of the function that will hold the variable and the type. The first variable declared in a function will have an offset of 0, the next an offset of 1, and so forth. Reserve a space in the stack for the variable by generating the following code:

```
push<type> 0
```

lab *label*: specifies a *label* that can be the target of a branch. *label* is a positive integer whose value is less than 1000.
compiler actions: *C*reate an entry in the symbol table. The entry will be of the form <key, value>, where the key will be *label* and the value will be the offset within the generated program code generated that the label appears.

subr *cnt, flabel*: declares an *flabel* that is the start of a subroutine, where *flabel* is a string with no more than eight alpha characters. The **main** procedure always has the *flabel* of main. No two functions share the same *flabel. cnt* is the number of argument **subr** takes and is an integer.

compiler actions: *C*reate an entry in the symbol table (see above). The entry will be of the form <key, value>, where the key will be the *flabel* and the value will be the offset within the program code generated that the label appears and the value of *cnt*. Note that variables and *flabels* may have the same string since the variable is always part of a function, and the function name is part of the variable's key.

ret: a subroutine return. Pop all local variables off of the stack, pop the return value off of the stack and into the PC (program counter). The next statement to be executed will be the one indicated by the updated PC.
compiler actions: Generate the following code:

bc.pop <count of local variables added to the stack + count of arguments>

```
bc.ret // jmp to the location at the top of the stack - the return
       // address
```

retr var: a subroutine return that returns a variable value. Pop all local variables off of the stack, pop the return value off of the stack and into the PC (program counter). The next statement to be executed will be the one indicated by the updated PC.
compiler actions: Generate the following code:

```
bc.pop <count of local variables added to the stack + count of
arguments>
bc.pushv var
bc.swp
bc.ret // jmp to the location at the top of the stack - the return
       // address
```

print<type> *literal*: prints the literal. Remember to increment the byte code offset in memory by the length of the literal in bytes.
compiler actions: Generate the following code:

```
bc.push<type> literal
bc.print
```

printv *var*: prints the value of the variable variable
compiler actions: Generate the following code:

```
bc.pushv (var)
bc.print
```

jmp *label***:** jump to the statement immediately following the label. The jmp statement pops the offset at the top of the stack off of the stack when it performs a jump.
compiler actions: Generate the following code:

```
pushi (label)
bc.jmp
```

jmpc *label***:** jump to the statement immediately following the label if the top of the stack has a 1, otherwise do nothing. Typically used after a compe, complt or compgt
compiler actions: Generate the following code:

```
pushi (label)
bc.jmpc
```

cmpe: Let *t* be a pointer to the top of the stack, the result of this is $*t = *(t\text{-}1) == *t$. 1 will be at the top of the stack if the comparison is true, 0 otherwise. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:

```
bc.cmpe
```

cmplt: Let *t* be a pointer to the top of the stack, the result of this is $*t = *(t\text{-}1) < *t$. 1 will be at the top of the stack if the comparison is true, 0 otherwise. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:

```
bc.cmplt
```

cmpgt*:* Let *t* be a pointer to the top of the stack, the result of this is $*t = *(t\text{-}1) > *t$. 1 will be at the top of the stack if the comparison is true, 0 otherwise. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:
```
bc.cmpgt
```

call *cnt, vara*$_0$*, vara*$_2$*, …, vara*$_{n-1}$*, flabel*: jump to the subroutine specified by *flabel*, i.e., jump to the offset that is in the symbol table for the *label*. The address of the next instruction after the call is pushed onto the stack. *cnt* is the number of arguments passed to the subroutine, and is an integer.

compiler actions:

If *flabel* is not in the symbol table, add it. You will not be able to add the location of *flabel* yet. Add the argument count, *cnt*, to the symbol table. If *flabel* is in the symbol table check that *cnt* is the same as the *cnt* in the symbol tabel. If not, print an error and terminate the compiler.

Generate the following code:
```
bc.pushi PC     // compute the next instruction byte offset after the
bc.pushi 6+n+x+1 // call. add the current instruction (PC)+8
                 // non-arg push instructions +n arg push
                 // instructions +x positions taken by the arg values
                 // and (flabel) value pushed
                 // + 1 to skip past the last position to the
                 // next instruction after the call.
                 // 8+n+x+1 can be determined at compile time,
                 // and the resulting integer variable pushed
             // note that n may be zero. Note the extra space for
             // the return argument varr which sits in the old stack
             // frame immediately before the arguments are pushed
bc.add
bc.push<type> var0 // push the arguments (n pushes)
bc.push<type> var1
. . .
bc.push<type> varn-1
bc.pushi n+1      // compute the stack depth added by arguments,
                  // (flabel) (n args + 1 for (flabel)
bc.pushi (flabel)
bc.call
```

callr *cnt, varr, vara*$_0$*, vara*$_2$*, …, vara*$_{n-1}$*, flabel*:: jump to the subroutine specified by *flabel*, i.e., jump to the offset that is in the symbol table for the *label*. *cnt* is the number of arguments passed to the subroutine, and is an integer. Space is reserved on the stack to hold the return value. After the call returns, the value in this stack location is moved to *varr*. The address of the next instruction after the call is pushed onto the stack. **NOTE: *varr* must the same type for all calls and the type is set by the first call.**

compiler actions: Generate the following code:

