Introduction to the Course

This course will cover four main themes: techniques for specifying programming languages, implementation of programming languages, exposure to alternative programming language paradigms, and issues in the design of programming languages.

Note: frequently in these course notes you will find this symbol: ⇒ in the margin. Wherever this occurs, we will do this work during a mini-lecture, recording our results either in a snapshot of the board or a text file.

Some Introductory Questions

⇒ What is a programming language?

⇒ In the Web search for `programming paradigms` and note the main four that we will study.

⇒ In the Web, search for `history of programming languages` and specifically look for a timeline, and ponder the list, including noting the paradigm(s) some of them belong to.

⇒ Go to `www.tiobe.com/tiobe-index` and observe the popularity of various programming languages at this moment in time.

A Very Primitive Language

For the rest of our introduction to the course (including in-depth study of a number of important issues relating to designing and implementing imperative programming languages), we will study a toy language known as VPL (for "very primitive language").

To begin, we will specify the syntax and semantics of VPL, and then you will have a project to implement VPL (which will turn out to be fairly easy).

The main requirement for this language is that it be very easy to specify and implement. One major consequence of this requirement is that we will only work with integer values, and input/output operations and other utility functions that could easily be provided are kept to a minimum.

This work provides a first example of specification of a programming language, begins our study of implementation of programming languages, will be used later as the platform for discussing all sorts of imperative language implementation issues, and will provide a very gentle introduction to language design issues.

In a sense, all procedural languages are just spiffed up versions of a language such as this.

The following description of VPL mostly only makes sense in the context of the implementation we have in mind, so implementation suggestions will be integrated with the language description. This makes sense, since we are sort of creating a virtual processor as we describe the language.

The VPL Memory Model

The memory model consists of a rather large one-dimensional array, named `mem` in the following discussion, together with a number of individual variables, including ones named `ip`, `sp`, `bp`, `rv`, `hp`, and `gp` that are referred to as *registers*.

Each location in this array is known as a memory cell, or just cell. This array holds the stored program, starting in cell 0 and continuing as needed. The global memory area begins where the stored program ends. The system stack begins where the global memory area ends. The area for dynamic memory, known as the "heap," begins at the upper end of the array and continues downward.

While the program is executing, the instruction pointer register `ip` will hold the index of the cell containing the operation code for the instruction that is currently executing.

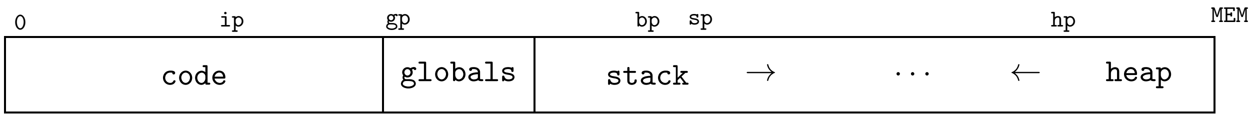
The "global pointer" register `gp` holds the first index after the stored program.

At all times the base pointer register `bp` will hold the index of the smallest cell in the region of memory used for the currently active subprogram, which is known as the "stack frame," and the stack pointer register `sp` will hold the index of the first cell above the stack frame.

The heap pointer register `hp` will point to the first cell of the most recently allocated chunk of memory, so that the region with indices smaller than `hp` will be available for allocation.

The return value register `rv` is used to hold the single value being returned by a subprogram.

Here is a picture of the memory layout:



The Instructions and Execution Semantics

We will now see our first example of specifying a programming language—we will specify the *syntax*—the rules for what sequences of symbols constitute a legal program in VPL—and the *semantics*—the meaning of a VPL program. For this very simple language, we will be able to specify these things quite easily. In fact, the previous "memory model" discussion has begun the description of the semantics of VPL.

Execution starts with `ip` set to 0. Note that all the registers need to be properly initialized. After each instruction has executed, `ip` has either been moved to a different value, or it moves to the sequentially next instruction.

When input arguments are passed, a subprogram is called, or a subprogram is returned from, we will need to manage the details of the stack frame to make recursion possible. In particular, in addition to allocating space in a stack frame for the input arguments and local variables, we will need to have space for storing the values of any registers that need to be restored when you return from a subprogram. However we do this we want to set things up so that all "variables" in a subprogram, including arguments and locals, will be named 0, 1, 2, and so on, with the input arguments being 0, 1, 2, and so on.

We will assume that arguments are always passed immediately before calling a subprogram.

