bincfg.normalization package

This subpackage provides classes to tokenize and normalize assembly lines, as well as the ability to easily create new tokenization/normalization methods.

This library currently supports the following architectures:

- x86/x86_64

And disassembler output from the following binary analysis tools:

- Rose https://github.com/rose-compiler/rose
- Ghidra https://github.com/NationalSecurityAgency/ghidra

`Normalizer` classes should all inherit from `BaseNormalizer`. By default, `BaseNormalizer` instances will perform and 'unnormalized' normalization: only removing extraneous information/spacing. Various built-in normalizers will perform different levels of normalization, and new classes can be made inheriting from `BaseNormalizer` which can override parent methods for different functionality.

An example of using a default `BaseNormalizer` on some x86_64 assembly:

```
from bincfg.normalization import BaseNormalizer

asm_lines = [
   '0x00402cdd: add    rsp, 0x08',
   '0x00402cf0: push   qword ds:[rip + 0x0000000000252312<absolute=0x0000000000655008>]',
   'CALL   0x0000000000403360'
]
normalizer = BaseNormalizer()

for line in asm_lines:
   print(normalizer.normalize(line))
```

Which would give the output:

```
>>> add rsp 8
>>> push qword [ rip + 2433810 ]
>>> call 4207456
```

The `BaseNormalizer` class by default does some simple cleaning while keeping all of the necessary information for the assembly line itself. For example: removing memory addresses of the instruction itself if it exists, converting all values to decimal, removing extra whitespace/commas, removing segment identifiers, etc.

This process is split into two main parts: *tokenization*, and *normalization*.

Tokenization

`Tokenizer` classes should likely inherit from or modify the `BaseTokenizer` class. When called, this class will tokenize incoming strings so that they can be later normalized. It is unlikely you would ever need to build a new tokenizer class as `BaseTokenizer` can be easily modified when instantiated to change the tokenizer behavior.

The tokenization process uses python's `re` module to perform tokenization, converting strings into streams of (token_name, token_string) tuples. For more information on how to use regex to create tokenizers, see: https://docs.python.org/3/library/re.html#writing-a-tokenizer

If you wish to modify the tokens that `BaseTokenizer` should identify, or the order it should identify them in, you can pass a list of (token_name: str, token_regex: str) tuples to `BaseTokenizer` on initialization. The default list of tokens should work for virtually any use case of the supported architectures, but it you may be required to modify the list of tokens should you wish to add in new supported architectures.

Normalization

`Normalizer` classes will normalize incoming strings. They do this by first tokenizing the strings (using either a user-defined or default tokenizer), then normalizing that stream of (token_name, token_string) tuples into strings.

Normalization has two possible *Tokenization Levels* for the incoming strings:

- 'op': opcode/operand level tokenization. Each individual opcode/operand gets normalized into its own token

- 'instruction': instruction level tokenization. Each instruction line gets normalized into a single token, with all opcodes/operands in that instruction joined together, separated by some separator string (defaults to ' ' for `BaseNormalizer`, and '_' for all other normalizers)

This library has a few built-in normalization methods based on literature:

- InnerEye: https://arxiv.org/pdf/1808.04706.pdf
- Deep Bin Diff: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24311-paper.pdf
- SAFE: https://github.com/gadiluna/SAFE
- Deep Semantic: https://arxiv.org/abs/2106.05478

This module also provides a `normalize_cfg_data()` function to normalize CFG data.

Custom Normalizers

Creating custom normalizers is quite simple. In fact, multiple of the built-in normalization techniques are as simple as a few lines of code:

```
class InnerEyeNormalizer(BaseNormalizer):
   DEFAULT_TOKENIZATION_LEVEL = TokenizationLevel.INSTRUCTION
   handle_immediate = return_immstr(include_negative=True)
   handle_memory_size = ignore
   handle_function_call = replace_function_call_immediate(FUNCTION_CALL_STR)
```

Custom normalizers should inherit from `BaseNormalizer`, and override parent methods to alter functionality. Most methods do exactly as they say, "handling" the tokens in their names:

- *handle_opcode()*
- *handle_memory_size()*
- *handle_register()*
- *handle_immediate()*
- *handle_memory_expression()*
- *handle_rose_info()*
- *handle_ignored()*
- *handle_mismatch()*

There are some handlers that have slightly different functionality:

- *handle_newline()*: this gets called after each full string has been parsed, or a new line character was found, indicating the end of a single assembly instruction. The full instruction will then be parsed, modified if necessary, specific opcodes handled, and converted into the final string (or list of strings if using 'op' tokenization level).
- *handle_instruction()*: this gets called by *handle_newline()*. It will parse the full instruction, checking for any specifc opcodes that need to be handled. This method does not do any other cleaning/converting of the instruction.

Specific opcodes can be handled differently after the full line has been parsed. The *register_opcode_handler()* function allows you to pass in a string regular expression to identify the opcodes to handle, and a function to handle those opcodes. There are also a few built-in opcode handler functions:

- *handle_jump()*: handles jump instructions
- *handle_call()*: handles call instructions
- 'nop' instructions: all 'nop' instructions will have everything stripped from them except the 'nop' opcode itself, since there is often a large amount of useless/extraneous information alongside those filler instructions

Finally, one can add in behavior for brand new token types using the *handle_unknown_token()* method, which will have passed to it the token_name and token_string whenever an unknown token_name is found. This way, you need not create an entirely new `Normalizer` class, and can still use `BaseNormalizer` as a parent, if you wish to add in new token types to parse.

For info on method signatures/expected return values, see their documentation below.

As shown above, you need only set the handler to the desired function to change behavior. This can be done either when building the class definition, or during the __init__ call.

There are multiple utility functions defined under *bincfg.normalization.norm_utils* that can be used to set the handlers above to different common behaviors without having to implement those functions yourself.

One may also set the *DEFAULT_TOKENIZATION_LEVEL* attribute on the class definition/instances to change what the default tokenization level behavior will be.

Extra info

- Currently, normalizers cannot be pickled. Instead, the `dill` package may be used to serialize them. This happens by default whenever using any of the *.save* or *.load* methods. This may change in the future if I can figure out ways to make them pickle-able without removing any capabilities.
- If you wish to pickle cfg objects without using `dill`, you can simply set their *.normalizer* attribute to None (or some other pickleable information about the normalizer being used), then pickle them.

Submodules

bincfg.normalization.base_normalizer module

Classes for normalizing assembly instructions.

class `bincfg.normalization.base_normalizer.`**`BaseNormalizer`**(**args*, ***kwargs*)

Bases: `object`

A base class for a normalization method.