If *flabel* is not in the symbol table, add it. You will not be able to add the location of *flabel* yet. Add the argument count, *cnt*, to the symbol table. If *flabel* is in the symbol table check that *cnt* is the same as the *cnt* in the symbol tabel. If not, print an error and terminate the compiler.

```
bc.push<type> varr
bc.pushi PC    // compute the next instruction byte offset after the
bc.pushi 8+n+x+1 // call. add the current instruction (PC)+8
                 // non-arg push instructions +n arg push
                 // instructions +x positions taken by the arg values
                 // and (flabel) value pushed
                 // + 1 to skip past the last position to the
                 // next instruction after the call.
                 // 8+n+x+1 can be determined at compile time,
                 // and the resulting integer variable pushed
             // note that n may be zero. Note the extra space for
             // the return argument varr which sits in the old stack
             // frame immediately before the arguments are pushed
bc.add
bc.push<type> var_0 // push the arguments (n pushes)
bc.push<type> var_1
. . .
bc.push<type> var_n-1
bc.pushi n        // compute the stack depth added by arguments
bc.pushi (flabel)
bc.call
bc.pop varr
```

push<type> *val*: push the *val* onto the stack. *<type>* specifies the type, either an s, i or f for short, int and float, respectively. The type of the operand to be pushed is inferred from the format of *val.*
compiler actions: Generate the following code:

```
bc.push<type> val
```

push<type> *var*: push the value of the variable whose name is given by *var* onto the stack. *<type>* specifies the type, either an s, i or f for short, int and float, respectively. The type of the operand to be pushed is inferred from the type of the variable.
compiler actions: Generate the following code:

```
bc.pushs<type> (var)
```

popm *val*: pop the top entry *val* entries from the stack. The values in the stack are lost. *val* will be an integer.
compiler actions: Generate the following code:

```
bc.pushi val
bc.popm
```

popv *var*: pop the top of the stack and put the value into the variable *var*.
compiler actions: Generate the following code:

```
pushi (var)
bc.popv
```

peek var, val: *var = stack[sp+val]*. The types of the stack entry and the variable *var* must be the same. *sp+val* should be a valid stack entry. The primary use is to examine arguments. If there are n arguments then peek var, k-n get's the value of the kth argument. *val* is typically negative
compiler actions: Generate the following code:
```
push<type> (var)
pushi val
bc.peek
```

poke val, var: *stack[sp+val] = var*. The types of the stack entry and the variable *var* must be the same. *sp+val* should be a valid stack entry. The primary use is to change the value of arguments.
compiler actions: Generate the following code:
```
pushi (var)
pushi val
bc.poke<type>
```

swp: Swap top two stack elements, i.e., t = stack[sp]; stack[sp] = stack[sp-1]; stack[sp-1] = t.
compiler actions: Generate the following code:
```
bc.swp
```

add: add the top two elements of the stack and push the result onto the stack. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:
```
bc.add
```

sub: subtract the top two elements of the stack and push the result onto the stack. Thus, if *t* is a pointer to the top of the stack, the result of this is **t = *(t-1) - *t*. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:
```
bc.sub
```

mul: multiply operand *op1* and *op2*, and push the value onto the stack. he stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:
```
bc.mul
```

div: Divide the top two elements of the stack and push the result onto the stack. Thus, if *t* is a pointer to the top of the stack, the result of this is **t = *(t-1) / *t*. The stack depth decreases by one at the end of this operation.
compiler actions: Generate the following code:
```
bc.div
```

Values of bytecodes:

no bytecode values
lab
subr

comparison bytecodes
cmpe: 132, or 10000100

cmplt: 136, or 10001000
cmpgt: 140, or 10001100

control flow bytecodes
jmp: 36, or 00100100
jmpc: 40, or 00101000
~~call: 44, or 00101100~~
~~callr: 48, or 00110000~~
~~ret: 52, or 00110100~~
~~retr: 56, or 00111000~~

stack manipulation byte codes
push<type>: 68+type, or 010001xx, where xx is 00 for char (pushc), 01 is for short (pushs), 10 is for int (pushi), 11 is for float (pushf).
popm: 76, or 01001100
popv: 80, or 010100, where the type is derived from the stack entry
peek: 84 or 01011000, where the type is derived from the stack entry
poke: 88 or 011000xx, where xx is 00 for char (pushc), 01 is for short (pushs), 10 is for int (pushi), 11 is for float (pushf).

arithmetic byte codes
add: 100, or 01100100
sub: 104, or 01101000
mul: 108, or 01101100
div: 112, or 01110000

special op codes
print, 132, or 10000100

Instruction Semantics:

Values of bytecodes

no bytecode values
lab
subr

comparison bytecodes
cmpe: 132, or 10000100
cmplt: 136, or 10001000
cmpgt: 140, or 10001100

control flow bytecodes
jmp: 36, or 00100100
jmpc: 40, or 00101000
call: 44, or 00101100
ret: 52, or 00110100
retr: 56, or 00111000

stack manipulation byte codes
push<type>: 68+type, or 010001xx, where xx is 00 for char (pushc), 01 is for short (pushs), 10 is for int (pushi), 11 is for float (pushf).
pushv<type>: 72, or 01001000, where the type is the variable type
popm: 76, or 01001100
popv: 80, or 01010000, where the type is derived from the stack entry
peek: 84 or 01011000, where the type is derived from the stack entry
poke<type>: 88 or 01100000, where xx is 00 for char (pushc), 01 is for short (pushs), 10 is for int (pushi), 11 is for float (pushf).

arithmetic byte codes
add: 100, or 01100100
sub: 104, or 01101000
mul: 108, or 01101100
div: 112, or 01110000

special op codes
print<type>, 132, or 10000100, where xx is 00 for char (printc), 01 is for short (prints), 10 is for int (printi), 11 is for float (printf).
print, 136, or 10001000