OVERVIEW

Control Flow:

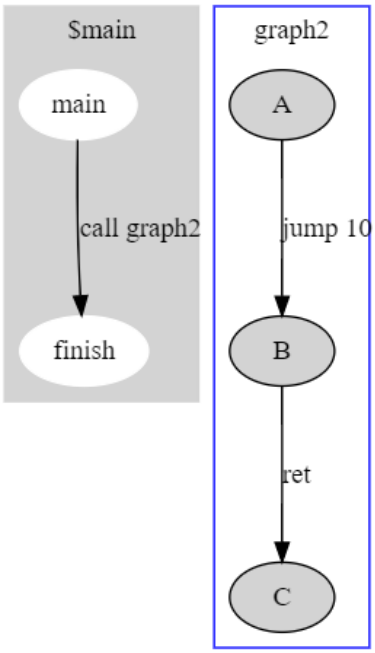
— States:

- Programs written in the PORC language represent the underlying parsing state machine. States describe the sequence of actions that one must take to parse - roughly, one state can be thought of as a line of code.
- States are declared *implicitly* - you define one simply by using it in a transition.
- We allow the state machine graph to be a *multigraph*, i.e. multiple transitions are allowed between two states.
- **Convention:** all state names must begin with a $.
- The initial state for a program is always $main.
- The final state for a program is always $end - this is needed in order to determine when to stop measuring the size of a PDU.
- The program will implicitly transition from $end back to $main after finishing parsing one PDU in order to prepare for parsing the next one.

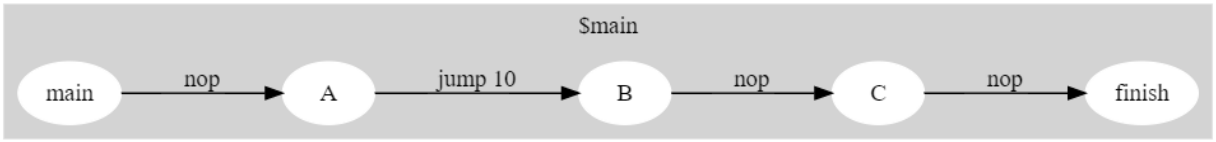
— Subgraphs:

- A subgraph represents a “function call” - a state can “call” into a subgraph, which would then return into a specified state. Subgraphs may assume that certain registers contain arguments that can be used and overwritten - caller graphs must set up these registers prior to calling the subgraph.
- Subgraphs are defined using labels, in the same way that functions in assembly are defined with labels.
- Note that these are not *true* function calls - recursion is not supported, and functions are *inlined* into the FSM graph.
- **Convention:** subgraph labels must start with the @ character.
- **Ex:**

```
$main $finish true (call @graph2)
@graph2:
$A $B true (jump 10)
$B $C true ret
```



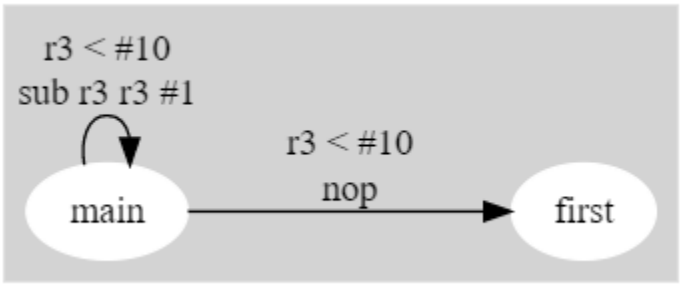
After inlining, this might look like:



— Conditional Transitions:

- All transitions between states are conditional. A transition only occurs when the condition is met. Note that programs **must** be deterministic state machines - it must never be the case that conditions for different transitions from a single state can simultaneously be satisfied.

```
$main $first (r3 < #10) (nop)
$main $main (r3 >= #10) (sub r3 r3 #1)
```



Here, there are two conditional transitions from main. The conditions depend on the state of the register r3.

Memory Model:

— Registers:

- `k` general-purpose 32-bit registers are used to keep track of intermediate information necessary for parsing, such as HTTP content-length, DNS number of records
- Registers can be used as parameters in conditions or actions - they tend to represent either lengths or quantities.
- Registers can also be use to pass parameters into subgraphs - however, it is the programmer's responsibility to keep track of which registers are being used and avoid clobbering data
- Registers are labeled as r0-rk - every register is fully general purpose, and can be written to and read from.

— Program Counter:

- The program counter keeps track of which state is active in the state machine. This is necessary, as the active state defines which transitions and actions are possible.

