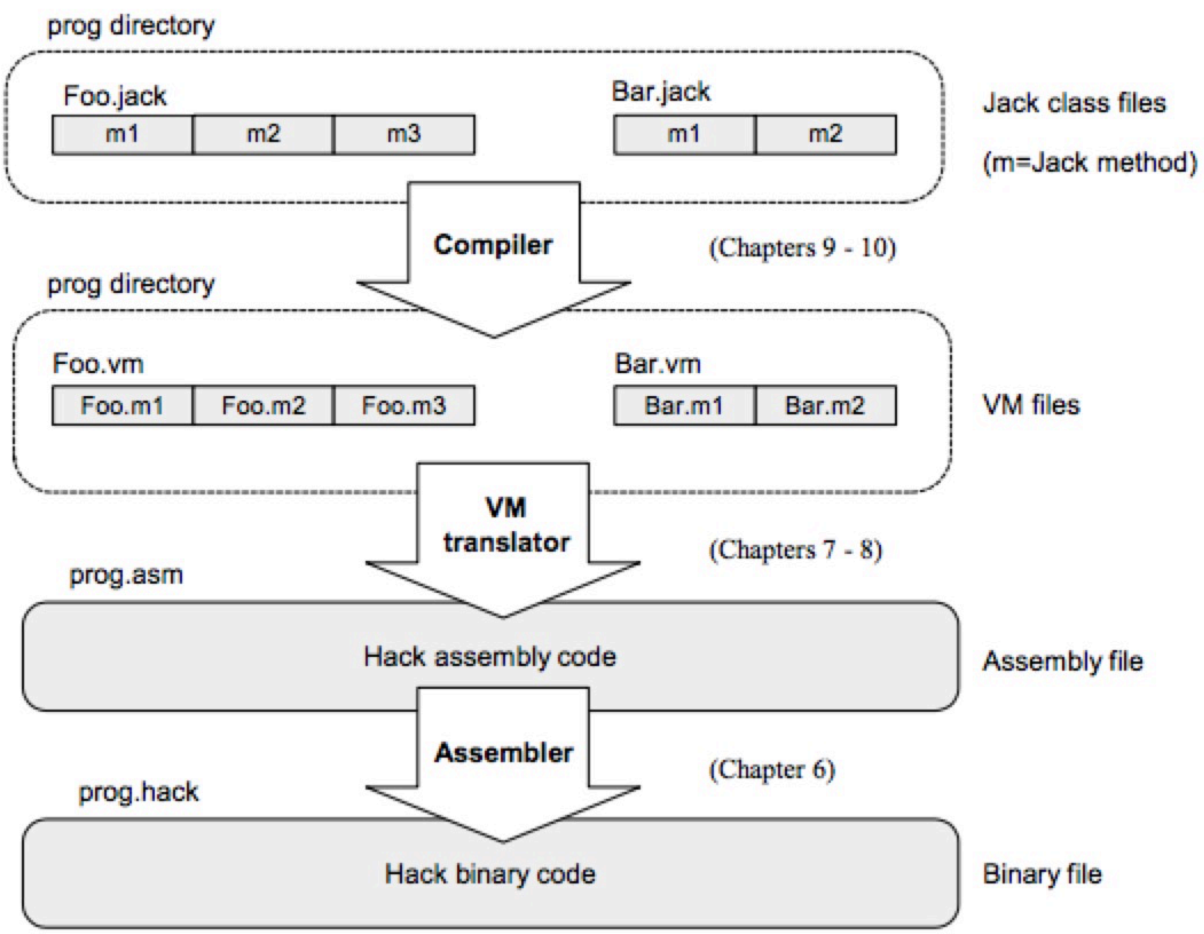
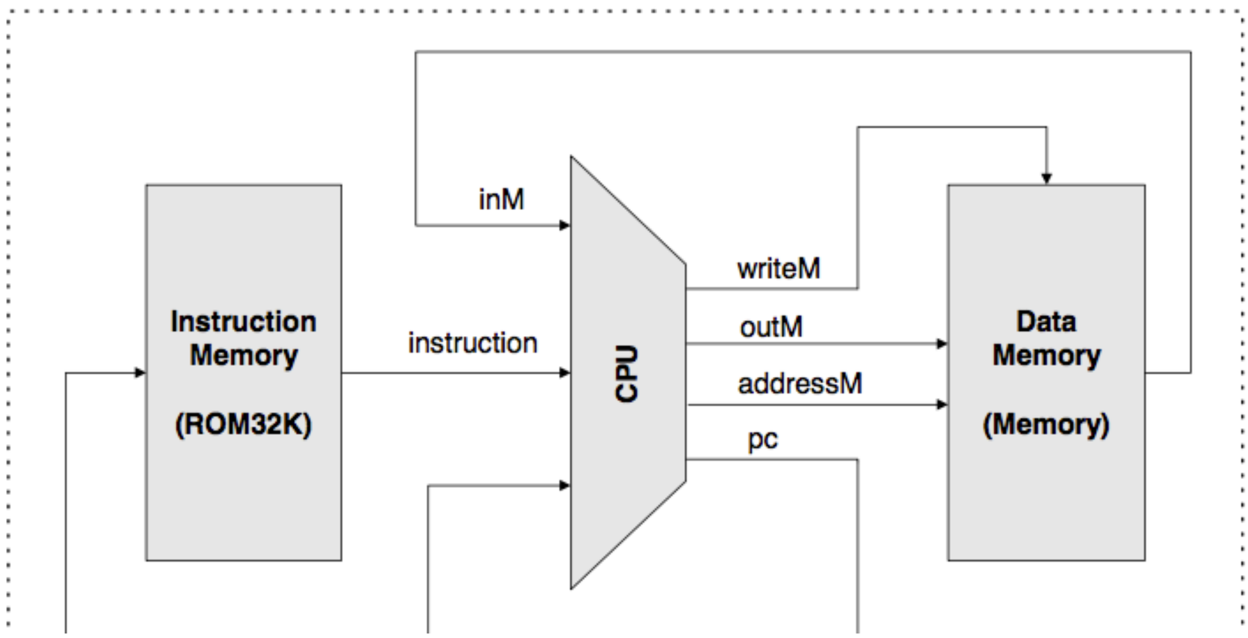
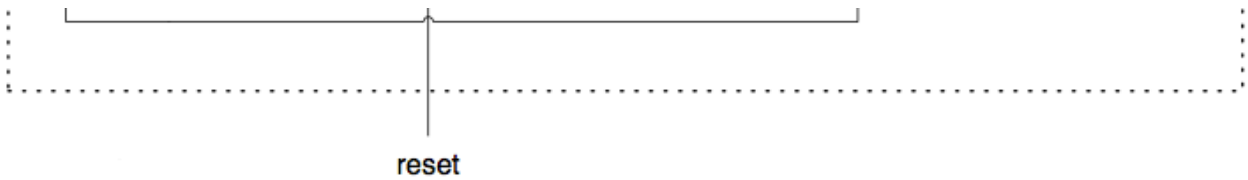
Nand2tetris note

