YASL Intermediate Code Generation

Brian Howard

Fall 2015, Version 1.1.2

Generate an entire program by creating an empty symbol table that maps identifiers to `Info`, then generate the block inside a new scope whose name is the program name. At the end, halt execution:

$$\mathcal{G}[\![\texttt{program ID ; Block .}]\!] = \begin{cases} \texttt{t = new SymbolTable[Info]} \\ \texttt{t.enter(ID)} \\ \mathcal{G}[\![\texttt{Block}]\!](t) \\ \texttt{t.exit()} \\ (\!|\texttt{HALT}|\!) \end{cases}$$

Generate a block by generating in turn each of the declarations, followed by each of the statements of the block. Create a new label, s, to skip around any generated procedure code. Wrap the body of the block in `ENTER`/`EXIT` and `RESERVE`/`DROP` pairs to manage local variables:

$$\mathcal{G}[\![\texttt{ConstDecls VarDecls ProcDecls begin Stmts end}]\!](t) = \begin{cases} s \texttt{ = newLabel} \\ (\!|\texttt{BRANCH } s|\!) \\ \texttt{foreach cd} \in \texttt{ConstDecls: } \mathcal{G}[\![\texttt{cd}]\!](t) \\ \texttt{t.offset = 0} \\ \texttt{foreach vd} \in \texttt{VarDecls: } \mathcal{G}[\![\texttt{vd}]\!](t) \\ \texttt{foreach pd} \in \texttt{ProcDecls:} \\ \quad \texttt{t.bind(pd.ID, new ProcInfo(newLabel, pd.Params))} \\ \texttt{foreach pd} \in \texttt{ProcDecls: } \mathcal{G}[\![\texttt{pd}]\!](t) \\ (\!|\texttt{LABEL } s|\!) \\ n \texttt{ = VarDecls.size;} \ell \texttt{ = t.level} \\ (\!|\texttt{ENTER } \ell|\!);(\!|\texttt{RESERVE } n|\!) \\ \texttt{foreach st} \in \texttt{Stmts: } \mathcal{G}[\![\texttt{st}]\!](t) \\ (\!|\texttt{DROP } n|\!);(\!|\texttt{EXIT } \ell|\!) \end{cases}$$

Generate a constant declaration by binding the identifier to a new `ConstInfo` holding the number:

$$\mathcal{G}[\![\texttt{const ID = NUM ;}]\!](t) = \texttt{t.bind(ID, new ConstInfo(NUM))}$$

Generate a variable declaration by binding the identifier to a new `VarInfo` holding the level of the current scope and the next available offset in the scope (starting at -1 and counting down):

$$\mathcal{G}[\![\texttt{var ID : Type ;}]\!](t) = \begin{cases} \texttt{t.offset = t.offset - 1} \\ \texttt{t.bind(ID, new VarInfo(t.level, t.offset))} \end{cases}$$

Generate a procedure declaration by binding the identifier to a new `ProcInfo` holding the list of parameters and a new label (this already happened above). Within a new scope where all of the parameters have been processed *in reverse*, generate code for the block preceded by the label and ending with a return:

$$\mathcal{G}[\![\texttt{proc ID Params ; Block ;}]\!](t) = \begin{cases} \texttt{ProcInfo(}s\texttt{, params) = t.lookup(ID)} \\ \texttt{t.enter(ID)} \\ \texttt{t.param = 1} \\ \texttt{foreach p} \in \texttt{Params.reverse: } \mathcal{G}[\![\texttt{p}]\!](t) \\ (\!|\texttt{LABEL } s|\!) \\ \mathcal{G}[\![\texttt{Block}]\!](t) \\ (\!|\texttt{RETURN}|\!) \\ \texttt{t.exit()} \end{cases}$$

Generate a value parameter declaration by binding the identifier to a new `VarInfo` holding the level of the current scope and the next available parameter offset in the scope (starting at 2 and counting up):

$$\mathcal{G}[\![\texttt{ID : Type}]\!](t) = \begin{cases} \texttt{t.param = t.param + 1} \\ \texttt{t.bind(ID, new VarInfo(t.level, t.param)} \end{cases}$$

Generate a variable parameter declaration by binding the identifier to a new `RefInfo` holding the level of the current scope and the next available parameter offset in the scope (starting at 2 and counting up):

$$\mathcal{G}[\![\texttt{var ID : Type}]\!](t) = \begin{cases} \texttt{t.param = t.param + 1} \\ \texttt{t.bind(ID, new RefInfo(t.level, t.param)} \end{cases}$$

Generate an assignment statement by generating the expression on the right, looking up the identifier on the left to find the address to store into, and then performing the store operation. The `lvalue` function generates the code to push the address of either a simple variable or a reference (var parameter):

$$\mathcal{G}[\![\texttt{ID = Expr ;}]\!](t) = \begin{cases} \mathcal{G}[\![\texttt{Expr}]\!](t) \\ \texttt{lvalue(ID, t)} \\ (\!|\,\texttt{STORE}\,|\!) \end{cases}$$

$$\texttt{lvalue(ID, t)} = \begin{cases} \texttt{info = t.lookup(ID)} \\ (\!|\,\texttt{ADDRESS info.level, info.offset}\,|\!) \\ \texttt{if info is a RefInfo: } (\!|\,\texttt{LOAD}\,|\!) \end{cases}$$

Generate a procedure call by looking up the identifier to get a `ProcInfo`, then generating the list of arguments (so each will be pushed to the stack)—this is handled by the `setup` function. A call statement is generated; when the procedure returns, the arguments must be dropped:

$$\mathcal{G}[\![\texttt{ID Args ;}]\!](t) = \begin{cases} \texttt{ProcInfo(}s\texttt{, params) = t.lookup(ID)} \\ \texttt{setup(params, Args, t)} \\ (\!|\,\texttt{CALL }s\,|\!) \\ (\!|\,\texttt{DROP params.size}\,|\!) \end{cases}$$

The `setup` function is defined by the following cases (note that `Nil` is an empty list, while `head :: tail` is a list with the given `head` element and rest of the list `tail`):

$$\begin{cases} \texttt{setup(Nil, Nil, t) = // nothing left} \\ \texttt{setup((ID1 : Type) :: params, arg :: args, t) =} \\ \quad \mathcal{G}[\![\texttt{arg}]\!](t) \\ \quad \texttt{setup(params, args, t)} \\ \texttt{setup((var ID1 : Type) :: params, ID2 :: args, t) =} \\ \quad \texttt{lvalue(ID2, t)} \\ \quad \texttt{setup(params, args, t)} \end{cases}$$

Generate a compound statement by generating each of the contained statements in turn:

$$\mathcal{G}[\![\texttt{begin Stmts end ;}]\!](t) = \texttt{foreach s} \in \texttt{Stmts: } \mathcal{G}[\![\texttt{s}]\!](t)$$

Generate an if-then statement by generating the test with two new labels—if true, branch to the statement; if false, branch past the statement:

$$\mathcal{G}[\![\texttt{if Expr then Stmt}]\!](t) = \begin{cases} y \texttt{ = newLabel} \\ n \texttt{ = newLabel} \\ \mathcal{G}[\![\texttt{Expr}]\!](t, y, n) \\ (\!|\texttt{ LABEL } y |\!) \\ \mathcal{G}[\![\texttt{Stmt}]\!](t) \\ (\!|\texttt{ LABEL } n |\!) \end{cases}$$

Generate an if-then-else statement by generating the test with two new labels—if true, branch to the first statement; if false, branch to the second statement. A third new label is used to branch past the second statement:

$$\mathcal{G}[\![\texttt{if Expr then Stmt1 else Stmt2}]\!](t) = \begin{cases} y \texttt{ = newLabel} \\ n \texttt{ = newLabel} \\ s \texttt{ = newLabel} \\ \mathcal{G}[\![\texttt{Expr}]\!](t, y, n) \\ (\!|\texttt{ LABEL } y |\!) \\ \mathcal{G}[\![\texttt{Stmt1}]\!](t) \\ (\!|\texttt{ BRANCH } s |\!) \\ (\!|\texttt{ LABEL } n |\!) \\ \mathcal{G}[\![\texttt{Stmt2}]\!](t) \\ (\!|\texttt{ LABEL } s |\!) \end{cases}$$

Generate a while statement by generating the test with two new labels—if true, branch to the statement; if false, branch past the statement. A third new label is used to branch from the end of the statement back to the test:

$$\mathcal{G}[\![\texttt{while Expr do Stmt}]\!](t) = \begin{cases} y \texttt{ = newLabel} \\ n \texttt{ = newLabel} \\ s \texttt{ = newLabel} \\ (\!|\texttt{ LABEL } s |\!) \\ \mathcal{G}[\![\texttt{Expr}]\!](t, y, n) \\ (\!|\texttt{ LABEL } y |\!) \\ \mathcal{G}[\![\texttt{Stmt}]\!](t) \\ (\!|\texttt{ BRANCH } s |\!) \\ (\!|\texttt{ LABEL } n |\!) \end{cases}$$

Generate a prompt statement by printing the message string (without a trailing newline) and waiting for a line of input. If an identifier is specified along with the string, then generate code to read an integer and assign it to that variable:

$$\mathcal{G}[\![\texttt{prompt STRING ;}]\!](t) = \begin{cases} \texttt{print(STRING)} \\ (\!|\,\texttt{READLINE}\,|\!) \end{cases}$$

$$\mathcal{G}[\![\texttt{prompt STRING , ID ;}]\!](t) = \begin{cases} \texttt{print(STRING + " ")} \\ (\!|\,\texttt{READINT}\,|\!) \\ \texttt{lvalue(ID)} \\ (\!|\,\texttt{STORE}\,|\!) \end{cases}$$

The `print` function generates code to print a string one character at a time:

$$\texttt{print(STRING)} = \begin{cases} \texttt{foreach c} \in \texttt{STRING:} \\ \quad (\!|\,\texttt{CONSTANT c}\,|\!) \\ \quad (\!|\,\texttt{WRITECHAR}\,|\!) \end{cases}$$

Generate a print statement by generating code to print each of its items (strings or integer expressions) on a single line:

$$\mathcal{G}[\![\texttt{print Items ;}]\!](t) = \begin{cases} \texttt{foreach it} \in \texttt{Items:} \\ \quad \texttt{if it is an Expr:} \\ \quad\quad \mathcal{G}[\![\texttt{it}]\!](t) \\ \quad\quad (\!|\,\texttt{WRITEINT}\,|\!) \\ \quad \texttt{else: // it is a STRING} \\ \quad\quad \texttt{print(it)} \\ (\!|\,\texttt{WRITELINE}\,|\!) \end{cases}$$

Generate a boolean binary operation expression as described in class. For relational operators, generate code to evaluate each operand and put the result on the stack, then generate appropriate tests and branches. For logical operators, generate code to evaluate the left operand and then either branch to the code for the right operand or branch directly to the provided "yes" or "no" label:

$$\mathcal{G}[\![\texttt{Expr1 op Expr2}]\!](t, y, n) = \begin{cases} \texttt{switch op:} \\ \quad \texttt{case And:} \\ \quad\quad s \texttt{ = newLabel} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t, s, n) \\ \quad\quad (\!|\texttt{ LABEL } s \texttt{ }|\!) \\ \quad\quad \mathcal{G}[\![\texttt{Expr2}]\!](t, y, n) \\ \quad \texttt{case Or:} \\ \quad\quad s \texttt{ = newLabel} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t, y, s) \\ \quad\quad (\!|\texttt{ LABEL } s \texttt{ }|\!) \\ \quad\quad \mathcal{G}[\![\texttt{Expr2}]\!](t, y, n) \\ \quad \texttt{case EQ:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t); \mathcal{G}[\![\texttt{Expr2}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHZERO } y \texttt{ }|\!); (\!|\texttt{ BRANCH } n \texttt{ }|\!) \\ \quad \texttt{case NE:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t); \mathcal{G}[\![\texttt{Expr2}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHZERO } n \texttt{ }|\!); (\!|\texttt{ BRANCH } y \texttt{ }|\!) \\ \quad \texttt{case LT:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t); \mathcal{G}[\![\texttt{Expr2}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHNEG } y \texttt{ }|\!); (\!|\texttt{ BRANCH } n \texttt{ }|\!) \\ \quad \texttt{case GE:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr1}]\!](t); \mathcal{G}[\![\texttt{Expr2}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHNEG } n \texttt{ }|\!); (\!|\texttt{ BRANCH } y \texttt{ }|\!) \\ \quad \texttt{case GT:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr2}]\!](t); \mathcal{G}[\![\texttt{Expr1}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHNEG } y \texttt{ }|\!); (\!|\texttt{ BRANCH } n \texttt{ }|\!) \\ \quad \texttt{case LE:} \\ \quad\quad \mathcal{G}[\![\texttt{Expr2}]\!](t); \mathcal{G}[\![\texttt{Expr1}]\!](t) \\ \quad\quad (\!|\texttt{ SUB }|\!); (\!|\texttt{ BRANCHNEG } n \texttt{ }|\!); (\!|\texttt{ BRANCH } y \texttt{ }|\!) \end{cases}$$

Generate an integer binary operation expression by generating code to evaluate each operand, then performing the desired operation.

$$\mathcal{G}[\![\texttt{Expr1 op Expr2}]\!](t) = \begin{cases} \mathcal{G}[\![\texttt{Expr1}]\!](t) \\ \mathcal{G}[\![\texttt{Expr2}]\!](t) \\ \texttt{switch op:} \\ \quad \texttt{case Plus:} \\ \quad\quad (\!|\, \texttt{ADD} \,|\!) \\ \quad \texttt{case Minus:} \\ \quad\quad (\!|\, \texttt{SUB} \,|\!) \\ \quad \texttt{case Times:} \\ \quad\quad (\!|\, \texttt{MUL} \,|\!) \\ \quad \texttt{case Div:} \\ \quad\quad (\!|\, \texttt{DIV} \,|\!) \\ \quad \texttt{case Mod:} \\ \quad\quad (\!|\, \texttt{MOD} \,|\!) \end{cases}$$

Generate a boolean unary operation expression (the `not` operator) by generating code to evaluate the operand with the y and n labels swapped:

$$\mathcal{G}[\![\texttt{not Expr}]\!](t, y, n) = \mathcal{G}[\![\texttt{Expr}]\!](t, n, y)$$

Generate an integer unary operation expression (the `-` operator) by generating code to subtract the expression from zero:

$$\mathcal{G}[\![\texttt{- Expr}]\!](t) = \begin{cases} (\!|\, \texttt{CONSTANT 0} \,|\!) \\ \mathcal{G}[\![\texttt{Expr}]\!](t) \\ (\!|\, \texttt{SUB} \,|\!) \end{cases}$$

Generate a boolean literal expression by branching to the appropriate label:

$$\mathcal{G}[\![\texttt{true}]\!](t, y, n) = (\!|\, \texttt{BRANCH } y \,|\!)$$
$$\mathcal{G}[\![\texttt{false}]\!](t, y, n) = (\!|\, \texttt{BRANCH } n \,|\!)$$

Generate an integer or boolean literal expression (in an integer context) by pushing the corresponding value:

$$\mathcal{G}[\![\texttt{NUM}]\!](t) = (\!|\, \texttt{CONSTANT } n \,|\!)\texttt{, if the NUM is } n$$
$$\mathcal{G}[\![\texttt{true}]\!](t) = (\!|\, \texttt{CONSTANT 1} \,|\!)$$
$$\mathcal{G}[\![\texttt{false}]\!](t) = (\!|\, \texttt{CONSTANT 0} \,|\!)$$

