1_dsl_rules_generation

July 23, 2025

[ ]:
```
import sys
import os
from pathlib import Path
from dotenv import load_dotenv

load_dotenv()

project_root = Path.cwd().parent
src_path = project_root / "sources"
sys.path.append(str(src_path))

from core.llm import OpenRouterClient
from utils.generate_dsl_docs import generate_symbolic_dsl_reference_markdown

from core.dsl_symbolic_interpreter import SymbolicRuleParser

sys.path.append(str(Path("/home/yann/ssd_storage/python/arcprize2025/tests/")))
sys.path.append(str(Path("/home/yann/ssd_storage/python/arcprize2025/sources/
 ↪")))
from test_dsl_symbolic_executor import TEST_CASES
from assets.symbols import ROM_VAL_MAP

print("Cell 1 executed: Environment paths, dotenv loaded, and core modules␣
 ↪imported.")
```

[ ]:
```
TRUE_RULES = {
    "007bbfb7": " ( ( (III), (III)), (III), (IX,IX, ))",
    "009d5c81": " ( ( (I)), [( (III,III, I ;III; I ), ( (VIII, II), (I, ))),␣
 ↪( (III,III,I I; I ;III), ( (VIII, III), (I, )))], ( (VIII, VII), (I, )))",
    "00d62c1b": " ( , ( ( ), ( (III))), ( ( ), ( (III))), ( ( ,IV), (I, ))))",
    "00dbd492":␣
 ↪" ( ( ,III), ( , (IX,IX,[[II,II,II,II,II,II,II,II,II],[II, , , , , , , ,II],[II, , , , , , , ,II],[I
}
```

[ ]:
```
doc_sigil = generate_symbolic_dsl_reference_markdown()
if not doc_sigil or not isinstance(doc_sigil, str) or not doc_sigil.strip():
```

```
    raise ValueError("`doc_sigil` is empty or malformed. Ensure␣
 ↪`generate_symbolic_dsl_reference_markdown()` correctly extracts DSL grammar␣
 ↪from `SYMBOL_RULES`.")
else:
    print("Cell 2 executed: Successfully generated DSL Grammar from␣
 ↪`SYMBOL_RULES` via `doc_sigil`.")
```