Overview







Harvard architecture

VMTranslator

RAM Usage

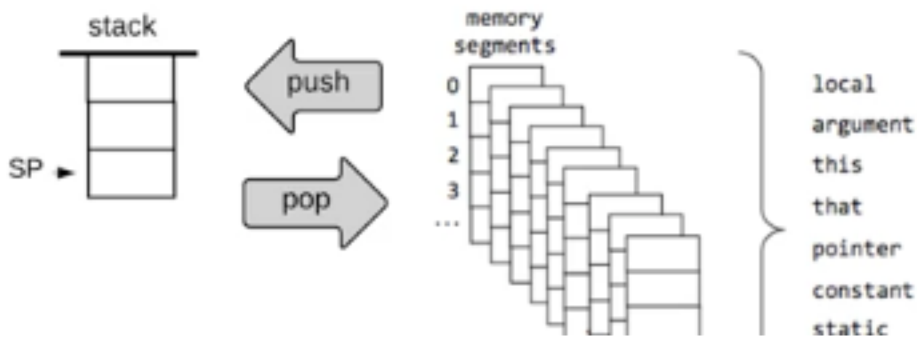
RAM addresses	***Usage***
0–15	Sixteen virtual registers, whose usage is described below
16–255	Static variables (of all the VM functions in the VM program)
256–2047	Stack
2048–16483	Heap (used to store objects and arrays)
16384–24575	Memory mapped I/O