Performs an 'unnormalized' normalization, removing what is likely extraneous information, and providing a base class for other normalization methods to inherit from.

Parameters:

- **tokenizer** (*Optional[Tokenizer]*) – the tokenizer to use, or None to use the default BaseTokenizer
- **token_sep** (*Optional[str]*) – the string to use to separate each token in returned instruction lines. Only used if tokenization_level is 'instruction'. If None, then a default value will be used (' ' for unnormalized using BaseNormalizer(), '_' for everything else)
- **tokenization_level** (*Optional[Union[TokenizationLevel, str]]*) –

  the tokenization level to use for return values. Can be a string, or a `TokenizationLevel` type. Strings can be:

  - 'op': tokenized at the opcode/operand level. Will insert a 'INSTRUCTION_START' token at the beginning of each instruction line
  - 'inst'/'instruction': tokenized at the instruction level. All tokens in each instruction line are joined together using token_sep to construct the final token
  - 'auto': pick the default value for this normalization technique
- **anonymize_tokens** (*bool*) – if True, then tokens will be annonymized by taking their 4-byte shake_128 hash. Why does this exist? Bureaucracy.

`DEFAULT_TOKENIZATION_LEVEL` = *['inst', 'instruction', 'line']*

The default tokenization level used for this normalizer

`handle_function_call`(*idx*, *line*, *cfg=None*, *block=None*)

Handles function calls. Defaults to returning raw call values

This is an opcode handler. It should modify the list of token tuples `line` in-place, then return the integer index in `line` of the last token that has been 'handled' by this function call.

Parameters:

- **idx** (*int*) – the index in `line` of the 'call' opcode
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **cfg** (*Optional[Union[CFG, MemCFG]], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Optional[Union[CFGBasicBlock, int]], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.

Returns: index in line of last handled token

Return type: int

`handle_ignored`(*name*, *token*, *line*, *sentence*)

Handles ignored tokens. Defaults to doing nothing

EG: spacing, commas, instruction memory address, etc.

Parameters:

- **name** (*str*) – the name of this token
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

`handle_immediate`(*token*, *line*, *sentence*)

Handles an immediate value. Defaults to converting into decimal

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Returns: the token in decimal

Return type: str

`handle_instruction`(*line*, *cfg=None*, *block=None*)

Handles an entire instruction once reaching a new line.

Allows for extra manipulations like checking call/jump destinations, etc. If nothing is returned, then it is assumed line itself has been edited.

Parameters:
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **cfg** (*Optional[Union[CFG, MemCFG]], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Optional[Union[CFGBasicBlock, int]], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.

`handle_jump`(*idx*, *line*, *cfg=None*, *block=None*)

Handles jumps. Defaults to returning raw jump values

This is an opcode handler. It should modify the list of token tuples `line` in-place, then return the integer index in `line` of the last token that has been 'handled' by this function call.

Parameters:
- **idx** (*int*) – the index in `line` of the 'jump' opcode
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **cfg** (*Optional[Union[CFG, MemCFG]], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Optional[Union[CFGBasicBlock, int]], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.

Returns: index in line of last handled token

Return type: int

`handle_memory_expression`(*memory_start*, *token*, *line*, *sentence*)

Handles memory expressions. Defaults to doing nothing special

Parameters:
- **memory_start** (*int*) – integer index in line where the full memory expression starts. The full memory expression
- **line[memory_start:]** (*would then be the list of tokens*) –
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

`handle_memory_size`(*token*, *line*, *sentence*)

Handles a memory size. Defaults to returning the raw memory size

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Returns: the original token

Return type: str

`handle_mismatch`(*name*, *token*, *line*, *sentence*)

What to do when the normalizaion method finds a token mismatch (in case they were ignored in the tokenizer)

Defaults to raising a TokenMismatchError()

Parameters:
- **name** (*str*) – the name of this token
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Raises: **TokenMismatchError** – by default

handle_newline(*token*, *line*, *sentence*, *cfg=None*, *block=None*)

Handles a newline token depending on what this normalizer's tokenization_level is

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*str*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*
- **cfg** (*Optional[Union[CFG, MemCFG]], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Optional[Union[CFGBasicBlock, int]], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.

Raises: **NotImplementedError** – If a TokenizationLevel was added but not implemented here

handle_opcode(*token*, *line*, *sentence*)

Handles an opcode. Defaults to returning the raw opcode

NOTE: This should only be used to determine how all opcode strings are handled. For how to handle specific opcodes to give them different behaviors, see `register_opcode_handler()`

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Returns: the original token

Return type: str

handle_prefix(*token*, *line*, *sentence*)

Handles an instruction prefix. Defaults to returning the original prefix

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Returns: the original prefix

Return type: str

handle_register(*token*, *line*, *sentence*)

Handles a register. Defaults to returning the raw register name

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Returns: the original token

Return type: str

handle_rose_info(*token*, *line*, *sentence*)

Checks to see if the rose info is telling us an immediate value is negative, otherwise ignores it

Parameters:
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

`handle_unknown_token`(*name*, *token*, *line*, *sentence*)

Handles an unknown token. Currently just raises a TypeError

Can be overridden in subclasses to add new token types

Parameters:
- **name** (*str*) – the name of this token
- **token** (*str*) – the current string token
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **sentence** (*List[str]*) – the current sentence, a list of strings. These will be either full assembly instructions if *tokenization_level='instuction'*, or single tokens with a separator between each assembly line if *tokenization_level='op'*

Raises: **TypeError** – by default

`hash_token`(*token*)

Hashes tokens during annonymization

By default, converts each individual token into its 4-byte shake_128 hash

Parameters: **token** (*str*) – the string token to hash

Returns: the 4-byte shake_128 hash of the given token

Return type: str

`normalize`(**strings*, *cfg=None*, *block=None*, *enforce_asm_rules=None*, *newline_tup=('newline', '\n')*, *match_instruction_address=True*)

Normalizes the given iterable of strings.

Parameters:
- **strings** (*str*) – arbitrary number of strings to normalize
- **cfg** (*Union[CFG, MemCFG], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Union[CFGBasicBlock, int], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **enforce_asm_rules** (*Optional[bool]*) – if True, then extra processing and checks will be done to make sure the tokenized assembly language matches the rules of assembly. See self.check_assembly_rules() for more info. If False, these checks aren't done and bad assembly could make its way through without error, but should be noticeably faster. If None, will use the default value. The default value starts as False at the beginning of program execution but can be modified using `_set_default_enforce_asm_rules()`
- **newline_tup** (*Tuple[str, str], optional*) – the tuple to insert inbetween each passed string, or None to not insert anything. Defaults to DEFAULT_TOKENIZER.DEFAULT_NEWLINE_TUPLE.
- **match_instruction_address** (*bool, optional*) – if True, will assume there will be an instruction address at the start of the string. This only has an effect on ghidra-like instruction addresses where that address could be interpreted as either an immediate, or an instruction address. If True, then any immediates found at the start of a line will be assumed to be instruction addresses instead of immediates. If False, then instruction addresses can still be matched, but they must end with a colon ':', otherwise they will be considered immediates. Defaults to True.

