

CSC488S Source Language Code Generation Rules

This handout documents the code generation that is necessary to implement the project source language. It assumes that the program has passed all semantic analysis checks and that symbol table entries have been built for all declared identifiers. Nothing in this document is intended to change the syntax of the language.

The code generation operators and their location in the grammar rules are a suggested approach to code generation for the source language. You are free to invent new operators or to move existing operators to different locations depending on your code generation template design.

```

program:      C00 scope C01 C02
statement:    variable ':' '=' expression C77 ,
              'if' expression C42 'then' statement C43 ,
              'if' expression C42 'then' statement C40 C43 'else' statement C41 ,
              'while' C46 expression C42 'do' statement C47 C43 C45 ,
              'repeat' C46 statement 'until' expression C44 C45
              'exit' C48 ,
              'exit' integer C48 ,
              'exit' 'when' expression C57 ,
              'exit' integer 'when' expression C57 ,
              'return' 'with' expression C18 ,
              'return' C19 ,
              'write' output ,
              'read' input ,
              procedurename C55 ,
              procedurename '(' C27 arguments C28 ')' C56 ,
              C03 scope C04 ,
              statement statement
declaration   'var' variablenames ':' type ,
              C35 'function' functionname ':' type C34 C10 C33 scope C11 C36 ,
              C35 'function' functionname C34 '(' C20 parameters C21 ')' C12 ':' type
              C33 scope C13 C36 ,
              C35 'procedure' procedurename C34 C14 scope C15 C36 ,
              C35 'procedure' procedurename C34 '(' C22 parameters C23 ')'
              C16 scope C17 C36 ,
              declaration declaration
variablenames: variablename C30 ,
               variablename '[' bound ']' C31 ,
               variablename '[' bound ':' bound ']' C37 ,
               variablenames ';' variablenames
bound:         integer ,
               generalBound ':' generalBound
generalBound:  integer ,
               '-' integer
scope         '{' declaration statement '}',
               '{' statement '}',
               '{' '}'

```

output:	expression C51 , text C52 , 'newline' C53 , output ',' output
input:	variable C54 , input ',' input
type:	'integer' , 'boolean'
arguments:	expression C29 , arguments ',' arguments
parameters:	parametername ':' type C32 C24 , parameters ',' parameters
variable:	variablename C75 , arrayname C81 '[' expression ']' C82 arrayname C81 '[' expression ',' expression ']' C86
expression:	integer C80 , '-' expression C60 , expression '+' expression C61 , expression '-' expression C62 , expression '*' expression C63 , expression '/' expression C64 , 'true' C79 , 'false' C78 , 'not' expression C65 , expression 'and' expression C66 , expression 'or' expression C67 , expression '=' expression C69 , expression 'not' '=' expression C70 , expression '<' expression C71 , expression '<' '=' expression C72 , expression '>' expression C73 , expression '>' '=' expression C74 , '(' expression ')', '(' expression C42 '?' expression C40 ':' C43 expression C41 ')', variable C76 , functionname C49 , functionname '(' C25 arguments C26 ')' C50 , parametername C68 C76
variablename:	identifier
arrayname:	identifier
functionname:	identifier
procedurename:	identifier
parametername:	identifier

Code Generation Operators

The operators described below document the code generation actions required to support the source language. An attempt has been made to include all necessary hooks for code generation. This document attempts to be independent of any particular code template design.

Depending on the decisions you made in designing your code templates,

- some of these operators may do nothing in your implementation.
- several different operators may be combined because they generate exactly the same code.
- it may not be necessary to "Emit instructions" for some operators.
- you may need to invent additional operations and/or relocate existing operations

Program and ordinary scopes

These code generation operators implement entry and exit to all kinds of scopes.

- C00** Emit code to prepare for the start of program execution.
- C01** Emit code to end program execution.
- C02** Set pc, msp and mlp to values for starting program execution.
- C03** Emit code (if any) to enter an ordinary scope.
- C04** Emit code (if any) to exit an ordinary scope.

Functions and procedures

These code generation operators deal with function and procedure scopes.

- C10** Emit code for the start of a function with no parameters.
- C11** Emit code for the end of a function with no parameters.
- C12** Emit code for the start of a function with parameters.
- C13** Emit code for the end of a function with parameters.
- C14** Emit code for the start of a procedure with no parameters.
- C15** Emit code for the end of a procedure with no parameters.
- C16** Emit code for the start of a procedure with parameters.
- C17** Emit code for the end of a procedure with parameters.
- C18** Emit code to return from a function.
- C19** Emit code to return from a procedure.

Declarations

These code generation operators deal with declarations.

- C30** Allocate storage for a scalar variable. Save address in symbol table.
- C31** Allocate storage for a 1 dimensional array variable. Save address in symbol table.
- C32** Allocate storage for a parameter. Save address in symbol table.
- C33** Allocate storage for the return value of a function. Save address in symbol table.
- C34** Save entry point address of procedure or function in symbol table.
- C35** Emit a forward branch around a function or procedure body.
- C36** Fill in address of forward branch generated by **C35**.
- C37** Allocate storage for a 2 dimensional array variable. Save address in symbol table

Expressions, Variables and Assignment

These code generation actions deal with variables and expressions.

C60	Emit instruction(s) to perform negation.
C61	Emit instruction(s) to perform addition.
C62	Emit instruction(s) to perform subtraction.
C63	Emit instruction(s) to perform multiplication.
C64	Emit instruction(s) to perform division.
C65	Emit instruction(s) to perform logical not operation.
C66	Emit instruction(s) to perform logical and operation.
C67	Emit instruction(s) to perform logical or operation.
C68	Emit instruction(s) to obtain address of parameter.
C69	Emit instruction(s) to perform equality comparison.
C70	Emit instruction(s) to perform inequality comparison.
C71	Emit instruction(s) to perform less than comparison.
C72	Emit instruction(s) to perform less than or equal comparison.
C73	Emit instruction(s) to perform greater than comparison.
C74	Emit instruction(s) to perform greater than or equal comparison.
C75	Emit instruction(s) to obtain address of variable.
C76	Emit instruction(s) to obtain value of variable or parameter.
C77	Emit instruction(s) to store a value in a variable.
C78	Emit instruction(s) to load the value MACHINE_FALSE.
C79	Emit instruction(s) to load the value MACHINE_TRUE.
C80	Emit instruction(s) to load the value of the integer constant.
C81	Emit instructions(s) to obtain address of an array variable.
C82	Emit instruction(s) to create address of a 1 dimensional array element.
C86	Emit instruction(s) to create address of a 2 dimensional array element.

Statements

These code generation actions deal with statements.

C40	Emit unconditional branch. Save address of branch instruction.
C41	Fill in address of branch instruction generated by C40 .
C42	Emit branch on FALSE. Save address of branch instruction.
C43	Fill in address of branch instruction generated by C42 .
C44	Emit branch on FALSE to address saved by C46 .
C45	Fill in address of branch instructions, if any, generated by C48 and C57 in the appropriate loop.
C46	Save current code address for backward branch.
C47	Emit branch to address saved by C46 .
C48	Emit unconditional branch. Save address of branch instruction. Save level if any.
C49	Emit code to call a function with no arguments.
C50	Emit code to call a function with arguments.
C51	Emit code to print an integer expression.
C52	Emit code to print a text string.
C53	Emit code to implement newline.
C54	Emit code to read one integer value and save it in a variable.
C55	Emit code to call a procedure with no arguments.
C56	Emit code to call a procedure with arguments.
C57	Emit branch on TRUE. Save address of branch instruction. Save level if any.

Parameters and Arguments

These code generation operators deal with formal parameters and argument lists.

- C20** Emit any code required before the parameter list of a function.
- C21** Emit any code required after the parameter list of a function.
- C22** Emit any code required before the parameter list of a procedure.
- C23** Emit any code required after the parameter list of a procedure.
- C24** Emit any code required for a parameter.
- C25** Emit any code required before a function argument list.
- C26** Emit any code required after a function argument list.
- C27** Emit any code required before a procedure argument list.
- C28** Emit any code required after a procedure argument list.
- C29** Emit any code required for an argument.

Code Generator Interface to the Machine

To make code generation easier to implement, mechanisms have been provided to allow the code generator to write its generated code directly to the memory of the pseudo machine. The pseudo machine is described in a separate handout. The assumed model for code generation and machine execution is illustrated in the figure on the next page.

During code generation the code generator writes machine instructions into memory starting at address zero and working upward in memory. The code area is read only during program execution. The top end of memory starting at address `memorySize - 1` and working downward is available to the code generator for storing constants, miscellaneous data and/or code fragments. The machine interface (`Machine.java`) provides two functions for accessing machine memory during code generation

Machine.writeMemory(short addr , short value)

Write *value* into memory at location *addr*.

short Machine.readMemory(short addr)

Return the contents of the memory location addressed by *addr*

During machine execution, the unused area between the highest program address and the lowest address used to store information at the top of memory is used as a run time stack.

The code generator **must** set the address boundaries of these regions and the initial value of the program counter before the program is executed. Usually this is done at or near the end of code generation by operation **C02** using the functions provided by the Machine interface:

Machine.setMLP(short addr)

Set the address of the upper limit of the run time stack to the highest address not used at the top of stack.

Machine.setMSP(short addr)

Set the address of the bottom of the run time stack to the first address above the code area

Machine.setPC(short addr)

Set the initial value of the program counter to the address of the first instruction to be executed.

Code Generation and Execution Model