— Bytestream Read Index:

- A PORC language program requires an associated bytestream that it will parse through. The bytestream read index simply points to the index in the bytestream the program is currently reading.

Instruction Format:

Each instruction represents one transition between states.

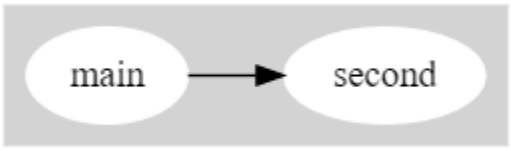
```
[[starting state]] [[ending state]] [[condition]] [[action to take]]
```

Both conditions and actions can take arguments (either a hardcoded immediate value or a register to read).

Note that register reads and immediate values are specified in different ways when passed as instruction arguments. Immediate values are written as #n, where n is the actual decimal value being passed into the action or condition. Registers, on the other hand, are specified as rA, where A is the register number being read. In this case, the literal value A is not passed in, but instead a previously stored value is read and used.

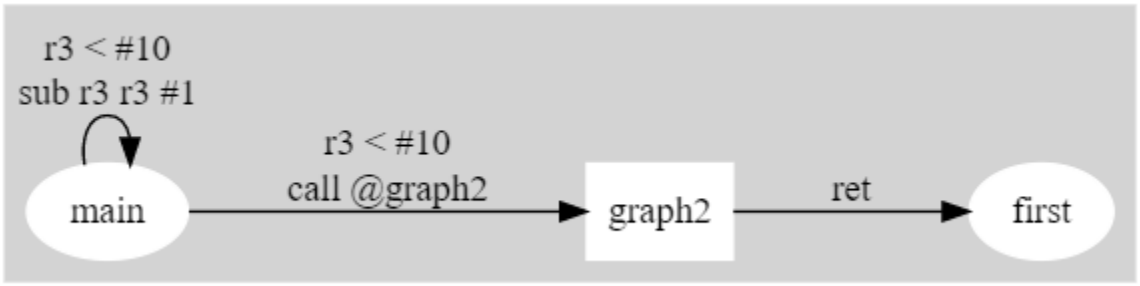
Examples

```
$main $second true jmp #8
```



This instruction unconditionally moves the bytestream read index forward by 8 bytes, and transitions to a new state labelled `$second`

```
$main $first (r3 < #10) (call @graph2)
$main $main (r3 >= #10) (sub r3 r3 #1)
```



This instruction sets up a simple loop - while the r3 register contains a value larger than 10, we decrement r3. However, once r3 gets below 10, we escape the loop and jump into a subgraph called graph2. The code for graph2 is not shown, but once it returns, it will jump to the $first state

Condition Definitions:

Note: condition cannot modify the state of the parser. It can only access elements of the state (a register, read index, or bytestream).

Condition	Explanation
`match <str>`	This transition is only taken if <str> is found in the bytestream at the current read index.

`regA > immediateN`
`regA < immediateN`
`regA = immedateN`
	These comparison conditions each require both a register (`regA`) and an immediate value (`immediateN`) to compare it to.

`true`	Makes a transition unconditional, often used if you need to skip forward in the bytestream

Instruction Definitions:

Action	Explanation
`jmp regA/immN`	This instruction moves the read index forward by `regA` or `immN` bytes.
`nop`	Does nothing, can be used if you want to transition states without performing any other action.
`Latch_num_ascii regA`	This instruction reads the ASCII numerical value at the bytestream read index into `regA`. It reads this in greedily, consuming bytes until it finds a non-digit byte or exceeds the maximum ascii length. After this action, the read index will be pointing to the byte immediately following the number.
`latch_num_bin regA #immN/regB`	This instruction reads the next `immN` or `regB` bytes into `regA`. It also advances the read index by `immN` or `regB` bytes.
`call @subgraph`	The call instruction provides functionality to use subgraphs as reusable functions. This instruction takes two immediate parameters - `@subgraph` is a label that represents the subgraph you are calling into. The instruction's destination state is stored to be used as the subgraph's return destination . **Note:** these subgraphs are not truly jumped to like functions - rather, these subgraphs will be *inlined* into the graph, between the start state and ending state of this instruction. This limits us to relatively simple subgraphs, but reduces the amount of state we are required to carry.
`ret`	This jumps back to the state that was passed with the call instruction.
`jmp_match`	This instruction jumps the read index to the end of the matched string - must **only** be used in conjunction with a `match` condition
`add/sub regD regA immN/regB`	Stores the result of one of: ● regA+regB