Raises: **TokenMismatchError** – on a bad branch prediction string

Returns: a list of normalized string instruction lines

Return type: List[str]

`register_opcode_handler`(*op_regex*, *func_or_str_name*)

Registers an opcode handler for this normalizer

Adds the given *op_regex* as an opcode to handle during self._handle_instruction() along with the given function to call with token/cfg arguments. *op_regex* can be either a compiled regex expression, or a string which will be compiled into a regex expression. *func_or_str_name* can either be a callable, or a string. If it's a string, then that attribute will be looked up on this normalizer dynamically to find the function to use.

Notes for registering opcode handlers:

1. passing instance method functions converts them to strings automatically
2. passing lambda's or inner functions (not at global scope) would not be able to be pickled

3. opcodes will be matched in order starting with ‘nop’, ‘call’, and ‘j.*’, then all in order of those passed to *register_opcode_handler()*

Parameters:
- **op_regex** (*Union[str, Pattern]*) – a string or compiled regex
- **func_or_str_name** (*Union[Callable, str]*) – the function to call with token/cfg arguments when an opcode matches op_regex, or a string name of a callable attribute of this normalizer to be looked up dynamically

Raises: **TypeError** – Bad *func_or_str_name* type

token_sep = *None*
The separator string used for this normalizer

Will default to ‘ ‘ for `BaseNormalizer`, and ‘_’ for all other normalizers.

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

tokenize(**strings*, *enforce_asm_rules=None*, *newline_tup=('newline', '\n')*, *match_instruction_address=True*)
Tokenizes the given strings using this normalizer’s tokenizer

Parameters:
- **strings** (*str*) – arbitrary length list of strings to tokenize
- **enforce_asm_rules** (*Optional[bool]*) – if True, then extra processing and checks will be done to make sure the tokenized assembly language matches the rules of assembly. See self.check_assembly_rules() for more info. If False, these checks aren’t done and bad assembly could make its way through without error, but should be noticeably faster. If None, will use the default value. The default value starts as False at the beginning of program execution but can be modified using `_set_default_enforce_asm_rules()`
- **newline_tup** (*Tuple[str, str], optional*) – the tuple to insert inbetween each passed string, or None to not insert anything. Defaults to DEFAULT_TOKENIZER.DEFAULT_NEWLINE_TUPLE.
- **match_instruction_address** (*bool, optional*) – if True, will assume there will be an instruction address at the start of the string. This only has an effect on ghidra-like instruction addresses where that address could be interpreted as either an immediate, or an instruction address. If True, then any immediates found at the start of a line will be assumed to be instruction addresses instead of immediates. If False, then instruction addresses can still be matched, but they must end with a colon ‘:’, otherwise they will be considered immediates. Defaults to True.

Raises: **TokenMismatchError** – on a bad branch prediction string

Yields: *Tuple[str, str]* – (token_name, token) tuples

tokenizer = *None*
The tokenizer used for this normalizer

class `bincfg.normalization.base_normalizer.`**MetaNorm**(*name*, *bases*, *dct*)
Bases: `type`

A metaclass for BaseNormalizer.

The Problem:
If you change instance functions within the __init__ method (EG: see the SAFE _handle_immediate() function being changed in __init__), then ‘self’ will not automatically be passed to those functions.

NOTE: this is specifically useful when the effect of a normalization method depends on parameters sent to the instance, not inherent to the class

NOTE: this is not the case for any functions that are set during class initialization (EG: outside of the __init__() block)

So, any functions changed within __init__ methods must be altered to also pass ‘self’. I ~could~ force the users to have to call a ‘__post_init__()’ function or something, but can we count on them (IE: myself) to always do that?…

The Solution:
This metaclass inserts extra code before and after any normalizer’s __init__ method is called. That code keeps track of all instance functions before intitialization, and checks to see if any of them change after initialization. This means someone re-set a function within __init__ (IE: self._handle_immediate = …). When this happens, ‘self’ will not automatically be passed when that function is called. These functions are then wrapped to also automatically pass ‘self’.

NOTE: to determine if a function changes, we just check equality between previous and new functions using getattr(self, func_name). I don’t know why basic ‘==’ works but ‘is’ and checking id’s do not, but I’m not going to question it…

NOTE: We also have to keep track of the instance functions as an instance variable in case a parent class needs their function updated, or if a child class also changes a parent class’s function in init

NOTE: this will mean you cannot call all of that class’s methods and expect them to always be the same as calling instance methods if you change functions in __init__

bincfg.normalization.base_tokenizer module

Class for tokenizing assembly lines

class bincfg.normalization.base_tokenizer.**BaseTokenizer**(*tokens=[('inst_addr', '(?:(?<=\\n|[|])|^)0x[0-9a-f]*:?(?=)'), ('newline', '[|\\n]'), ('rose', '<[^>]*>'), ('inst_start', '#start_instr#'), ('spacing', '[, \\t]+'), ('ptr', 'ptr'), ('open_bracket', '\\['), ('close_bracket', '\\]'), ('plus_sign', '\\+'), ('times_sign', '*'), ('prefix', '(?:lock|rep(?:ne|nz|e|z)?)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('segment_addr', '(?:(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$)):(?:(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$))|(?:[cdefgs]s):(?:(?:[xyz]?mm[0-9]+|st\\(?[0-9]+\\)?|(?:[sb]p|[ds]i)|[re]?(?:flags|ip|[bs]p|[sd]i|[abcd]x)|[cd]r[0-9]+|r[0-9]+[dwb]?|[abcd][lh]|[cst]w|fp_(?:[id]p|[cd]s|opc)|[cdefgs]s|(?:[gil]d)?tr|msw|mxcsr)(?=[*]|\\]|[, \\t]+|[|\\n]|<[^>]*>|$))'), ('immediate', '(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$)'), ('segment', '(?:[cdefgs]s):(?=(?:[, \\t]+)?\\[)'), ('memory_size', '(?:v[0-9]+)?(?:byte|[dqt]?word|t?float|l?double|[xyz]mmword)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('register', '(?:[xyz]?mm[0-9]+|st\\(?[0-9]+\\)?|(?:[sb]p|[ds]i)|[re]?(?:flags|ip|[bs]p|[sd]i|[abcd]x)|[cd]r[0-9]+|r[0-9]+[dwb]?|[abcd][lh]|[cst]w|fp_(?:[id]p|[cd]s|opc)|[cdefgs]s|(?:[gil]d)?tr|msw|mxcsr)(?=[*]|\\]|[, \\t]+|[|\\n]|<[^>]*>|$)'), ('opcode', '(?:[a-z][a-z0-9_.,]*)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('mismatch', '.')]*)

Bases: `object`

A default class to tokenize assembly line input

Currently, this tokenization schema can handle outputs from:

- rose disassembly
- ghidra disassembly (in BB data)

The tokenizer will tokenize essentially anything, so long as it fits known tokens.