[ ]:
```
import random

def create_pure_rule_generation_prompt(num_rules_to_generate=5,␣
 ↪complexity_focus="simple", target_command=None):
    prompt_examples = ""
    example_count = 0

    available_rule_strings = []
    # Add your existing TEST_CASES rules
    for test_case_item in TEST_CASES:
        if isinstance(test_case_item, dict) and "rule_string" in test_case_item␣
 ↪and test_case_item["rule_string"]:
            available_rule_strings.append(test_case_item["rule_string"].strip())
        elif isinstance(test_case_item, str) and test_case_item.strip():
            available_rule_strings.append(test_case_item.strip())

    # Add your TRUE_RULES as strings
    true_rule_strings = list(TRUE_RULES.values())

    # Decide which pool of examples to draw from based on complexity
    # For complex/advanced, heavily prioritize or exclusively use TRUE_RULES
    if "complex" in complexity_focus or "advanced" in complexity_focus:
        eligible_rules_pool = true_rule_strings
        # If true_rules are fewer than needed, supplement with the longest of␣
 ↪TEST_CASES
        if len(eligible_rules_pool) < num_rules_to_generate:
            sorted_test_cases = sorted(available_rule_strings, key=len,␣
 ↪reverse=True)
            eligible_rules_pool.extend(sorted_test_cases[:num_rules_to_generate␣
 ↪- len(eligible_rules_pool)])
    else:
        # For simple/moderate, use the length-based selection from␣
 ↪available_rule_strings
        sorted_rules_by_length = sorted(available_rule_strings, key=len)
        if "simple" in complexity_focus:
            eligible_rules_pool = sorted_rules_by_length[:max(1,␣
 ↪len(sorted_rules_by_length) // 5)]
        elif "moderate" in complexity_focus:
```

```
            eligible_rules_pool =␣
↪sorted_rules_by_length[len(sorted_rules_by_length) // 5 :␣
↪len(sorted_rules_by_length) * 4 // 5]
        else: # Fallback, should not happen if complexity_focus is one of the␣
↪defined levels
            eligible_rules_pool = available_rule_strings

    if not eligible_rules_pool:
        # Fallback if both pools are empty
        prompt_examples = "DSL Program Example:  \nDSL Program Example:  (I,␣
↪II)\nDSL Program Example:  ( ,  )\n"
    else:
        random.shuffle(eligible_rules_pool)
        for rule_str in eligible_rules_pool:
            prompt_examples += f"DSL Program Example: {rule_str}\n"
            example_count += 1
            if example_count >= num_rules_to_generate:
                break

    # The rest of your instruction construction remains the same
    instruction = f"""You are an expert in the Abstract Reasoning Challenge␣
↪(ARC) and a master of a Domain-Specific Language (DSL) designed for grid␣
↪manipulation.Your task is to generate {num_rules_to_generate} NEW and UNIQUE␣
↪DSL programs.These programs are not for a specific ARC puzzle, but rather to␣
↪serve as diverse examples of valid DSL syntax and logic for training␣
↪purposes.---**DSL Grammar and Symbols Reference:**{doc_sigil}---**Value␣
↪Mappings:*** Roman numerals (I, II, III, IV, V, VI, VII, VIII, IX, X) map to␣
↪integers 1-10.*  maps to 0.* Katakana symbols provided by ROM_VAL_MAP (e.g.,␣
↪" " for 1, " " for 2, etc.) also map to integers 0-9.---**Example DSL Program␣
↪Structures:**{prompt_examples}---**Generation Requirements:*** Each␣
↪generated program must be a **single, valid DSL string**.* Each program␣
↪should be on a new line.* Focus on programs that are `{complexity_focus}` in␣
↪terms of their command usage and nesting."""
    if target_command:
        instruction += f"* Each program must include the '{target_command}'␣
↪command at least once.\n"
    else:
        instruction += "* Vary the commands and their compositions.\n"

    instruction += """* Do not include any input/output grids, explanations,␣
↪numbering, or any other text.* Just provide the DSL programs."""
    return instruction

# You'll need to define TRUE_RULES somewhere before this function is called.
# For example, after your TEST_CASES are imported.
```

```
print("Cell 3 executed: `create_pure_rule_generation_prompt` function defined␣
 ↪with dynamic example selection and true rules integration.")
```

