



Los Alamos
NATIONAL LABORATORY

Hank Wikle

Post-Bac Student

Programming and Runtime Environments Team

Software is for People

A Pavilion Case Study

What is Pavilion?

- YAML test specification
- Building and running tests in parallel
- Automatic result parsing and evaluation



Filtering Tests

- Test name
- Username
- Creation time
- Status (PASS/FAIL)
- Partition
- Etc.



The Grammar

expr: or_expr

or_expr: and_expr (OR and_expr)*

and_expr: not_expr (AND not_expr)*

not_expr: NOT? (comp_expr | special | paren_expr)

comp_expr: keyword (LT | GT | LT_EQ | GT_EQ | EQ | NOT_EQ) literal

paren_expr: "(" expr ")"

special: COMPLETE

| ALL_STARTED

| PASSED

| FAILED

| RESULT_ERROR

keyword: NAME

| STATE

| USER

| SYS_NAME

| HAS_STATE

| CREATED

| FINISHED

| PARTITION

| NODES

| NUM_NODES

| STARTED

?literal: INT

| GLOB

| TIMESTAMP

I. Usability

II. Readability

III. Extensibility

I. Usability

The Interface

```
name=mytest and  
sysname=batcave and  
(user=bwayne or user=apennyworth) and  
finished > 1 day and  
not FAILED
```

Boolean Operators

&

|

Boolean Operators

~~&~~

and

or

~~|~~

The Equals Sign

num_nodes=8

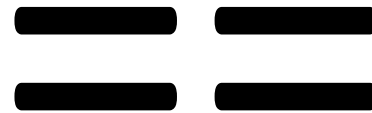
name=foo.*.bar

The Equals Sign



Assignment

Equality



II. Readability

Readability

```
class TestRun(TestAttributes):
    """The central pavilion test object. Handle saving, monitoring and running
    tests.

    **Test LifeCycle**
    1. Test Object is Created -- ``TestRun.__init__``

        1. Test id and directory (``working_dir/test_runs/0000001``) are created.
        2. Most test information files (config, status, etc) are created.
        3. Build script is created.
        4. Build hash is generated.
        5. Run script dry run generation is performed.

    2. Test is built. -- ``test.build()``
    3. Test is finalized. -- ``test.finalize()``

        1. Variables and config go through final resolution.
        2. Final run script is generated.

    4. Test is run. -- ``test.run()``
    5. Results are gathered. -- ``test.gather_results()``
```

✓  unittests / style (pull_request) Successful in 54s

Type Annotations

```
ThreeValue = Union[bool, None]
```

```
def or_expr(self, expr: List[Any]) -> ThreeValue:
    """Called on OR expressions (which may or may not actually
    contain the OR connective). Implements a 3-valued logic,
    where a value of None is only returned if all operands
    are None. Otherwise, the operator behaves as if it is
    a 2-valued OR operator and the None values have been removed
    from the list of operands."""

    bools = list(filter(lambda x: isinstance(x, bool), expr))

    if len(bools) > 0:
        return any(bools)
```


Economy of Code

```
def or_expr(self, expr: List[Any]) -> ThreeValue:
    """Called on OR expressions (which may or may not actually
    contain the OR connective). Implements a 3-valued logic,
    where a value of None is only returned if all operands
    are None. Otherwise, the operator behaves as if it is
    a 2-valued OR operator and the None values have been removed
    from the list of operands."""

    bools = list(filter(lambda x: isinstance(x, bool), expr))

    if len(bools) > 0:
        return any(bools)
```

III. Extensibility

Addition of Filter Properties

```
keyword: NAME  
        | STATE  
        | USER  
        | SYS_NAME  
        | HAS_STATE  
        | CREATED  
        | FINISHED  
        | PARTITION  
        | NODES  
        | NUM_NODES  
        | STARTED
```

```
NAME: "NAME"i  
STATE: "STATE"i  
USER: "USER"i  
SYS_NAME: "SYS_NAME"i  
HAS_STATE: "HAS_STATE"i  
CREATED: "CREATED"i  
FINISHED: "FINISHED"i  
PARTITION: "PARTITION"i  
NODES: "NODES"i  
NUM_NODES: "NUM_NODES"i  
STARTED: "STARTED"i
```

```
@validate_glob  
def _user(self) -> Optional[str]:  
    """Fetch the name of the user who ran the test or series."""  
  
    return self.attrs.get("user")
```