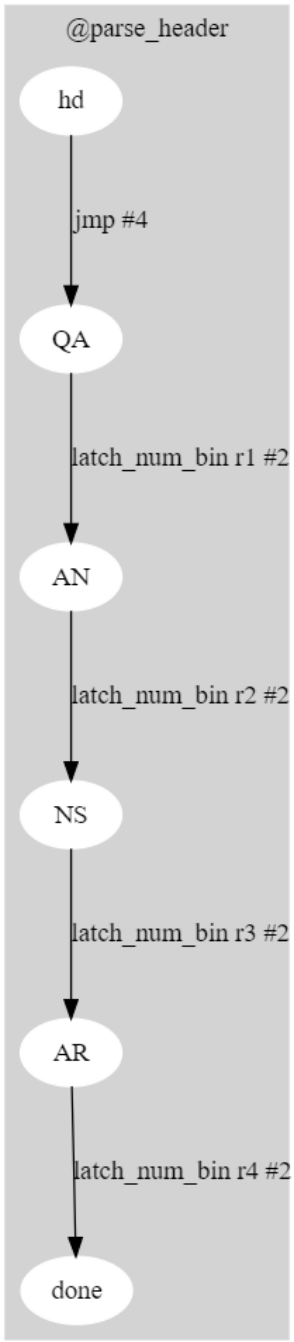
	• regA-regB • regA+immN • regA-immN into regD.

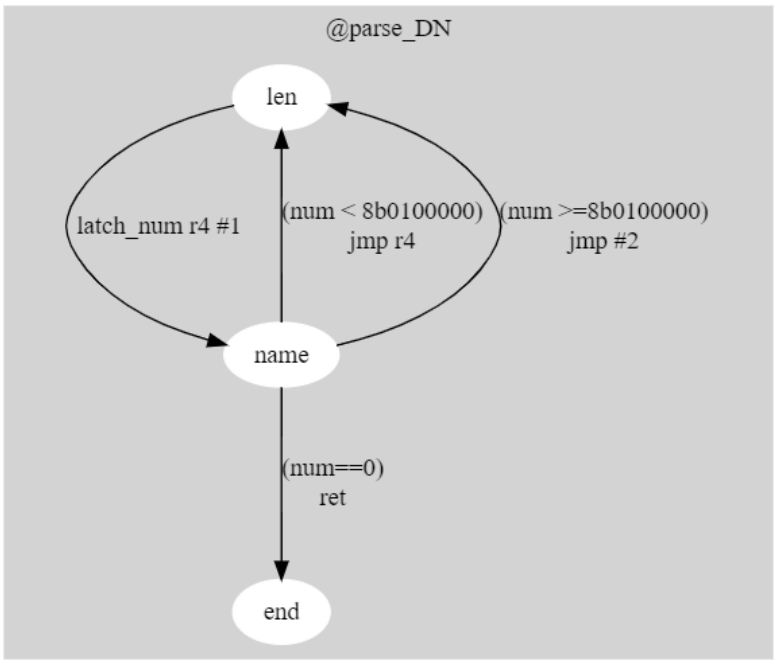
DNS: Instructions and State Machine Diagram

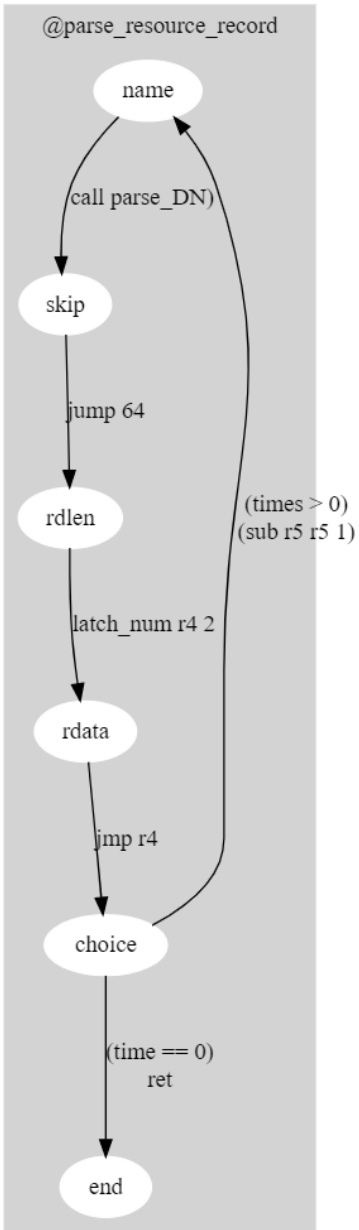
Program:

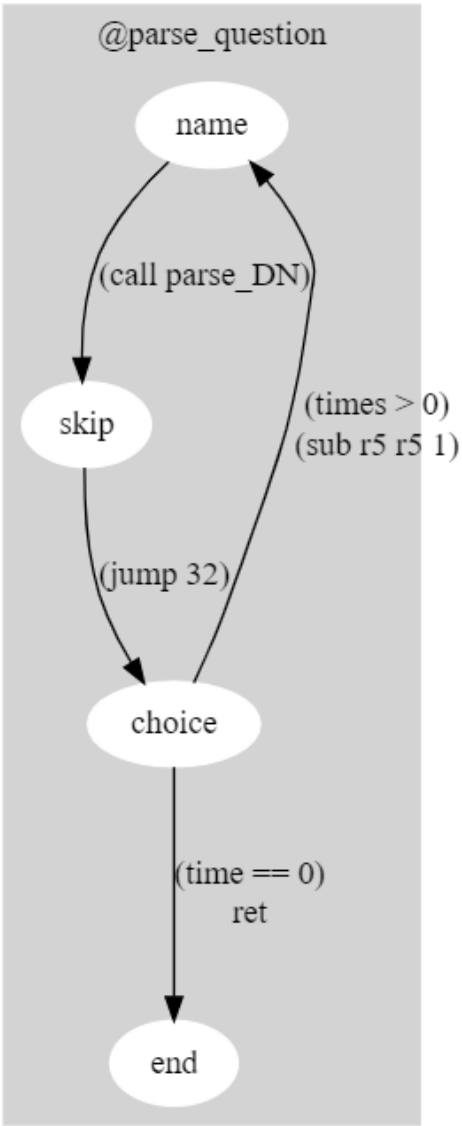
```
@parse_header:
$hd $QA true (jmp #4)
// Next three lines store the number of records
$QA $AN true (latch_num_bin r1 #2)
$AN $NS true (latch_num_bin r2 #2)
$NS $AR true (latch_num_bin r3 #2)
$AR $done true (latch_num_bin r3 #2)
$done $end true ret

@parse_DN:
$len $name true (latch_num r4 #1)
$name $len (num < 8b0100000) (jmp r4)
$name $len (num >=8b0100000) (jmp #2)
$name $end (num == 0) ret

@parse_resource_record: //r5 = num times
$name $skip true (call parse_DN)
$skip $rdlen true (jump 64)
$rdlen $rdata true (latch_num r4 2)
$rdata $choice true (jmp r4)
$choice $name (times > 0) (sub r5 r5 1)
choice end (time == 0) ret

@parse_question: //r5 = num times
name skip true (call parse_DN)
skip choice true (jump 32)
choice name (times > 0) (sub r5 r5 1)
choice end (time == 0) ret

@parse_dns:
$main $Q true parse_header
$Q $Q1 true (add r5 r0 r1) //store times variable
$Q1 $An true (call parse_question)
```

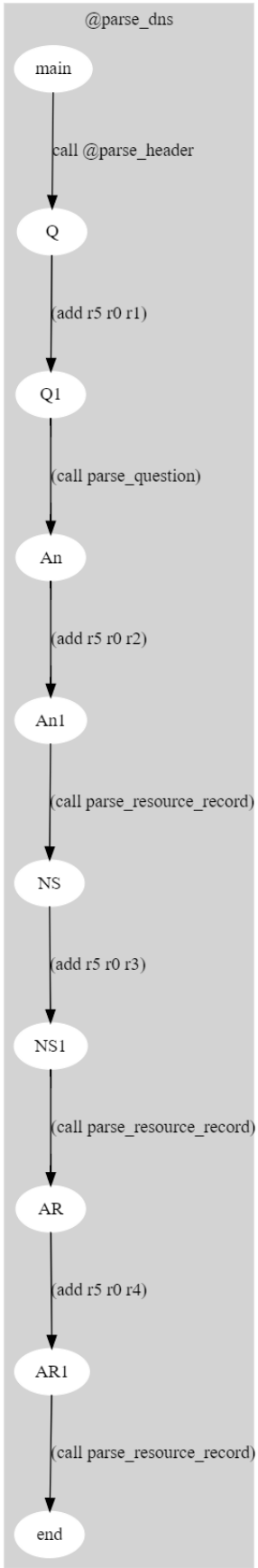
```
$An $An1 true (add r5 r0 r2)
$An1 $NS true (call parse_resource_record)
$NS $NS1 true (add r5 r0 r3)
$NS1 $AR true (call parse_resource_record)
$AR $AR1 true (add r5 r0 r4)
$AR1 $end true (call parse_resource_record)
```