Known Tokens:

- Instruction start token: used for op-level normalization techniques
- Instruction address: the address sometimes present and only at the very start of the instruction that is used to specify the address of that instruction. Should be an immediate value
- Newline token: used to specify new instructions in case multiple instructions are being tokenized in one string. Can be a newline character or a pipe character '|'
- Rose Information: Any information contained within <> brackets is considered rose information (including the brackets)
- Spacing: one or more spaces, tabs, or commas in a row. Can also be the characters '.' and '_' when used as delimiters for instruction prefixes built onto opcodes (see below)
- PTR: the literal string 'ptr' that appears in some memory accesses (EG: dword ptr […])
- Single character tokens used for memory accesses: '[', ']', '+', '*'
- Instruction prefixes: lock, rep, repne, etc. These can appear as plain string separated by spacing, or attatched to opcodes. If attatched to an opcode, its order does not matter, and it must be delimited by either a '.' or a '_'. If this occurs, there can only be one opcode, and all of the other substrings must be known instruction prefixes
- Branch prediction tokens: the literal strings ',pt' and ',pn' for branch predictions. Must come immediately following an opcode (or opcode + attatched instruction prefix(es)) if present
- Segment addresses: far-jump address information, eg: "fs:0x123456"
- Immediate values: can be in binary, octal, decimal, and hex
- Segment token: memory segment specifier used for memory accesses, eg: "qword ptr ds:[…]", the "ds:" bit
- Memory size: tokens that specify memory access size, eg: "qword", "dword", "byte", etc.
- Registers: known register names
- Opcodes: any string of alphabet characters that does not fit one of the tokens above is considered an opcode

Anything that does not fit one of the above tokens will be considered a 'token mismatch'

This will do the following transformations to the incomming token stream:

- Any instruction prefixes or branch predictions will be moved to immediately before their opcode in the same order that they appear in the string (separated by the same spacings that were used before them in the original string). This means any branch prediction will be immediately before the opcode and after any other instruction prefixes since they can only appear at the end of opcode + prefix strings

Parameters: **tokens** (*List[Tuple[str, str]]*) – the tokens to use. Should be a list of 2-tuples. Each tuple is a pair of (name, regex) where name is the string name of the token, and regex is a regular expression to find that token. These tuples should be ordered in the preferred order to search for tokens with a 'mismatch' token matching all characters at the very end to find mismatch lines. Defaults to *bincfg.normalization.tokenization_constants.DEFAULT_TOKENS*

DEFAULT_NEWLINE_TUPLE = *('newline', '\n')*

The default (token_name, token) tuple to use for newlines

on_token_mismatch(*token*, *string*, *mo*)

What to do when there is a token mismatch in a string. Raises a `TokenMismatchError` with info on the mismatch

Parameters:

- **token** (*str*) – the token that is mismatched
- **string** (*str*) – the string in which the mismatch occurred
- **mo** (*Match*) – the re match object

Raises: **TokenMismatchError** – by default

tokenize(**strings*, *enforce_asm_rules=None*, *newline_tup=('newline', '\n')*, *match_instruction_address=True*)

Tokenizes some number of strings in the order they were recieved returning a list of 2-tuples.

Each tuple is (name, token) where name is the string name of the token, and token is the substring in the given string corresponding to that token. Extra 'newline' tuples will be added inbetween each string.

Initially cleans the string. See `clean_incoming_instruction()` for more details.

Also pulls prefixes out of opcodes. See top of file for possible placements of instruction prefixes. These prefixes are returned in order before the opcode, with no extra newlines or anything.

Parameters:

- **strings** (*str*) – arbitrary number of strings to tokenize.
- **enforce_asm_rules** (*Optional[bool]*) – if True, then extra processing and checks will be done to make sure the tokenized assembly language matches the rules of assembly. See self.check_assembly_rules() for more info. If False, these checks aren't done and bad assembly could make its way through without error, but should be noticeably faster. If None, will use the default value. The default value starts as False at the beginning of program execution but can be modified using `_set_default_enforce_asm_rules()`
- **newline_tup** (*Tuple[str, str], optional*) – the tuple to insert inbetween each passed string, or None to not insert anything. Defaults to *self.__class__.DEFAULT_NEWLINE_TUPLE*.
- **match_instruction_address** (*bool, optional*) – if True, will assume there will be an instruction address at the start of the string. This only has an effect on ghidra-like instruction addresses where that address could be interpreted as either an immediate, or an instruction address. If True, then any immediates found at the start of a line will be assumed to be instruction addresses instead of immediates. If False, then instruction addresses can still be matched, but they must end with a colon ':', otherwise they will be considered immediates. Defaults to True.

Raises: **TokenMismatchError** – on a bad branch prediction string

Returns: list of (token_name, token) tuples

Return type: List[Tuple[str, str]]

bincfg.normalization.builtin_normalizers module

A bunch of builtin normalization methods based on literature.