Addition of Filter Properties

```
keyword: NAME  
        | STATE  
        | USER  
        | SYS_NAME  
        | HAS_STATE  
        | CREATED  
        | FINISHED  
        | PARTITION  
        | NODES  
        | NUM_NODES  
        | STARTED
```

```
NAME: "NAME"i  
STATE: "STATE"i  
USER: "USER"i  
SYS_NAME: "SYS_NAME"i  
HAS_STATE: "HAS_STATE"i  
CREATED: "CREATED"i  
FINISHED: "FINISHED"i  
PARTITION: "PARTITION"i  
NODES: "NODES"i  
NUM_NODES: "NUM_NODES"i  
STARTED: "STARTED"i
```

```
@validate_glob  
def _user(self) -> Optional[str]:  
    """Fetch the name of the user who ran the test or series."""  
  
    return self.attrs.get("user")
```

Addition of Filter Properties

```
keyword: NAME  
        | STATE  
        | USER  
        | SYS_NAME  
        | HAS_STATE  
        | CREATED  
        | FINISHED  
        | PARTITION  
        | NODES  
        | NUM_NODES  
        | STARTED
```

```
NAME: "NAME"i  
STATE: "STATE"i  
USER: "USER"i  
SYS_NAME: "SYS_NAME"i  
HAS_STATE: "HAS_STATE"i  
CREATED: "CREATED"i  
FINISHED: "FINISHED"i  
PARTITION: "PARTITION"i  
NODES: "NODES"i  
NUM_NODES: "NUM_NODES"i  
STARTED: "STARTED"i
```

```
@validate_alob  
def _user(self) -> Optional[str]:  
    """Fetch the name of the user who ran the test or series."""  
  
    return self.attrs.get("user")
```

Addition of Filter Properties

```
keyword: NAME  
        | STATE  
        | USER  
        | SYS_NAME  
        | HAS_STATE  
        | CREATED  
        | FINISHED  
        | PARTITION  
        | NODES  
        | NUM_NODES  
        | STARTED
```

```
NAME: "NAME"i  
STATE: "STATE"i  
USER: "USER"i  
SYS_NAME: "SYS_NAME"i  
HAS_STATE: "HAS_STATE"i  
CREATED: "CREATED"i  
FINISHED: "FINISHED"i  
PARTITION: "PARTITION"i  
NODES: "NODES"i  
NUM_NODES: "NUM_NODES"i  
STARTED: "STARTED"i
```

```
@validate_glob  
def _user(self) -> Optional[str]:  
    """Fetch the name of the user who ran the test or series."""  
  
    return self.attrs.get("user")
```

Validators

```
def comp_glob(lval: str, comp: str, rval: str) -> bool:
    """Determine whether the provided glob (passed as the righthand
    value) matches the string (passed as the lefthandvalue). The match
    can be inverted by passing '!=' as the operator rather than '='.
    Uses case-insensitive matching."""

    lval = lval.upper() # fnmatch is case sensitive, despite docs
    rval = rval.upper()

    if comp == '=':
        return fnmatch(lval, rval)
    if comp == '!=':
        return not fnmatch(lval, rval)

    raise FilterParseError(f"Invalid comparator {comp} for (str, glob).")
```

```
validate_glob = make_validator(comp_glob)
```

Validators

```
def make_validator(comp_func: Callable[[T, str, T], bool],
                  rtype: Callable[[str], T] = identity
                  ) -> Callable[
    [Callable[[object], T]],
    Callable[[object, str, str], ThreeValue]]:
    """Make a decorator that validates a comparison expression, ensuring that its
    righthand operand is of type rtype, produces the lefthand operand by calling
    the decorated function (intended to be a method of FilterTransform), then
    compares the lefthand and righthand operands using comp_func, which also
    takes the expression's operator as an argument."""

    def validate(func: Callable[[object], T]) -> Callable[[object, str, str], ThreeValue]:

        def compare(self, comp: str, rval: str) -> ThreeValue:
            try:
                rval = rtype(rval)
            except ValueError:
                raise FilterParseError(f"Invalid value {rval} for function {rtype.__name__}.")

            lval = func(self)

            if lval is None:
                return None

            return comp_func(lval, comp, rval)

        return compare

    return validate
```


Interfaces

```
class SeriesInfoBase(Mapping):  
    """Shared base class for series info and test set info."""
```

```
def __getitem__(self, key: str) -> Any:  
    """Dictionary like access."""  
  
    if not key in self.list_attrs():  
        raise KeyError("Unknown key in SeriesInfo: {}".format(key))  
  
    return getattr(self, key)  
  
def __iter__(self) -> Iterator[str]:  
    return iter(self.list_attrs())  
  
def __len__(self) -> int:  
    return len(list(iter(self)))
```

```
class TestAttributes(Mapping):  
    """A object for accessing test attributes. TestRuns  
    inherit from this, but it can be used by itself to  
    access test information with less overhead.
```

```
def __getitem__(self, key: str) -> Any:  
    if key in iter(self):  
        return getattr(self, key)  
  
    raise KeyError(str(key))  
  
def __iter__(self) -> Iterator[str]:  
    attr_list = self.list_attrs() + self.LIST_ATTRS_EXCEPTIONS  
    attr_list.append('path')  
  
    return iter(attr_list)  
  
def __len__(self) -> int:  
    return len(list(iter(self)))
```

Why does this matter?

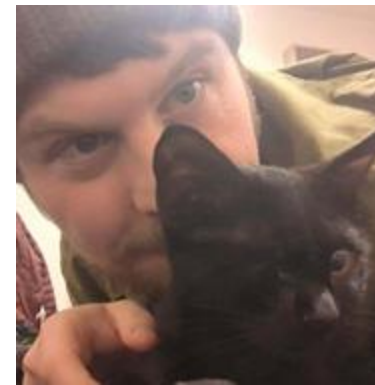
Thank You



Paul Ferrell



David DeBonis



Ty Goetsch



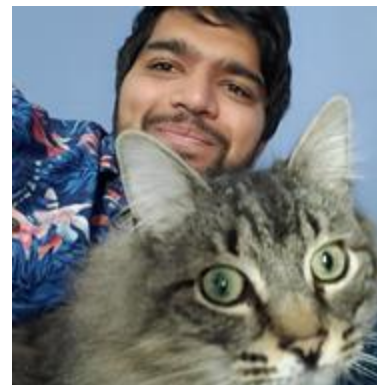
8/5/2024

Francine Lapid



Chris DeJager

LA-UR-24-28293



Shivam Mehta



Adam Good

Any Questions?

More on Validators

```
# Create all the validators
# Procedural abstraction FTW!
validate_int = make_validator(comp_num, int)
validate_glob = make_validator(comp_glob)
validate_glob_list = make_validator(comp_glob_list)
validate_str_list = make_validator(comp_str_list)
validate_datetime = make_validator(comp_num, parse_time)
validate_str = make_validator(comp_str)
validate_name_glob = make_validator(comp_name_glob)
```

More on Validators

```
@validate_str_list
def _has_state(self) -> Iterator[str]:
    """Fetch the state history of the test or series."""

    history = self.attrs.get("state_history", [])

    return map(lambda x: x.state, history)

@validate_datetime
def _created(self) -> Optional[datetime]:
    """Fetch the date and time at which the test or series
    was created."""

    created = self.attrs.get('created')

    if isinstance(created, float):
        return datetime.fromtimestamp(created)

    return created

@validate_str
def _state(self) -> Optional[str]:
    """Fetch the current state of the test or series."""

    state = self.attrs.get("state")

    if state is not None:
        return self.attrs.get("state").state
```

Example: Extensibility

```
class SeriesInfo(SeriesInfoBase):  
    """This class is a stop-gap. It's not meant to provide the same  
    functionality as test_run.TestAttributes, but a lazily evaluated set  
    of properties for a given series path. It should be replaced with  
    something like TestAttributes in the future."""
```