Register	***Name***	***Usage***
RAM[0]	`SP`	Stack pointer: points to the next topmost location in the stack;
RAM[1]	`LCL`	Points to the base of the current VM function's `local` segment;
RAM[2]	`ARG`	Points to the base of the current VM function 's `argument` segment;
RAM[3]	`THIS`	Points to the base of the current `this` segment (within the heap);
RAM[4]	`THAT`	Points to the base of the current `that` segment (within the heap);
RAM[5-12]		Holds the contents of the `temp` segment;
RAM[13-15]		Can be used by the VM implementation as general-purpose registers.

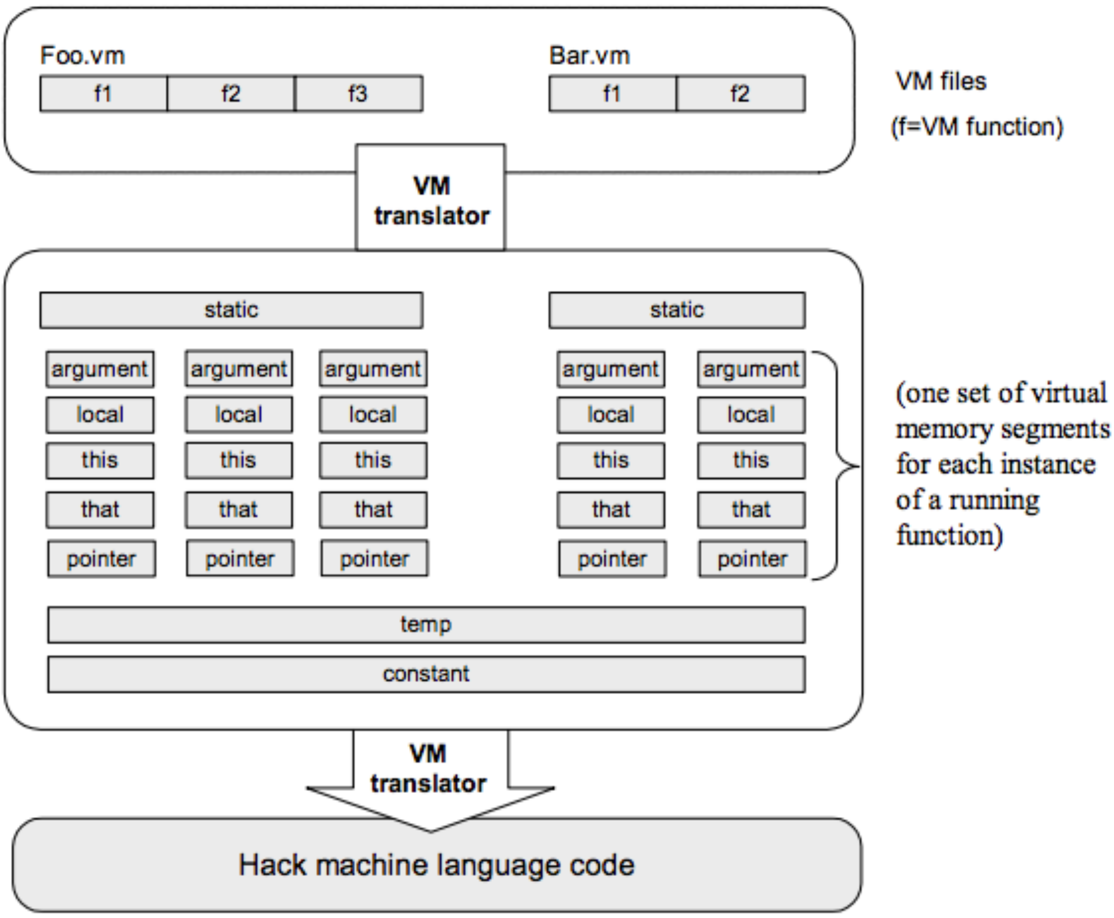
FIGURE 7.13: Usage of the Hack registers in the standard mapping

Memory Segment

HackVM consists of 8 virtual memory segments, and compute at Stack

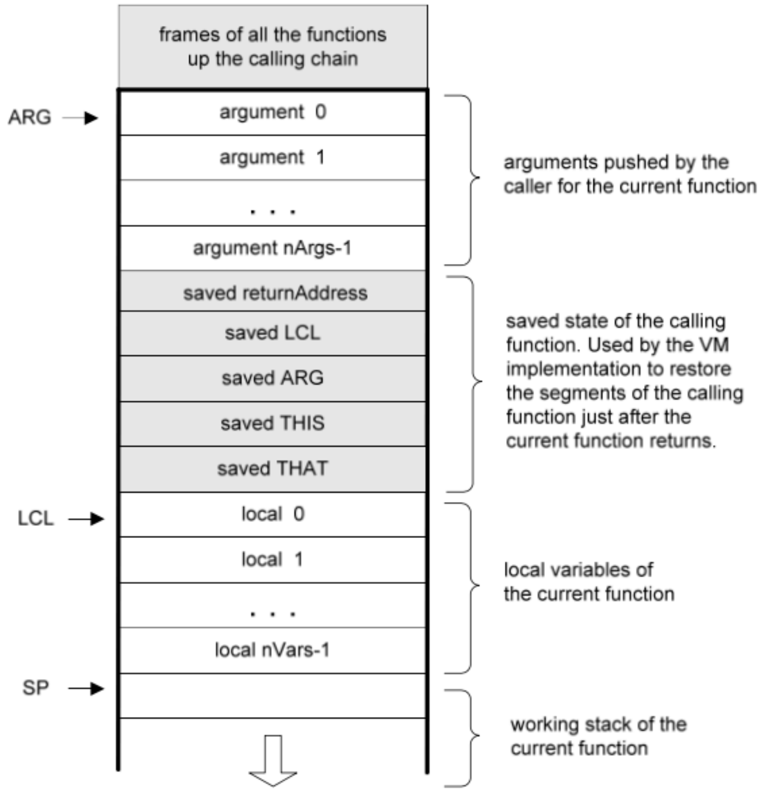


Segment	Purpose	Comments
`argument`	Stores the function's arguments.	Allocated dynamically by the VM implementation when the function is entered.
`local`	Stores the function's local variables.	Allocated dynamically by the VM implementation and initialized to 0 when the function is entered.
`static`	Stores static variables shared by all functions in the same `.vm` file.	Allocated by the VM implementation for each `.vm` file; shared by all functions in the `.vm` file.
`constant`	Pseudo-segment that holds all the constants in the range 0 ... 32767.	Emulated by the VM implementation; Seen by all the functions in the program.
`this` `that`	General-purpose segments. Can be made to correspond to different areas in the heap. Serve various programming needs.	Any VM function can use these segments to manipulate selected areas on the heap.
`pointer`	A two-entry segment that holds the base addresses of the `this` and `that` segments.	Any VM function can set `Pointer 0` (or `1`) to some address; this has the effect of aligning the `this` (or `that`) segment to the area on the heap beginning in that address.
`temp`	Fixed eight-entry segment that holds temporary variables for general use.	May be used by any VM function for any purpose. Shared by all functions in the program.



Save currently executive function's frame, before call another function

The implementation of the VM's stack on the host Hack RAM



<u>Global stack:</u>
the entire RAM area dedicated to hold the stack

<u>Working stack:</u>
from SP onwards: the stack that the current function sees

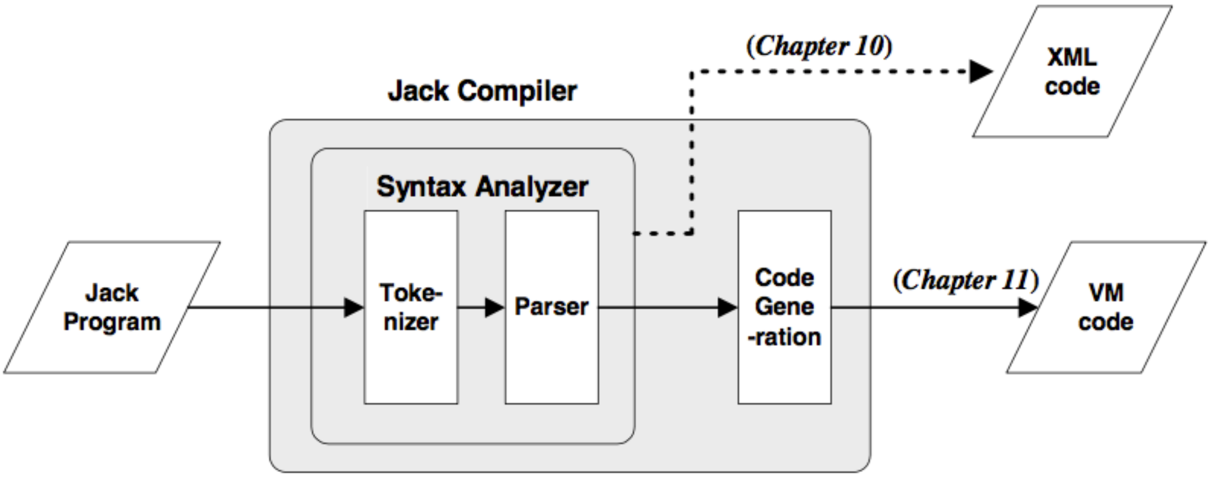
- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, as well as its virtual memory segments
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.

VM command	*Generated (pseudo) code emitted by the VM implementation*	
`call f n`	`push return-address`	// (Using the label declared below)
(calling a function `f`	`push LCL`	// Save `LCL` of the calling function
after `n` arguments	`push ARG`	// Save `ARG` of the calling function
have been pushed	`push THIS`	// Save `THIS` of the calling function
onto the stack)	`push THAT`	// Save `THAT` of the calling function
	`ARG = SP-n-5`	// Reposition `ARG` (n = number of args.)
	`LCL = SP`	// Reposition `LCL`
	`goto f`	// Transfer control
	`(return-address)`	// Declare a label for the return-address
`function f k`	`(f)`	// Declare a label for the function entry
(declaring a function	`repeat k times:`	// k = number of local variables
`f` that has `k` local	`PUSH 0`	// Initialize all of them to 0
variables)		
`return`	`FRAME = LCL`	// `FRAME` is a temporary variable
(from a function)	`RET = *(FRAME-5)`	// Put the return-address in a temp. var.
	`*ARG = pop()`	// Reposition the return value for the caller
	`SP = ARG+1`	// Restore `SP` of the caller
	`THAT = *(FRAME-1)`	// Restore `THAT` of the caller
	`THIS = *(FRAME-2)`	// Restore `THIS` of the caller
	`ARG = *(FRAME-3)`	// Restore `ARG` of the caller
	`LCL = *(FRAME-4)`	// Restore `LCL` of the caller
	`goto RET`	// Goto return-address (in the caller's code)

When a VM function starts running, it assumes that

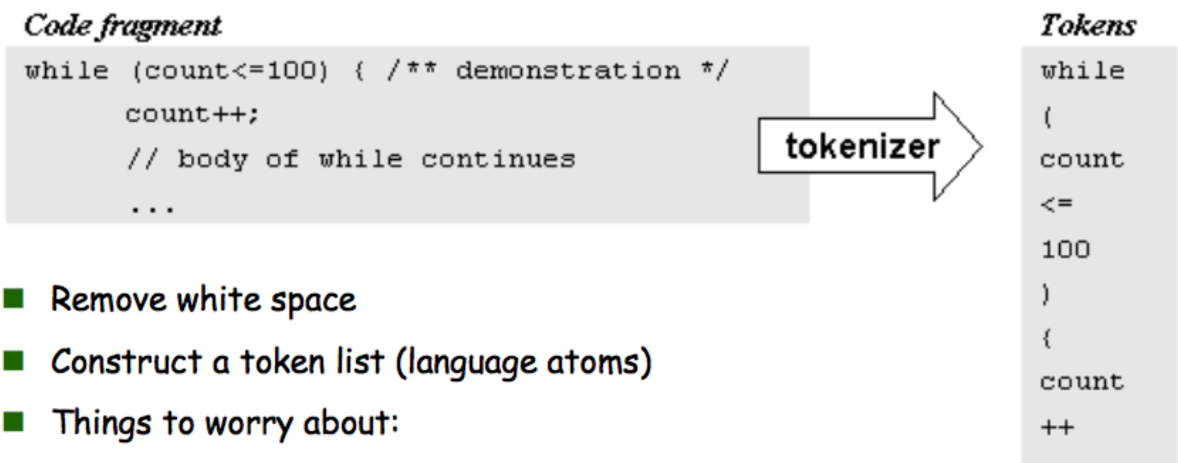
- the stack is empty
- the argument values on which it is supposed to operate are located in the argument segment
- the local variables that it is supposed to use are initialized to 0 and located in the local segment