[ ]:
```
import random

print("--- Starting Bulk DSL Rule Generation Process ---")

openrouter_client = OpenRouterClient(model="meta-llama/llama-3.1-70b-instruct",␣
 ↪temperature=0.8)

total_rules_needed = 1000
rules_per_batch = 20
save_interval =5
output_file = Path("generated_dsl_rules.txt")

complexity_prompts = [
#   # --- Level 1: Foundation & Basic Operations ---
#     "simple, involving one or two operations like  or , and simple␣
 ↪compositions with , sometimes including literals like I or .",
#     "simple, focusing on basic operations like , , , and their direct␣
 ↪application with Roman numerals (I-X) and  as arguments.",
#     "simple, emphasizing rules that involve the  operator with basic␣
 ↪symbolic inputs or simple  expressions.",
#     "simple, generating rules that only use constant values (I-X, ) as␣
 ↪inputs to single-arity or binary operators.",
#     "simple, strictly focusing on rules that map one literal value to another␣
 ↪using ' ', ensuring diversity in value pairs.",

#     # --- Level 2: Basic Composition & Argument Variety ---
#     "moderate, involving up to two nested operations, primarily using , , ,␣
 ↪and , with mixed literal and simple symbolic arguments.",
#     "moderate, focusing on rules that use a diverse set of arguments,␣
 ↪including Roman numerals I-X, , and complex nested sub-expressions as␣
 ↪arguments.",
#     "moderate, prioritizing rules that involve comparison operators like = or␣
 ↪<, applied to numeric literals or results of simple operations.",
#     "moderate, focusing on rules that perform basic arithmetic operations␣
 ↪like , ,  on literals or simple expressions, with minimal nesting (1-2␣
 ↪levels).",
#     "moderate, generating rules that apply transformation operators like ␣
 ↪(compose) or  (apply) to symbolic inputs or outputs of simple operations.",
#     "moderate, emphasizing the use of aggregation operators like  (sum) or ␣
 ↪(product) on sets of literals or results of simple transformations.",
#     "moderate, exploring rules that generate constant outputs (e.g., always␣
 ↪returning  or a specific Roman numeral) based on simple operations.",
```

```
#     "moderate, focusing on rules that use the  (complement/negation)␣
 ↪operator applied to other simple DSL expressions.",

    # --- Level 3: Intermediate Nesting & Broader Operator Set ---
    # Example for '007bbfb7' insights:
    "complex, focus on rules that use sequence (` `) or conditional (` `)␣
 ↪operations where arguments are themselves complex transformations or logical␣
 ↪checks (e.g., ` `, ` `). Incorporate operators like ` ` (sum/combine) in␣
 ↪nested structures. Aim for 3-4 levels of nesting.",

    # Example for '00d62c1b' insights:
    "complex, generate rules that integrate logical checks (` `, ` `, ` `, ` `)␣
 ↪within nested ` ` or ` ` structures. Explore patterns where conditions are␣
 ↪chained, leading to complex control flow.",

    # --- Level 4: Advanced Composition & Specific ARC Concepts ---
    # Example for '009d5c81' insights:
    "advanced, emphasize rules using ` ` (switch/case-like) with ` ` (pattern␣
 ↪matching) and multiple conditional branches, where each branch contains a␣
 ↪distinct, nested sub-rule. Maximize the number of branches and the␣
 ↪complexity within each.",

    # Example for '00dbd492' insights:
    "advanced, generate highly unique and elaborate DSL programs that involve␣
 ↪recursive application of operations like ` ` (apply/map) to literal grid␣
 ↪structures (` `). Focus on rules that show a clear pattern of transformation␣
 ↪applied to progressively smaller or modified data. These rules should be␣
 ↪very long and demonstrate deep, structured repetition of complex␣
 ↪sub-patterns. Aim for 5+ levels of logical nesting.",

    # General Advanced prompt:
    "advanced, construct rules with maximal nesting (5+ levels), integrating a␣
 ↪broad spectrum of DSL commands, including arithmetic, logical, comparison,␣
 ↪transformation, and pattern matching operators. The structure should reflect␣
 ↪a multi-step problem-solving process.",

]
switch_complexity_every_n_batches = 1

target_op = None

all_validated_rules = []
parser = SymbolicRuleParser()

if output_file.exists():
    with open(output_file, 'r') as f:
```

```
        existing_rules = set(line.strip() for line in f if line.strip())
    all_validated_rules.extend(list(existing_rules))
    print(f"Resuming generation. Loaded {len(all_validated_rules)} existing␣
 ↪rules from {output_file}.")
else:
    print(f"Starting new generation. Output will be saved to {output_file}.")

print(f"\n### Target: {total_rules_needed} valid rules. Requesting␣
 ↪{rules_per_batch} per batch.")
print(f"### Complexity Prompts Defined: {len(complexity_prompts)}")
print(f"### Switching complexity every {switch_complexity_every_n_batches}␣
 ↪batches.")
if target_op:
    print(f"### Target Command: '{target_op}'")
print("\nCell 4 executed: Bulk generation configuration and initialization␣
 ↪complete.")
```

