CSC488 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:	**C**00 scope **C**01 **C**02
statement:	variable ':' '=' expression **C**77 , '**if**' expression **C**42 '**then**' statement **C**43 , '**if**' expression **C**42 '**then**' statement **C**40 **C**43 '**else**' statement **C**41 , '**while**' **C**46 expression **C**42 '**do**' statement **C**47 **C**43 **C**45 , '**repeat**' **C**46 statement '**until**' expression **C**44 **C**45 , '**exit**' **C**48 , '**exit**' integer **C**48 , '**exit**' '**when**' expression **C**57 , '**exit**' integer '**when**' expression **C**57 , '**return**' '**with**' expression **C**18 , '**return**' **C**19 , '**write**' output , '**read**' input , procedurename **C**55 , procedurename '(' **C**27 arguments **C**28 ')' **C**56 , **C**03 scope **C**04 , statement statement
declaration:	'**var**' variablenames ':' type , **C**35 '**function**' functionname ':' type **C**34 **C**10 **C**33 scope **C**11 **C**36 , **C**35 '**function**' functionname **C**34 '(' **C**20 parameters **C**21 ')' **C**12 ':' type **C**33 scope **C**13 **C**36 , **C**35 '**procedure**' procedurename **C**34 **C**14 scope **C**15 **C**36 , **C**35 '**procedure**' procedurename **C**34 '(' **C**22 parameters **C**23 ')' **C**16 scope **C**17 **C**36 ,
variablenames:	variablename **C**30 , variablename '[' integer ']' **C**31 , variablename '[' bound '..' bound ']' **C**31 , variablenames ',' variablenames
bound:	integer , '-' integer
scope:	'{' declaration statement '}' , '{' statement '}' , '{' '}' ,

output: expression **C**51 ,
text **C**52 ,
'**newline**' **C**53 ,
output ',' output

input: variable **C**54 ,
input ',' input

type: '**Integer**' ,
'**Boolean**'

arguments: expression **C**29 ,
arguments ',' arguments

parameters: parametername ':' type **C**32 **C**24 ,
parameters ',' parameters

variable: variablename **C**75 ,
arrayname **C**81 '[' expression ']' **C**82

expression: integer **C**80 ,
'-' expression **C**60 ,
expression '+' expression **C**61 ,
expression '-' expression **C**62 ,
expression '*' expression **C**63 ,
expression '/' expression **C**64 ,
'**true**' **C**79 ,
'**false**' **C**78 ,
'**not**' expression **C**65 ,
expression '**and**' expression **C**66 ,
expression '**or**' expression **C**67 ,
expression '=' expression **C**69 ,
expression '**not**' '=' expression **C**70 ,
expression '<' expression **C**71 ,
expression '<' '=' expression **C**72 ,
expression '>' expression **C**73 ,
expression '>' '=' expression **C**74 ,
'(' expression ')' ,
'(' expression **C**42 '?' expression **C**40 ':' **C**43 expression **C**41 ')' ,
variable **C**76 ,
functionname **C**49 ,
functionname '(' **C**25 arguments **C**26 ')' **C**50 ,
parametername **C**68 **C**76

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 an 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 **C**35 .

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 calculate address of 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 **C**40 .
C42 Emit branch on FALSE. Save address of branch instruction.
C43 Fill in address of branch instruction generated by **C**42 .
C44 Emit branch on FALSE to address saved by **C**46 .
C45 Fill in address of branch instructions, if any, generated by **C**48 and **C**57 in the appropriate loop.
C46 Save current code address for backward branch.
C47 Emit branch to address saved by **C**46 .
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 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 interpreter (`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.

