Parsing semi-structured text with Ansible

The :ref:`cli_parse <ansible_collections.ansible.utils.cli_parse_module>` module parses semi-structured data such as network configurations into structured data to allow programmatic use of the data from that device. You can pull information from a network device and update a CMDB in one playbook. Use cases include automated troubleshooting, creating dynamic documentation, updating IPAM (IP address management) tools and so on.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\user_guide\[ansible-devel][docs][docsite][rst][network][user_guide]cli_parsing.rst`, line 7); *backlink*

Unknown interpreted text role "ref".

Understanding the CLI parser

The ansible.utils collection version 1.0.0 or later includes the :ref:`cli_parse <ansible_collections.ansible.utils.cli_parse_module>` module that can run CLI commands and parse the semi-structured text output. You can use the `cli_parse` module on a device, host, or platform that only supports a command-line interface and the commands issued return semi-structured text. The `cli_parse` module can either run a CLI command on a device and return a parsed result or can simply parse any text document. The `cli_parse` module includes cli_parser plugins to interface with a variety of parsing engines.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\user_guide\[ansible-devel][docs][docsite][rst][network][user_guide]cli_parsing.rst`, line 17); *backlink*

Unknown interpreted text role "ref".

Why parse the text?

Parsing semi-structured data such as network configurations into structured data allows programmatic use of the data from that device. Use cases include automated troubleshooting, creating dynamic documentation, updating IPAM (IP address management) tools and so on. You may prefer to do this with Ansible natively to take advantage of native Ansible constructs such as:

- The `when` clause to conditionally run other tasks or roles
- The `assert` module to check configuration and operational state compliance
- The `template` module to generate reports about configuration and operational state information
- Templates and `command` or `config` modules to generate host, device, or platform commands or configuration
- The current platform `facts` modules to supplement native facts information

By parsing semi-structured text into Ansible native data structures, you can take full advantage of Ansible's network modules and plugins.

When not to parse the text

You should not parse semi-structured text when:

- The device, host, or platform has a RESTAPI and returns JSON.
- Existing Ansible facts modules already return the desired data.
- Ansible network resource modules exist for configuration management of the device and resource.

Parsing the CLI

The `cli_parse` module includes the following cli_parsing plugins:

`native`
: The native parsing engine built into Ansible and requires no addition python libraries

`xml`
: Convert XML to an Ansible native data structure

`textfsm`
: A python module which implements a template based state machine for parsing semi-formatted text

`ntc_templates`
: Predefined `textfsm` templates packages supporting a variety of platforms and commands

`ttp`
: A library for semi-structured text parsing using templates, with added capabilities to simplify the process

`pyats`
: Uses the parsers included with the Cisco Test Automation & Validation Solution

`jc`
: A python module that converts the output of dozens of popular Linux/UNIX/macOS/Windows commands and file types to python dictionaries or lists of dictionaries. Note: this filter plugin can be found in the `community.general` collection.

`json`
: Converts JSON output at the CLI to an Ansible native data structure

Although Ansible contains a number of plugins that can convert XML to Ansible native data structures, the `cli_parse` module runs the command on devices that return XML and returns the converted data in a single task.

Because `cli_parse` uses a plugin based architecture, it can use additional parsing engines from any Ansible collection.

Note

The `ansible.netcommon.native` and `ansible.utils.json` parsing engines are fully supported with a Red Hat Ansible Automation Platform subscription. Red Hat Ansible Automation Platform subscription support is limited to the use of the `ntc_templates`, pyATS, `textfsm`, `xmltodict`, public APIs as documented.

Parsing with the native parsing engine

The native parsing engine is included with the `cli_parse` module. It uses data captured using regular expressions to populate the parsed data structure. The native parsing engine requires a YAML template file to parse the command output.

Networking example

This example uses the output of a network device command and applies a native template to produce an output in Ansible structured data format.