[ ]:
```
import time
import random

print("--- Beginning LLM Calls and Validation Loop ---")
loop_count = 0
current_complexity_prompt = None

MAX_RETRIES = 5
INITIAL_BACKOFF_SECONDS = 5

while len(all_validated_rules) < total_rules_needed:
    loop_count += 1

    if loop_count == 1 or (loop_count - 1) % switch_complexity_every_n_batches␣
 ↪== 0:
        chosen_complexity = random.choice(complexity_prompts)
        current_complexity_prompt = chosen_complexity
        print(f"\n--- Switching Complexity! ---")
        print(f"New complexity focus: '{current_complexity_prompt}'")

    print(f"\n--- Batch {loop_count} --- (Current valid rules:␣
 ↪{len(all_validated_rules)}/{total_rules_needed})")

    generation_prompt = create_pure_rule_generation_prompt(
        num_rules_to_generate=rules_per_batch,
        complexity_focus=current_complexity_prompt,
        target_command=target_op
    )

    generated_text = None
```

```
    retries = 0
    while retries < MAX_RETRIES:
        try:
            print(f"Calling LLM for {rules_per_batch} rules (Attempt {retries +␣
↪1}/{MAX_RETRIES})...")
            generated_text = openrouter_client(
                generation_prompt
            )
            if generated_text:
                break
            else:
                print("LLM returned empty text. Retrying...")
        except Exception as e:
            print(f"LLM call failed for batch {loop_count} (Attempt {retries +␣
↪1}/{MAX_RETRIES}): {e}")

        retries += 1
        if retries < MAX_RETRIES:
            wait_time = INITIAL_BACKOFF_SECONDS * (2 ** (retries - 1)) #␣
↪Exponential backoff
            print(f"Waiting {wait_time} seconds before retrying...")
            time.sleep(wait_time)
        else:
            print(f"Max retries ({MAX_RETRIES}) reached for batch {loop_count}.␣
↪Skipping this batch.")
            break # Exit retry loop, no more attempts for this batch

    if not generated_text: # If after all retries, still no text
        print(f"Skipping batch {loop_count} due to persistent LLM errors.")
        continue # Skip to next outer loop iteration (next batch)

    generated_rules_list = [
        line.strip() for line in generated_text.split('\n')
        if line.strip() and not line.strip().startswith('#')
    ]

    if not generated_rules_list:
        print("No parsable rules found in raw LLM output for this batch.␣
↪Retrying in next loop.")
        time.sleep(2)
        continue

    print(f"Attempting to Parse and Validate {len(generated_rules_list)} DSL␣
↪Rules from LLM response...")

    batch_valid_count = 0
```

```
    for i, rule_str in enumerate(generated_rules_list):
        if rule_str not in all_validated_rules:
            try:
                parser.parse_rule(rule_str)
                all_validated_rules.append(rule_str)
                batch_valid_count += 1
            except Exception as e:
                pass

    print(f"Batch {loop_count} Summary: {batch_valid_count} new unique valid␣
 ↪rules added. Total valid rules: {len(all_validated_rules)}.")

    if len(all_validated_rules) >= total_rules_needed:
        pass
    elif len(all_validated_rules) // save_interval > (len(all_validated_rules)␣
 ↪- batch_valid_count) // save_interval:
        print(f"Saving {len(all_validated_rules)} rules to {output_file}...")
        try:
            with open(output_file, 'w') as f:
                for rule in all_validated_rules:
                    f.write(rule + '\n')
            print("Save complete.")
        except IOError as e:
            print(f"Error saving rules to file: {e}")

print(f"\n--- Generation Loop Complete! ---")
print(f"Final count: {len(all_validated_rules)} unique valid rules collected.")
print(f"Performing final save of all {len(all_validated_rules)} rules to␣
 ↪{output_file}...")
try:
    with open(output_file, 'w') as f:
        for rule in all_validated_rules:
            f.write(rule + '\n')
    print("Final save complete.")
except IOError as e:
    print(f"Error during final save: {e}")

print("Cell 5 executed: Bulk rule generation loop completed, rules saved.")
```

[ ]:
```
# Cell 6: Parse Raw Output and Perform Syntactic Validation

# This cell now just displays the final outcome after the loop in Cell 5 has␣
 ↪completed.

print("\n### Final DSL Rule Generation Summary:")
print(f"Total Rules Targeted: {total_rules_needed}")
print(f"Total Unique Valid Rules Collected: {len(all_validated_rules)}")
```

```
print(f"All collected rules saved to: {output_file.resolve()}")

# Optional: Display a few generated rules
print("\n--- A few examples of generated rules (first 10) ---")
if all_validated_rules:
    for i, rule in enumerate(all_validated_rules[:10]):
        print(f"{i+1}. {rule}")
    if len(all_validated_rules) > 10:
        print(f"... and {len(all_validated_rules) - 10} more.")
else:
    print("No rules were generated or collected.")

print("\nCell 6 executed: Final summary displayed.")
```

[ ]:

[ ]:

[ ]:
```
generation_prompt
```

[ ]:
```
all_validated_rules
```

[ ]: