CSC488 Winter 2014 A4 - Code Generation Templates

Thursday, March 19, 2014

Group 09:
g2mccorm - Peter McCormick
g0dbloem - Daniel Bloemendal
g2watson - Watson Oren Isaac
c4qindai - Mike Qin
g2scotts - Simon Scott

1. Introduction

In this document we will show how our group plans to generate pseudo-machine assembly for the CSC488 language in the final assignment. We intend to implement an AST tree walker using the visitor pattern, which will generate a lightly sugared assembly intermediate representation (IR). Our IR will include the following:

- Definition of named labels
- Symbolic constants as operands to PUSH (including the address of labels)
- An extended instruction set using macro conveniences (where the macros expand at assembly-time to a well-defined sequence of underlying machine instructions, given below)

We will implement a final IR-to-machine-code pass which will produce executable code after reifying all symbolic names to constant values, as well as an IR pretty-printer as a development aid.

In this document we will use our IR to specify the code generation templates, and describe both the overall structure and how code will be generated for each AST node type. We will utilize the following conventions:

- At the point in the templates that our code generator will call upon some method to inject code for some child node, we will signify this with “>>” plus some remarks
- Here is a typical syntax for the IR:

```
# This is a comment, but the next line is a literal pseudo-machine instruction
PUSH True   # pushes MACHINE_TRUE onto stack
PUSH False  # pushes MACHINE_FALSE
# Stack (read left to right, top down towards bottom): False, True
DUP
# Stack: False, False, True

>> inject code for AST node XYZ
```

```
PUSH _label  # target label address as symbolic constant
%JMP _label  # macro instruction
HALT # never executed
_label: # label definition
        # At this point we are done
```

In what follows, LHS and RHS stand for left-hand and right-hand side, respectively.

2. Storage & Activation Frames

Occasionally a temporary variable is required locally within a block of generated code, and our convention will be that the highest available memory `memorySize - 1` will be reserved for that use. Consequently, this means that the system MLP pointer will be bounded above by `memorySize - 2`.

Our code generator will lay out activation frame according to major and minor scopes: nested functions create new major scopes (which are at a +1 lexical level) and all others are minor. In the standard sense, two sibling minor scopes with declared variables will have overlapping space reserved in the activation frame, since those variables will never be live at the same instant. Integers and booleans will each occupy 1 word in the frame, and arrays their natural size given their dimensionality and upper/lower bound(s).

Our generated code uses the display, indexed by the lexical level of a major scope, starting from 0 at the top level and ascending as scopes nest. Our routine calling and scope management conventions dictate that following:

- The caller pushes space for all the negative offset items below (return value, the return branch address and 0...N arguments)
- The callee will be responsible to pop the N arguments
- The caller can expect to have execution continue afterwards at the return address, at which point it must pop the return value (even if it is unused, as in the case of a procedure.)
- Any major scope which reserves space for an activation frame must clean it up afterwards
- A major scope is responsible for saving the previous display address for its lexical level (by design at offset ON=0, see below), and restoring it before returning

For ADDR <LL> ON:

ON=-N - 2	return value (always present, but ignored if procedure)
ON=-N - 1	return address
ON=-N	argument 1
...	...

ON=-1	argument N
ON=0	saved value of previous ADDR <LL> 0 (offset 0 of display at level <LL>)
ON=1	1st word of local variable memory in activation frame
...	...
ON=M	M'th word of local variable memory in activation frame

3. Macro Conveniences

Our IR includes an instruction set enriched by the following pseudo-instructions, which will be expanded at assembly-time to raw instructions. Label names will be guaranteed unique to a given instance of a macro expansion, unless it is a label name passed in as an operand to the instruction itself:

%JMP <label> - unconditional jump to <label>:

```
PUSH <label>
BR
```

%BFALSE <label> - branch to <label> if top of stack is False:

```
PUSH <label>
BF
```

%NOT - Pop a boolean value off the stack, and push its logical negation:

```
PUSH MACHINE_FALSE
EQ
```

%SAVE_CONTEXT <LL> - save previous <LL> display value, and start a new scope:

```
ADDR <LL> 0
PUSHMT
SETD <LL>
```

%RESTORE_CONTEXT <LL> <NARGS> - discard scope and restore previous <LL> display:

```
ADDR <LL> 0
SETD <LL>
POP % discard old `ADDR <LL> 0` save
PUSH <NARGS>
POPN
```

%RESERVE <num_words> - reserves <num_words> words for local variables

```
PUSH 0
PUSH <words>
```

```
DUPN
```

%PUTSTR <"string"> - prints the string
large amount of code, see AssemblerIREmitter::emitPutString(). Code length 23.

%PUTNEWLINE – prints the newline character

```
PUSH '\n'
PRINTC
```

%SETUPCALL <label> - setup return value and return address, label is the address of the final BR instruction

```
PUSH 0
PUSH <label> + 1
```

Example:

```
SETUPCALL __call_main_foo
…
PUSH foo
__call_main_foo: BR
```

4. Evaluating Expressions

Every expression in a correctly type-checked 488 source language program will evaluate to either an integer or boolean value. Anytime our tree walker calls for code to be generated corresponding to the evaluation of an expression, the following conventions will be adhered to:

- The generated expression code MUST assumes nothing about the contents of the stack
- While the expression code may internally contain labels and branches, at the point it is injected it MUST flow execution through to the next statement following that injection point
- Once completed, the stack MUST have exactly one new element on top, typed according to the expressions' evaluation type
- The surrounding code MUST be responsible for inspecting and popping that evaluated value and to restore the stack (if it so desires) to the state it was in prior to calling the expression evaluation code

With these ground rules in place, we will give short template sequences corresponding to the evaluation code for each expression node type in our AST class hierarchy. Where labels are used, they are understood to be local to the template, and our implementation in A5 will generate appropriately unique label identifiers to avoided any conflicts.