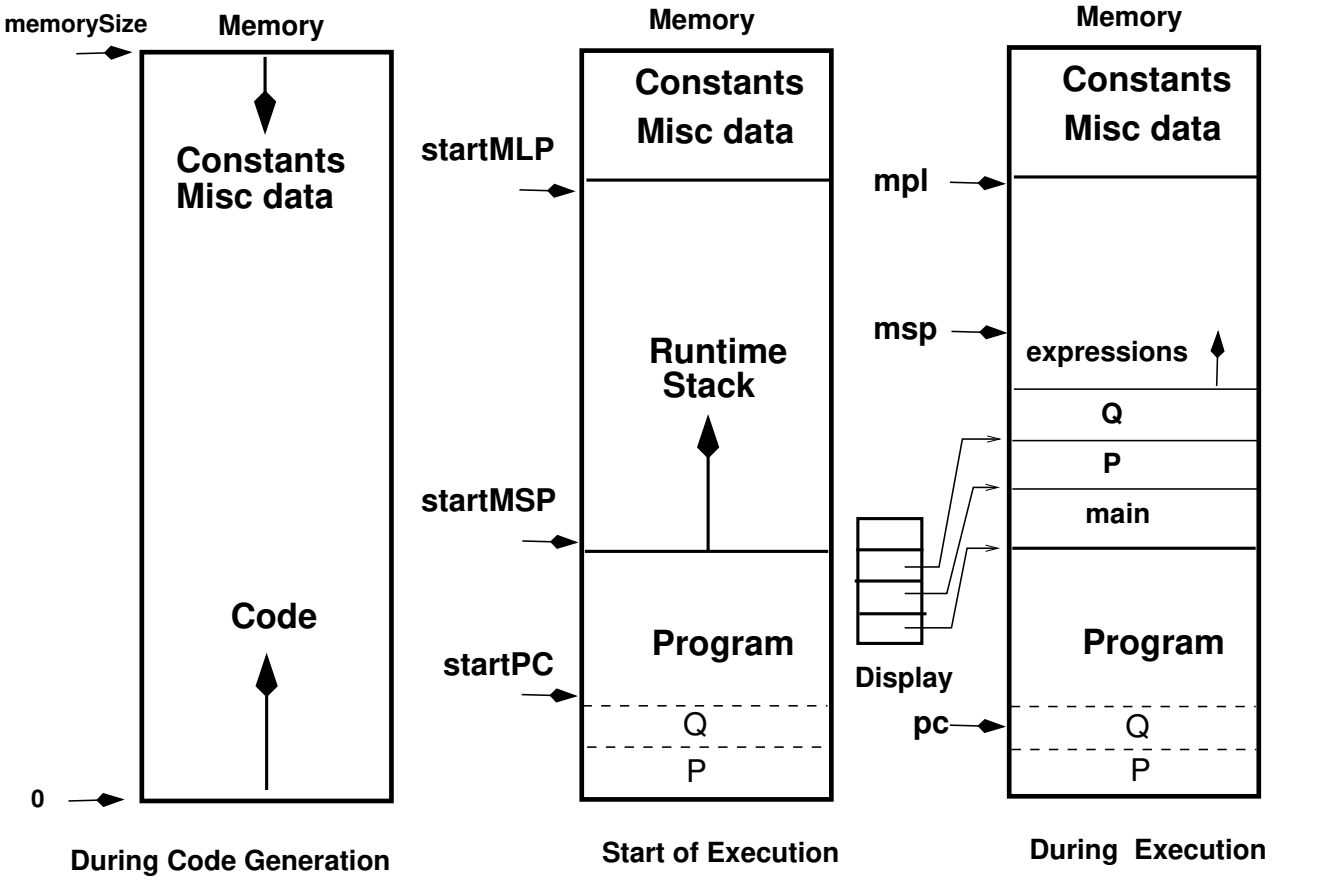
The code generator **must** set the address boundaries of these regions and the initial value of the program counter before the program is executed. Usually this is done at or near the end of code generation by operation **C**02 using the functions provided by the Machine class:

Machine.setMLP(short addr)
Set the address of the upper limit of the run time stack.

Machine.setMSP(short addr)
Set the address of the bottom of the run time stack.

Machine.setPC(short addr)
Set the initial value of the program counter for machine execution.

Code Generation and Execution Model



main calls P calls Q