NOTE: some of these are slightly modified from their original papers either for code purposes, or because we are using decompiled binaries instead of compiled assembly and thus lose out on some information (EG: symbol information for jump instructions)

class `bincfg.normalization.builtin_normalizers.`**CompressedStatsNormalizer**(**args*, ***kwargs*)

Bases: `BaseNormalizer`

A normalizer I created for use in CFG.get_compressed_stats()

Rules:

- Immediates are treated like in safe, but with a much lower default threshold
- function calls are either self vs. intern vs. extern func, no special functions
- jump destinations are 'jmpdst'
- registers are handled the same as deepsem/deepbindiff
- memory pointers/memory expressions are handled the same as in deepsemantic
- Tokenized at the instruction-level

Parameters:

- **imm_threshold** (*int*) – immediate values whose absolute value is <= imm_threshold will be left alone, those above it will be replaced with the string 'immval'. Defaults to a small value
- **special_functions** (*Optional[Set[str]]*) – a set of special function names. All external functions whose name (ignoring the '@plt' at the end) is in this set will have their name kept, otherwise they will be replaced with 'externfunc'. If None, this will default to not using any special functions
- **tokenizer** (*Optional[Tokenizer]*) – the tokenizer to use, or None to use the default BaseTokenizer
- **token_sep** (*Optional[str]*) – the string to use to separate each token in returned instruction lines. Only used if tokenization_level is 'instruction'. If None, then a default value will be used (' ' for unnormalized using BaseNormalizer(), '_' for everything else)
- **tokenization_level** (*Optional[Union[TokenizationLevel, str]]*) –

  the tokenization level to use for return values. Can be a string, or a `TokenizationLevel` type. Strings can be:

  - 'op': tokenized at the opcode/operand level. Will insert a 'INSTRUCTION_START' token at the beginning of each instruction line
  - 'inst'/'instruction': tokenized at the instruction level. All tokens in each instruction line are joined together using token_sep to construct the final token
  - 'auto': pick the default value for this normalization technique

DEFAULT_TOKENIZATION_LEVEL = *['inst', 'instruction', 'line']*

handle_function_call(*idx*, *line*, *special_functions=None*, *cfg=None*, *block=None*)

handle_immediate(*token*, *line*, *sentence*, **args*, ***kwargs*) -> str

handle_jump(*idx*, *line*, **args*, ***kwargs*)

handle_memory_size(*token*, **args*)

handle_register(*token*, **args*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

class bincfg.normalization.builtin_normalizers.**DeepBinDiffNormalizer**(**args*, ***kwargs*)
Bases: BaseNormalizer

A normalizer based on the Deep Bin Diff method

From the DeepBinDiff paper: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24311-paper.pdf

Rules:

- Constant values are ignored and replaced with 'immval'
- General registers are renamed based on length, special ones are left as-is (with number information removed.
  EG: st5 -> st, rax -> reg8, r14d -> reg4, rip -> rip, zmm13 -> zmm)
- Memory expressions are replaced with 'memexpr'
- Can't really tell what's supposed to be done with function calls, will just assume they should be 'call immval'
- Jump destinations are 'immval'
- Doesn't say anything about memory sizes, so they are ignored
- Tokens are at the op-level

DEFAULT_TOKENIZATION_LEVEL = *['op', 'opcode', 'operand']*

handle_function_call(*idx*, *line*, **args*, ***kwargs*)

handle_immediate(*, *include_negative=False*)

handle_memory_expression(*memory_start*, *token*, *line*, **args*)

handle_memory_size(**args*, ***kwargs*)

handle_register(*token*, **args*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

class bincfg.normalization.builtin_normalizers.**DeepSemanticNormalizer**(**args*, ***kwargs*)
Bases: BaseNormalizer

A normalizer based on the Deepsemantic method

from the DeepSemantic paper: https://arxiv.org/abs/2106.05478

Rules:

- Immediates can fall into multiple categories:
  a. Function calls:
     - libc function name(): "libc[name]" (not used)
     - recursive call: 'self'
     - function within the binary: 'innerfunc'
     - function outside the binary: 'externfunc'
  b. Jump (branching) family: "jmpdst"
  c. Reference: (NOTE: This is not done as I don't know how to do it with ROSE…)
     - String literal: 'str'
     - Statically allocated variable: "dispbss"
     - Data (data other than a string): "dispdata"
  d. Default (all other immediate values): "immval"
- Registers can fall into multiple categories:
  a. Stack/Base/Instruction pointer: Keep track of type and size
     [e|r]*[b|s|i]p[l]* -> [s|b|i]p[1|2|4|8]

b. Special purpose (IE: flags): Keep track of type
cr[0-15], dr[0-15], st([0-7]), [c|d|e|f|g|s]s -> reg[cr|dr|st], reg[c|d|e|f|s]s

c. AVX registers: Keep track of type
[x|y|z]*mm[0-7|0-31] -> reg[x|y|z]*mm

d. General purpose registers: Keep track of size
[e|r]*[a|b|c|d|si|di][x|l|h]*, r[8-15][b|w|d]* -> reg[1|2|4|8]

- Pointers can fall into multiple categories:

a. Direct, small: keep track of size
byte,word,dword,qword,ptr -> memptr[1|2|4|8]

b. Direct, large: keep track of size
tbyte,xword,[x|y|z]mmword -> memptr[10|16|32|64]

c. Indirect, string:
[base+index*scale+displacement] -> [base+index*scale+dispstr]

d. Indirect, not string:
[base+index*scale+displacement] -> [base+index*scale+disp]

- Tokenized at instruction-level

Parameters:

- **special_functions** (*Optional[Set[str]]*) – a set of special function names. All external functions whose name (ignoring the '@plt' at the end) is in this set will have their name kept, otherwise they will be replaced with 'externfunc'. If None, will attempt to load the default special function names from `bincfg.utils.cfg_utils.get_special_function_names()`. If you do not wish to use any special function names, then pass an empty set.
- **tokenizer** (*Optional[Tokenizer]*) – the tokenizer to use, or None to use the default BaseTokenizer
- **token_sep** (*Optional[str]*) – the string to use to separate each token in returned instruction lines. Only used if tokenization_level is 'instruction'. If None, then a default value will be used (' ' for unnormalized using BaseNormalizer(), '_' for everything else)
- **tokenization_level** (*Optional[Union[TokenizationLevel, str]]*) –

the tokenization level to use for return values. Can be a string, or a `TokenizationLevel` type. Strings can be:

- 'op': tokenized at the opcode/operand level. Will insert a 'INSTRUCTION_START' token at the beginning of each instruction line
- 'inst'/'instruction': tokenized at the instruction level. All tokens in each instruction line are joined together using token_sep to construct the final token
- 'auto': pick the default value for this normalization technique

DEFAULT_TOKENIZATION_LEVEL = *['inst', 'instruction', 'line']*

handle_function_call(*idx*, *line*, *special_functions=None*, *cfg=None*, *block=None*)

handle_immediate(*, *include_negative=False*)

handle_jump(*idx*, *line*, **args*, ***kwargs*)

handle_memory_size(*token*, **args*)

handle_register(*token*, **args*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

class bincfg.normalization.builtin_normalizers.**InnerEyeNormalizer**(**args*, ***kwargs*)
Bases: `BaseNormalizer`

A normalizer based on the Innereye method

Inherited-members: BaseNormalizer

From the InnerEye paper: https://arxiv.org/pdf/1808.04706.pdf

Rules:

- Constant values are ignored and replaced with 'immval' or '-immval' for negative values
- Function names are ignored and replaced with 'func'
- Jump destinations are 'immval'
- Registers are left as-is
- Doesn't say anything about memory sizes, so they are ignored
- Tokens are at the instruction-level

DEFAULT_TOKENIZATION_LEVEL = *['inst', 'instruction', 'line']*

handle_function_call(*idx*, *line*, **args*, ***kwargs*)

handle_immediate(*token*, *line*, *sentence*, **args*, ***kwargs*) -> str

handle_memory_size(**args*, ***kwargs*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

class bincfg.normalization.builtin_normalizers.**MyNormalizer**(**args*, ***kwargs*)
Bases: BaseNormalizer

A normalizer I created. Combines safe, deepsem, and deepbindiff methods, also uses opcode tokenization

Rules:

- Immediates are handled the same as in safe
- function calls are handled the same as deepsemantic
- jump destinations are 'jmpdst'
- registers are handled the same as deepsem/deepbindiff
- memory pointers/memory expressions are handled the same as in deepsemantic
- Tokenized at the opcode-level

Parameters:

- **imm_threshold** (*int*) – immediate values whose absolute value is <= imm_threshold will be left alone, those above it will be replaced with the string 'immval'
- **special_functions** (*Optional[Set[str]]*) – a set of special function names. All external functions whose name (ignoring the '@plt' at the end) is in this set will have their name kept, otherwise they will be replaced with 'externfunc'. If None, will attempt to load the default special function names from `bincfg.utils.cfg_utils.get_special_function_names()`. If you do not wish to use any special function names, then pass an empty set.
- **tokenizer** (*Optional[Tokenizer]*) – the tokenizer to use, or None to use the default BaseTokenizer
- **token_sep** (*Optional[str]*) – the string to use to separate each token in returned instruction lines. Only used if tokenization_level is 'instruction'. If None, then a default value will be used (' ' for unnormalized using BaseNormalizer(), '_' for everything else)
- **tokenization_level** (*Optional[Union[TokenizationLevel, str]]*) –

  the tokenization level to use for return values. Can be a string, or a `TokenizationLevel` type. Strings can be:

  - 'op': tokenized at the opcode/operand level. Will insert a 'INSTRUCTION_START' token at the beginning of each instruction line
  - 'inst'/'instruction': tokenized at the instruction level. All tokens in each instruction line are joined together using token_sep to construct the final token
  - 'auto': pick the default value for this normalization technique

DEFAULT_TOKENIZATION_LEVEL = *['op', 'opcode', 'operand']*

handle_function_call(*idx*, *line*, *special_functions=None*, *cfg=None*, *block=None*)

handle_jump(*idx*, *line*, **args*, ***kwargs*)

handle_memory_size(*token*, **args*)

handle_register(*token*, **args*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

class bincfg.normalization.builtin_normalizers.**SafeNormalizer**(**args*, ***kwargs*)
Bases: BaseNormalizer

A normalizer based on the SAFE method

From the SAFE paper: https://github.com/gadiluna/SAFE

Rules:

- All base memory addresses (IE: memory addresses that are constant values) are replaced with 'immval'
- All immediate values greater than some threshold (safe_threshold parameter, they use 5000 in the paper) are replaced with 'immval'
- Function calls are replaced with 'self' if a recursive call, 'innerfunc' if the function is within the binary, and 'externfunc' if the function is external.

  NOTE: this is different to how it is done in the SAFE paper (I believe they just keep the function names), but I made the executive decision to change it for OOV problems, and changed it instead to how it is done in the deepsemantic paper to try and give it the most information possible)
- Jump destinations are 'immval'
- Doesn't say anything about memory sizes, so they are ignored
- Doesn't say anything about registers, so they are left as-is

- Tokens are at the instruction-level

Parameters:
- **imm_threshold** (*int*) – immediate values whose absolute value is <= imm_threshold will be left alone, those above it will be replaced with the string 'immval'
- **special_functions** (*Optional[Set[str]]*) – a set of special function names. All external functions whose name (ignoring the '@plt' at the end) is in this set will have their name kept, otherwise they will be replaced with 'externfunc'. If None, will attempt to load the default special function names from `bincfg.utils.cfg_utils.get_special_function_names()`. If you do not wish to use any special function names, then pass an empty set.
- **tokenizer** (*Optional[Tokenizer]*) – the tokenizer to use, or None to use the default BaseTokenizer
- **token_sep** (*Optional[str]*) – the string to use to separate each token in returned instruction lines. Only used if tokenization_level is 'instruction'. If None, then a default value will be used (' ' for unnormalized using BaseNormalizer(), '_' for everything else)
- **tokenization_level** (*Optional[Union[TokenizationLevel, str]]*) –

  the tokenization level to use for return values. Can be a string, or a `TokenizationLevel` type. Strings can be:

  - 'op': tokenized at the opcode/operand level. Will insert a 'INSTRUCTION_START' token at the beginning of each instruction line
  - 'inst'/'instruction': tokenized at the instruction level. All tokens in each instruction line are joined together using token_sep to construct the final token
  - 'auto': pick the default value for this normalization technique

DEFAULT_TOKENIZATION_LEVEL = *['inst', 'instruction', 'line']*

handle_function_call(*idx*, *line*, *special_functions=None*, *cfg=None*, *block=None*)

handle_memory_size(**args*, ***kwargs*)

tokenization_level: *TokenizationLevel*
The tokenization level to use for this normalizer

bincfg.normalization.builtin_normalizers.**get_normalizer**(*normalizer*)
Returns the normalizer being used.

Parameters: **normalizer** (*Union[str, Normalizer]*) – either a `Normalizer` object (IE: has a callable 'normalize' function), or a string name of a built-in normalizer to use Accepted strings include: 'innereye', 'deepbindiff', 'safe', 'deepsem'/'deepsemantic', 'none'/'unnormalized'

Raises:
- **ValueError** – for unknown string name of normalizer
- **TypeError** – if *normalizer* was not a string or `Normalizer` object

Returns: a `Normalizer` object

Return type: Normalizer

bincfg.normalization.norm_utils module

An assortment of helper/utility functions for tokenization/normalization.

bincfg.normalization.norm_utils.**clean_incoming_instruction**(*s*)
Performs a first pass cleaning input strings.

Currently:

1. converts to all lowercase
2. strip()'s extra whitespace at the ends
3. Replaces all strings (like those in rose info) with __STR__

Parameters: **s** (*str*) – the string to clean

Returns: the clean string

Return type: str

bincfg.normalization.norm_utils.**clean_nop**(*idx*, *line*, **args*, ***kwargs*)
Cleans any line with the opcode 'nop' to only contain the opcode

Parameters:
- **idx** (*int*) – the index in `line` of the 'nop' opcode
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **args** – unused
- **kwargs** – unused

Returns: integer index in line of last handled token

Return type: int

bincfg.normalization.norm_utils.**eq_special_funcs**(*s1*, *s2*)
Returns True if the given two sets of special function names are equal, false otherwise

bincfg.normalization.norm_utils.**ignore**(*self*, **args*, ***kwargs*)
Ignores information (if using for rose info, then it will also ignore negatives)

bincfg.normalization.norm_utils.**imm_to_int**(*token*)
Convert the given value to integer

If token is an integer, returns token. Otherwise, converts a string token to an integer, then back to a string, accounting for hexadecimal, decimal, octal, and binary values

Parameters: **token** (*Union[str, int]*) – the immediate token to convert to integer

Returns: integer value of given token

Return type: int

bincfg.normalization.norm_utils.**memsize_value**(*self*, *token*, **args*)
Replaces memory size pointers with 'memsize' followed by the value of that memsize in bytes

Parameters: **token** (*str*) – the current string token

Returns: normalized memory size string

Return type: str

bincfg.normalization.norm_utils.**replace_function_call_immediate**(**args*)
Builds a function that replaces function call immediate values with the given replacement string

This will return a function to be called as a part of a normalizer. This only takes one argument: the replacement string. If no arguments are passed, then the replacement string will default to 'func'

NOTE: This is meant to be a higher-order function. But, just in case the user forgets that (or is too lazy to add in two extra characters to call this function), if you pass multiple args then it will be assumed this is being called as if it is the _repl_func() function below and will simply return the default result

Parameters: **args** – args for this function. Ideally either empty to use default function call string, or a string to replace all function callsa with.

Returns: either a function that will handle function calls (if this function was called correctly), or a handled function call

Return type: Union[Callable[…, None], None]

bincfg.normalization.norm_utils.**replace_general_register**(*self*, *token*, **args*)
Replaces general registers with a default string and their size, keeping special registers the same (while removing their numbers)

Parameters: **token** (*str*) – the current string token

Returns: normalized name of register

Return type: str

bincfg.normalization.norm_utils.**replace_jmpdst**(*self*, *idx*, *line*, **args*, ***kwargs*)
Replaces the jump destination immediate with 'jmpdst' iff the jump destination is an immediate value, not a segment address

Parameters:
- **idx** (*int*) – the index in `line` of the 'jump' opcode
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line

Returns: integer index in line of last handled token

Return type: int

bincfg.normalization.norm_utils.**replace_memory_expression**(**args*)
Builds a function that replaces memory expressions with the given replacement string

This will return a function to be called as a part of a normalizer. This only takes one argument: the replacement string. If no arguments are passed, then the replacement string will default to 'memexpr'

NOTE: This is meant to be a higher-order function. But, just in case the user forgets that (or is too lazy to add in two extra characters to call this function), if you pass multiple args then it will be assumed this is being called as if it is the _repl_func() function below and will simply return the default result

Parameters: **args** – args for this function. Ideally either empty to use default memory expression string, or a string to replace all memory expressions with.

Returns: either a function that will handle memory expressions (if this function was called correctly), or a handled memory expression

Return type: Union[Callable[…, None], None]

bincfg.normalization.norm_utils.**return_immstr**(**args*, *include_negative=False*)
Builds a function that replaces immediate values with the IMMEDIATE_VALUE_STR.

This will return a function to be called as a part of a normalizer. This function takes no arguments and only 1 keyword argument: whether or not to include a negative sign '-' in front of the immediate string when the input is negative.

NOTE: This is meant to be a higher-order function. But, just in case the user forgets that (or is too lazy to add in two extra characters to call this function), if you pass multiple args then it will be assumed this is being called as if it is the _repl_func() function below and will simply return the default result

Parameters:
- **args** – args for this function. Ideally empty
- **include_negative** (*bool, optional*) – if True, will include a negative sign in front of the returned immediate string when the input is negative. Defaults to False.

Returns: either a function that will handle immediate strings (if this function was called correctly), or a handled immediate string

Return type: Union[Callable[…, str], str]

bincfg.normalization.norm_utils.**special_function_call**(*self*, *idx*, *line*, *special_functions=None*, *cfg=None*, *block=None*)
Handles special function calls

Special external functions have their name kept. Recursive calls are replaced with 'self', other internal function calls are replaced with 'internfunc', other external function calls are replaced with 'externfunc'. If a block has multiple function calls out, then it will be replaced with 'multifunc'.

NOTE: This can all only happen if cfg and block information is passed. If it is not passed, then all function calls will be replaced with 'func'

Parameters:
- **idx** (*int*) – the index in `line` of the 'call' opcode
- **line** (*List[TokenTuple]*) – a list of (token_name, token) tuples. the current assembly line
- **special_functions** (*Set[str], optional*) – If passed, should be a set of string special function names. Otherwise the default special functions from `bincfg.utils.cfg_utils.get_special_function_names()` will be used. Defaults to None.
- **cfg** (*Union[CFG, MemCFG], optional*) – either a `CFG` or `MemCFG` object that these lines occur in. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.
- **block** (*Union[CFGBasicBlock, int], optional*) – either a `CFGBasicBlock` or integer block_idx in a `MemCFG` object. Used for determining function calls to self, internal functions, and external functions. If not passed, then these will not be used. Defaults to None.

Returns: integer index in line of last handled token

Return type: int

bincfg.normalization.norm_utils.**threshold_immediate**(**args*)
Builds a function that replaces immediate values with *immval* iff abs(immediate) > some threshold

This will return a function to be called as a part of a normalizer. This only takes one argument: the immediate value threshold. If no arguments are passed, then the threshold will default to *DEFAULT_IMMEDIATE_THRESHOLD*.

NOTE: This is meant to be a higher-order function. But, just in case the user forgets that (or is too lazy to add in two extra characters to call this function), if you pass multiple args then it will be assumed this is being called as if it is the _repl_func() function below and will simply return the default result

Parameters: **args** – args for this function. Ideally either empty to use the default thresholding value, or a single positive integer for the immediate threshold

Returns: either a function that will handle thresholded immediate strings (if this function was called correctly), or a handled thresholded immediate string

Return type: Union[Callable[…, str], str]

bincfg.normalization.normalize module

Provides function(s) to perform normalization techniques on CFG's

bincfg.normalization.normalize.**normalize_cfg_data**(*cfg_data*, *normalizer*, *inplace=False*, *using_tokens=None*, *force_renormalize=False*, *convert_to_mem=False*, *unpack_cfgs=False*, *progress=False*)
Normalizes some cfg data.

Parameters:

- **cfg_data** (*Union[CFG, MemCFG, CFGDataset, MemCFGDataset, Iterable]*) – some cfg data. Can be either: CFG, MemCFG, CFGDataset, MemCFGDataset, or iterable of previously mentioned types. Will return the same type as that passed.
- **normalizer** (*Union[str, Normalizer]*) – the normalizer to use. Can be either a `Normalizer` class with a *.normalize()* method, or a string to use a built-in normalizer. See `bincfg.normalization.get_normalizer()` for acceptable strings.
- **inplace** (*bool, optional*) – if True, will modify data in-place instead of creating new objects. Defaults to False. NOTE: if *inplace=False*, and the incoming data has already been normalized with the passed *normalizer*, then the original cfg will be returned, NOT a copy.
- **using_tokens** (*TokenDictType, optional*) – only used for `MemCFG`'s. If not None, then a dictionary mapping string tokens to integer token values that will be used as any `MemCFG`'s tokens. Defaults to None.
- **force_renormalize** (*bool, optional*) – by default, this method will only normalize cfg's whose .normalizer != to the passed normalizer. However if *force_renormalize=True* , then all cfg's will be renormalized even if they have been previously normalized with the same normalizer. Defaults to False.
- **convert_to_mem** (*bool, optional*) – if True, will convert all `CFG`'s and `CFGDatasets` to their memory-efficient versions after normalizing. Defaults to False.
- **unpack_cfgs** (*bool, optional*) – by default, this method will return the same types that were passed to be normalized. However if *unpack_cfgs=True*, then instead, a list of all cfgs unpacked (EG: unpacked from lists, and pulled out of datasets) will be returned. Defaults to False. NOTE: if only a single `CFG`/`MemCFG` was passed, a list will still be returned of only that single element.
- **progress** (*bool, optional*) – if True, will show a progressbar for normalizations of multiple cfg's. Defaults to False.

Raises: **TypeError** – Unknown input *cfg_data* type(s)

Returns: the normalized data

Return type: Union[CFG, MemCFG, CFGDataset, MemCFGDataset, List, Tuple]

bincfg.normalization.tokenization_constants module

Constants and regular expressions involving tokenization of assembly lines

class `bincfg.normalization.tokenization_constants.`**`ARCH`**(*value*)

Bases: `Enum`

An enumeration.

`X86` = *'x86'*

`X86_64` = *'x86_64'*

`bincfg.normalization.tokenization_constants.`**`DEFAULT_TOKENS`** = *[('inst_addr', '(?:(?<=\\n|[|])|^)0x[0-9a-f]*:?(?=)'), ('newline', '[|\\n]'), ('rose', '<[^>]*>'), ('inst_start', '#start_instr#'), ('spacing', '[, \\t]+'), ('ptr', 'ptr'), ('open_bracket', '\\['), ('close_bracket', '\\]'), ('plus_sign', '\\+'), ('times_sign', '*'), ('prefix', '(?:lock|rep(?:ne|nz|e|z)?)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('segment_addr', '(?:(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$)): (?:(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$))|(?:[cdefgs]s):(?:(?:[xyz]?mm[0-9]+|st\\(?[0-9]+\\)?|(?:[sb]p|[ds]i)|[re]?(?:flags|ip|[bs]p|[sd]i|[abcd]x)|[cd]r[0-9]+|r[0-9]+[dwb]?|[abcd][lh]|[cst]w|fp_(?:[id]p|[cd]s|opc)|[cdefgs]s|(?:[gil]d)?tr|msw|mxcsr)(?=[*]|\\]|[, \\t]+|[|\\n]|<[^>]*>|$))), ('immediate', '(?:-?0x[0-9a-f]+|-?0o[0-7]+|-?0b[01]+|-?[0-9]+)(?=\\]|[, \\t]+|[|\\n]|<[^>]*>|$)'), ('segment', '(?:[cdefgs]s):(?=(?:[, \\t]+)?\\[)'), ('memory_size', '(?:v[0-9]+)?(?:byte|[dqt]?word|t?float|l?double|[xyz]mmword)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('register', '(?:[xyz]?mm[0-9]+|st\\(?[0-9]+\\)?|(?:[sb]p|[ds]i)|[re]?(?:flags|ip|[bs]p|[sd]i|[abcd]x)|[cd]r[0-9]+|r[0-9]+[dwb]?|[abcd][lh]|[cst]w|fp_(?:[id]p|[cd]s|opc)|[cdefgs]s|(?:[gil]d)?tr|msw|mxcsr)(?=[*]|\\]|[, \\t]+|[|\\n]|<[^>]*>|$)'), ('opcode', '(?:[a-z][a-z0-9_.,]*)(?=[, \\t]+|[|\\n]|<[^>]*>|$)'), ('mismatch', '.')]*

Default list of (token_name, regex) token tuples to match to

exception `bincfg.normalization.tokenization_constants.`**`TokenMismatchError`**

Bases: `Exception`

class `bincfg.normalization.tokenization_constants.`**`TokenizationLevel`**(*value*)

Bases: `Enum`

An enumeration.

`AUTO` = *['auto', 'automatic']*

`INSTRUCTION` = *['inst', 'instruction', 'line']*

`OPCODE` = *['op', 'opcode', 'operand']*

class `bincfg.normalization.tokenization_constants.`**`Tokens`**

Bases: `object`

`BRANCH_PREDICTION` = *'branch_prediction'*

`CLOSE_BRACKET` = *'close_bracket'*

`IMMEDIATE` = *'immediate'*

`INSTRUCTION_ADDRESS` = *'inst_addr'*

`INSTRUCTION_PREFIX` = *'prefix'*

`INSTRUCTION_START` = *'inst_start'*

`MEMORY_SIZE` = *'memory_size'*

MISMATCH = *'mismatch'*

NEWLINE = *'newline'*

OPCODE = *'opcode'*

OPEN_BRACKET = *'open_bracket'*

PLUS_SIGN = *'plus_sign'*

PTR = *'ptr'*

REGISTER = *'register'*

ROSE_INFO = *'rose'*

SEGMENT = *'segment'*

SEGMENT_ADDRESS = *'segment_addr'*

SPACING = *'spacing'*

TIMES_SIGN = *'times_sign'*

Module contents