4.1. The Printable Interface

The parent class Expn implements the Printable interface, which works in conjunction with the PUT statement to generate code to evaluate themselves and print the result. While `NewlineConstExpn` and `TextConstExpn` will override this to provide their own implementaions, parent `Expn` will provide the following as default.

```
>> evaluate own self expression
# only if evaluated type is integer:
        PRINTI

# only if evaluated type is boolean:
        %BFALSE _false:
        PUSH 'T'  # for ASCII character 'T'
        %JMP _end
        _false:
        PUSH 'F'
        _end:
        PRINTC
```

4.2. BinaryExpn

There are four subclasses of BinaryExpn in the AST: ArithExpn, CompareExpn, EqualsExpn and BoolExpn. Each will generally evaluate both the left and right operands (themselves expressions) and then perform an operation to produce their result, although BoolExpn may not evaluate its right operand due to short circuiting. In each case, the type checker will have guaranteed that both the left and right expressions will evaluate to a type appropriate for the operation.

4.2.1. ArithExpn

The AST for arithmetic expressions come in four varieties: plus, minus, times and divide. Fortunately these map directly to opcodes in the pseudo-machine: ADD, SUB, MUL and DIV, respectively. An arithmetic expression takes integers as operands and evaluates to an integer.

Template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
ADD/SUB/MUL/DIV
```

4.2.2. CompareExpn

The AST for comparison expression come in four varieties: less than, less than or equal to, greater than, and greater than or equal to. The operands will be integers, and the comparison expression itself will evaluate to a boolean. Unlike in the arithmetic expression sub-cases, the pseudo-machine only has two comparison opcodes, one for less-than and a second for equals-to comparisons, meaning that we will have some more work to do for the other three. We will, however, maintain two conditions: the LHS and RHS sub-expressions are evaluated exactly once, and the left expression is evaluated first, before the right.

Less-than template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
LT
```

Greater-than-or-equal-to template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
# (L >= R) == !(L < R)
LT
%NOT
```

Less-than-or-equal-to template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
# (L <= R) == !(R < L)
SWAP
LT
%NOT
```

Greater-than template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
# (L > R) == (R < L)
SWAP
LT
```

4.2.3. EqualsExpn

This AST node comes in a straight-ahead equals and a negated, not-equals variety.

Equals template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
EQ
```

Not-equals template:

```
>> inject evaluation of LHS
>> inject evaluation of RHS
EQ
%NOT
```

4.2.4. BoolExpn

OR Template:

```
>> inject evaluation of LHS
DUP
# Stack: LHS, LHS
# If LHS==False, will jump to _checkRHS
%BFALSE _checkRHS

# Stack: LHS==True (hence we can short-circuit)
%JMP _theEnd

_checkRHS:
# Stack: False (from LHS==False)
# False || RHS == RHS
POP
>> inject evaluation of RHS
# Stack: RHS

_theEnd:
```

AND Template:

```
>> inject evaluation of LHS
DUP
```

```
# Stack: LHS, LHS

# If the following branch is taken, then LHS==False and
# remaining stack top is False,  i.e. False && RHS == False
%BFALSE _theEnd

# By this point, True && RHS == RHS
# Stack: True (because LHS==True, otherwise the branch would have been taken)
POP
# Stack: (empty)

>> inject evaluation of RHS
# Stack: RHS

_theEnd:
```

4.3. UnaryExpn

There are two types of unary expressions in the AST, one for boolean value operands, and the other for integers.

4.3.1. NotExpn

```
>> inject evaluation of boolean-valued expression
PUSH False
EQ
```

4.3.2. UnaryMinusExpn

```
>> inject evaluation of integer-valued expression
NEG
```

4.4. ConstExpn

In the AST there are two kinds of constant expressions: the kind that can occur anywhere (integer or boolean valued), and the kind that only ever occur as arguments to PUT (those classe implement the Printable interface.)

4.4.1. IntConstExpn

```
PUSH <integer constant>
```

4.4.2. BoolConstExpn

```
# if the AST was for a True value
PUSH MACHINE_TRUE
```

```
# otherwise, if for a False:
PUSH MACHINE_FALSE
```

4.4.3. NewlineConstExpn

By convention, AST nodes which implement Printable can generate code to evaluate themselves and print the result. To echo a new line, this is not too difficult.

```
# assuming UNIX newline ('\n' aka 0x0A aka 10) convention.
PUSH 10
PRINTC
```

4.4.4. TextConstExpn

By convention, AST nodes which implement Printable can generate code to evaluate themselves and print the result. To print a string constant, this node will make use of an auxiliary support routine `_PUTS` (full definition given in an appendix.) At assembly time, code address space will be reserved to locate a constant wide ASCIIZ string, that is, consecutive 16-bit wide, zero extended 8-bit ASCII characters with a trailing NUL word to signal the end.

```
PUSH _return
PUSH <address of string constant>
%JMP _PUTS  # the globally visible address label where the support routine is located
_return:
```

4.5. ConditionalExpn

```
        >> evaluate conditional expression
        %BFALSE _wasFalse
        >> evaluate true case expression
        %JMP _end
_wasFalse:
        >> evaluate false case expression
_end:
```

4.6. FunctionCallExpn

This follows the calling convention described in the `Storage` section.

```
PUSH 0
PUSH _return
PUSH <arg_1>
PUSH <arg_…>
PUSH <arg_n>
%JMP _target
_return:
```

4.7. VarRefExpn

A variable reference can either appear in a LHS or RHS context. When generating code for this node in an LHS context, the convention is that upon executing the resultant code, the stack top result will be a memory address, while if generating for an RHS context, executing the code will leave the actual value at the top of the stack. The only time this node appears in a LHS context is as the L-value for an `AssignStmt`, in all other instances it will be RHS.

At code generation time, the symbol table will be searched to find the correct lexical level <LL> of the identifier relative to the context of this AST node, and its display-relative offset <ON> in the activation frame for the encompassing scope. If the identifier is a function or procedure argument, <ON> will be in the range of -N-1 to -1 inclusive (where N is the number of arguments for that routine) and if a local variable, it will be between 1 and M inclusive (where M is the number of words allocated to the local variables of given lexical level scope.)

4.7.1. IdentExpn

```
ADDR <LL> <ON>
# only if used in a RHS context:
        LOAD
```

4.7.2. SubsExpn

```
ADDR <LL> <ON>
>> evaluate subscript 1 expression
DUP # of sub_1
PUSH <lowerbound_1>
LT # is sub_1  < lowerbound_1 ?
%BFALSE _checkUpper1
%JMP _BOUNDS_ERROR # a bound error handler/reporter, see appendix

_checkUpper1:
        DUP # of sub_1
        PUSH <upperbound_1>
        SWAP
        LT # is upperbound_1 < sub_1 ?
        %BFALSE _sub1Ready
        %JMP bound_error # a bound error handler/reporter

_sub1Ready:
        PUSH <lowerbound_1>
        SUB

# Stack: (sub_1 - lowerbound_1), base array address
```

```
# only if 1D array:
	ADD
	# Stack: (address of array element at [sub_1])

# only if 2D array:
	PUSH <stride_size>
	MUL
	# Stack: offset from base of first element in the second dimension at first
dimension index `sub_1`, base array address
	ADD

	>> evaluate subscript 2 expression
	DUP # of sub_2
	PUSH <lowerbound_2>
	LT # is sub_2  < lowerbound_2 ?
	%BFALSE _checkUpper
	%JMP _BOUNDS_ERROR

	_checkUpper2:
		DUP # of sub_2
		PUSH <upperbound_2>
		SWAP
		LT # is upperbound_2 < sub_2 ?
		%BFALSE _sub2Ready
		%JMP bound_error #FIXME

	_sub2Ready:
		PUSH <lowerbound_2>
		SUB

	ADD
	# Stack: (address of element at [sub_1][sub_2])

# only if used in a RHS context:
	LOAD
```

5. Functions/Procedures

The code generator for routines will provide the body scope and statements with an distinguished function epilogue label, which can be used by RESULT and RETURN statements to leave.

5.1. Prologue

%SAVE_CONTEXT <LL>
%RESERVE <locals_size>

5.2. Epilogue

<function epilogue label>:
PUSH <locals_size>
POPN
%RESTORE_CONTEXT <LL> <argument_count>

6. Statements

6.1. Program

Just emit code for the enclosing Scope.

6.2. Scope

If the scope is a major scope (i.e. a function), generating code for it will include the prologue and epilogue code. All scopes, both major and minor, will provide their body statements with access to a distinguish `end label`, which will be used by EXIT statements. The activation frames will be organized with major scopes beginning a new lexical level, and adjacent and nested minor scopes being packed into a single frame. Generating code for a minor scope will just emit its statements' code, and major scopes will emit those plus a prologue and epilogue as described in the `Functions/Procedures` section.

6.3. PutStmt

Each argument implements the Java interface Printable, and every kind of expression that implements that can emit its own code for evaluating and printing a value. The code generator for the PUT will then simply emit each of these.

6.4. GetStmt

The Java interface Readable is intended to indicate that the AST node can generate code which will push the address of an L-value expression (i.e. a local variable or an element of an array.) Only VarRefExpn implements it, which covers both the scalar and array element cases.

For each parameter:

```
>> evaluate Readable to give L-value address
READI
STORE
```

6.5. AssignStmt

```
>> evaluate VarRefExpn l-value in LHS context (gives an address)
>> evaluate r-value expresssion
SWAP # so that we evaluate all expressions left to right in source program order
STORE
```

6.6. ProcedureCallStmt

```
PUSH 0 # return value
PUSH _return # return code address
PUSH <arg_1>
PUSH <arg_…>
PUSH <arg_n>
%JMP _target
_return:
POP # ignore return value
```

6.7. If Statement

```
>>> inject evaluation of condition expression
PUSH _else
BF
>>> inject code for ``whenTrue'' statement list
%JMP _end
_else:
>>> inject code for ``whenFalse'' statement list
_end:
```

6.8. LoopingStmt

The common parent class for both while-do and repeat-until loops will provide the body statements with an distinguished end label name, which can be used by EXIT statements to jump out.

6.8.1. WhileDoStmt

```
_start:
>> inject evaluation of condition
PUSH _end
BF
>> inject code for ``body`` statement list
PUSH _start
```

```
BR
_end:
```

6.8.2. RepeatUntilStmt

```
_start:
>> inject code for ``body`` statement list
>> inject evaluation of condition
%NOT
PUSH _start
BF
_end: #needed for exit even if not used normally.
```

6.9. ReturnStmt

```
%JMP <function epilogue label>
```

6.10. ResultStmt

Given the encompassing function scope, at compile time we will know the value of <ON>, which will be `-N-2`, where N is the number of arguments to that function, and <LL> is its lexical level.

```
>> evaluate value expression
ADDR <LL> <ON>
STORE
%JMP <function epilogue label>
```

6.11. ExitStmt

```
%JMP <end label of enclosing loop>
```

If the ExitStmt actually represents an “EXIT WHEN <condition>”:

```
>> inject evaluation of condition
%NOT
%BFALSE <end label of enclosing loop>
```

7. Conclusion

We are now ready to tackle A5!

Appendix 1 - Support Routines

The _PUTS function is a library support routine designed to print 16-bit wide ASCIIZ strings. It does not follow the usual calling convention. It will be assembled and incorporated into the program code, and the `_PUTS` label itself visible throughout.

_PUTS:

```
_top:
        DUP # of string_addr
        LOAD # Stack: cur_char=*string_addr
        DUP
        PUSH 0
        EQ
        BF _puts_print # if cur_char != 0
        %JMP _puts_done
_puts_print:
        PRINTC # of cur_char
        PUSH 1
        ADD # add 1 to string_addr
        %JMP _top
_puts_done:
        POP # from dup of string_addr
        BR # to _returnPC
```

Example usage:

```
                PUSH _returnPC
                PUSH <string_addr>
                %JMP _puts
        _returnPC:
                # Carry on.
```

Bound Errors:

A global label `_BOUNDS_ERROR` will be available, which will be called if an out-of-bounds subscript is used against an array. We may choose to give a nicer user interface by accurately reporting the source location and mismatch, but for now this in a to-be-determined placeholder, which we will sort out for A5.

Appendix 2 - Sample Programs

Here we are each of the three sample programs hand-assembled.

Sample Program A4-2 (Peter):

% 2-2: {
ADDR 0 0
PUSHMT
SETD 0

% NB: Because scope at 2-2 is the major scope to all enclosing scopes,
% the variables declared at 2-3, 2-4, 2-15, 2-20, 2-25 will have their
% space all be reserved together at a single lexical level
% (they also have to be nested, so there is no overlap)

% 2-3: var a, b, c , d : integer
% a -> ADDR 0 1
% b -> ADDR 0 2
% c -> ADDR 0 3
% d -> ADDR 0 4

% 2-4: var p, q, r : boolean
% p -> ADDR 0 5
% q -> ADDR 0 6
% r -> ADDR 0 7

% 2-15: var b1, b2 : boolean
% b1 -> ADDR 0 8
% b2 -> ADDR 0 9

% 2-20: var w, x , A[100] : integer
% w -> ADDR 0 10
% x -> ADDR 0 11
% A[1] -> ADDR 0 12
% A[100] -> ADDR 0 111

% 2-24: var t, u : integer
% t -> ADDR 0 112
% u -> ADDR 0 113

PUSH 0
PUSH 113 % local size
DUPN

% 2-5: a := ((b + c) - (d * c)) + (b / c)

```
ADDR 0 1 % &a
ADDR 0 2 % b
LOAD
ADDR 0 3 % c
LOAD
ADD % b+c
ADDR 0 4 % d
LOAD
ADDR 0 3 % c
LOAD
MUL % d*c
SUB % (b+c)-(d*c)
ADDR 0 2 % b
LOAD
ADDR 0 3 % c
LOAD
DIV % b/c
ADD % (b+c)-(d*c) + (b/c)
STORE % *a = ...
```

% 2-6: p := (not false) or ((not q) and r)

```
PUSH false % false
PUSH false
EQ % not false
DUP
PUSH _checkRHS_26
BF % if (not false) == false
PUSH _end_26
BR % if (not false) == true, short circuit

_checkRHS_26:
POP
ADDR 0 6 % &q
LOAD % q
PUSH false
EQ % not q
DUP
PUSH _end_26
BF % if (not q) == false, bail out

POP
ADDR 0 7 % &r
LOAD % r

_end_26:
```

% 2-7: if p then a := 3 fi

```
ADDR 0 5 % &p
LOAD % p
PUSH _end_27
BF

ADDR 0 1 % &a
PUSH 3
STORE % a := 3

_end_27:
```

% 2-8: if p or not p then b := 2 else b := 0 fi

```
ADDR 0 5 % &p
LOAD
DUP
PUSH _checkRHS_28
BF
PUSH _end_cond_28
BR

_checkRHS_28:
POP
ADDR 0 5 % &p
PUSH false
EQ

_end_cond_28:

PUSH _else_28
BF

_then_28:
ADDR 0 2 % &b
PUSH 2
STORE % b := 2

PUSH _end_28
BR

_else_28:
ADDR 0 2 % &b
PUSH 0
STORE % b := 0
```

```
_end_28:
```

% 2-9: while c < 7 do c := 6 end

```
_start_29:
        ADDR 0 3 % &c
        LOAD % c
        PUSH 7
        LT
        PUSH _end_29
        BF

        ADDR 0 3 % &c
        PUSH 6 % c
        STORE % c := 6

        PUSH _start_29
        BR

_end_29:
```

% 2-10: while true do b := b + 1 end

```
_start_210:
        PUSH true
        PUSH _end_210
        BF

        ADDR 0 2 % &b
        ADDR 0 2 % &b
        LOAD % b
        PUSH 1
        ADD % b + 1
        STORE % b := b + 1

        PUSH _start_210
        BR

_end_210:
```

% 2-11: repeat { a := 3 exit b := 7 } until false

```
_start_211:

        ADDR 0 1 % &a
        PUSH 3
        STORE % a := 3

        PUSH _end_211
```

```
        BR

        ADDR 0 2 % &b
        PUSH 7
        STORE % b := 7

        PUSH false % <cond>
        PUSH false
        EQ % not <cond>
        PUSH _start_211
        BF

_end_211:
```

% 2-12: while (q or (r and (not p))) do exit when b not= 10 end

```
_start_212:
        ADDR 0 6 % &q
        LOAD % q
        DUP
        PUSH _chechRHS_1_212 % q==false, check RHS
        BF
        PUSH _end_cond1_212 % q==true, true OR _ == true so we're done
        BR

        _checkRHS_1_212:
        POP

        ADDR 0 7 % &r
        LOAD % r
        DUP
        PUSH _end_cond2_212  % q==false, false AND _ == false so we're done
        BF

        POP
        ADDR 0 5 % &p
        LOAD % p
        PUSH false
        EQ % not p

        _end_cond2_212:
        _end_cond1_212:

        PUSH _end_212
        BF

        ADDR 0 2 % &b
```

```
        LOAD % b
        PUSH 10
        EQ % b == 10
        PUSH false
        EQ % b not= 10

        PUSH false
        EQ % not (b not= 10)  NB: according to the `ExitStmt` template!!
        PUSH _end_212
        BF

        PUSH _start_212
        BR

_end_212:
```

% 2-13: put "Value is " , a / b, " or not ", newline

```
PUSH _return_1_213
PUSH _string_constant1 % where _string_constant1 is "Value is \0" somewhere in the code address space
PUSH _PUTS
BR
_return_1_213:

ADDR 0 1
LOAD
ADDR 0 2
LOAD
DIV
PRINTI

PUSH _return_2_213
PUSH _string_constant2 % where _string_constant2 is " or not \0" somewhere in the code address space
PUSH _PUTS
BR
_return_2_213:

PUSH 10
PRINTC
```

% 2-14: while true not= false do {

```
_start_214:
        PUSH true
        PUSH false
        EQ % (true == false)
```

```
PUSH false
EQ % !(true == false) == (true != false)

PUSH _end_214
BF

% 2-16: get a, c, b
ADDR 0 1 % &a
READI
STORE

ADDR 0 3 % &c
READI
STORE

ADDR 0 2 % &b
READI
STORE

% 2-17: exit when (p or not r)
ADDR 0 5 % &p
LOAD % p
DUP
PUSH _checkRHS_217
BF % p==false, so check RHS `not r`
PUSH _end_217
BR

_checkRHS_217:
POP
ADDR 0 7 % &r
LOAD % r
PUSH false
EQ % not r

_end_217:

PUSH false
EQ % not <cond>
PUSH _end_214 % enclosing loop end label
BF

% 2-18: b1 := not p or q
ADDR 0 8 % &b1
ADDR 0 5 % &p
LOAD % p
```

```
PUSH false
EQ % not p
DUP
PUSH _checkRHS_218
BF
PUSH _end_218
BR

_checkRHS_218:
POP
ADDR 0 6 % &q
LOAD % q

_end_218:
STORE % b1 := ...

% 2-19: repeat  {
_start_219:
        % 2-21: p := ( b2 ? q : d <= 7)
        ADDR 0 5 % &p
        ADDR 0 9 % &b2
        LOAD
        PUSH _false_221
        BF

        ADDR 0 6 % &q
        LOAD % q
        PUSH _end_221

        _false_221:
        ADDR 0 4 % &d
        PUSH 7
        SWAP
        LT % 7 < d
        PUSH false
        EQ % !(7 < d) == (d <= 7)

        _end_221:
        STORE % p := ...

        % 2-22: if w <= a then exit fi
        ADDR 0 10 % &w
        LOAD % w
        ADDR 0 1 % &a
        LOAD % a
        SWAP
```

```
LT % a < w
PUSH false
EQ % !(a < w) == (w <= a)

PUSH _else_222
BF

_then_222:
% exit from 2-19 repeat loop
PUSH _end_219
BR

_else_222:
_end_222:
```

% 2-23: while ((p or (not q)) or r) do

```
_start_223:
        ADDR 0 5 % &p
        LOAD % p
        DUP
        PUSH _checkRHS_1_223
        BF % false OR rhs == rhs
        PUSH _end_cond_1_223
        BR % true OR _ == true

        _checkRHS_1_223:
        ADDR 0 6 % &q
        LOAD % q
        PUSH false
        EQ % not q

        _end_cond_1_223:
        DUP
        PUSH _checkRHS_2_223
        BF % false OR r == r
        PUSH _end_cond_2_223
        BR % true OR _ == true

        _chechkRHS_2_223:
        ADDR 0 7 % &r
        LOAD % r

        _end_cond_2_223:

        PUSH _end_223
        BF
```

% 2-26: p := not q

```
ADDR 0 5 % &p
ADDR 0 6 % &q
LOAD % q
PUSH false
EQ % not q
STORE % p := (not q)
```

% 2-27: t := ((p or q) ? (t + 1) : (t - 1))

```
ADDR 0 112 % &t

ADDR 0 5 % &p
LOAD % p
DUP
PUSH _checkRHS_227
BF
PUSH _end_cond_227
BR

_checkRHS_227:
POP
ADDR 0 6 % &q
LOAD % q

_end_cond_227:

PUSH _false_227
BF

_true_227:
ADDR 0 112 % &t
LOAD % t
PUSH 1
ADD % t + 1

_false_227:
ADDR 0 112 % &t
LOAD % t
PUSH 1
SUB % t - 1

_done_227:
STORE % t := ...
```

% 2-28: exit when t > 12

```
    ADDR 0 112 % &t
    LOAD % t
    PUSH 12
    SWAP
    LT % (12 < t) == (t > 12)
    PUSH false
    EQ % !(t > 12)
    PUSH _end_223 % exit from the 2-23 repeat loop
    BF

    % 2-29: }
    % 2-30: end   % while
    PUSH _start_223
    BR

_end_223:

% 2-31: exit when A[w] < d
% <cond> == (A[w] < d)
ADDR 0 12 % &A[lowerbound] -> &A[1]

ADDR 0 10 % &w
LOAD % w
DUP
PUSH 1 % lowerbound of A
LT % w < 1
PUSH _checkUpper_231
BF
PUSH _BOUNDS_ERROR
BR

_checkUpper_231:
DUP % w
PUSH 100 % upperbound of A
SWAP
LT % is upperbound < w ?
PUSH _sub1Ready_231
BF
PUSH _BOUNDS_ERROR
BR

_sub1Ready_231:
PUSH 1 % lowerbound
SUB % offset frome base of `w - 1`
ADD % &A[w] (i.e. base_addr + w - 1)
LOAD % A[w] (because of RHS context)
```

```
        ADDR 0 4 % &d
        LOAD % d
        LT % A[w] < d

        PUSH false
        EQ % not <cond>
        PUSH _end_219 % exit from the 2-19 repeat loop
        BF

        % 2-32: } until p and r  % repeat
        % <cond> == (p and r)
        ADDR 0 5 % &p
        LOAD % p
        DUP
        PUSH _checkRHS_232
        BF
        PUSH _end_cond_232
        BR

        _checkRHS_232:
        POP
        ADDR 0 7 % &r
        LOAD % r

        _end_cond_232:

        PUSH false
        EQ % not <cond>
        PUSH _start_219
        BF

    _end_219:

    % 2-34:   end  % while
    PUSH _start_214
    BR

_end_214:

% 2-35: }
PUSH 113 % local size
POPN % throw away local variables

% restore previous LL=0 context (unnecessary for top-level, but alas..)
ADDR 0 0
```

```
SETD 0
POP

HALT
% The End.
```

Sample Program A4-3 (Oren):

```
_START:
SETD 0    # 3-2
PUSH 0
PUSH 8
DUPN    # 3-3 3-4
PUSH 0
PUSH _RET6
ADDR 0 5    # p
LOAD
PUSH MACHINE_FALSE
EQ
DUP
PUSH _CHECKOR3
BF
PUSH _ENDOR3
BR
_CHECKOR3:
POP
ADDR 0 6    # q
LOAD
_ENDOR3:
ADDR 0 2    # b
LOAD
ADDR 0 3    # c
LOAD
MUL
ADDR 0 5    # p
LOAD
ADDR 0 6    # q
LOAD
EQ
PUSH MACHINE_FALSE
EQ
PUSH _Q
BR
_RET6:
POP     # 3-30
HALT              # 3-31
_P:   ADDR 1 0    # 3-6
   PUSHMT
   SETD 1
   PUSH 0
   PUSH 1
   DUPN          # 3-7
```

```
   _START_WHILE1:
        ADDR 0 6    # q
        LOAD
        PUSH _END_WHILE1
        BF
        PUSH 0
        PUSH _RET1
        ADDR 0 5    # p
        LOAD
        ADDR 1 1    # e
        LOAD
        ADDR 0 1    # a
        LOAD
        ADD
        PUSH 1
        SUB
        ADDR 0 3    # c
        LOAD
        ADDR 0 4    # d
        LOAD
        SWAP
        LT
        PUSH MACHINE_FALSE
        EQ
        PUSH _Q
        BR      # 3-9
        _RET1:
        POP     # unused return
        PUSH _ENDFUNC1
        BR      # 3-10
        PUSH _START_WHILE1
        BR
   _END_WHILE1:    # 3-11
   _ENDFUNC1:
   PUSH 1       # free locals
   POPN
   SETD 1
   BR           # 3-12
_F:   ADDR 1 0
   PUSHMT
   SETD 1       # 3-13
   ADDR 1 -1    # n
   LOAD
   PUSH _ELSE1:
   BF
   ADDR 1 -2    # m
```

```
    LOAD
    ADDR 0 2    # b
    LOAD
    ADD
    PUSH _ENDIF1:
    _ELSE1:
    PUSH 0       # return val
    PUSH _RET2
    ADDR 1 -2    # m
    LOAD
    ADDR 0 2    # b
    LOAD
    SUB
    ADDR 1 -1   # n
    LOAD
    DUP
    PUSH _ENDAND
    BF
    POP
    ADDR 0 8    #s
    _ENDAND1:
    PUSH _F
    BR
    _RET2:
    ADDR 1 -4    #retval
    SWAP
    STORE
    _ENDFUNC2:
    SETD 1       #restore display
    PUSH 2       #arguments
    POPN
    BR           # 3-16
_Q:   ADDR 1 0
    PUSHMT
    SETD 1       # 3-17
    PUSH 0
    PUSH 3
    DUPN         # 3-19
    PUSH 0
    PUSH _RET3
    PUSH _Q_G
    BR
    _RET3:
    PUSH 7
    LT
    PUSH _ELSE2
```

```
   BF
   PUSH _ENDFUNC3
   BR
   PUSH _ENDIF2
   BR
   _ELSE2:
   _ENDIF2:    # 3-26
   PUSH 0
   PUSH _RET4
   ADDR 1 1    # t
   LOAD
   ADDR 0 7    # r
   LOAD
   PUSH MACHINE_FALSE
   EQ
   PUSH _F
   BR
   _RET4:
   PUSH 17
   EQ
   PUSH _ELSE3
   BF
   PUSH _ENDFUNC3
   BR
   PUSH _ENDIF3
   BR
   _ELSE3:
   _ENDIF3:    # 3-27
   PUSH 0
   PUSH _RET5
   PUSH _P
   BR
   _RET5:
   POP         # 3-28
   _ENDFUNC3:
   PUSH 3
   POPN        # locals
   SETD 1      # restore display
   PUSH 3      # arguments
   POPN
   BR          # 3-29
_Q_G:   ADDR 2 0
   PUSHMT
   SETD 2      # 3-20 3-21
   PUSH 0
   PUSH 2
```

```
DUPN          # 3-22
PUSH 0
PUSH _RET6
ADDR 1 -3    # m
LOAD
PUSH MACHINE_FALSE
EQ
ADDR 0 1    # a
LOAD
ADDR 1 2    # u
LOAD
ADD
ADDR 2 2    # x
LOAD
SUB
ADDR 1 -1    # p
DUP
PUSH _CHECKOR1
BF
PUSH _ENDOR1
BR
_CHECKOR1:
POP
ADDR 0 8     # s
LOAD
_ENDOR1:
PUSH _Q
BR
_RET6:
POP
ADDR 1 -3    # m
LOAD
DUP
PUSH _CHECKOR2
BF
PUSH _ENDOR2
BR
_CHECKOR2:
POP
ADDR 1 -1    # p
LOAD
_ENDOR2:
PUSH _ELSE4
BF
ADDR 1 3    # v
LOAD
```

```
ADDR 1 -2    # n
LOAD
ADD
PUSH _ENDIF4
BR
_ELSE4:
ADDR 1 2    # u
LOAD
ADDR 0 2    # b
LOAD
SUB
_ENDIF4:
ADDR 2 -1    # retval
SWAP
STORE
_ENDFUNC5:
PUSH 2
POPN         # locals
SETD 1       # restore display
BR           # 3-25
```

Sample Program A4-1 (Daniel & Oren):

```
%SAVE_CONTEXT 0
%RESERVE 5 #var i, j, k, n, m : integer
%RESERVE 5 #var p, q, r, s, t : boolean
%RESERVE 7 #var A[7] : integer
%RESERVE 151 #var B[-100 .. 50] : integer
%RESERVE 5 #var C[-7 .. -3] : boolean
%RESERVE 400 #var D[400] : boolean

ADDR 0 2 # j
LOAD
ADDR 0 3 # k
LOAD
PUSH 1
SUB
MUL
ADDR 0 4 # n
LOAD
PUSH 2
ADD
DIV
ADDR 0 4 # n
SWAP
STORE # 1-16

ADDR 0 7 # q
LOAD
%NOT
DUP
%BFALSE _AND1
POP
ADDR 0 6 # p
LOAD
_AND1:
DUP
%BFALSE _CHECKOR1
%JMP _ENDOR1
_CHECKOR1:
POP
ADDR 0 7 # q
LOAD
DUP
%BFALSE _AND2
POP
ADDR 0 6 # p
```

```
LOAD
%NOT
_AND2:
_ENDOR1:
ADDR 0 8
SWAP
STORE # 1-17

ADDR 0 1 # i
LOAD
ADDR 0 2 # j
LOAD
LT
DUP
%BFALSE _CHECKOR2
%JMP _ENDOR2
_CHECKOR2:
POP
ADDR 0 3 # k
LOAD
ADDR 0 4 # n
LOAD
SWAP
LT
%NOT
_ENDOR2:
ADDR 0 6 # p
SWAP
STORE # 1-18

ADDR 0 2 # j # begin 1-19
LOAD
ADDR 0 4 # n
LOAD
EQ
DUP
%BFALSE _AND3
POP
ADDR 0 3 # k
LOAD
ADDR 0 5 # m
LOAD
EQ
%NOT
_AND3:
ADDR 0 8
```

```
SWAP
STORE # 1-19

ADDR 0 2 # j # begin 1-20
LOAD
ADDR 0 3 # k
LOAD
SWAP
LT
%BFALSE _ELSE1
ADDR 0 8 # r
LOAD
ADDR 0 9 # s
LOAD
EQ
%JMP _ENDIF1
_ELSE1:
ADDR 0 1 # i
LOAD
ADDR 0 2 # j
LOAD
EQ
%NOT
_ENDIF1:
ADDR 0 10 # t
SWAP
STORE #  1-20

PUSH 5 # begin 1-21
ADDR 0 11 # A
ADDR 0 1 # i
LOAD
PUSH 0
PUSH _RET1
PUSH -4
ADDR 0 11 # A
ADDR 0 4 # n
LOAD
PUSH 3
ADD
DUP
PUSH 1
LT
%BFALSE _CHECKUPPER1
%JMP _BOUNDS_ERROR # a bound error handler/reporter, see appendix
_CHECKUPPER1:
```

```
DUP # of sub_1
PUSH 7
SWAP
LT # is upperbound_1 < sub_1 ?
%BFALSE _SUB1READY
%JMP _BOUNDS_ERROR # a bound error handler/reporter
_SUB1READY:
PUSH 1
SUB
ADD
LOAD
%JMP _FUNC_F
_RET1:
ADD
ADD
SWAP
STORE # 1-21

PUSH 0 # begin 1-22
PUSH _RET2
PUSH 17
PUSH 5
%JMP _FUNC_F
_RET2:
ADDR 0 18 % B
ADDR 0 18 % B
ADDR 0 18 % B
ADDR 0 1 % i
LOAD
PUSH 1
ADD
DUP
PUSH -100
LT
%BFALSE _CHECKUPPER2
%JMP _BOUNDS_ERROR # a bound error handler/reporter, see appendix
_CHECKUPPER2:
DUP # of sub_1
PUSH 50
SWAP
LT # is upperbound_1 < sub_1 ?
%BFALSE _SUB2READY
%JMP _BOUNDS_ERROR # a bound error handler/reporter
_SUB2READY:
PUSH -100
SUB
```

```
ADD
LOAD
DUP
PUSH -100
LT
%BFALSE _CHECKUPPER3
%JMP _BOUNDS_ERROR # a bound error handler/reporter, see appendix
_CHECKUPPER3:
DUP # of sub_1
PUSH 50
SWAP
LT # is upperbound_1 < sub_1 ?
%BFALSE _SUB3READY
%JMP _BOUNDS_ERROR # a bound error handler/reporter
_SUB3READY:
PUSH -100
SUB
ADD
LOAD
DUP
PUSH -100
LT
%BFALSE _CHECKUPPER4
%JMP _BOUNDS_ERROR # a bound error handler/reporter, see appendix
_CHECKUPPER4:
DUP # of sub_1
PUSH 50
SWAP
LT # is upperbound_1 < sub_1 ?
%BFALSE _SUB4READY
%JMP _BOUNDS_ERROR # a bound error handler/reporter
_SUB4READY:
PUSH -100
SUB
ADD
SWAP
STORE # 1-22

ADDR 0 6 # p
LOAD
DUP
%BFALSE _CHECKOR3
%JMP _ENDOR3
_CHECKOR3:
POP
ADDR 0 7 # q
```

```
LOAD
_ENDOR3:
DUP
%BFALSE _CHECKOR4
%JMP _ENDOR4
_CHECKOR4:
POP
ADDR 0 2 # j
LOAD
PUSH 0
PUSH _RET3
ADDR 0 3 # k
LOAD
PUSH 7
%JMP _FUNC_F
_RET3:
LT
%NOT
_ENDOR4:
ADDR 0 71 # C[-4]
SWAP
STORE # 1-25

%RESERVE 100 # E
%RESERVE 49 # B # 1-26

PUSH 149
POPN # 1-31



_FUNC_F:
   %SAVE_CONTEXT 1
   ADDR 0 0 # &i
   LOAD
   PUSH 0
   SWAP
   LT
   PUSH _FUNC_F_ELSE1
   BF
   PUSH 0
   PUSH _FUNC_F_CALLRET1
   ADDR 0 0 # &i
   LOAD
   PUSH 1
   SUB # i - 1
```

```
    ADDR 0 1 # &j
    LOAD
    PUSH 1
    ADD # j + 1
    BR _FUNC_F # f(i - 1, j + 1)
    _FUNC_F_CALLRET1:
    ADDR 1 -4
    STORE # result f(i - 1, j + 1)
    BR _FUNC_F_END
    _FUNC_F_ELSE1:
      ADDR 0 0 # &i
      LOAD
      PUSH 0
      LT
      PUSH _FUNC_F_ELSE2
      BF
      PUSH 0
      PUSH _FUNC_F_CALLRET2
      ADDR 0 0 # &i
      LOAD
      PUSH 1
      ADD # i + 1
      ADDR 0 1 # &j
      LOAD
      PUSH 1
      SUB # j - 1
      BR _FUNC_F # f(i + 1, j - 1)
      _FUNC_F_CALLRET2:
      ADDR 1 -4
      STORE # result f(i + 1, j - 1)
      BR _FUNC_F_END
      _FUNC_F_ELSE2:
      ADDR 0 1 # &j
      LOAD
      ADDR 1 -4
      STORE # result j
      BR _FUNC_F_END
    _FUNC_F_END:
    %RESTORE_CONTEXT 1 2

  % RESTORE_CONTEXT 0 0
```