Compiler



JackAnalyzer

Lexcical Analysis(also can be call Tokenizing or Sacnning)



- Language specific rules:
  e.g. how to treat "++"

;

...

- Language-specific classifications:
  keyword, symbol, identifier, integerCconstant, stringConstant,...

- While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).

Parsing

The act of checking whether a grammar "accepts" an input text as valid is called parsing

- The parsing process:
  - A text is given and tokenized
  - The parser determines weather or not the text can be generated from the grammar
  - In the process, the parser performs a complete structural analysis of the text

Grammar

<table>
<tr><td>Lexical elements:</td><td>The Jack language includes five types of terminal elements (tokens):</td></tr>
<tr><td>keyword:</td><td>'class' | 'constructor' | 'function' | 'method' | 'field' | 'static' | 'var' | 'int' | 'char' | 'boolean' | 'void' | 'true' | 'false' | 'null' | 'this' | 'let' | 'do' | 'if' | 'else' | 'while' | 'return'</td></tr>
<tr><td>symbol:</td><td>'{' | '}' | '(' | ')' | '[' | ']' | '.' | ',' | ';' | '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '=' | '~'</td></tr>
<tr><td>integerConstant:</td><td>A decimal number in the range 0 .. 32767.</td></tr>
<tr><td>StringConstant</td><td>'"' A sequence of Unicode characters not including double quote or newline '"'</td></tr>
<tr><td>identifier:</td><td>A sequence of letters, digits, and underscore ('_') not starting with a digit.</td></tr>
<tr><td>Program structure:</td><td>A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:</td></tr>
<tr><td>class:</td><td>'class' className '{' classVarDec* subroutineDec* '}'</td></tr>
<tr><td>classVarDec:</td><td>('static' | 'field') type varName (',' varName)* ';'</td></tr>
<tr><td>type:</td><td>'int' | 'char' | 'boolean' | className</td></tr>
<tr><td>subroutineDec:</td><td>('constructor' | 'function' | 'method') ('void' | type) subroutineName '(' parameterList ')' subroutineBody</td></tr>
<tr><td>parameterList:</td><td>((type varName) (',' type varName)*)?</td></tr>
<tr><td>subroutineBody:</td><td>'{' varDec* statements '}'</td></tr>
<tr><td>varDec:</td><td>'var' type varName (',' varName)* ';'</td></tr>
<tr><td>className:</td><td>identifier</td></tr>
<tr><td>subroutineName:</td><td>identifier</td></tr>
<tr><td>varName:</td><td>identifier</td></tr>
<tr><td>Statements:</td><td></td></tr>
<tr><td>statements:</td><td>statement*</td></tr>
<tr><td>statement:</td><td>letStatement | ifStatement | whileStatement | doStatement | returnStatement</td></tr>
<tr><td>letStatement:</td><td>'let' varName ('[' expression ']')? '=' expression ';'</td></tr>
<tr><td>ifStatement:</td><td>'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?</td></tr>
</table>

```
   whileStatement:  'while' '(' expression ')' '{' statements '}'
      doStatement:  'do' subroutineCall ';'
  ReturnStatement   'return' expression? ';'

Expressions:
       expression:  term (op term)*
             term:  integerConstant | stringConstant | keywordConstant | varName |
                    varName '[' expression ']' | subroutineCall | '(' expression ')' | unaryOp term
   subroutineCall:  subroutineName '(' expressionList ')' | ( className | varName) '.' subroutineName
                    '(' expressionList ')'
   expressionList:  (expression (',' expression)* )?
               op:  '+' | '-' | '*' | '/' | '&' | '|' | '<' | '>' | '='
          unaryOp:  '-' | '~'
  KeywordConstant:  'true' | 'false' | 'null' | 'this'
```

FIGURE 10.5: Complete grammar of the Jack language

JackCompiler

Symbol table

- Whenever a new identifier is encountered in the source code for the first time (e.g., in a variable declaration), the compiler adds its description to the table.
- Whenever an identifier is encountered elsewhere in the code, the compiler looks it up in the symbol table and gets all the necessary information about it.

name	type	kind	index
identifier	int,char,boolean,identifier(String,Array,className)	argument,var(subroutine scope) field,static(class scope)	>=0

example

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

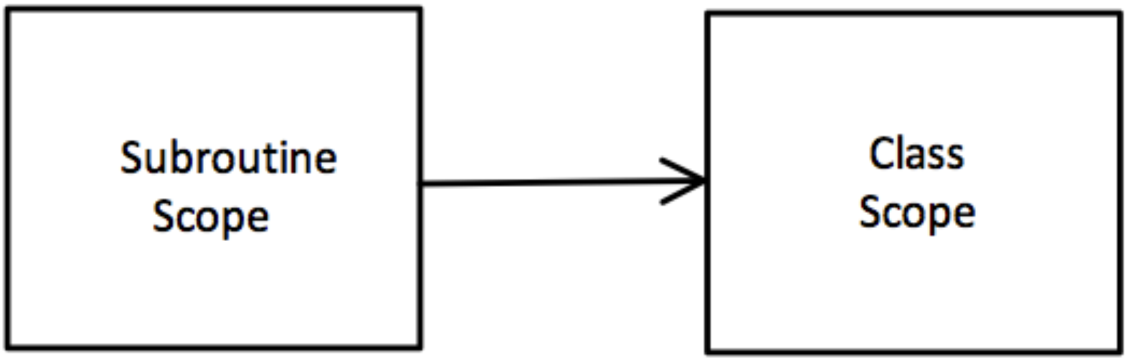
Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0

j	int	var	1
due	Date	var	2

kind	argument	var	field	static
correspond memory segment	argument	local	this	static

Scope

Jack language only have two scope,when account a variable first search at subroutine scope if can't found and then search at class scope.



Subroutine Translation

Subroutine type

- constructor
- method
- funtcion

Subroutine call

obj.subroutine(args) => Obj.method(this,args) Obj is type of obj look up at symbol table

look up obj at subroutine scope symbol table,if can't found look up at class scope symbol table

if obj declare in subroutine scope

push local obj's index
push args
call Obj.method

else if obj declare in class scope

push this obj'index
push args
Obj.method

Obj.method

else if can not found type of obj at nor subroutine scope and class scope symbol table,assume that obj is a className or subroutine is function or constructor,

for example Array.new(),BankAccount.sumup(a,b) and so on,there is no need to deliver 'this' pointer.

push args
Obj.method(args)

Subroutine declare

- constructor

  function className.new number of local
  push constant n (n means number of field of this class)
  call Memory.alloc n(in order to get an available address)
  pop pointer 0

- method

  function className.subroutineName number of local
  push arg 0
  pop pointer 0

- function(the static function in Jack language)

  function className.subroutineName number of local

Date Translation

Array,Object

- The array declaration results in the allocation of a single pointer only, which, eventually, may point to the array's base address
- The array proper is created in memory later, if and when the array is actually constructed at run-time. This type of dynamic memory allocation is done fromthe heap, using the memory management services of the operating system.
- Object declaration similar to array declaration

Array Access

- get value of array[i]

  push array's baseAddress
  push i
  add
  pop pointer 1
  pop that 1

- arrayA[i] = arrayB[j]