The `show interface` command output from the network device looks as follows:

```
Ethernet1/1 is up
admin state is up, Dedicated Interface
  Hardware: 100/1000/10000 Ethernet, address: 5254.005a.f8bd (bia 5254.005a.f8bd)
  MTU 1500 bytes, BW 1000000 Kbit, DLY 10 usec
  reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, medium is broadcast
  Port mode is access
  full-duplex, auto-speed
  Beacon is turned off
  Auto-Negotiation is turned on  FEC mode is Auto
  Input flow-control is off, output flow-control is off
  Auto-mdix is turned off
  Switchport monitor is off
  EtherType is 0x8100
  EEE (efficient-ethernet) : n/a
  Last link flapped 4week(s) 6day(s)
  Last clearing of "show interface" counters never
<...>
```

Create the native template to match this output and store it as `templates/nxos_show_interface.yaml`:

```
---
- example: Ethernet1/1 is up
  getval: '(?P<name>\S+) is (?P<oper_state>\S+)'
  result:
    "{{ name }}":
      name: "{{ name }}"
      state:
        operating: "{{ oper_state }}"
  shared: true

- example: admin state is up, Dedicated Interface
  getval: 'admin state is (?P<admin_state>\S+),'
  result:
    "{{ name }}":
      name: "{{ name }}"
      state:
        admin: "{{ admin_state }}"

- example: "  Hardware: Ethernet, address: 5254.005a.f8b5 (bia 5254.005a.f8b5)"
```

```
  getval: '\s+Hardware: (?P<hardware>.*), address: (?P<mac>\S+)'
  result:
    "{{ name }}":
      hardware: "{{ hardware }}"
      mac_address: "{{ mac }}"
```

This native parser template is structured as a list of parsers, each containing the following key-value pairs:

- `example` - An example line of the text line to be parsed
- `getval` - A regular expression using named capture groups to store the extracted data
- `result` - A data tree, populated as a template, from the parsed data
- `shared` - (optional) The shared key makes the parsed values available to the rest of the parser entries until matched again.

The following example task uses `cli_parse` with the native parser and the example template above to parse the `show interface` command from a Cisco NXOS device:

```
- name: "Run command and parse with native"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.netcommon.native
    set_fact: interfaces
```

Taking a deeper dive into this task:

- The `command` option provides the command you want to run on the device or host. Alternately, you can provide text from a previous command with the `text` option instead.
- The `parser` option provides information specific to the parser engine.
- The `name` suboption provides the fully qualified collection name (FQCN) of the parsing engine (`ansible.netcommon.native`).
- The `cli_parse` module, by default, looks for the template in the templates directory as `{{ short_os }}_{{ command }}.yaml`.
    - The `short_os` in the template filename is derived from either the host `ansible_network_os` or `ansible_distribution`.
    - Spaces in the network or host command are replace with `_` in the `command` portion of the template filename. In this example, the `show interfaces` network CLI command becomes `show_interfaces` in the filename.

Note

`ansible.netcommon.native` parsing engine is fully supported with a Red Hat Ansible Automation Platform subscription.

Lastly in this task, the `set_fact` option sets the following `interfaces` fact for the device based on the now-structured data returned from `cli_parse`:

```
Ethernet1/1:
    hardware: 100/1000/10000 Ethernet
    mac_address: 5254.005a.f8bd
    name: Ethernet1/1
    state:
    admin: up
    operating: up
Ethernet1/10:
    hardware: 100/1000/10000 Ethernet
    mac_address: 5254.005a.f8c6
<...>
```

Linux example

You can also use the native parser to run commands and parse output from Linux hosts.

The output of a sample Linux command (`ip addr show`) looks as follows:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s31f6: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether x2:6a:64:9d:84:19 brd ff:ff:ff:ff:ff:ff
3: wlp2s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether x6:c2:44:f7:41:e0 brd ff:ff:ff:ff:ff:ff permaddr d8:f2:ca:99:5c:82
```

Create the native template to match this output and store it as `templates/fedora_ip_addr_show.yaml`:

```
---
- example: '1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000'
  getval: |
    (?x)                                                # free-spacing
    \d+:\s                                              # the interface index
    (?P<name>\S+):\s                                    # the name
    <(?P<properties>\S+)>                               # the properties
    \smtu\s(?P<mtu>\d+)                                 # the mtu
    .*                                                  # gunk
    state\s(?P<state>\S+)                               # the state of the interface
  result:
    "{{ name }}":
        name: "{{ name }}"
        loopback: "{{ 'LOOPBACK' in stats.split(',') }}"
        up: "{{ 'UP' in properties.split(',')  }}"
        carrier: "{{ not 'NO-CARRIER' in properties.split(',') }}"
        broadcast: "{{ 'BROADCAST' in properties.split(',') }}"
        multicast: "{{ 'MULTICAST' in properties.split(',') }}"
        state: "{{ state|lower() }}"
        mtu: "{{ mtu }}"
  shared: True

- example: 'inet 192.168.122.1/24 brd 192.168.122.255 scope global virbr0'
  getval: |
   (?x)                                                 # free-spacing
   \s+inet\s(?P<inet>([0-9]{1,3}\.){3}[0-9]{1,3})       # the ip address
   /(?P<bits>\d{1,2})                                   # the mask bits
  result:
    "{{ name }}":
        ip_address: "{{ inet }}"
        mask_bits: "{{ bits }}"
```

Note

The `shared` key in the parser template allows the interface name to be used in subsequent parser entries. The use of examples and free-spacing mode with the regular expressions makes the template easier to read.

The following example task uses `cli_parse` with the native parser and the example template above to parse the Linux output:

```
- name: Run command and parse
  ansible.utils.cli_parse:
    command: ip addr show
    parser:
      name: ansible.netcommon.native
    set_fact: interfaces
```

This task assumes you previously gathered facts to determine the `ansible_distribution` needed to locate the template. Alternately, you could provide the path in the `parser/template_path` option.

Lastly in this task, the `set_fact` option sets the following `interfaces` fact for the host, based on the now-structured data returned from `cli_parse`:

```
lo:
  broadcast: false
  carrier: true
  ip_address: 127.0.0.1
  mask_bits: 8
  mtu: 65536
  multicast: false
  name: lo
  state: unknown
  up: true
enp64s0u1:
  broadcast: true
  carrier: true
  ip_address: 192.168.86.83
  mask_bits: 24
  mtu: 1500
  multicast: true
  name: enp64s0u1
  state: up
  up: true
<...>
```

Parsing JSON

Although Ansible will natively convert serialized JSON to Ansible native data when recognized, you can also use the `cli_parse` module for this conversion.

Example task:

```
- name: "Run command and parse as json"
  ansible.utils.cli_parse:
    command: show interface | json
    parser:
      name: ansible.utils.json
    register: interfaces
```

Taking a deeper dive into this task:

- The `show interface | json` command is issued on the device.
- The output is set as the `interfaces` fact for the device.
- JSON support is provided primarily for playbook consistency.

Note

The use of `ansible.netcommon.json` is fully supported with a Red Hat Ansible Automation Platform subscription

Parsing with ntc_templates

The `ntc_templates` python library includes pre-defined `textfsm` templates for parsing a variety of network device commands output.

Example task:

```
- name: "Run command and parse with ntc_templates"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.netcommon.ntc_templates
    set_fact: interfaces
```

Taking a deeper dive into this task:

- The `ansible_network_os` of the device is converted to the ntc_template format `cisco_nxos`. Alternately, you can provide the `os` with the `parser/os` option instead.
- The `cisco_nxos_show_interface.textfsm` template, included with the `ntc_templates` package, parses the output.
- See the ntc_templates README for additional information about the `ntc_templates` python library.

Note

Red Hat Ansible Automation Platform subscription support is limited to the use of the `ntc_templates` public APIs as documented.

This task and and the predefined template sets the following fact as the `interfaces` fact for the host:

```
interfaces:
- address: 5254.005a.f8b5
  admin_state: up
  bandwidth: 1000000 Kbit
  bia: 5254.005a.f8b5
  delay: 10 usec
  description: ''
  duplex: full-duplex
  encapsulation: ARPA
  hardware_type: Ethernet
  input_errors: ''
  input_packets: ''
  interface: mgmt0
  ip_address: 192.168.101.14/24
  last_link_flapped: ''
  link_status: up
  mode: ''
  mtu: '1500'
  output_errors: ''
  output_packets: ''
  speed: 1000 Mb/s
- address: 5254.005a.f8bd
  admin_state: up
  bandwidth: 1000000 Kbit
  bia: 5254.005a.f8bd
  delay: 10 usec
```

Parsing with pyATS

`pyATS` is part of the Cisco Test Automation & Validation Solution. It includes many predefined parsers for a number of network platforms and commands. You can use the predefined parsers that are part of the `pyATS` package with the `cli_parse` module.

Example task:

```
- name: "Run command and parse with pyats"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.netcommon.pyats
    set_fact: interfaces
```

Taking a deeper dive into this task:

- The `cli_parse` modules converts the `ansible_network_os` automatically (in this example, `ansible_network_os` set to `cisco.nxos.nxos`, converts to `nxos` for pyATS. Alternately, you can set the OS with the `parser/os` option instead.
- Using a combination of the command and OS, the pyATS selects the following parser: https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/parsers/show%2520interface.
- The `cli_parse` module sets `cisco.ios.ios` to `iosxe` for pyATS. You can override this with the `parser/os` option.
- `cli_parse` only uses the predefined parsers in pyATS. See the pyATS documentation and the full list of pyATS included parsers.

Note

Red Hat Ansible Automation Platform subscription support is limited to the use of the pyATS public APIs as documented.

This task sets the following fact as the `interfaces` fact for the host:

```
mgmt0:
  admin_state: up
  auto_mdix: 'off'
  auto_negotiate: true
  bandwidth: 1000000
  counters:
    in_broadcast_pkts: 3
    in_multicast_pkts: 1652395
    in_octets: 556155103
    in_pkts: 2236713
    in_unicast_pkts: 584259
    rate:
      in_rate: 320
      in_rate_pkts: 0
      load_interval: 1
      out_rate: 48
      out_rate_pkts: 0
    rx: true
    tx: true
  delay: 10
  duplex_mode: full
  enabled: true
  encapsulations:
    encapsulation: arpa
  ethertype: '0x0000'
  ipv4:
    192.168.101.14/24:
      ip: 192.168.101.14
      prefix_length: '24'
  link_state: up
  <...>
```

Parsing with textfsm

`textfsm` is a Python module which implements a template-based state machine for parsing semi-formatted text.

The following sample``textfsm`` template is stored as `templates/nxos_show_interface.textfsm`

```
Value Required INTERFACE (\S+)
Value LINK_STATUS (.+?)
Value ADMIN_STATE (.+?)
Value HARDWARE_TYPE (.\*)
Value ADDRESS ([a-zA-Z0-9]+.[a-zA-Z0-9]+.[a-zA-Z0-9]+)
Value BIA ([a-zA-Z0-9]+.[a-zA-Z0-9]+.[a-zA-Z0-9]+)
Value DESCRIPTION (.\*)
Value IP_ADDRESS (\d+\.\d+\.\d+\.\d+\/\d+)
Value MTU (\d+)
Value MODE (\S+)
Value DUPLEX (.+duplex?)
Value SPEED (.+?)
Value INPUT_PACKETS (\d+)
Value OUTPUT_PACKETS (\d+)
Value INPUT_ERRORS (\d+)
Value OUTPUT_ERRORS (\d+)
Value BANDWIDTH (\d+\s+\w+)
Value DELAY (\d+\s+\w+)
Value ENCAPSULATION (\w+)
Value LAST_LINK_FLAPPED (.+?)