Here are the draft details of the operations. In this description, the syntax of each instruction is given implicitly through an abstract example. In the abstract examples, every symbol represents an integer, but as an aid to understanding the semantics, integer literals are operation codes, `n` represents an integer literal, `a`, `b`, and `c` represent offsets in the current stack from `bp` (and "cell a" actually means the array location with index `bp+2+a`), and `L` represents a label.

`Instruction`	Mnemonic	Semantic Description
`0`	no op	Do nothing.
`1 L`	label	During program loading this instruction disappears, and all occurrences of `L` are replaced by the actual index in `mem` where the opcode `1` would have been stored.
`2 L`	call	Do all the steps necessary to set up for execution of the subprogram that begins at label `L`.
`3 a`	pass	Push the contents of cell `a` on the stack.
`4 n`	locals	Increase `sp` by `n` to make space for local variables in the current stack frame.
`5 a`	return	Do all the steps necessary to return from the current subprogram, including putting the value stored in cell `a` in `rv`.
`6 a`	get retval	Copy the value stored in `rv` into cell `a`.
`7 L`	jump	Change `ip` to `L`.
`8 L a`	cond	If the value stored in cell `a` is non-zero, change `ip` to `L` (otherwise, move `ip` to the next instruction).
`9 a b c`	add	Add the values in cell `b` and cell `c` and store the result in cell `a`.
`10 a b c`	subtract	Same as 9, but do cell `b` − cell `c`.
`11 a b c`	multiply	Same as 9, but do cell `b` ∗ cell `c`.
`12 a b c`	divide	Same as 9, but do cell `b` / cell `c`.
`13 a b c`	remainder	Same as 9, but do cell `b` % cell `c`.
`14 a b c`	equal	Same as 9, but do cell `b` `==` cell `c`.
`15 a b c`	not equal	Same as 9, but do cell `b` `!=` cell `c`.
`16 a b c`	less than	Same as 9, but do cell `b` `<` cell `c`.
`17 a b c`	less than or equal	Same as 9, but do cell `b` `<=` cell `c`.
`18 a b c`	and	Same as 9, but do cell `b` && cell `c`.
`19 a b c`	or	Same as 9, but do cell `b` `||` cell `c`.

`20 a b`	not	If cell `b` holds zero, put 1 in cell `a`, otherwise put 0 in cell `a`.
`21 a b`	opposite	Put the opposite of the contents of cell `b` in cell `a`.
`22 a n`	literal	Put `n` in cell `a`.
`23 a b`	copy	Copy the value in cell `b` into cell `a`.
`24 a b c`	get	Get the value stored in the heap at the index obtained by adding the value of cell `b` and the value of cell `c` and copy it into cell `a`.
`25 a b c`	put	Take the value from cell `c` and store it in the heap at the location with index computed as the value in cell `a` plus the value in cell `b`.
`26`	halt	Halt execution.
`27 a`	input	Print a `?` and a space in the console and wait for an integer value to be typed by the user, and then store it in cell `a`.
`28 a`	output	Display the value stored in cell `a` in the console.
`29`	newline	Move the console cursor to the beginning of the next line
`30 a`	symbol	If the value stored in cell `a` is between 32 and 126, display the corresponding symbol at the console cursor, otherwise do nothing.
`31 a b`	new	Let the value stored in cell `b` be denoted by m. Decrease `hp` by m and put the new value of `hp` in cell `a`.
`32 n`	allocate global space	This instruction must occur first in any program that uses it. It simply sets the initial value of `sp` to `n` cells beyond the end of stored program memory, and sets `gp` to the end of stored program memory.
`33 n a`	Copy to global	Copy the contents of cell `a` to the global memory area at index `gp+n`.
`34 a n`	Copy from global	Copy the contents of the global memory cell at index `gp+n` into cell `a`.

The C strategy of “non-zero is true, zero is false” is used where appropriate. C style psuedo-code is used to describe the various operations 9–19.

Exercise 1

Working in whiteboard groups (ideally 4 or 5 people), write the following simple programs in VPL.

a. Create a single global variable and store 17 in it. Display a question mark in the console, get input from the keyboard, and store it in a local cell. Add that input value to the global variable value and display the result in the console.

b. Create a VPL program that asks the user for an input value and displays 1 if that input is an odd number, and 2 if it is an even number.

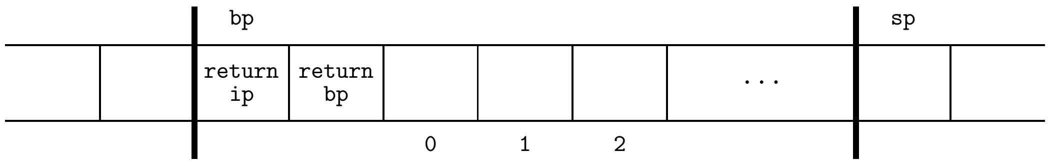
c. Create a VPL program that asks the user for a value of `number` and then displays on-screen (in the console) the values 1, 2, . . ., `number`, one per line.

d. Create a VPL program that gets an input value, say x, and repeatedly divides x by 2 if x is even or multiplies x by 3 and adds 1, until x reaches 1.

The Stack Frame Model

In order to implement subprograms in the usual way (later we will discuss simpler memory models) we need to give each subprogram call its own memory area, known as a *stack frame* (or "activation record").

Here is a picture of the layout of a *stack frame* the way we will do it in VPL. This layout, combined with commands 2–6, allows the VPL programmer to use *modularity*—to be able to use subprograms.



With this model, a *stack frame* consists of all the memory cells from index `bp` up to but not including index `sp`.

To call a subprogram, first we have to use command 3 once for each value that we want to pass over to the next stack frame. Then we use command 2 to call the desired subprogram. This command stores the correct values of the current base pointer and instruction pointer in the first two cells of the new stack frame. Note that the arguments passed by command 3 sit just to the right of those two cells.

And, command 2 has to change the instruction pointer to the starting point in the code segment of the subprogram being called.

The first command in the subprogram must be command 4, which simply moves the stack pointer to the right to make space for the local variables in the stack frame. So, if for example we passed two arguments, then the cells conceptually labeled (underneath) 0 and 1 will hold the values of those parameters (to use Java terminology), while cells conceptually labeled 2, 3, and so on, will be local variables for the subprogram.

When command 5 executes, it puts the value to be returned in the return value register, and then restores the base pointer, the stack pointer, and the instruction pointer, thus returning both in code and in the stack to the stack frame from which the call was made.

Then in the calling stack frame, command 6 moves the returned value into a local cell.

Exercise 2

Here is a VPL program (a very bad one in the sense that it has no comments!):

```
4 3
27 0
3 0
2 1001
6 1
28 1
29
26

1 1001
4 6
22 2 2
22 1 1
16 6 0 2
8 2001 6
10 3 0 1
3 3
2 1001
6 4
11 5 0 4
5 5
1 2001
5 1
```

Trace the execution of this program. Your group will need to draw a bunch of memory cells. Since you have a narrow piece of whiteboard, break the long strip of cells into convenient horizontal pieces, say using 10 cells per piece.

As you write the numbers for the code in the code section, be sure to remove the label commands and to change all the "jump" commands to use the index where the label command would be.

As you trace the commands, keep track of the `ip`, `bp`, and `sp` by writing them in above the correct memory cells and erasing the previous values.

And, of course, trace what is written in the console.

Wrap-Up Exercise 1

My solutions to Exercise 1 are in the VPL folder. Make sure that you finish Exercise 1 on your own time, as needed, with these to help.

Also, use these programs as some of your test cases for Project 1.

Exercise 3

Working in whiteboard groups (ideally 4 or 5 people), write this program, which includes two useful subprograms:

> Write a subprogram, starting with label 100000, that takes as its single input argument a positive integer n, and then allocates $n + 1$ cells in the heap, stores n in the first of these cells, and then asks the user for n numbers, storing them in the remaining n cells, and returns the address in memory where the list begins.
>
> Write a subprogram, starting with label 200000, that takes as its single input argument the index in memory of the starting point of a list, stored as for subprogram 100000, and then displays the list on a single line, with items separated by spaces.
>
> Then write a main program that will call the list input subprogram and then call the list display subprogram.
>
> Someone in your group should record your final work in a computer file and email it to everyone in the group, because you will want to use this program as a test case for your Project 1 work.

Exercise 4

Trace the execution of the code shown below. Draw on the whiteboard memory cells from 70 on, and trace all the operations, assuming the user inputs 4 and then 2.

`ip`

0	1	2	3	4	5	6	7	8	9
27	0	27	1	3	0	3	1	2	16

10	11	12	13	14	15	16	17	18	19
6	2	28	2	29	26	4	5	22	2

20	21	22	23	24	25	26	27	28	29
1	22	5	0	14	6	1	5	8	68

30	31	32	33	34	35	36	37	38	39
6	14	6	1	0	8	68	6	10	3

40	41	42	43	44	45	46	47	48	49
0	2	10	4	1	2	3	3	3	4

50	51	52	53	54	55	56	57	58	59
2	16	6	6	3	3	3	1	2	16

60	61	62	63	64	65	66	67	68	69
6	7	9	6	6	7	5	6	5	2

Project 1 [first serious submission due by Monday, September 10]

Implement VPL by starting from the given file `VPL.java` found in the `VPL` folder at the course web site (supported by the little class `IntPair.java`), which (along with some other useful stuff) does the irritating task of replacing symbolic labels by memory indices.

