jvst

JSON schema validation while streaming tool

jvst

- JSON schema
  - Describes valid structure of JSON document
  - Like DTDs for XML
- Can it be done faster and leaner?
- Proposed by Kate Flavel
  - Joint design
  - Kate: AST and most of the parsing bits
  - Me: SJP library and most of the backend

jvst: goals

Validate JSON with a schema

- Maximize throughput!
- Minimize memory!

Typical validators:

- Parse JSON schema
- Parse JSON data into a tree
- Descend the tree

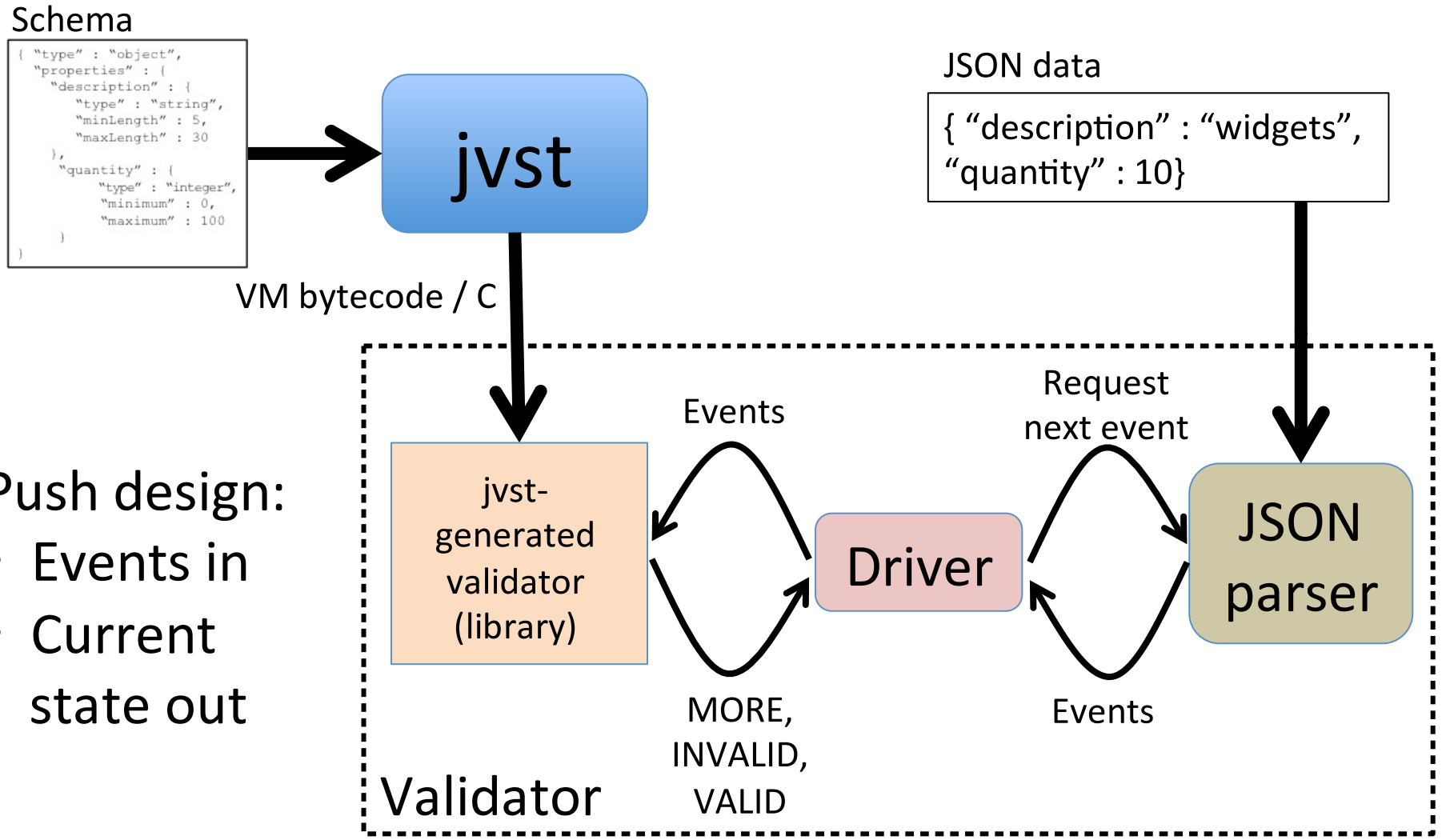
Big ideas:

- Combine parsing and descent
- Schema → automaton
- String matching → fast DFAs
- Ultimately emit C

jvst: what it uses

- C99
- SJP: streaming JSON lexer and parser
  - Push, not pull
  - Minimizes changing input
- libfsm: DFA manipulation
  - Union / intersection / subtraction
  - Output: table or C state machine
  - libre: regular expressions → DFAs

jvst: general design



Push design:

- Events in
- Current state out

jvst: schema

- JSON describing other valid JSON
- Example:
  - MUST be an object
  - “description”: must be a string, length 5-30
  - “quantity”: must be an integer, 0–100

Schema

```
{"type" : "object",
   "properties" : {
    "description" : {
      "type" : "string",
      "minLength" : 5,
      "maxLength" : 30
    },
    "quantity" : {
        "type" : "integer",
        "minimum" : 0,
        "maximum" : 100
    }
}
```

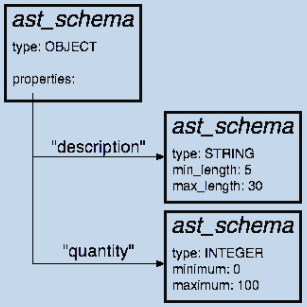
jvst: stages

Schema

```
{ "type" : "object",
  "properties" : {
    "description" : {
        "type" : "string",
        "minLength" : 5,
        "maxLength" : 30
    },
    ...
```

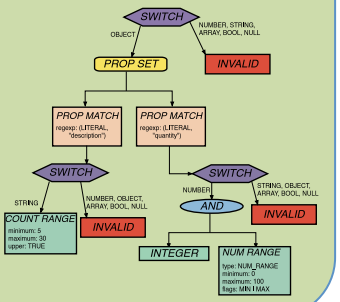
Parse to *ast_schema*

AST



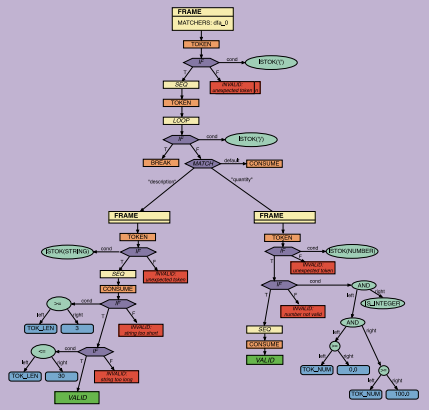
Translate to *jvst_cnode*

Constraint tree



Translate to *jvst_ir_stmt*

IR tree



Linearize

Linear IR

```
PROGRAM
FRAME(1):
    MATCHERS[MATCHER(0, "dfa_0")]
entry_0:        TOKEN,
                CBRANCH(
                    ISTOK($OBJECT_BEG),
                    loop_4,
                    invalid_1_17)

invalid_1_17:   INVALID(1, "unexpected token")

loop_4:         TOKEN
                CBRANCH(
                    ISTOK($OBJECT_END),
                    valid_15,
                    false_7)

false_7:        MOVE(ITEMP(0), EMATCH(0)),
                CBRANCH(
                    EQ(ITEMP(0), 0),
                    M_9,
                    M_next_10)

M_next_10:      CBRANCH(
                    EQ(ITEMP(0), 1),
                    M_11,
                    M_next_12)
```

Translate to *jvst_vm_program*

VM bytecode

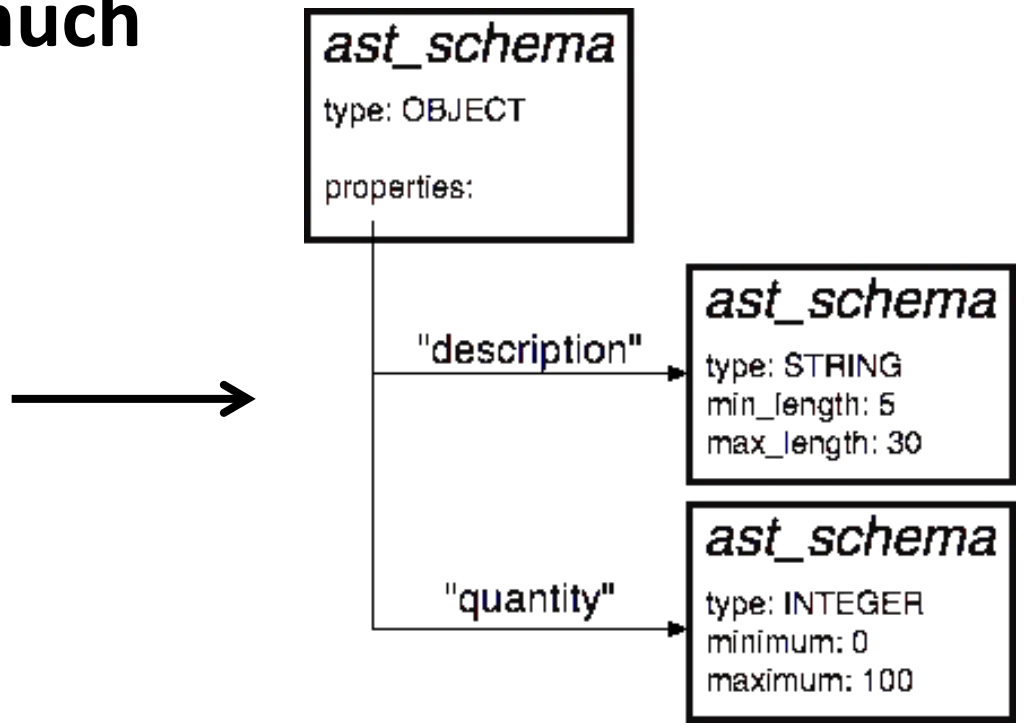
```
00001   TOKEN     $0, $0
00002   IEQ       %TT, $6
00003   CBT       2
00004   INVALID   $1, $0
00005   TOKEN     $0, $0
00006   IEQ       %TT, $7
00007   CBT       16
00008   MATCH     $0, $0
00009   MOVE      %R0, %M
00010   IEQ       %R0, $0
00011   CBT       6
```

jvst: parsing

AST nodes do too much

Schema

```
{"type" : "object",
   "properties" : {
    "description" : {
        "type" : "string",
        "minLength" : 5,
        "maxLength" : 30
    },
     "quantity" : {
           "type" : "integer",
           "minimum" : 0,
           "maximum" : 100
     }
}
```