Generate a boolean variable expression by pushing its address and performing a load, then branching to the appropriate label:

$$\mathcal{G}[\![\texttt{ID}]\!](t, y, n) = \begin{cases} \texttt{lvalue(ID)} \\ (\!|\, \texttt{LOAD} \,|\!) \\ (\!|\, \texttt{BRANCHZERO } n \,|\!) \\ (\!|\, \texttt{BRANCH } y \,|\!) \end{cases}$$

Generate an integer variable expression by pushing its address and performing a load. If it is a constant, just push its value:

$$\mathcal{G}[\![\texttt{ID}]\!](t) = \begin{cases} \texttt{info = t.lookup(ID)} \\ \texttt{if info is ConstInfo(}n\texttt{):} \\ \quad (\!|\, \texttt{CONSTANT } n \,|\!) \\ \texttt{else:} \\ \quad \texttt{lvalue(ID)} \\ \quad (\!|\, \texttt{LOAD} \,|\!) \end{cases}$$

If a boolean expression is used in an integer context, wrap it in code to push either 0 (false) or 1 (true) on the stack. Note that this case is only used if none of the above applied:

$$\mathcal{G}[\![\texttt{Expr}]\!](t) = \begin{cases} y \texttt{ = newLabel} \\ n \texttt{ = newLabel} \\ s \texttt{ = newLabel} \\ \mathcal{G}[\![\texttt{Expr}]\!](t, y, n) \\ (\!|\, \texttt{LABEL } y \,|\!) \\ (\!|\, \texttt{CONSTANT 1} \,|\!) \\ (\!|\, \texttt{BRANCH } s \,|\!) \\ (\!|\, \texttt{LABEL } n \,|\!) \\ (\!|\, \texttt{CONSTANT 0} \,|\!) \\ (\!|\, \texttt{LABEL } s \,|\!) \end{cases}$$

The symbol table needs one additional method: `t.level()` returns the level number of the topmost scope (the first scope entered has level 0, and each successive scope entered has a level one more than its parent). If the symbol table is a stack of maps, then this is just the size of the stack minus one. The symbol table also holds an `offset` for the latest local variable, and a `param` offset for the latest generated parameter of the current procedure.

The `newLabel` operation creates a unique label each time it is called. An easy way to do this is to maintain a global sequence number, n, and return the next label (`_`n) in the sequence `_0`, `_1`, ... on each call.

The symbol table needs to store the information about each symbol to know what it is and where it can be found. There are four kinds of `Info` values needed:

- `ConstInfo(n)` records the value associated with an integer constant.
- `VarInfo(level, offset)` and `RefInfo(level, offset)` both tell the scope level (and hence, what frame pointer in the display) and offset within that level where the variable can be found; a `VarInfo` is for an ordinary local variable or value parameter, which stores an integer, while a `RefInfo` is for a reference (var) parameter, which stores the address (pointer) of the actual value.
- `ProcInfo(s, params)` reflects a procedure declaration, where `s` is the label of the start of the procedure and `params` is a list of the parameters (which is needed to know how many parameters are expected and which ones of them are reference parameters).

Be sure to notice the difference between expressions being generated in an integer context $\mathcal{G}[\![\texttt{Expr}]\!](t)$, where the code will result in a value being pushed on the stack, and expressions being generated in a boolean context $\mathcal{G}[\![\texttt{Expr}]\!](t, y, n)$, where the code will result in branching to either label y ("yes") if true or label n ("no") if false.

Stack Frame

If we have a procedure with k parameters and n local variables, then the stack frame will be laid out as follows:

	temp. stack
$\texttt{FP} - n$	$\texttt{local}_n$
	$\cdots$
$\texttt{FP} - 1$	$\texttt{local}_1$
FP	*saved* FP
$\texttt{FP} + 1$	*return addr.*
$\texttt{FP} + 2$	$\texttt{param}_k$
	$\cdots$
$\texttt{FP} + (k + 1)$	$\texttt{param}_1$

For a procedure defined at level ℓ (where the main program is at level 0), its frame pointer (`FP`) will be stored in `display[`ℓ`]`.