To begin, download `VPL.java` and `IntPair.java` to your working folder.

The given code assumes you are calling it from the command line. If you aren't, you'll need to change the code in obvious ways.

Put in code to implement all the operations. The Be sure to test your program thoroughly. At a minimum, run it on all the programs given in the `VPL` folder, plus your program from Exercise 3. The file `someUnitTests` does a sequence of simple tests of some commands, and you should be able to easily determine the output it should produce to compare to what your program actually does produce. If you submit your work when it is not working as desired on these VPL programs, you should also explain which ones have problems and describe what you are planning to do next to fix your program.

As you're working on this project, remember that this work will directly help you to prepare for Test 1, which will ask you to read, write, and execute VPL code, as well as to add new features to the VPL language.

When you are ready to submit your work (and recall that a serious first submission must be done by the due date or you will never be allowed to submit it) email `shultzj@msudenver.edu` with your file `VPL.java` as an attachment. And, yes, you must do this program in Java, purely for my and your convenience. If there is any feature of the program that you find difficult to do in Java, then you are probably doing things wrong! In other words, implementing a language this primitive should only require very primitive features of Java.

An Assembler for VPL?

⇒ Let's discuss for a while what is so horribly wrong with VPL (even assuming we only want to be able to work with integer values).

Your next Project, following some background work and Exercises to get ready, will be to implement, in Java, a translator that will take a program written in a language that is nicer to use than VPL and translate it to a VPL program.

To keep things from getting out of hand at this point, we will make the following crucial language design decision:

> *The new language should translate line-by-line to a VPL program, where each line of the new language can be translated to some VPL code as we go.*

Exercise 5

Working in groups, ideally of size four or five, *design* this new language. After you work on this a while, we will have a whole group discussion and come up with the draft (we might decide something which turns out later to be bad) design for our joint language.

Note that part of the design is the *name* of the language.

The First Step in Language Implementation: Lexical Analysis

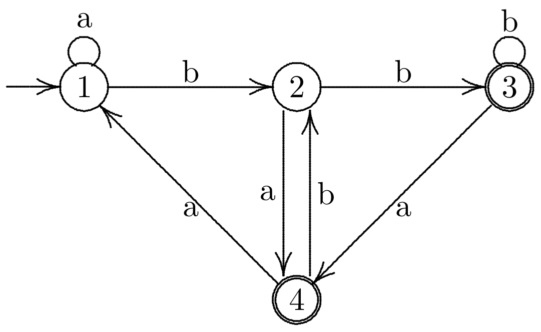
Because the new language is simple enough, the only interesting issue in its implementation will be turning sequences of physical symbols—actual single symbols—into conceptual symbols known as *tokens* or *lexemes.*

To do this, we will first study some ides in what is known as *lexical analysis.*

Finite Automata

A *finite automaton* (FA) consists of a directed graph, where the vertices are known as *states* and are typically drawn as circles, and each edge of the graph is labeled with a list of one or more symbols belonging to the alphabet. Each symbol in the alphabet can only occur on at most one edge leading out of any state. One state is marked by an arrow out of nowhere pointing to it as the *start state*, and one or more states are marked, typically by "double circling," as *accepting states.*

For example, here is an FA using the alphabet $\{a, b\}$:



Here is how an FA is used to decide whether a given string is legal:

> starting from the start state, process each symbol of the input string in turn, until you reach the end of the string, or detect an error. To process a symbol, look at the edges leading out of the current state. If the symbol is listed on one of them, follow that edge to the state on the other end and make that the current state. If the symbol is not listed on any edge, then the string is rejected. Finally, if all the symbols of the input string are successfully processed, then check the state that you end up in—if it is an accepting state, then the input string is accepted as legal, otherwise it is rejected.

For example, in the FA given above, the string `ababbba` is accepted, moving from 1 to 1 to 2 to 4 to 2 to 3 to 3 to 4, and state 4 is an accepting state. On the other hand, the string `bbbab` is not accepted, because it moves from 1 to 2 to 3 to 3 to 4 to 2, and state 2 is not an accepting state. And, the string `abc` is rejected because it moves from 1 to 1 to 2 and then detects no edge out of state 2 that processes a `c`.