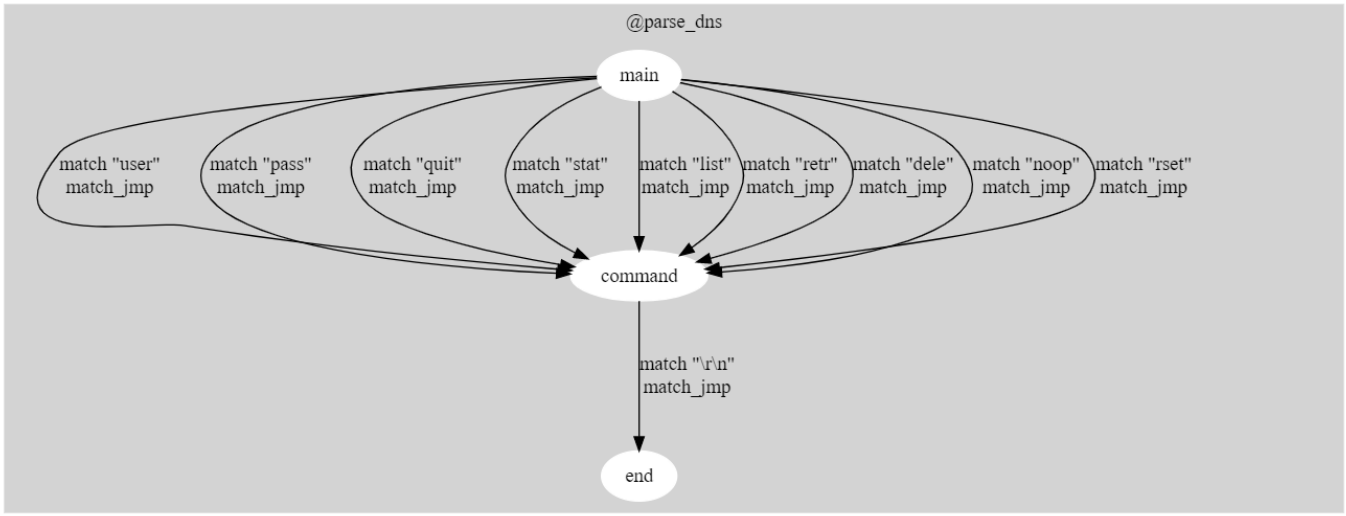

@parse_resource_record
name
call parse_DN)
skip
jump 64
rdlen
(times > 0)
(sub r5 r5 1)
latch_num r4 2
rdata
jmp r4
choice
(time == 0)
ret
end


@parse_question
name
(call parse_DN)
skip
(times > 0)
(sub r5 r5 1)
(jump 32)
choice
(time == 0)
ret
end


@parse_dns
main
call @parse_header
Q
(add r5 r0 r1)
Q1
(call parse_question)
An
(add r5 r0 r2)
An1
(call parse_resource_record)
NS
(add r5 r0 r3)
NS1
(call parse_resource_record)
AR
(add r5 r0 r4)
AR1
(call parse_resource_record)
end

POP3: Instructions and State Machine Diagram

```
$main $command (match “user”) match_jmp //moves bytestream index
$main $command (match “pass”) match_jmp
$main $command (match “quit”) match_jmp
$main $command (match “stat”) match_jmp
$main $command (match “list”) match_jmp
$main $command (match “retr”) match_jmp
$main $command (match “dele”) match_jmp
$main $command (match “noop”) match_jmp
$main $command (match “rset”) match_jmp
$command $end (match “\r\n”) match_jmp
```



SMTP: Instructions and State Machine Diagram

```
$main $command (match “HELO”) match_jmp
$main $command (match “EHLO”) match_jmp
$main $command (match “MAIL FROM”) match_jmp
$main $command (match “RCPT TO”) match_jmp
$main $commandLong (match “DATA”) match_jmp
$main $command (match “VRFY”) match_jmp
$main $command (match “TURN”) match_jmp
$main $command (match “EXPN”) match_jmp
$main $command (match “AUTH”) match_jmp
$main $command (match “RSET”) match_jmp
```

```
$main $command (match “EXPN”) match_jmp
$main $command (match “HELP”) match_jmp
$main $command (match “QUIT”) match_jmp
$command $end (match “\r\n”) match_jmp
$commandLong $end (match “.\r\n”) match_jmp
```