AST: direct representation

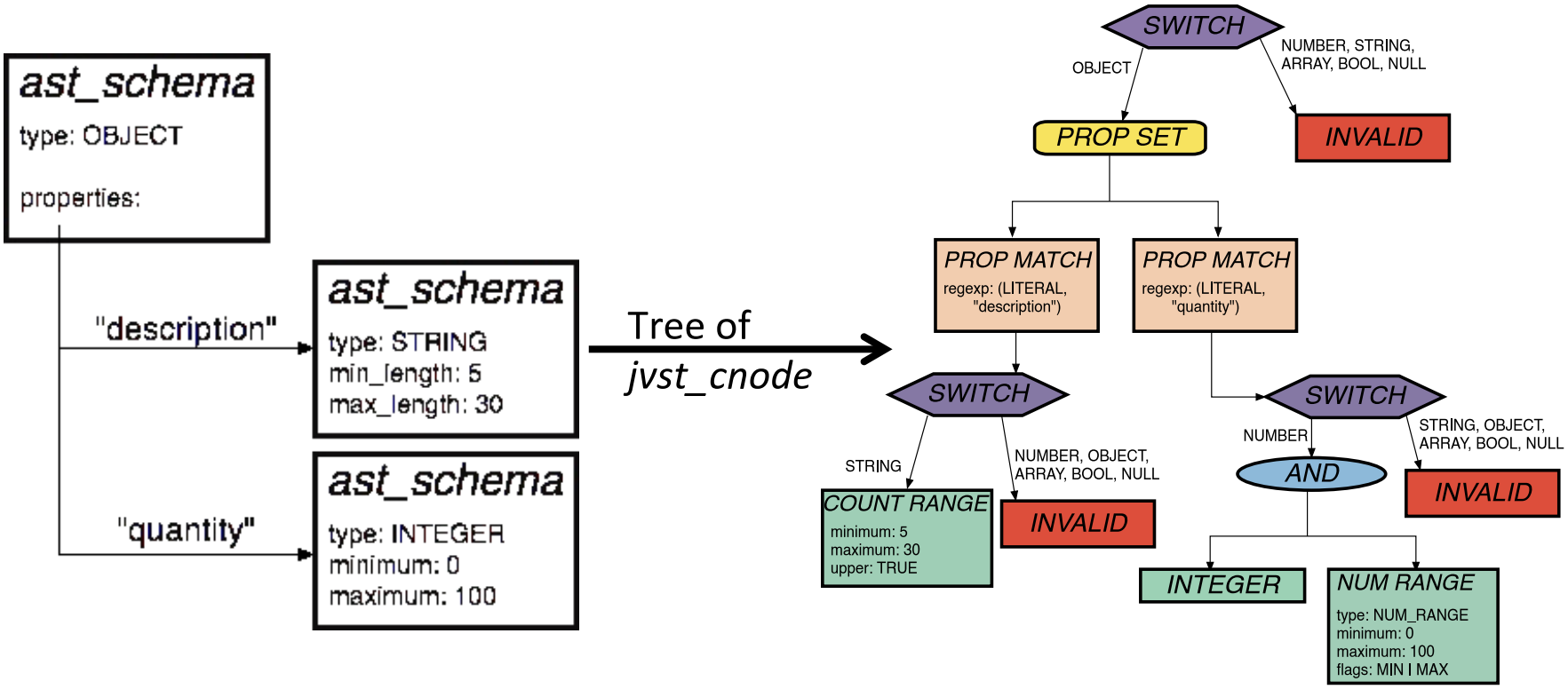
- SJP lexer: emits JSON tokens
- SID: LL(1) parser generator

AST nodes do too much

Too much is implicit

Lots of redundancy

jvst: constraint tree



jvst_cnode

tagged union

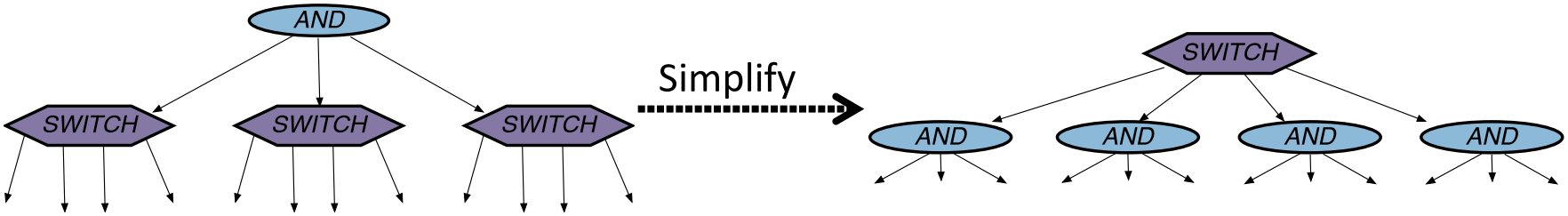
each node: one thing

Easier to reason about:

Rewrite subtrees

Rewrite complex constraints

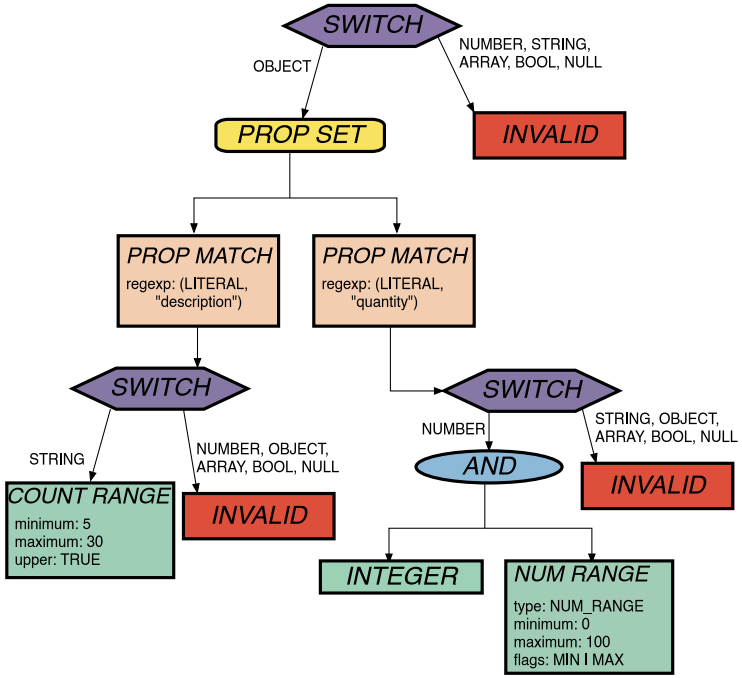
jvst: constraint tree



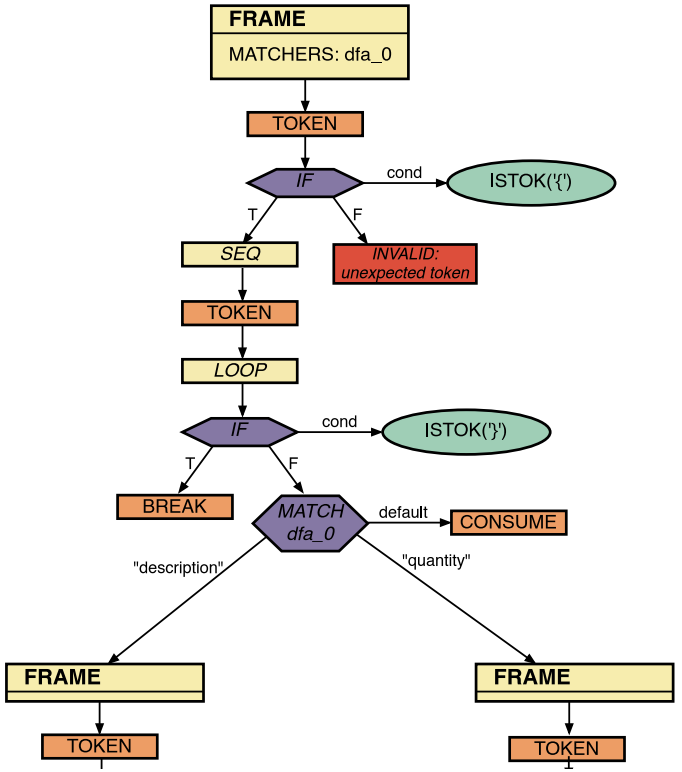
Example transform:

- Push ANDs down (also ORs/XORs)
- Gathers related logic
- Allows more simplification

jvst: IR tree



Tree of
jvst_ir_stmt

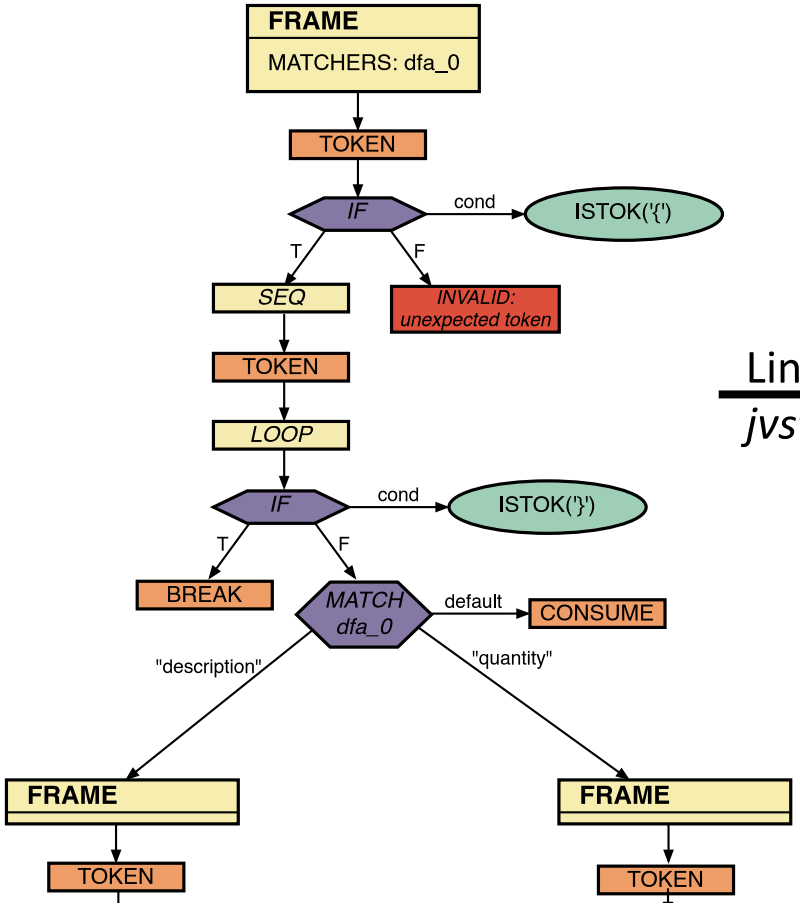


jvst_ir_stmt

- tagged union
- some nodes hold *jvst_ir_expr* trees

Constraint tree is declarative (eg: SQL)
IR tree is imperative (eg: C/Python)
Closer to runnable code

jvst: Linear IR



Linearize tree

jvst_ir_stmt

```
PROGRAM
FRAME(1):
      MATCHERS[MATCHER(0, "dfa_0")]
entry_0:        TOKEN,
                CBRANCH(
                      ISTOK($OBJECT_BEG),
                      loop_4,
                      invalid_1_17)

invalid_1_17:   INVALID(1, "unexpected token")

loop_4:         TOKEN
                CBRANCH(
                      ISTOK($OBJECT_END),
                      valid_15,
                      false_7)

false_7:        MOVE(ITEMP(0), EMATCH(0)),
                CBRANCH(
                      EQ(ITEMP(0), 0),
                      M_9,
                      M_next_10)

M_next_10:      CBRANCH(
                      EQ(ITEMP(0), 1),
                      M_11,
                      M_next_12)
```

same tree of *jvst_ir_stmt*

- simplifies nodes
- tree of blocks

Basic scheduling

Some optimization:

- Jumps to jumps, dead blocks

jvst: emit bytecode

```
PROGRAM
FRAME(1):
    MATCHERS[MATCHER(0, "dfa_0")]
entry_0:        TOKEN,
                CBRANCH(
                    ISTOK($OBJECT_BEG),
                    loop_4,
                    invalid_1_17)

invalid_1_17:   INVALID(1, "unexpected token")

loop_4:         TOKEN
                CBRANCH(
                    ISTOK($OBJECT_END),
                    valid_15,
                    false_7)

false_7:        MOVE(ITEMP(0), EMATCH(0)),
                CBRANCH(
                    EQ(ITEMP(0), 0),
                    M_9,
                    M_next_10)

M_next_10:      CBRANCH(
                    EQ(ITEMP(0), 1),
                    M_11,
                    M_next_12)
```

Emit *jvst_vm_program*

```
00000 PROC      $1, $0
00001 TOKEN     $0, $0
00002 IEQ       %TT, $6
00003 CBT       2
00004 INVALID   $1, $0
00005 TOKEN     $0, $0
00006 IEQ       %TT, $7
00007 CBT       16
00008 MATCH     $0, $0
00009 MOVE      %R0, %M
00010 IEQ       %R0, $0
00011 CBT       6
00012 IEQ       %R0, $1
00013 CBT       6
00014 IEQ       %R0, $2
00015 CBT       6
00016 INVALID   $9, $0
00017 CONSUME   $0, $0
00018 BR        -13
00019 CALL      5
00020 BR        -15
00021 CALL      20
00022 BR        -17
00023 VALID     $0, $0
```

jvst_vm_program

container for VM opcodes, constants, DFAs

Minor transformations: CBRANCH

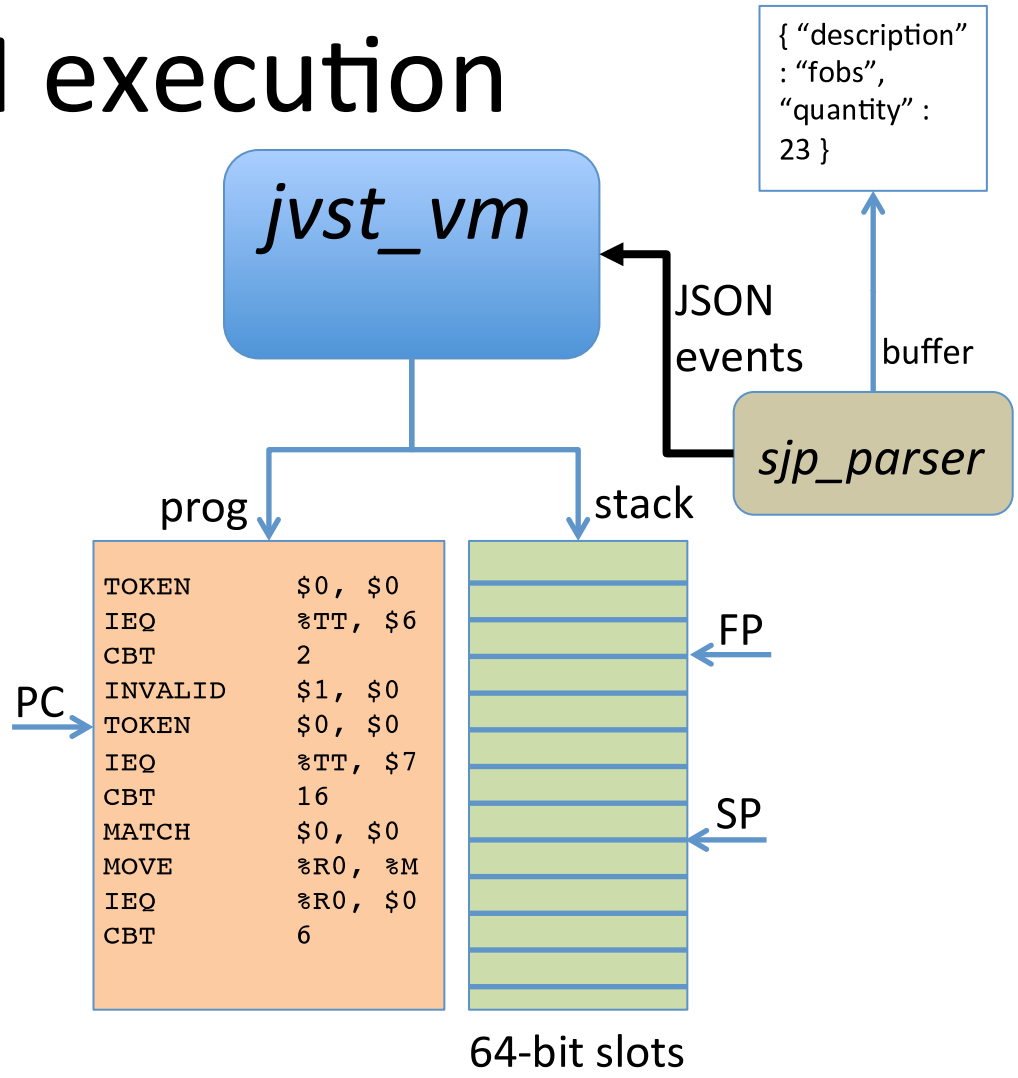
Map blocks → offsets

Encode instructions: fixed width (32-bits)

TODO: move coalescing

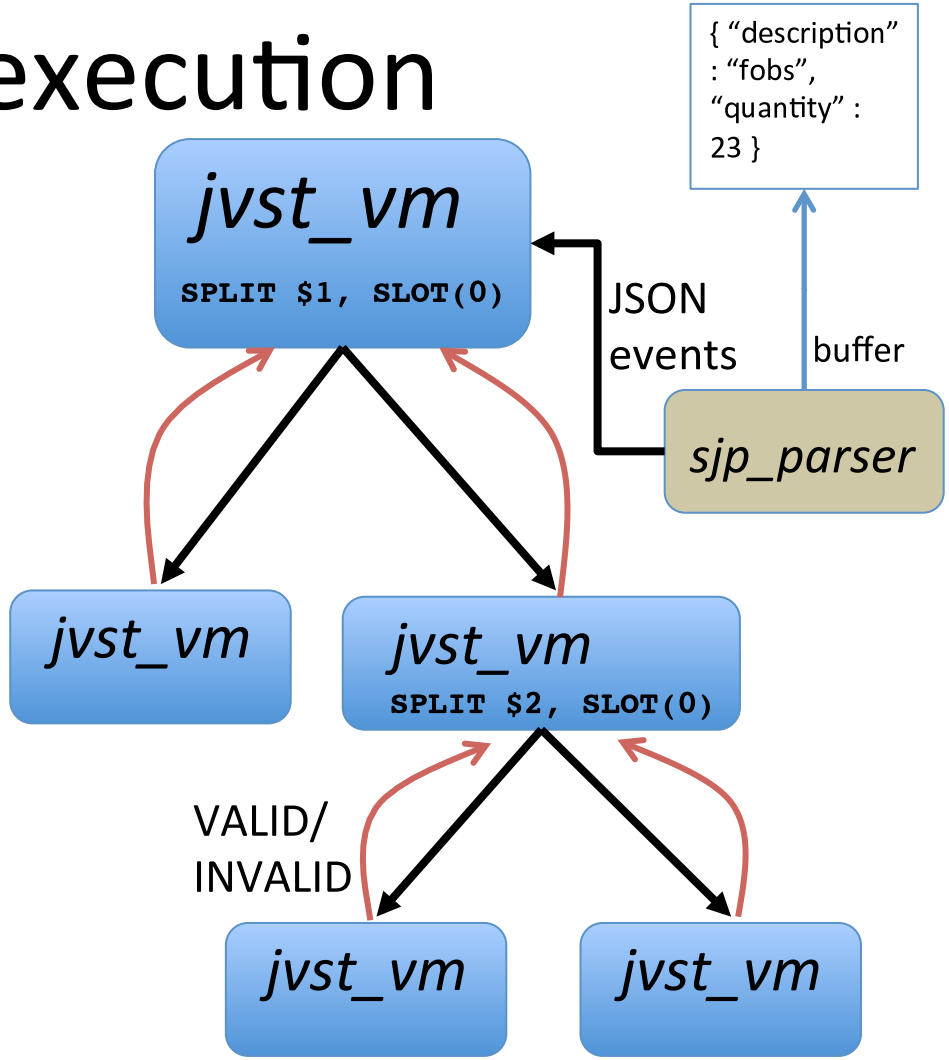
jvst: VM execution

- Simple “big switch” VM
- Push-validator
  - Fed stream of JSON event tokens
  - Stop on INVALID
  - Buffer exhausted, ask for more



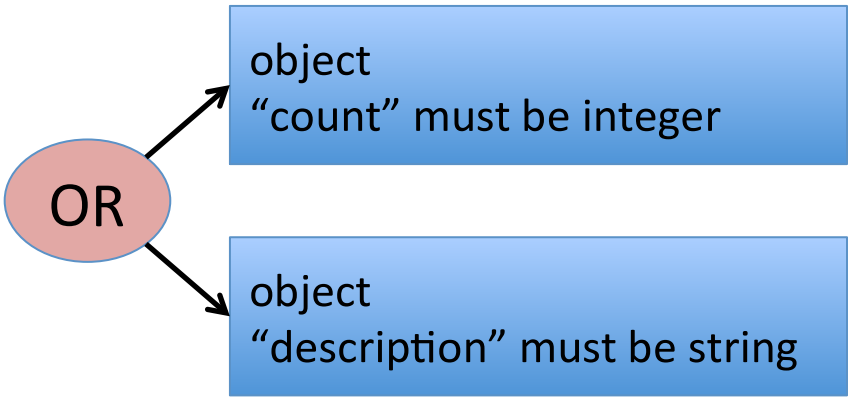
jvst: VM execution

- Split validator tree
  - Events pushed from parent to child
  - VALID/INVALID: child to parent
  - AND/OR/XOR/NOT
  - Synchronous on input tokens



jvst: Why PUSH?

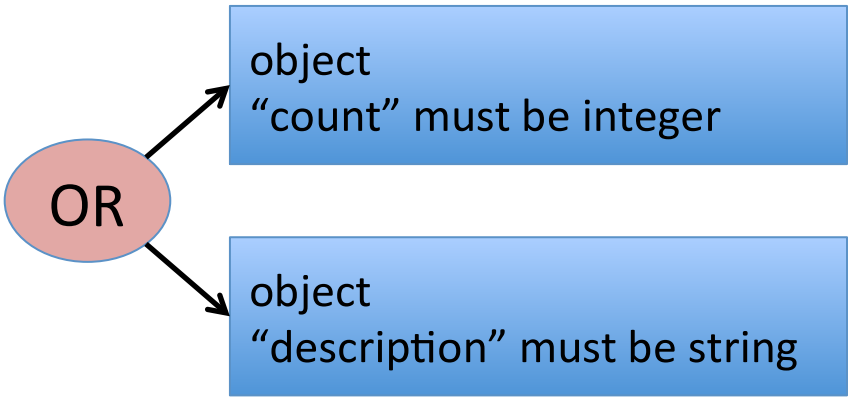
- **No JSON tree**
  - Goal: validate while parsing
- Consider anyOf




```
{ "anyOf" : [{ "type" : "object",
                 "properties" : {
                     "count" : { "type" : "integer" }
                 }
             },
             { "type" : "object",
                 "properties" : {
                     "description" : { "type" : "string" }
                 }
             }]
}
```

jvst: Why PUSH?

- **No JSON tree**
  - Goal: validate while parsing
- Consider anyOf

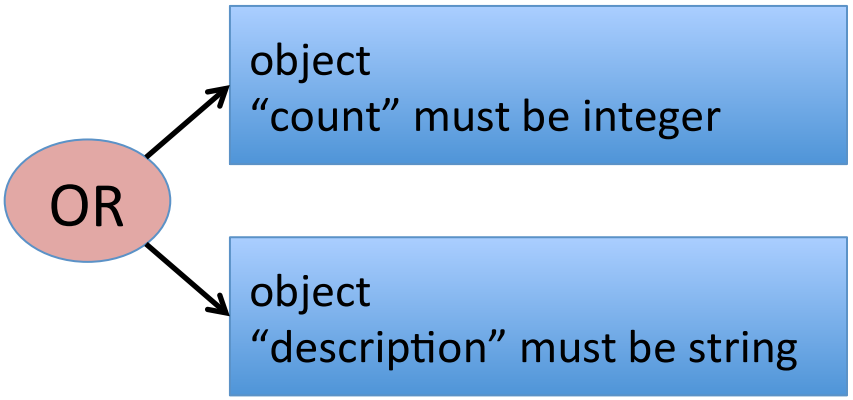


Valid JSON:

- { "description" : false, "count" : 23 }
- { "count" : [5,10], "description" : "foo" }

jvst: Why PUSH?

- Easy with a tree:
  - Descend tree, testing each schema

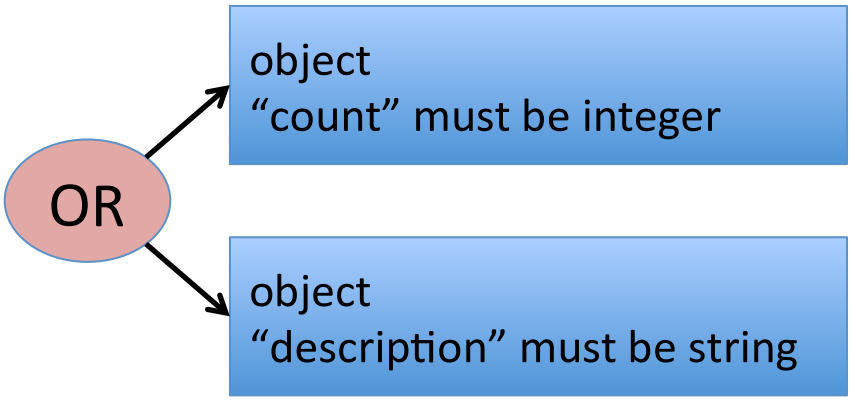


Valid JSON:

- { "description" : false, "count" : 23 }
- { "count" : [5,10], "description" : "foo" }

jvst: Why PUSH?

- Stream, not tree
  - Tokens are consumed
  - Validators in parallel?
- But!
  - Consume different numbers of events
  - Pull via callbacks is a mess!

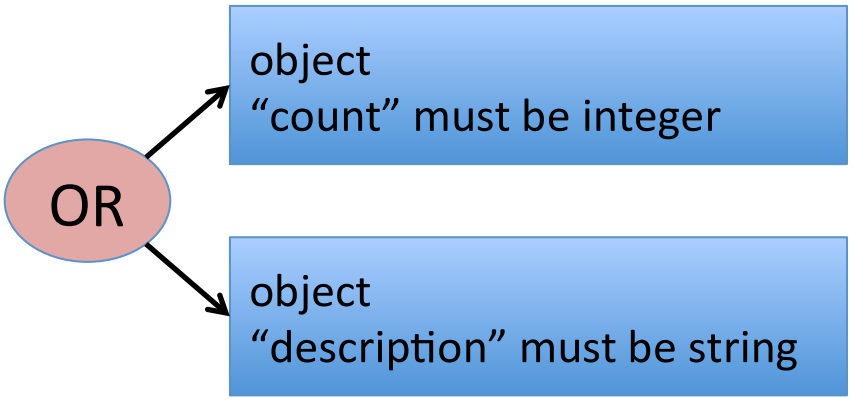


Valid JSON:

- { “description” : false, “count” : 23 }
- { “count” : [5,10], “description” : “foo” }

jvst: Why PUSH?

- Stream, not tree
  - Tokens are consumed
  - Validators in parallel?
- Solution: Push validator
  - Validators fed one token at a time
  - All validators work on same event



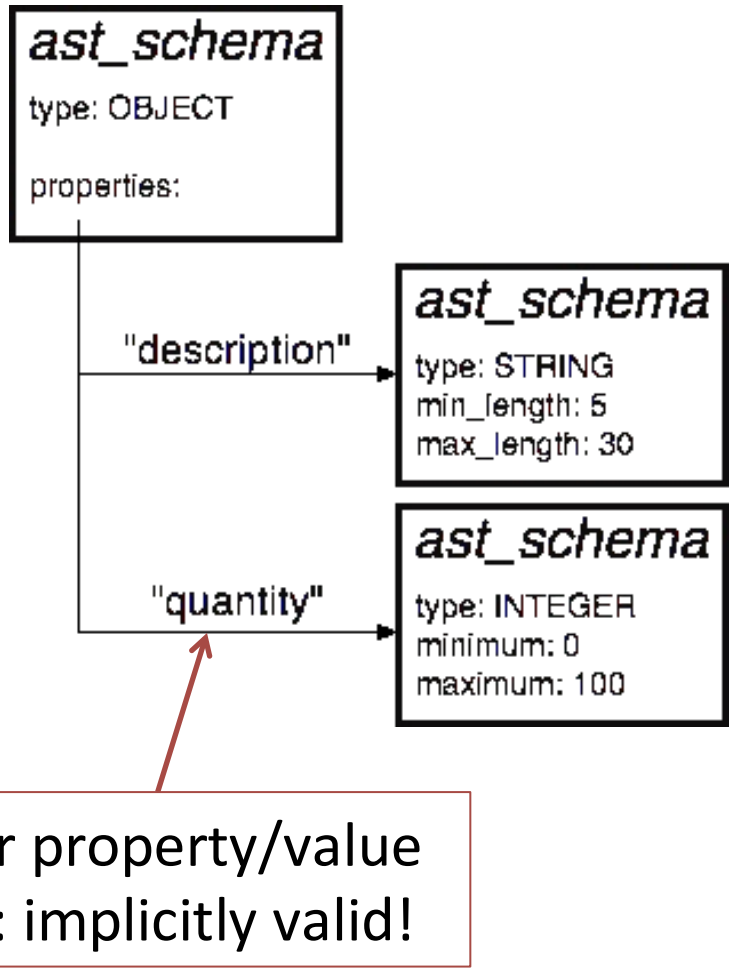
Valid JSON:

- { “description” : false, “count” : 23 }
- { “count” : [5,10], “description” : “foo” }

jvst: the AST

Schema nodes do too much

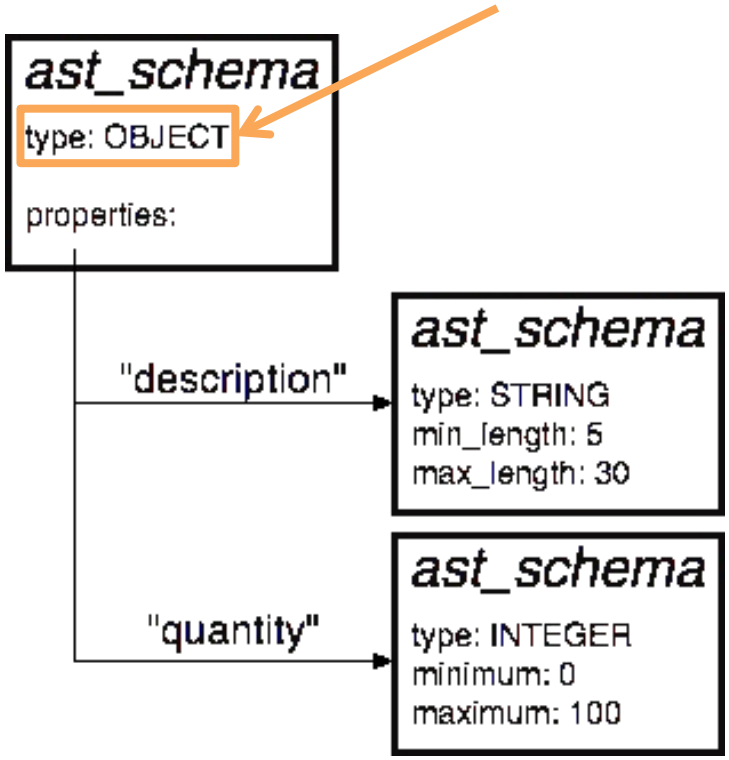
- Implicit valid/invalid
- Same node: several types
- Lots of redundant string matching
  - properties, patternProperties, propertyNames, required, dependencies
- **Hard to reason about**



jvst: the AST

Schema nodes do too much

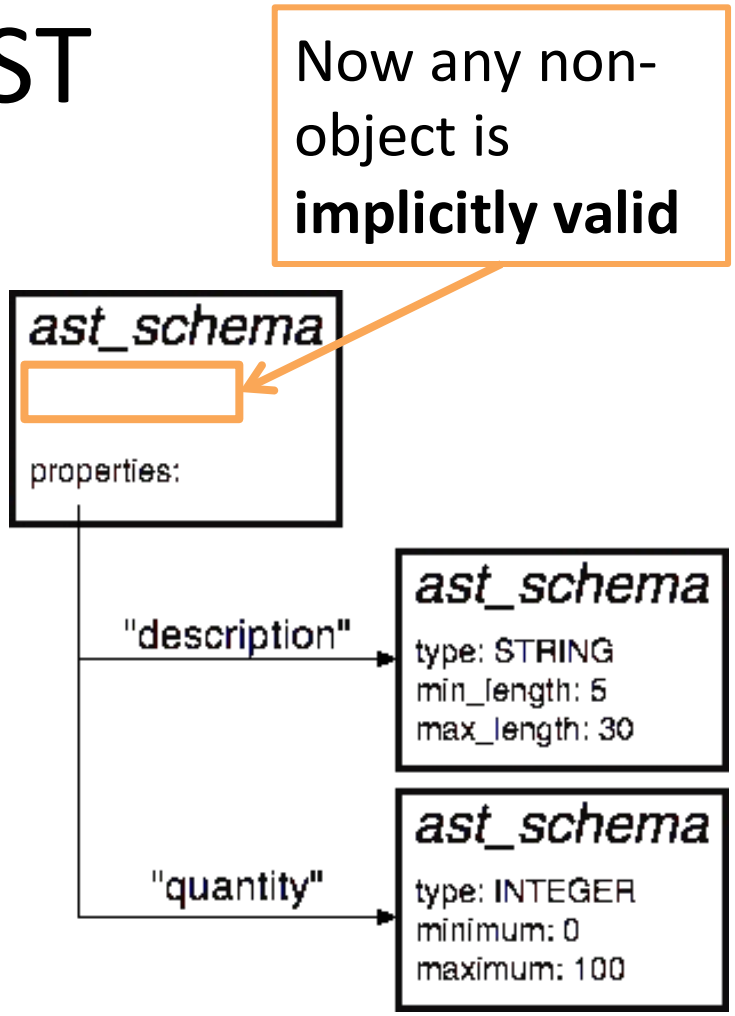
- Implicit valid/invalid
- Same node: several types
- Lots of redundant string matching
  - properties, patternProperties, propertyNames, required, dependencies
- **Hard to reason about**



jvst: the AST

Schema nodes do too much

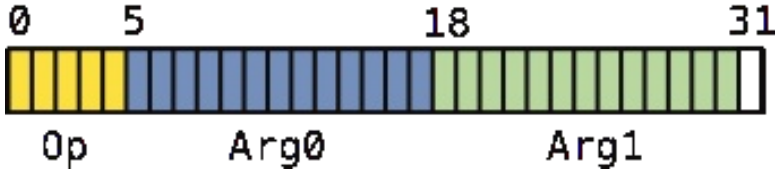
- Implicit valid/invalid
- Same node: several types
- Lots of redundant string matching
  - properties, patternProperties, propertyNames, required, dependencies
- **Hard to reason about**



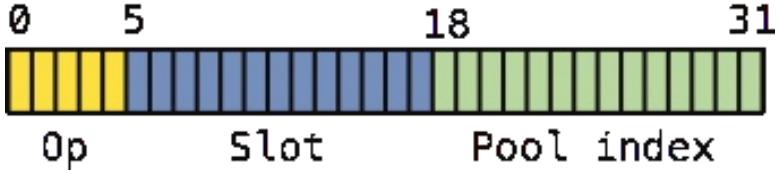
jvst: emit bytecode

- DFA:
  - Graphs → compact tables
- Constant pool
  - All floats
  - ints > 13 bits
- TODO: register allocation
  - Mainly to coalesce moves
- Emit opcodes
  - Fixed width: 32 bits
  - CBRANCH

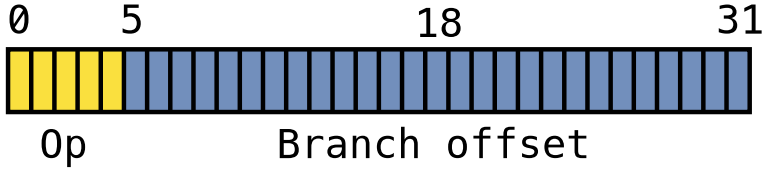
2-op args (IEQ, MOVE, etc.)



ILOAD / FLOAD

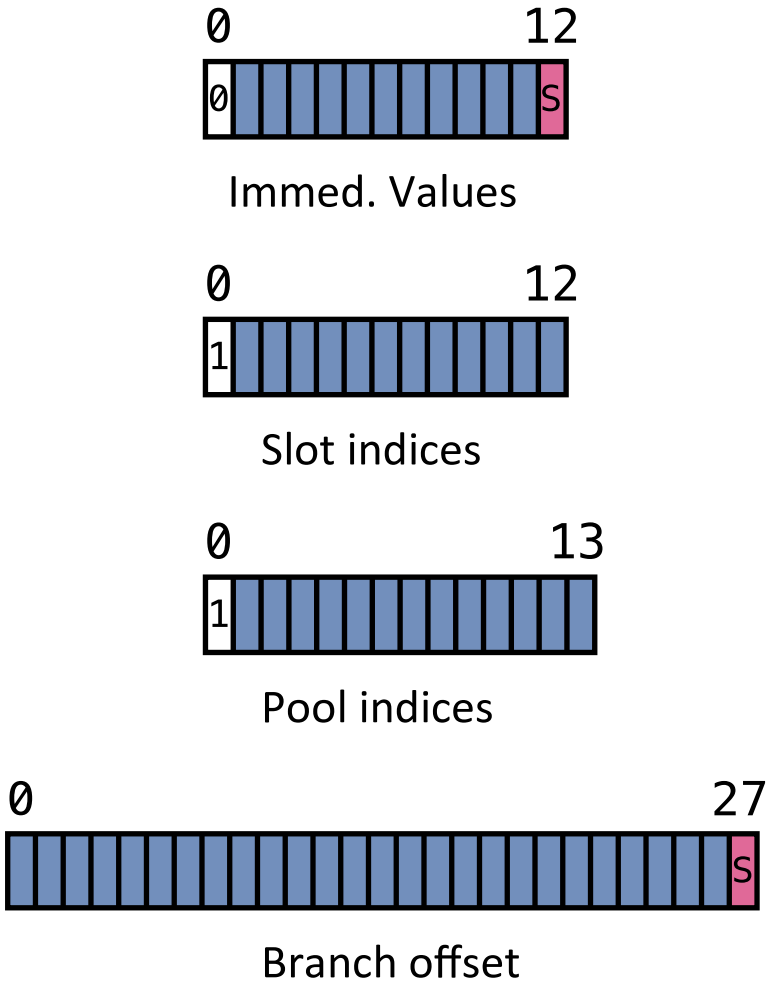


Branching (BR, CBT, CBF)



jvst: emit bytecode

- DFA:
  - Graphs → compact tables
- Constant pool
  - All floats
  - "Big" ints
- TODO: register allocation
  - Mainly to coalesce moves
- Emit opcodes
  - Fixed width: 32 bits
  - CBRANCH



jvst: stages

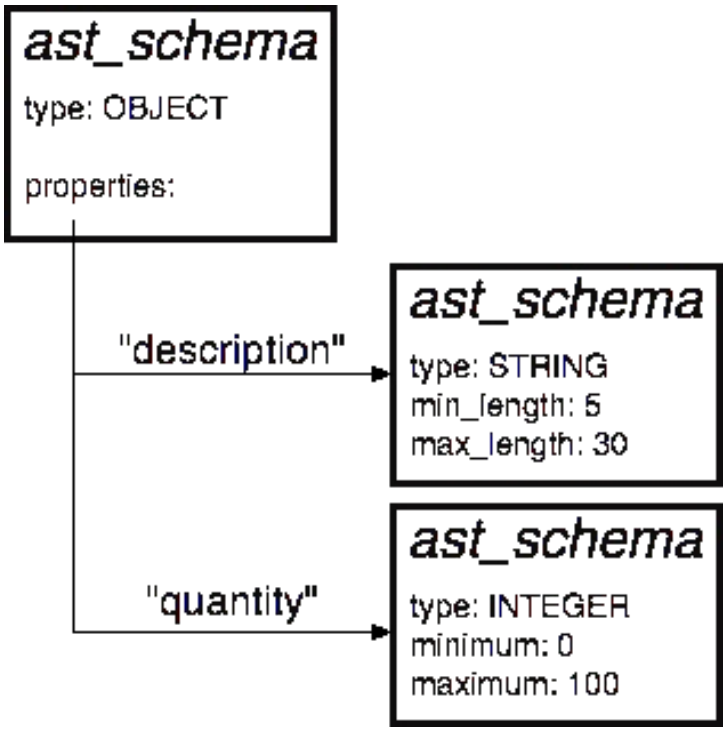
1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode

Schema

```
{"type" : "object",
   "properties" : {
    "description" : {
      "type" : "string",
      "minLength" : 5,
      "maxLength" : 30
    },
    "quantity" : {
        "type" : "integer",
        "minimum" : 0,
        "maximum" : 100
    }
}
```

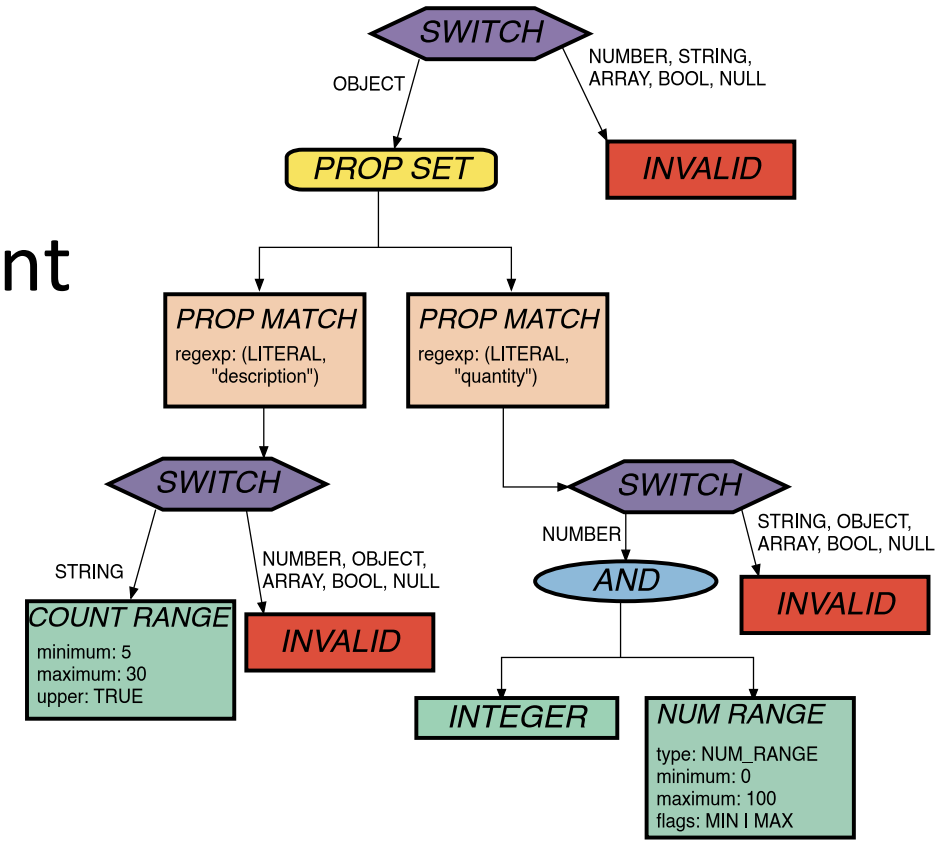
jvst: stages

1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode



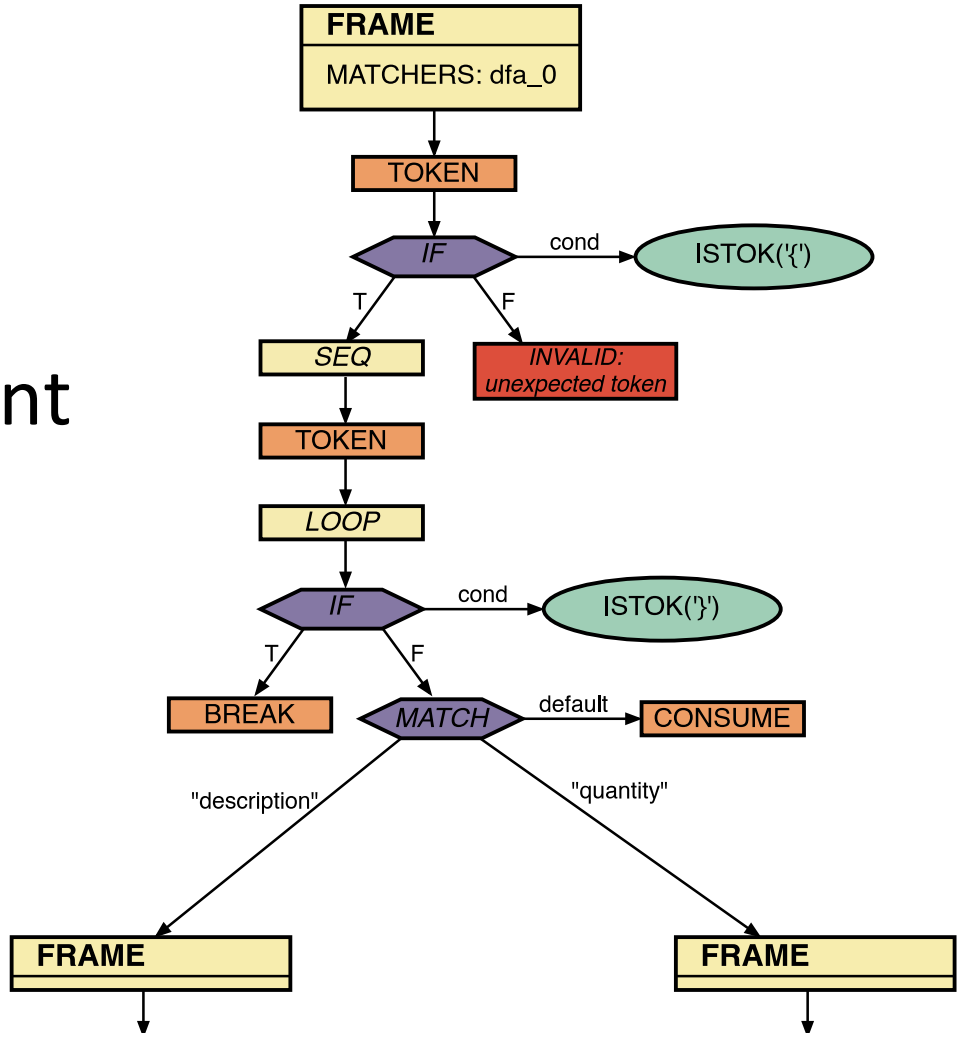
jvst: stages

1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode



jvst: stages

1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode



jvst: stages

1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode

```
PROGRAM
FRAME(1):
      MATCHERS[MATCHER(0, "dfa_0")]
entry_0:          TOKEN,
                  CBRANCH(
                        ISTOK($OBJECT_BEG),
                        loop_4,
                        invalid_1_17)

invalid_1_17:     INVALID(1, "unexpected token")

loop_4:           TOKEN
                  CBRANCH(
                        ISTOK($OBJECT_END),
                        valid_15,
                        false_7)

false_7:          MOVE(ITEMP(0), EMATCH(0)),
                  CBRANCH(
                        EQ(ITEMP(0), 0),
                        M_9,
                        M_next_10)

M_next_10:        CBRANCH(
                        EQ(ITEMP(0), 1),
                        M_11,
                        M_next_12)
```

jvst: stages

1. Input: JSON text
2. Parse into AST tree
3. Translate to constraint tree
4. Translate to IR tree
5. Linearize IR tree
6. Emit C / bytecode

```
00000     PROC      $1, $0
00001     TOKEN     $0, $0
00002     IEQ       %TT, $6
00003     CBT       2
00004     INVALID   $1, $0
00005     TOKEN     $0, $0
00006     IEQ       %TT, $7
00007     CBT       16
00008     MATCH     $0, $0
00009     MOVE      %R0, %M
00010     IEQ       %R0, $0
00011     CBT       6
00012     IEQ       %R0, $1
00013     CBT       6
00014     IEQ       %R0, $2
00015     CBT       6
00016     INVALID   $9, $0
00017     CONSUME   $0, $0
00018     BR        -13
00019     CALL      5
00020     BR        -15
00021     CALL      20
00022     BR        -17
00023     VALID     $0, $0
```

jvst: parsing

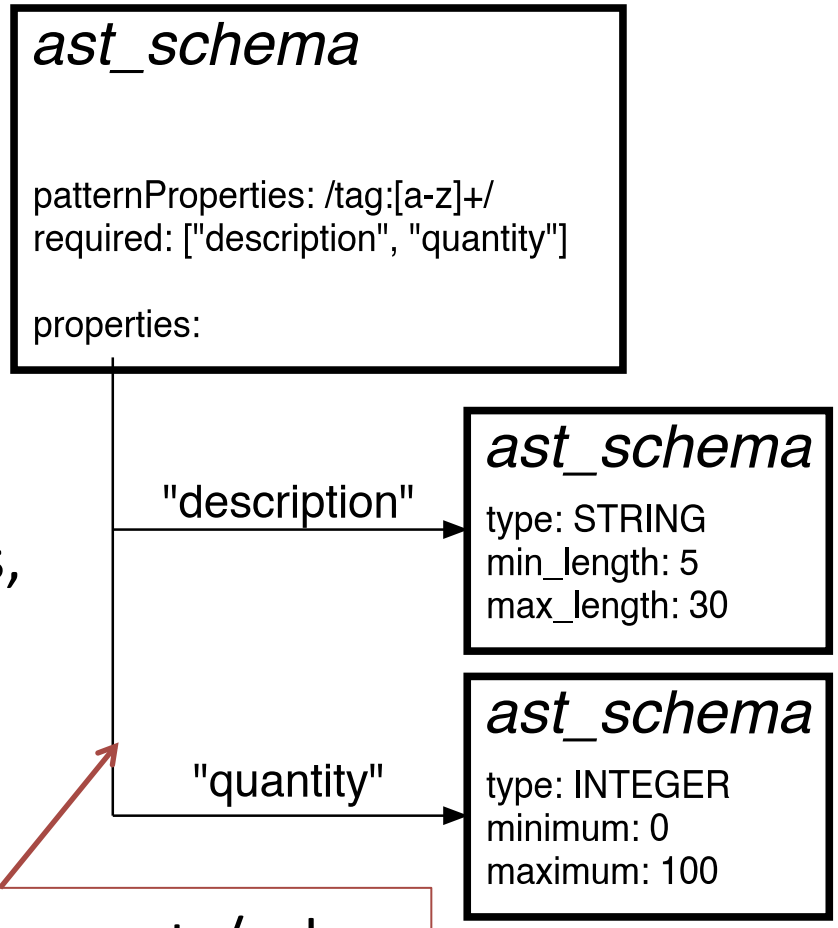
- SJP lexer: emits JSON tokens
- LL(1) parser generator
- AST: direct representation of the input
  - Each node is a sub-schema
  - Could round-trip back to JSON schema
- **General strategy**
  - Each stage does a simple thing
  - Introduce new nodes/trees when useful

jvst: the AST

Any non-object is implicitly valid

AST nodes do too much

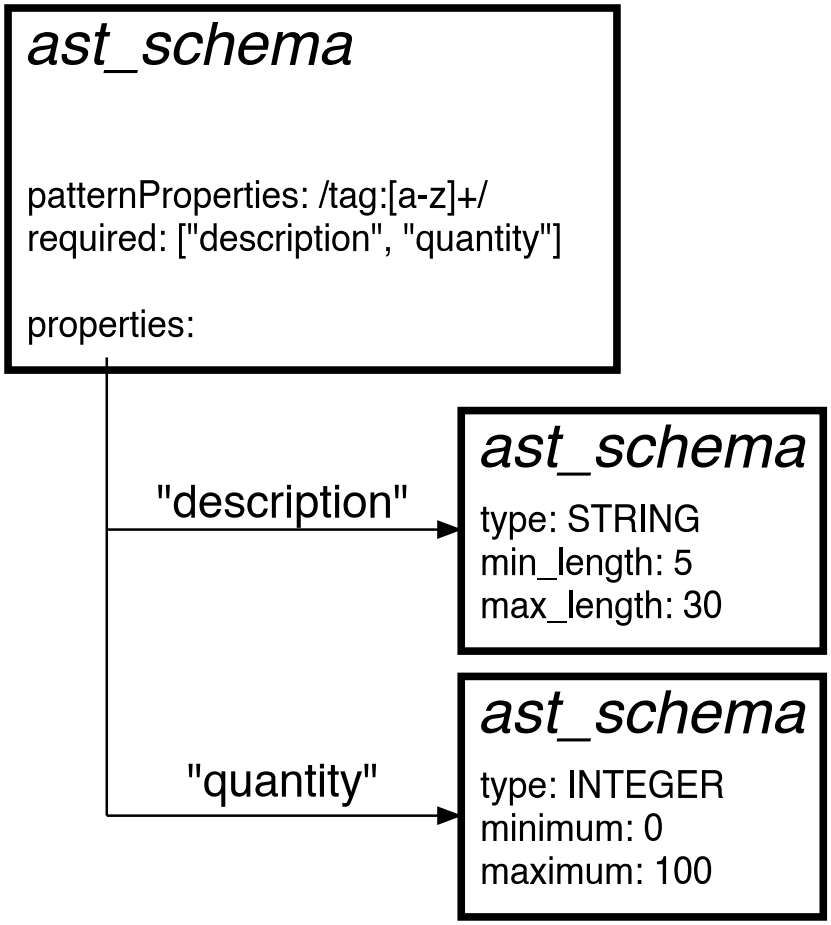
- Implicit valid/invalid
- Same node: several types
- Lots of redundant string matching
  - properties, patternProperties, propertyNames, required, dependencies
- **Hard to reason about**



jvst: the AST

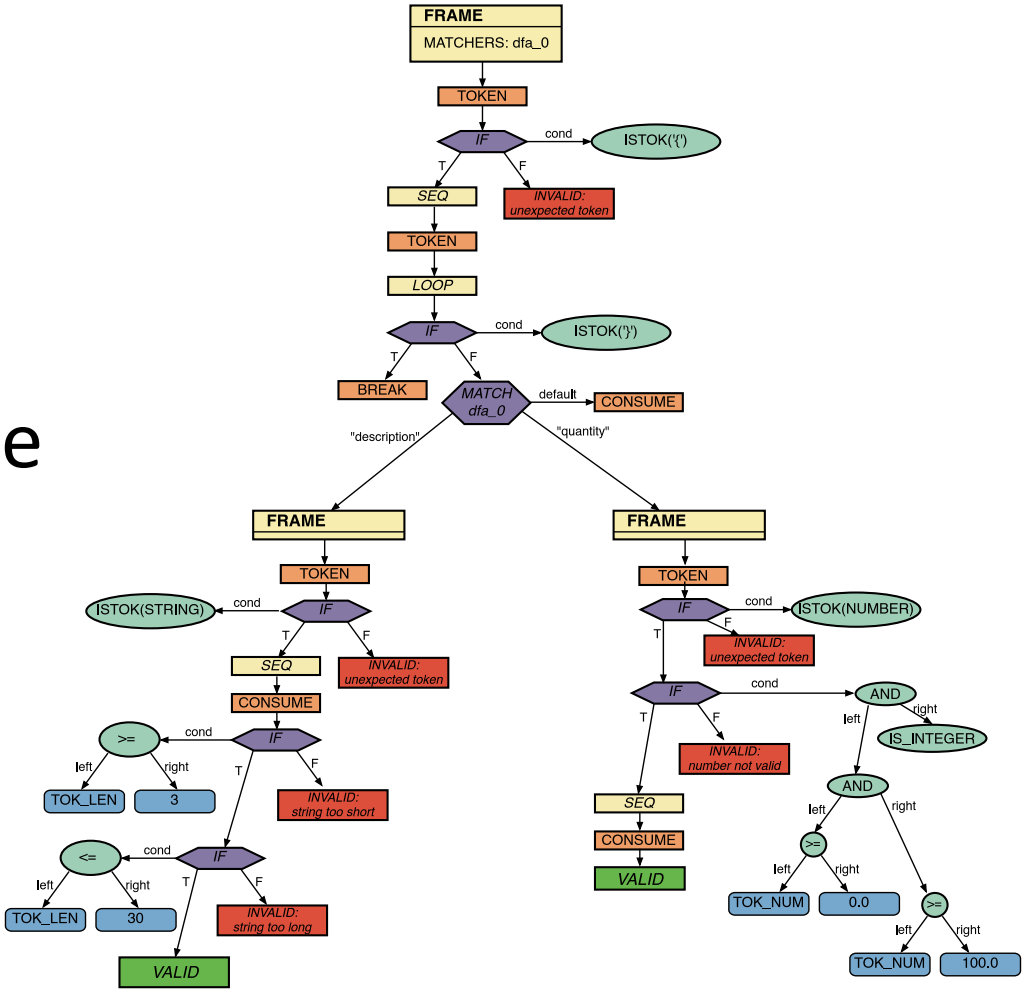
AST nodes do too much

- Too much is implicit
- Lots of redundancy



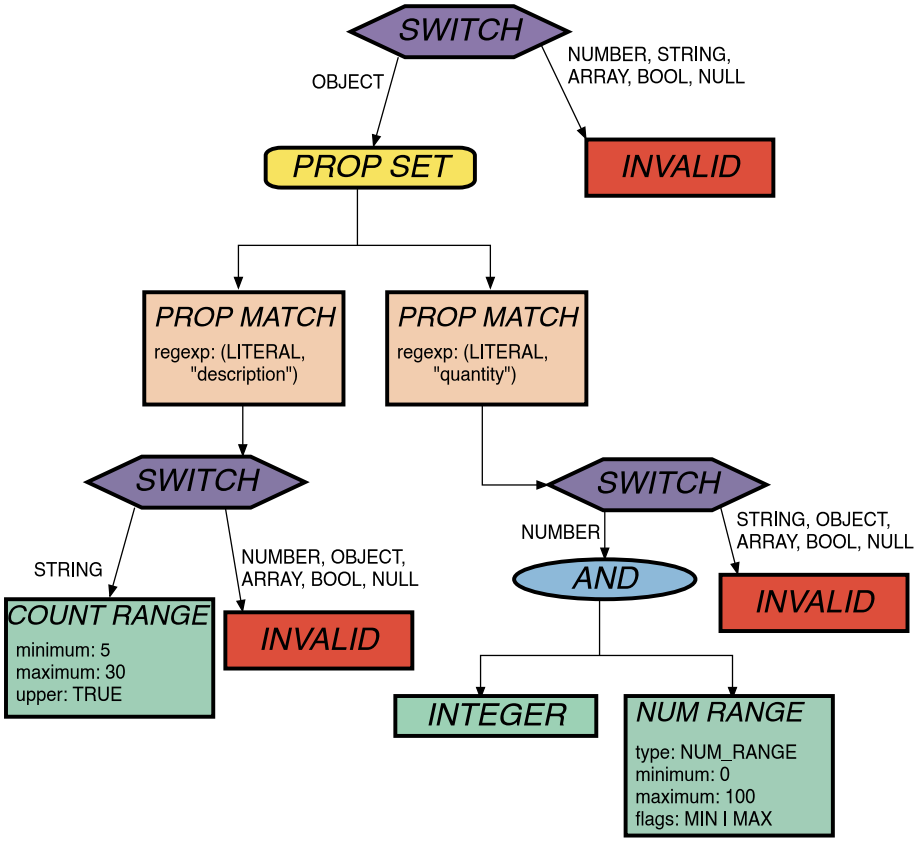
jvst: IR tree

- Constraint tree is declarative
  - SQL
- IR tree is imperative
  - Python / C
- Like a C AST

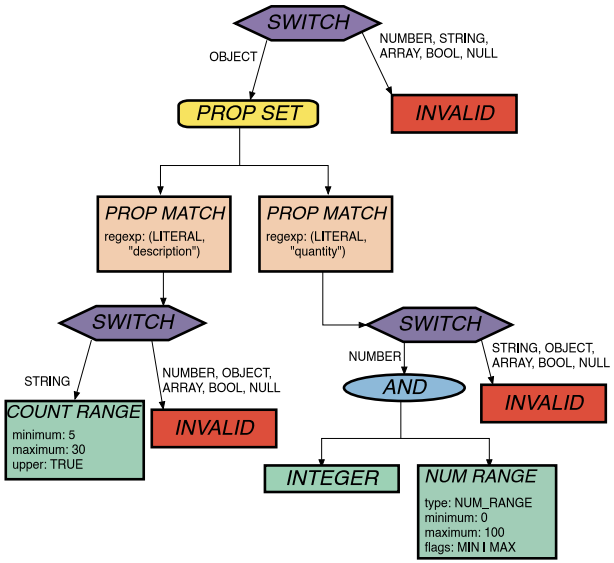


jvst: constraint tree

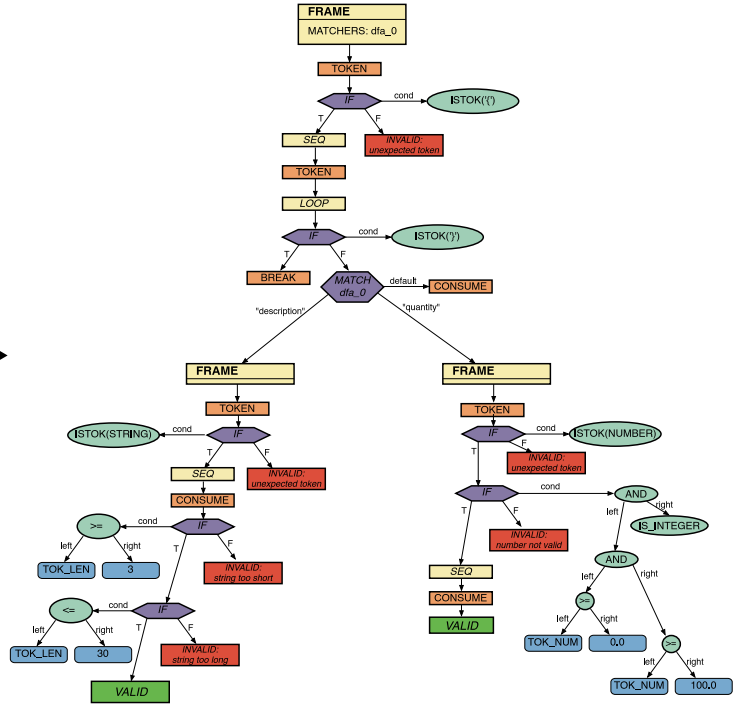
- Easy to reason about
  - Rewrite subtrees
  - Rewrite complex constraints
- Easy to simplify
  - Gather constraints on the same type
  - All string matching → DFAs
- Descriptive, not imperative



jvst: IR tree



Tree of
jvst_ir_stmt

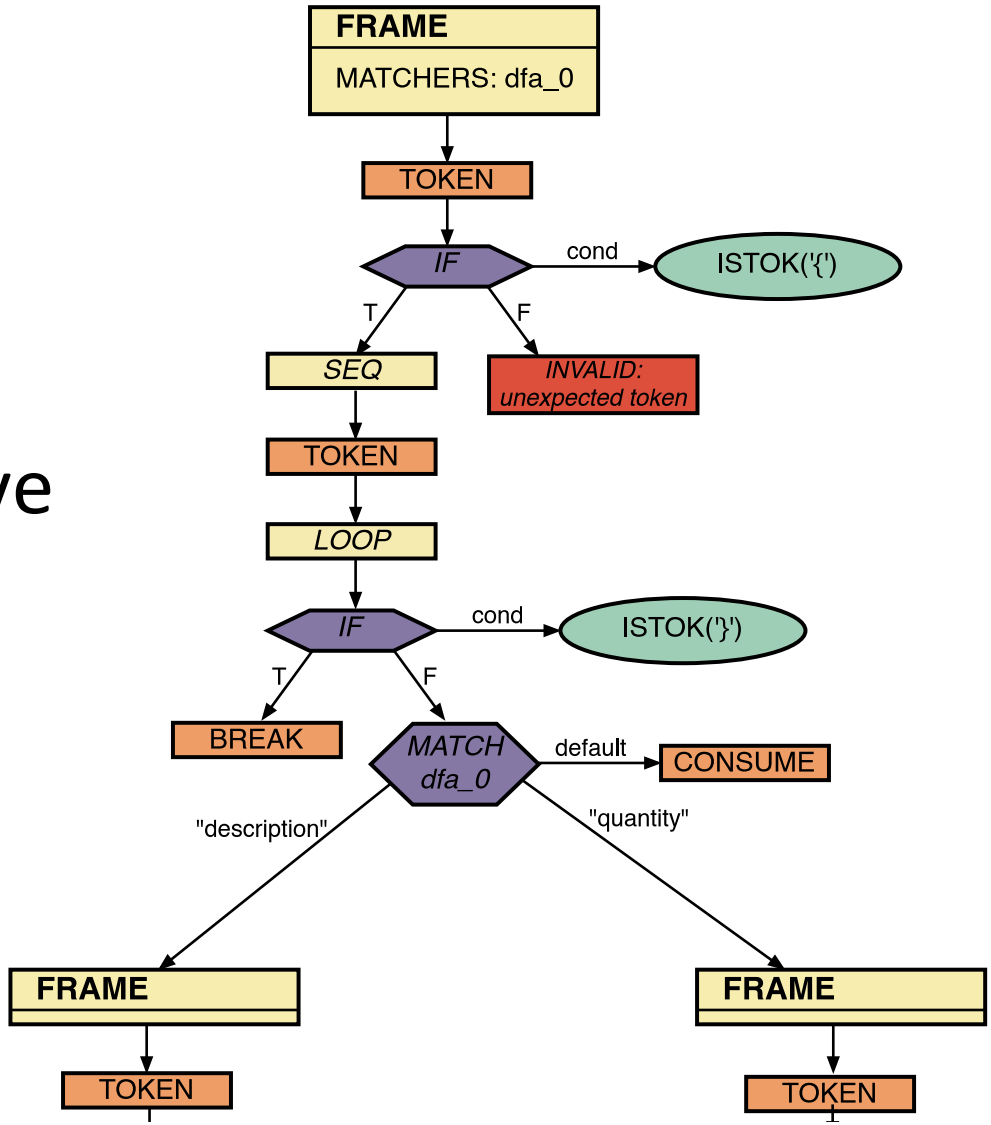


jvst_ir_stmt

- tagged union
- some nodes hold *jvst_ir_expr* subtrees

jvst: IR tree

- Constraint tree is declarative
  - SQL
- IR tree is imperative
  - Python / C
- Closer to code
  - Like a C AST



jvst: Linear IR

- Tree → Sequence of statements
- Gather in blocks
  - Single entry/exit
- Basic scheduling
- Some optimization
  - Jumps to jumps
  - Dead blocks

```
PROGRAM
FRAME(1):
      MATCHERS[MATCHER(0, "dfa_0")]
entry_0:          TOKEN,
                  CBRANCH(
                        ISTOK($OBJECT_BEG),
                        loop_4,
                        invalid_1_17)

invalid_1_17:     INVALID(1, "unexpected token")

loop_4:           TOKEN
                  CBRANCH(
                        ISTOK($OBJECT_END),
                        valid_15,
                        false_7)

false_7:          MOVE(ITEMP(0), EMATCH(0)),
                  CBRANCH(
                        EQ(ITEMP(0), 0),
                        M_9,
                        M_next_10)

M_next_10:        CBRANCH(
                        EQ(ITEMP(0), 1),
                        M_11,
                        M_next_12)
```

jvst: emit bytecode

- DFA:
  - Graphs → compact tables
- Constant pool
  - All floats
  - ints > 13 bits
- TODO: register allocation
  - Mainly to coalesce moves
- Emit opcodes
  - Fixed width: 32 bits
  - CBRANCH

```
00000     PROC      $1, $0
00001     TOKEN     $0, $0
00002     IEQ       %TT, $6
00003     CBT       2
00004     INVALID   $1, $0
00005     TOKEN     $0, $0
00006     IEQ       %TT, $7
00007     CBT       16
00008     MATCH     $0, $0
00009     MOVE      %R0, %M
00010     IEQ       %R0, $0
00011     CBT       6
00012     IEQ       %R0, $1
00013     CBT       6
00014     IEQ       %R0, $2
00015     CBT       6
00016     INVALID   $9, $0
00017     CONSUME   $0, $0
00018     BR        -13
00019     CALL      5
00020     BR        -15
00021     CALL      20
00022     BR        -17
00023     VALID     $0, $0
```