⇒ Create an FA for the language of legal identifiers in our new language.

Exercise 6

Working in your small group, create an FA for the language of integer literals.

Then look at the new language design and see all the different kinds of tokens. Make sure that you can create an FA for each kind of token.

Announcement

The Learning Assistants (Peter and Jimmy) have settled on **Office Hours** from 2–3:50 on Tuesday and Thursday, in AES 237 (check the board if they aren't there to see if they moved to another room for more quiet). Barring emergencies, one or the other of them will be available during these times to talk with you about the course material, Exercises, and Projects.

They provide another resource, which is not at all to say that I won't be happy to talk with you.

Practice Problems for Test 1

Test 1 will cover, among other things, our VPL-based introduction to the study of programming languages. You should be able to execute given VPL code by hand, write short chunks of VPL code to do certain things, and write Java code to add commands to the language.

The practice problems below are not intended to cover all issues. For example, any translation of Java-like code to VPL is "fair game."

- Here is a fairly simple VPL program, shown (with no documentation and labels replaced by actual array indices) after the code has loaded. Trace execution of the program, beginning with `ip` at 0, writing in the correct values in all the memory cells until the program halts. When a value is replaced by another value, draw a line through the first value and write the second value in the cell. You will need to track the values of the registers yourself, but you do not need to show them formally.

  You should draw boundaries between the code segment, global segment, and stack segment, and between the various stack frames.

  Also, show what will be displayed on screen.

`ip`

0	1	2	3	4	5	6	7	8	9
32	1	4	3	22	0	17	22	1	3

10	11	12	13	14	15	16	17	18	19
3	1	3	0	2	24	6	2	28	2

20	21	22	23	24	25	26	27	28	29
33	0	2	26	4	1	9	2	0	1

30	31	32	33	34	35	36	37	38	39
5	2								

40	41	42	43	44	45	46	47	48	49

50	51	52	53

- Here is a snapshot of VPL memory at a point of execution where `bp`, `sp`, and `ip` are as shown. Demonstrate your understanding of VPL by explaining this snapshot as follows. Figure out where the code segment ends and draw a clearly visible vertical line between the last cell of the VPL program and the first cell of the stack segment (there is no global segment in this program). Draw clearly visible boundary lines showing the various stack frames existing at this moment. Finally, fill in memory cells as you execution for another function call—start with `ip` at 10—and continue up to but not including the next function call after that.

0	1	2	3	4	5	6	7	8	9
4	1	22	0	1	3	0	2	10	26

`ip`									
10	11	12	13	14	15	16	17	18	19
4	3	28	0	29	22	3	2	11	1

20	21	22	23	24	25	26	27	28	29
3	0	3	1	2	10	6	2	5	2

30	31	32	33	34	35	36	37	38	39
0	0	1	30	9	1	2	0	2	33

40	41	42	43	44	45	46	47	48	49
26	2	4	0	2	39	26	4	8	0

50	51	52	53	54	55	56	57	58	59
2	45	26	8	16	0	2	51	26	16

			`bp`						`sp`
60	61	62	63	64	65	66	67	68	69
32	0	2	57	26	32	64	0	2	

70	71	72	73	74	75	76	77	78

- Write a fragment of VPL code that is a reasonable translation of each fragment of Java-like code shown below. Assume that `A` and `B` are sequences of 0 or more statements in the Java-like language, and denote the translation of a sequence of 0 or more statements `A` to VPL by the function call `v(A)`—so `v(A)` is the sequence of VPL statements that `A` translates to. Also assume that the variable `x` translates to local cell 7 in the function where this fragment occurs, and that the variable `y` translates to global cell 3. If you need any additional memory cells, assume that you can freely use local cells 10, 11, and so on, as needed. Use labels starting with 1000 in the code you generate.

```
if( x < y )
{
   A
}
else
{
  B
}
```

```
for( x=1; x<10; x++ ){
   A
}
```

- Suppose we want to add a command to VPL that will allow us to get the physical address—the index in the memory array—of a local cell in the currently active stack frame and store it in some local cell. This would be useful, for example, if we wanted to implement pass-by-reference.

  Specifically, suppose we want to add a command with this specification:

Instruction	Mnemonic	Semantic Description
`42 a b`	get address	Get the index in the memory array of local cell `b` and store it in local cell `a`.

  Your job is to write code that will implement this new command. Your code must fit in with the fragments of code shown. Specifically, assume that `op` holds the current operation number, which is stored in `mem[ip]`, and the variables `a`, `b`, and `c` hold the arguments to the command, from cells `mem[ip+1]`, `mem[ip+2]`, and `mem[ip+3]`.