Start
  ^\S+\s+is.+ -> Continue.Record
  ^${INTERFACE}\s+is\s+${LINK_STATUS},\sline\sprotocol\sis\s${ADMIN_STATE}$$
  ^${INTERFACE}\s+is\s+${LINK_STATUS}$$
  ^admin\s+state\s+is\s+${ADMIN_STATE},
  ^\s+Hardware(:|\s+is)\s+${HARDWARE_TYPE},\s+address(:|\s+is)\s+${ADDRESS}(.*bia\s+${BIA})*
  ^\s+Description:\s+${DESCRIPTION}
  ^\s+Internet\s+Address\s+is\s+${IP_ADDRESS}
  ^\s+Port\s+mode\s+is\s+${MODE}
  ^\s+${DUPLEX}, ${SPEED}(,|$$)
  ^\s+MTU\s+${MTU}.\*BW\s+${BANDWIDTH}.\*DLY\s+${DELAY}
  ^\s+Encapsulation\s+${ENCAPSULATION}
  ^\s+${INPUT_PACKETS}\s+input\s+packets\s+\d+\s+bytes\s\*$$
  ^\s+${INPUT_ERRORS}\s+input\s+error\s+\d+\s+short\s+frame\s+\d+\s+overrun\s+\d+\s+underrun\s+\d+\s+ignored
  ^\s+${OUTPUT_PACKETS}\s+output\s+packets\s+\d+\s+bytes\s\*$$
  ^\s+${OUTPUT_ERRORS}\s+output\s+error\s+\d+\s+collision\s+\d+\s+deferred\s+\d+\s+late\s+collision\s\*$$
  ^\s+Last\s+link\s+flapped\s+${LAST_LINK_FLAPPED}\s\*$$
```

The following task uses the example template for `textfsm` with the `cli_parse` module.

```
- name: "Run command and parse with textfsm"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.utils.textfsm
    set_fact: interfaces
```

Taking a deeper dive into this task:

- The `ansible_network_os` for the device (`cisco.nxos.nxos`) is converted to `nxos`. Alternately you can provide the OS in the `parser/os` option instead.
- The textfsm template name defaulted to `templates/nxos_show_interface.textfsm` using a combination of the OS and command run. Alternately you can override the generated template path with the `parser/template_path` option.
- See the textfsm README for details.
- `textfsm` was previously made available as a filter plugin. Ansible users should transition to the `cli_parse` module.

> **Note**
>
> Red Hat Ansible Automation Platform subscription support is limited to the use of the `textfsm` public APIs as documented.

This task sets the following fact as the `interfaces` fact for the host:

```
- ADDRESS: X254.005a.f8b5
  ADMIN_STATE: up
  BANDWIDTH: 1000000 Kbit
  BIA: X254.005a.f8b5
  DELAY: 10 usec
  DESCRIPTION: ''
  DUPLEX: full-duplex
  ENCAPSULATION: ARPA
  HARDWARE_TYPE: Ethernet
  INPUT_ERRORS: ''
  INPUT_PACKETS: ''
  INTERFACE: mgmt0
  IP_ADDRESS: 192.168.101.14/24
  LAST_LINK_FLAPPED: ''
  LINK_STATUS: up
  MODE: ''
  MTU: '1500'
  OUTPUT_ERRORS: ''
  OUTPUT_PACKETS: ''
  SPEED: 1000 Mb/s
- ADDRESS: X254.005a.f8bd
  ADMIN_STATE: up
  BANDWIDTH: 1000000 Kbit
  BIA: X254.005a.f8bd
```

Parsing with TTP

TTP is a Python library for semi-structured text parsing using templates. TTP uses a jinja-like syntax to limit the need for regular expressions. Users familiar with jinja templating may find the TTP template syntax familiar.

The following is an example TTP template stored as `templates/nxos_show_interface.ttp`:

```
{{ interface }} is {{ state }}
admin state is {{ admin_state }}{{ ignore(".\*") }}
```

The following task uses this template to parse the `show interface` command output:

```
- name: "Run command and parse with ttp"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.utils.ttp
    set_fact: interfaces
```

Taking a deeper dive in this task:

- The default template path `templates/nxos_show_interface.ttp` was generated using the `ansible_network_os` for the host and `command` provided.
- TTP supports several additional variables that will be passed to the parser. These include:
  - `parser/vars/ttp_init` - Additional parameter passed when the parser is initialized.
  - `parser/vars/ttp_results` - Additional parameters used to influence the parser output.
  - `parser/vars/ttp_vars` - Additional variables made available in the template.
- See the TTP documentation for details.

The task sets the follow fact as the `interfaces` fact for the host:

```
- admin_state: up,
  interface: mgmt0
  state: up
- admin_state: up,
  interface: Ethernet1/1
  state: up
- admin_state: up,
  interface: Ethernet1/2
  state: up
```

Parsing with JC

JC is a python library that converts the output of dozens of common Linux/UNIX/macOS/Windows command-line tools and file types to python dictionaries or lists of dictionaries for easier parsing. JC is available as a filter plugin in the `community.general` collection.

The following is an example using JC to parse the output of the `dig` command:

```
- name: "Run dig command and parse with jc"
  hosts: ubuntu
  tasks:
  - shell: dig example.com
    register: result
  - set_fact:
      myvar: "{{ result.stdout | community.general.jc('dig') }}"
  - debug:
      msg: "The IP is: {{ myvar[0].answer[0].data }}"
```

- The JC project and documentation can be found here.
- See this blog entry for more information.

Converting XML

Although Ansible contains a number of plugins that can convert XML to Ansible native data structures, the `cli_parse` module runs the command on devices that return XML and returns the converted data in a single task.

This example task runs the `show interface` command and parses the output as XML:

```
- name: "Run command and parse as xml"
    ansible.utils.cli_parse:
      command: show interface | xml
      parser:
        name: ansible.utils.xml
  set_fact: interfaces
```

> **Note**
>
> Red Hat Ansible Automation Platform subscription support is limited to the use of the `xmltodict` public APIs as documented.

This task sets the `interfaces` fact for the host based on this returned output:

```
nf:rpc-reply:
  '@xmlns': http://www.cisco.com/nxos:1.0:if_manager
  '@xmlns:nf': urn:ietf:params:xml:ns:netconf:base:1.0
  nf:data:
    show:
      interface:
        __XML__OPT_Cmd_show_interface_quick:
          __XML__OPT_Cmd_show_interface___readonly__:
            __readonly__:
              TABLE_interface:
                ROW_interface:
                - admin_state: up
                  encapsulation: ARPA
                  eth_autoneg: 'on'
                  eth_bia_addr: x254.005a.f8b5
                  eth_bw: '1000000'
```

Advanced use cases

The `cli_parse` module supports several features to support more complex uses cases.

Provide a full template path

Use the `template_path` option to override the default template path in the task:

```
- name: "Run command and parse with native"
  ansible.utils.cli_parse:
    command: show interface
    parser:
      name: ansible.netcommon.native
      template_path: /home/user/templates/filename.yaml
```

Provide command to parser different than the command run

Use the `command` suboption for the `parser` to configure the command the parser expects if it is different from the command `cli_parse` runs:

```
- name: "Run command and parse with native"
  ansible.utils.cli_parse:
    command: sho int
    parser:
      name: ansible.netcommon.native
    command: show interface
```

Provide a custom OS value

Use the `os` suboption to the parser to directly set the OS instead of using `ansible_network_os` or `ansible_distribution` to generate the template path or with the specified parser engine:

```
- name: Use ios instead of iosxe for pyats
  ansible.utils.cli_parse:
    command: show something
    parser:
      name: ansible.netcommon.pyats
      os: ios

- name: Use linux instead of fedora from ansible_distribution
  ansible.utils.cli_parse:
    command: ps -ef
    parser:
      name: ansible.netcommon.native
      os: linux
```

Parse existing text

Use the `text` option instead of `command` to parse text collected earlier in the playbook.

```
# using /home/user/templates/filename.yaml
- name: "Parse text from previous task"
  ansible.utils.cli_parse:
    text: "{{ output['stdout'] }}"
    parser:
      name: ansible.netcommon.native
      template_path: /home/user/templates/filename.yaml

 # using /home/user/templates/filename.yaml
- name: "Parse text from file"
  ansible.utils.cli_parse:
    text: "{{ lookup('file', 'path/to/file.txt') }}"
    parser:
      name: ansible.netcommon.native
      template_path: /home/user/templates/filename.yaml

# using templates/nxos_show_version.yaml
- name: "Parse text from previous task"
  ansible.utils.cli_parse:
    text: "{{ sho_version['stdout'] }}"
    parser:
      name: ansible.netcommon.native
      os: nxos
      command: show version
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\user_guide\[ansible-devel][docs][docsite][rst][network][user_guide]cli_parsing.rst`, line 742)

Unknown directive type "seealso".

```
.. seealso::

  * :ref:`develop_cli_parse_plugins`
```