  Note that you have to write code that will cause the desired behavior *and* correctly update `ip`.

```
    do{
      op = mem[ ip ];  ip++;
      // code to extract op and the args into a, b, c
      // for convenience omitted
      ...  (lots  of code omitted) ....
      else if( op == 22 )    // literal
      {
        mem[ bp+2+a ] = b;
        ip += 3;
      }
      else if( op == 23 )  // copy
      {
        mem[ bp+2+a ] = mem[ bp+2+b ];
        ip += 3;
      }
      else if( op == 24 )  // add
      {
        mem[ bp+2+a ] = mem[ mem[ bp+2+b] + mem[ bp+2+c ] ];
        ip += 4;
      }
      ... (lots more code omitted )...
      // write your code to implement command 42 here:







      else
      {
        System.out.println("Fatal error: unknown opcode [" + op + "]" );
        System.exit(1);
      }

    }while( !done );
```

A Survey of Memory Issues in Some Early Imperative Languages

We will now look at a number of memory models used in early imperative languages that differ from VPL.

> The VPL memory model is actually closest to the non-object-oriented parts of Java, which is pretty much the same as C, except C had powerful features—discussed later—that were eliminated from Java.

In this discussion we will refer to various memory models by the language that more or less follows that model, but we don't care about the details, just the main ideas.

In all this work, we will imagine that a Java program, with its own vast memory, is used to translate the source code into VPL-like language, using whatever memory resources and data structures it needs, but then at run-time the program is executed using just the single memory array, together with a fixed set of Java variables acting as "registers." This is identical to how we implemented VPL.

We will only look at features of these languages that correspond to VPL abilities, namely working with integers only.

Some Terminology

People talking about programming languages (including us in the following material) often use terms related to the *semantics* of a language, including:

> *scope (also known as "visibility"):* the region of code in which a particular variable has the same meaning, in the sense that it refers to the same cell in memory
>
> *binding time:* the time in the entire process when the meaning of an entity (usually a variable, sometimes other things) in the code is "bound" to its meaning (memory cell for a variable). Some things (e.g. `int` literals in Java) are bound at *language design time*. Other things (e.g. `final static` variables in Java) are bound at compile time
>
> > (probably—as the assembly language code is produced from the Java code, every occurrence of such a variable might be replaced by its value).
>
> Other things (e.g. the location of a local variable in a method in Java, which will be somewhere on the stack which is unknowable until the code is executed) are bound at *run time*.

We will tend in the following to be more precise, as we present memory models for the various languages at a level like we have done for VPL.

Language implementations fall into two main categories, namely *translated* (old timey word for this was "compiled") or *interpreted*. This distinction actually only refers to the typical approach for a given language—any language can in principle be translated—have the entire source code for some program in the language translated to code in some other language which can then somehow be executed—or interpreted—have individual statements be processed and executed—but it might be more common or convenient to do one or the other.

Java is typically translated—from Java source code for a class to Java virtual machine language code for that same class—with the Java "byte code"—the `.class` files—being interpreted. But, there are Java compilers that translate from Java source code to assembly language code for a particular platform (CPU and operating system).

The Basic Model

By "Basic' here we mean the old-time programming language "Beginner's All-Purpose Symbolic Instruction Code."

Here are the core features of this memory model:

- The language is best thought of as interpreted.
- All variables have global scope—they can be accessed anywhere in the program, with the same meaning. As each identifier (for a "variable") is encountered for the first time during execution of the code, it is assigned to a memory cell. Later occurrences of the same identifier mean the same memory cell.
- Support for subprograms is minimal—just the `gosub` and `return` commands, with no named subprograms, no parameters, and no local variables.
- No dynamic memory features, except sort of for arrays.
- Arrays are created in the global memory space, and are allowed to have the size of an array only known at run-time. Arrays are created by the `dim` command, such as `dim a(6)` meaning to create an array named `a` that has 6 memory cells allocated.

⇒ Develop a scheme for memory usage for this model, in the spirit of the memory pictures we drew for VPL.

⇒ Is recursion supported by this model?

⇒ Could Basic really be implemented by translation in the sense of scanning the code and mapping variables to memory cells before actually executing the translated code?

The Pre-90 Fortran Model

Note: the following is discussing Fortran as it was in the early days (Fortran 77, maybe?). Apparently modern Fortran (Fortran 90 and up) has lots of further features, including recursion and objects. This reminds me of the old joke: "what will the language of the future be like?" "I don't know, but it will be called 'Fortran' ."

- All variables have local scope—they can only be accessed in the subprogram where they occur.

- Parameter passing is done by "pass by reference." This means that the address of an argument is passed to the subprogram, and the compiler translates all occurrences of parameters to use the memory location with the address stored in the parameter.
- No dynamic memory. Array declarations must have a size that can be determined statically—that is, at compile time.
- No recursion.

⇒ Develop a scheme for memory usage for this model, in the spirit of the memory pictures we drew for VPL. Think about how a translator for this language would deal with variables. Note that with this model, values of local variables can have a lifetime that is the lifetime of the program execution.

⇒ Be careful to understand exactly how pass-by-reference works.

⇒ Why is recursion not allowed? Why must arrays have their size known at compile-time?

The Pascal Model

- Parameter passing is done by pass by value or pass by reference, depending on whether the parameter has a special key word (`var`, in case you care) occurring in its declaration to specify pass by reference.
- Dynamic memory is allowed, through the ability to allocate any number of bytes on the heap during execution.
- Scoping of variables is very complicated, because *nested subprograms* are allowed. In other words, subprograms are declared within subprogram bodies, much like local variables.

⇒ Consider why this seems like a good idea. Programming language experts speak of the idea of "orthogonality" in programming language design. The following discussion will indicate why this is not such a good idea, maybe—is the additional complexity worth it?

- In fact, there are two different ideas for scoping in such a language, known as static (or lexical) scoping and dynamic scoping. Pascal actually uses static scoping, but we will ponder both approaches here.
- To deal with the semantics of nested subprograms, we will use the following execution model:

  Instead of stack frames on a linear strip of memory like we have been doing, we will use a 2D picture. A rectangle is drawn to represent the local variables and subprograms for a subprogram. Each such rectangle is labeled in its upper right corner by the name of the subprogram. In the upper left corner of each rectangle we write all the identifiers—variables and subprogram names—that are declared within the subprogram.

  Start with a large box representing the entire program, and start executing the body of the program in this context. When a subprogram is called, draw a box for it either inside the active box (that's dynamic scoping) or immediately inside the box where the subprogram name is declared (that's static scoping).

The meaning of an identifier—both variables and subprograms—is determined by first looking at the upper left corner of the current box, and then scanning *outward* until the identifier is first located.

⇒ Find or write some Pascal code and execute it according to these rules. Execute the same code with both scoping models.

⇒ Draw a possible memory picture for this model, in the spirit of the memory pictures we drew for VPL. Note that the boxes we draw are essentially stack frames, and try to figure out how a stack frame could hold the information necessary to implement each scoping model.

The C Model

C was invented in an era when Pascal-like languages (actually as we read online the first day, Niiklaus Wirth invented Pascal as a simpler language reacting to the complexity of Algol 68) were dominant and represented a step toward simplicity.

- C has variables and functions similar to Java's `static` fields and methods. Variables declared outside any function are visible to any function (there are lots of details about occurrence within files, but we don't care much).
- C only implements pass-by-value, but it has operators * (with `int * x` meaning `x` holds an address, and `*x` used in an expression meaning the cell(s) that `x`'s address points to), and & (the "address of" operator)
- Dynamic memory is allowed.
- C only has functions. Earlier languages (notably Fortran and Pascal) had both functions (subprograms that return a value) and subroutines/procedures (subprograms that don't return a value).

⇒ Draw a possible memory picture for the C model, in the spirit of the memory pictures we drew for VPL. Be sure to include the fact that C allows arbitrarily large arrays and `struct` variables to be local variables.

Official Announcement of Policy Change

Some people have asked to work in groups of size larger than three on the Projects, so I am somewhat reluctantly changing the course policy to allow groups of up to five students submitting Projects together. If you want to have six people in a group, you must ask for permission. Larger groups will not be allowed.

Follow-up to Exercise 5 and 6

After the whole group discussion during class session 5, Jimmy and I (Peter was flooded out), but I'll take most of the blame, designed the new language to be named `Jive`.

I then proceeded over the weekend to begin implementing a Jive to VPL translator, as a sanity check before assigning the same work as Project 2. To avoid burying the lede (yes, that's not the word "lead" as I thought my whole life until recently), I'm not going to assign this work for Project 2, because it seemed too much. Instead, you will be responsible (on Test 1) for the concepts of this work: being able to hand-execute a given Jive program, being able to write a Jive program to perform a specified task, and, most importantly, being able to translate a given Jive program into VPL.

You can find the code I'm writing in the `Jive` folder at the course web site. I hope to finish it by 9/11/2018.

⇒ I will present the draft design for Jive, driven by the draft finite automata hand-written on the next page. Feedback will be considered (but I've done a fair amount of the implementation work, so I won't be too open to changes other than correcting problems.

⇒ Write a Jive program that will get a positive integer n from the user and will then compute and report $n!$.

⇒ Hand-translate this program to VPL, demonstrating the ideas of my partially-completed code.

Exercise 7

Working in your small group, write the following Jive programs, and translate each into VPL:

- The same goal as Exercise 1, part c.
- The same goal as Exercise 1, part d.
- The same goal as Exercise 3.

Exercise 8

Note: do this before Exercise 7!

I've been working as time allows on the Jive implementation (current version available at the course web site). You can help me debug it!

Here is the `factorial.jive` program next to the translated VPL version:

```
                                       1 1000
                                       4 3
                                       27 0
                                       23 1 0
                                       3 1
                                       2 2000
                                       6 0
                                       23 2 0
/* main function                       23 0 2
   for computing                       28 0
   factorial of                        29
   input value                         26
*/                                     1 2000
                                       4 5
Keys -> n                              22 2 2
Fact n -> f                            22 3 1
f -> Prt   NL                          16 1 0 2
Halt                                   8 2001 1
                                       10 1 0 3
Def Fact n .                           23 4 1
                                       3 4
Less n 2 -> Jmp small:                 2 2000
                                       6 1
   Sub n 1 -> temp                     23 5 1
   Fact temp -> f                      11 1 0 5
   Mult n f -> Ret                     5 1
                                       1 2001
small:                                 23 1 3
  1 -> Ret                             5 1
```

The Jive Manual

In the following, quantities in angle brackets, like `<funcName>`, represent single non-terminal symbols in the given grammar (we'll talk more carefully about context free grammars later). Other symbols represent physical symbols that can be typed in a Jive program file.

A context free grammar is basically a collection of rules that say the quantity on the left can be replaced by the quantities on the right of the arrow (the arrow used in CFG's is different from the two physical symbols `->` that appear in Jive code).

Note that I'm using a CFG to specify Jive, instead of a finite automaton like I gave you earlier, because it is easier to type grammar rules than to typeset diagrams.

Quantities that appear in square brackets, like `[ <var> ]`, are optional.

Here are the *syntax* rules (with some English mixed in to the rules, purely for convenience).

`<Jive program>` → `[ <globalDec> ] <main> [ <functions> ]`

`<globalDec>` → `Globs [ <params> ] .`

`<params>` → zero or more `<param>`

`<functions>` → zero or more `<function>`

`<main>` → zero or more `<statement>`

`<statements>` → zero or more `<statement>`

`<function>` → `Def <funcName> [ <params> ] .  <statements>`

`<statement>` → `/*` any words that are not */ `*/`

`<statement>` → `Halt`

`<statement>` → `NL`

`<statement>` → `<label>`

`<statement>` → `Jmp <label>`

`<statement>` → `<part1> -> <part2>`

`<part1>` → `<bif2> <var> <var>`

`<part1>` → `<bif1> <var>`

`<part1>` → `Keys`

`<part1>` → `<funcCall> [ <vars> ] .`

`<part1>` → `<var>`

`<part1>` → `Fet <param>`

`<funcCall>` → `<funcName>` followed by 0 or more `<var>`

`<part2>` → `.`

`<part2>` → `Prt`

`<part2>` → `Sym`

`<part2>` → `Ret`

`<part2>` → `<var>`

`<part2>` → `Jmp <label>`

`<part2>` → `Put <var> <var>`

`<part2>` → `Sto <params>`

`<param>` → a lowercase letter followed by zero or more letters or digits

`<funcName>` → an uppercase letter followed by zero or more letters or digits

`<label>` → any sequence of letters or digits ending with a colon

`<var>` → either an integer literal or any sequence of letters and digits starting with a lowercase letter

`<bif2>` → a built-in function with two arguments, namely one of
`Add, Sub, Mult, Quot, Rem, Eq, NotEq, Less, LessEq, And, Or, Get`

`<bif1>` → a built-in function with one argument, namely one of `Not, Opp, New`

Semantics of Jive

The semantics (meaning) of Jive programs is intended to be obvious to the experienced VPL programmer, because Jive constructs translate fairly directly to VPL commands (that was our design goal for this new language).

For clarity we should say that every `<part1>` somehow generates a value which is then used in the matching `<part2>` in some fairly obvious way.