PROSE-TO-P4: LEVERAGING HIGH LEVEL LANGUAGES

Mihai-Valentin DUMITRU[1], Vlad-Andrei BĂDOIU[2], Costin RAICIU[3]

Languages such as P4 and NPL have enabled a wide and diverse range of networking applications that take advantage of programmable data planes. However, software development in these languages is difficult. To address this issue, high-level languages have been designed to offer programmers powerful abstractions that reduce the time, effort and domain-knowledge required for developing networking applications, as well as to allow writing portable and modular code. These languages are then translated by a compiler into P4/NPL code.

Inspired by the recent success of Large Language Models (LLMs) in the task of code generation, we propose to raise the level of abstraction even higher, employing LLMs to translate prose into high-level networking code. We analyze this problem, focusing on the motivation and opportunities, as well as the challenges involved, and sketch out a roadmap for the development of a system that can generate high-level data plane code from natural language instructions. We present some promising preliminary results on generating Lucid code from natural language.

Keywords: P4, programmable data planes, LLM, code generation

1. Introduction

During the past decade, the introduction of programmable data planes and associated languages, such as P4 [7] and NPL [8], has opened the door for a broad variety of networking applications, such as in-band network telemetry, firewalls and load-balancing [10], to name a few.

Software development in these languages has proven to be difficult, for multiple reasons [6, 5]. For example, limited hardware resources require programmers to be familiar with the target and customize programs for it, reducing portability. The nature of these languages induces programs to be monolithic: removing or adding support for a protocol requires changes in multiple parts of the program (parser, deparser, control), making it hard to compose programs or to compartmentalize functionality into separate libraries.

[1]PhD Student, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: mihai.dumitru2201@upb.ro

[2]PhD Student, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: vlad_andrei.badoiu@upb.ro

[3]Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: costin.raiciu@upb.ro

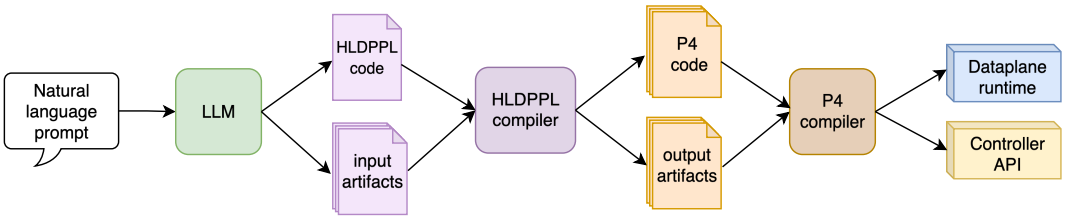

FIGURE 1. Proposed pipeline of going from prose to concrete data plane applications on actual switches. Some languages may require network configurations or other extra information, grouped here under "input artifacts". Compilation produces one or more data plane programs, perhaps together with program-to-device mappings or other information, grouped here as "output artifacts".

One strategy to address these problems is the development of high-level data plane programming languages (HLDPPLs) that compile to P4/NPL code for one or more switches. These enable programmers to focus on higher-level aspects of the resulting network application, without concerning themselves with the low-level details of a particular target. Multiple such languages have been designed, including Graph-to-P4 [1], Lucid [4, 5], Lucid 2.0 [9], Lyra [6], O4 [2], P4All [16], P4rrot [3], pcube [17]. They provide various levels of abstraction that help lessen burden on the developer; many of them have simpler grammars than P4 or NPL. However, the field remains volatile: new languages appear periodically and none of the existing ones has been widely adopted and given the opportunity to mature.

Inspired by the recent success of Large Language Models (LLMs) in the task of code generation [11, 12, 14, 13, 15, 20], we wish to push the move towards higher-level network programming languages even further, by employing natural language as the top layer for the development of data planes. Ideally, the network programmer should be able to express desired functionalities in the form of a natural language prompt that is then progressively transformed into code that runs on actual devices (Figure 1). The user of such a system can not only ignore platform-specific constructs and hardware constraints, but can also do away with learning to program in a language that may easily be abandoned or superseded. All that is needed is a rudimentary knowledge of basic networking architectures and protocols.

One of the main obstacles in developing such a system is the low amount of data available. The authors of The Stack v2 [13], one of the largest publicly available coding datasets, refer to languages in their dataset such as Julia (450K files totalling 6.12 GB) and Perl (1.12M files totalling 7.82 GB) as "low-resource languages". We wish to address "no-resource languages", for which only several programs are available.

The experiments presented in this paper target exclusively the Lucid language, but our focus is not on code-generation for one particular, already

existing data plane language, but rather code-generation for recently published, non-mature HLDPPLs, for which there is no existing program dataset or active community.

One may ask what makes data plane networking a good candidate for code generation under the no-resources constraints. Computations intended to be run in the data plane are linear in nature; each program can be modeled as a chain of simple instructions, with occasional branching for conditional blocks, but no loops. Due to the nature of the applications themselves, as well as the constraints ultimately mandated by the target switches, programs are also quite limited in length. We believe these features make HLDPPLs ideal candidates for using LLMs to generate "no-resource languages".

We explore code generation for HLDPPLs and not directly for P4 or NPL, as this offers unique opportunities and challenges. The grammars and semantics of these high-level languages are simpler, with programs being generally shorter than their low-level counterparts; this should make it easier for programmers to read, understand, validate and correct the generated code. Programs written in HLDPPLs are also not tightly coupled to a specific architecture, which makes them portable. This lifts the burden of managing hardware from the generating LLM and places it instead on the compiler. Just like in general-purpose programming settings, time-tested conventional compiler techniques can be levereaged to reliably transform high-level code into correct, safe, highly performant programs.

P4 also differs from these HLDPPLs by having an active programming community which produces publicly available code, allowing for techniques such as fine-tuning for obtaining a code-generating LLM. The problem of collecting a P4 dataset and fine-tuning LLMs of various architectures and sizes to generate P4 code has been addressed by [27].

The aim of this work is to leverage code-generating LLMs and conventional transpilers to transform natural languages prompts into concrete code that can run on a programmable switch. The opportunity to do so is enabled by the simple nature of HLDPPLs and the relatively short program length. The main challenge stems from the lack of a dataset of programs written in these languages: code-generating LLMs are usually trained or fine-tuned on gigabytes, or even terabytes of data; we only assume the existence of a handful of examples.

We make the following key contributions:

- we identify a series of characteristics that make HLDPPLs good candidates for the task of using LLMs to generate "no-resource languages"
- we analyze the challenges and opportunities involved in this task
- we evaluate the plausibility of our methodology using two state-of-the-art LLMs (ChatGPT 4 [26] and Gemini Ultra [20, 21]) to generate code in the Lucid language [4]

2. High-level data plane programming languages

The difficult nature of software development in P4 has lead to research into new data plane programming languages which abstract away target-specific details, offering programmers high-level constructs to aid with portability, composability, ease of development etc. These language differ in their motivations, scope, structure, constructs and level of abstraction, but in general they all aim to reduce the time, effort and domain-knowledge needed to develop new data plane applications.

We briefly survey existing languages, focusing on features relevant to our goal of LLM-assisted code generation and present our rationale for singling out Lucid [4] for our preliminary experiments.

O4 [2], **pcube** [17], **P4All** [16] add to P4 several syntactical constructs such as fixed-size loops and arrays (O4, pcube) and elastic structures (P4All). They are essentially *thin wrappers* over the P4 syntax, offering a few convenient constructs.

Graph-to-P4 [1] can translate state diagrams (designed visually and represented as an XML) into parser graphs. Its scope is limited to P4 parsers and does not address other components of data plane programming, such as control blocks.

P4rrot [3] aims to ease the implementation of "application-layer tasks" in the data plane. It is designed as a Python library, which has interesting implications, as Python seems to be at the center of LLM-based code generation efforts. However, for the purposes of this work, we choose to focus on a language with a standalone syntax, as this is more common among the languages considered.

Lyra [6] addresses the issues of portability, extensibility and composition. The Lyra compiler takes as input code in the Lyra language, an algorithm scope describing the placement of algorithms (e.g. on a specific subset of switches) and a description of the network's topology and configuration; from these it first produces a "context-aware intermediate representation" that can be then turned by the backend into P4 or NPL code. Unfortunately neither a complete grammar, a compiler implementation, detailed documentation or complete examples are publicly available.

Lucid [4]'s goal is "putting control functionality into the data plane." Lucid introduces events (such as an arriving packet) and event handlers, which are procedures that describe stateful computation to be executed when an event occurs, as well as a novel type system and memory operations. We consider Lucid to be the most mature HLDPPL currently available. It has been followed by **Lucid 2.0** [9], which introduces new constructs (such as polymorphism and type inference) to aid with modular programming. The presentation has also been extended to a PhD thesis [5] that provides additional details on the language design and motivation. Both the compiler and example

programs are publicly available[1], which makes Lucid an ideal candidate for our experiments, because it offers all the resources that we might expect from such a language.

At the time of writing, none of these languages has been standardised or widely-adopted; they are research projects in various degrees of active development. We expect this volatility to continue in the near future, with existing languages being improved or modified and new languages being designed. The lack of stability in the field makes it difficult for novices (and perhaps experienced programmers alike) to enjoy the benefits of these languages, since the learning effort can be considered wasted if a language is soon abandoned by its creators and the community or superseded by a new one.

We wish to offer potential network programmers the possibility of rapid prototyping with low effort and little domain-knowledge, harnessing the power of HLDPPLs without requiring the commitment to familiarize oneself with a particular language: its syntax, semantics and idiosyncrasies. To this end, we propose using LLMs to translate natural language specifications into code. In this paper we focus on the Lucid language, for reasons discussed above and in the next section.

3. Generating "no-resource languages"

Our goal is to translate natural language specifications into HLDPPL code using LLMs. Due to their maturity and impressive results on a wide range of applications, we choose ChatGPT 4 and Gemini Ultra. While LLMs have, in recent times, shown very good results in the task of prose-to-code, they usually generate code in popular programming languages, for which a large number of programs written by human developers is available; models are fine-tuned (or come pre-trained, as is the case for our two LLMs) on these large code datasets. For a newly published programming language with no active community, such a wide range of examples is not available, making fine-tuning or training non-viable strategies.

We assume the resources available for such a language are a subset of the following:

- the contents of the scientific paper that introduces the language
- additional documentation
- description of the formal grammar
- several code examples
- a compiler implementation

The documentation, grammar and examples could be available as appendices to the original paper, or as part of a public repository; a compiler could be useful for the possibility of extracting a formal grammar description from its parser, as well as for offering a better understanding of the semantics of the

[1] https://github.com/PrincetonUniversity/lucid

language. We focus our experiments on the Lucid language, because it is the only HLDPPL we are aware of that ticks all the five boxes above.

Lacking a dataset of programs to retrain (or fine-tune) the LLMs, we have to rely on *prompt engineering* techniques, which involve carefully constructing the inputs to the LLM, guiding it to produce high-quality answers. Prompt engineering has already developed into a wide and diverse field [22], with techniques such as Chain-of-Thought [23] and Tree-of-thoughts [24] achieving impressive results on tasks such as arithmetic reasoning, symbolic reasoning, creative writing etc.

The technique of "grammar prompting" developed by Wang et al. [18] is most relevant to our goals. The authors tackle the problem of generating programs in a Domain Specific Language (DSL) that is absent (or present in small quantities) in the LLM's training set. To this end, a few-shot approach is employed, attaching to each example a small subset of the DSL's grammar, "minimally sufficient" for generating that particular example. The model is instructed to first produce a specialized grammar for the requested program, then generate the code conditioned on this grammar.

That same paper presents state-of-the-art results on five DSLs for the tasks of semantic parsing, AI planning and molecule generation. Jain et al. [19] experiment with grammar prompting, as well as other techniques, for the task of prose-to-diagram: translating a natural language description into the syntax used by the Penrose framework[2] [25], which produces visual diagrams.

Because our goal is to develop a system for generating newly published languages, it is relevant to know whether data associated with the tested language (Lucid), such as the text of the original paper or the public repository code, are part of these LLMs' training sets. Unfortunately, for ChatGPT 4 and Gemini Ultra, this information is not publicly available; basic interrogation of the two LLMs reveal that they have at least some knowledge of Lucid. This is a limitation of our experiments and results, because the language information does not come solely from the contents of the prompt, which may cast doubt on whether the results could transfer to brand new programming languages, completely absent from the training dataset. However, we illustrate that this is not a critical limitation, by building a baseline of results that shows how, without advanced prompt-engineering techniques, neither ChatGPT 4 nor Gemini Ultra can generate code in the considered languages.

4. Preliminary results

We evaluate the grammar prompting technique [18] for few-shot HLDPPL code generation in the Lucid programming language, because of its standalone syntax and public availability of all the resources mentioned in section "Generating "no-resource languages"". To this end, we leverage the Gemini Ultra and ChatGPT 4 models due to their overall performance on a wide variety of

[2] https://penrose.cs.cmu.edu/

tasks, including natural language understanding, symbolic reasoning and code generation, which we believe are useful skills for the task at hand.

One important constraint for prompt engineering is the *context length* of the LLM – the number of tokens that the LLM can effectively consider when producing the answer. A naive prompt for Lucid, consisting of task-specific instructions, the contents of the original paper [4], a complete grammar in Backus-Naur form (extracted from the publicly available parser) and the full code of the ten applications presented in the paper has a length of 35K tokens; this is more than the web interfaces of both ChatGPT 4 and Gemini Ultra can handle. However, state of the art models have context lengths in the hundreds of thousands, even millions of tokens. We ran a few simple tests using OpenAI's API to query GPT 4. The results seem syntactically coherent but more testing is needed. For the rest of our experiments, we used the web interface of ChatGPT and Gemini (to avoid API costs) and employed grammar prompting. This has the advantage of using a context size six times smaller than the naive prompt, while generating syntactically correct code.

To integrate Lucid into the framework, we wrote a grammar for it in Lark [3] and designed 20 samples for few-shot learning. The samples vary from basic concepts (e.g. defining the type for IPv4 packets), to more complex prompts (i.e. MAC address learning). The resulting grammar-learning prompts are structured as shown in Figure 2. They sum up to around 5.5K tokens; the responses are just several hundred tokens in length.

Besides the program written in the Lucid language, a P4 template is also required, containing basic elements such as parsers and deparser, as well as annotations for where the high-level parts should fit. For the purposes of this paper, we do not require the LLM to generate it.

The model is provided with the full Lucid grammar and several example queries consisting of a specialized grammar (subset of the complete grammar) and the resulting code that implements the query. Lastly, the model is asked to predict a grammar for a given query and generate the program that implements a query conforming to it (Figure 3).

We have tested several simple queries, requiring one-line responses and three complex ones: writing a firewall that can drop packets based on the connection 5-tuple (partial answer in Figure 3), a stateful NAT and a 6to4 tunnel. We noticed that Gemini Ultra sometimes fails to conform to the requirements and does not produce a grammar subset to drive the generated code, which results in an incorrect syntax even for simple examples. ChatGPT-4 is able to follow the guidance of grammar prompting and always produces a grammar for its output; however, this grammar is not always correct, so neither is the accompanying program; however, the errors are not of a conceptual nature and can be easily fixed by a human programmer. Figure 3 shows a fragment of ChatGPT 4's answer for a firewall query.

[3] https://github.com/lark-parser/lark

Prompt to LLM

```
You are an expert data plane programmer, and you need to write
a program for the given language query.
A complete grammar of the language is presented below. First,
you should write grammar rules by choosing from the following
BNF rules.
Then, you should write programs that conform to your predicted
rules.

[BEGIN RULES]
// Full definition of the Lucid grammar
[END RULES]

query: generate a global array "nat_to_port" with NUM_PORTS
values

BNF grammar rules:
prog ::= decls
decls ::= decl
decl ::= GLOBAL cid poly ID ASSIGN cid paren_args SEMI
GLOBAL ::= "global"
cid ::= ID "." cid | ID
ID ::= "Array" | "t" | "nat_to_port" | "create" | "NUM_PORTS"
poly ::= LLEFT polys RRIGHT
LLEFT ::= "<<"
polys ::= size
size ::= NUM
NUM ::= "32"
RRIGHT ::= ">>"
ASSIGN ::= "="
paren_args ::= LPAREN args RPAREN
LPAREN ::= "("
args ::= exp
exp ::= cid
RPAREN ::= ")"
SEMI ::= ";"

program based on the BNF grammar rules:
global Array.t<<32>> nat_to_port  = Array.create(NUM_PORTS);

// other examples

query: <the desired query>
```

FIGURE 2. Sample of prompt engineering for Lucid

```
BNF grammar rules:

prog ::= decls
decls ::= decl decls | decl
decl ::= "const" ty ID ASSIGN exp SEMI | GLOBAL cid poly ID
    ASSIGN cid paren_args SEMI | event_decl SEMI | HANDLE ID
    paramsdef "{" statement "}" | FUN ty ID paramsdef "{"
    statement "}"
ty ::= TINT | TINT single_poly | TBOOL
[...]
FUN ::= "fun"
RETURN ::= "return"

program based on the BNF grammar rules:

const int HASH_SEED = 1234;
const int NUM_PORTS = 6;
global Array.t<<32>> nat_to_ip = Array.create(NUM_PORTS);
global Array.t<<32>> nat_to_port = Array.create(NUM_PORTS);

[...]

handle outside_packet(int<<32>> dst_port) {
   int ip = Array.get(nat_to_ip, dst_port);
   int<<32>> port = Array.get(nat_to_port, dst_port);
   generate outside_continue(ip, port);
}
```

FIGURE 3. ChatGPT response for the query: "a stateful NAT".

Table 1 summarizes ChatGPT's performance on the three complex tasks; for each task, it lists the number of rules in the output grammar ("`A ::= B | C`" counts as two rules), the number of lines of code in the output Lucid program and the number of lines that need to be fixed for the program to compile. The task prompts are admittedly vague, so it is hard to assess the "correctness" of the implementation, which is why we focus on whether programs compile. The generated code is aligned with the input query and could serve well as a starting point for a more particular implementation.

While always generating a grammar fragment, ChatGPT doesn't always respect it. For the tunnel implementation, it uses hexadecimal constants with the "0x" prefix, which are invalid both in the grammar fragment generated and in the complete Lucid grammar; it also uses a bitwise "&" without generating rules for it. Conversely, it includes rules that it does not use; for example it

Task	# of rules	LoC	LoC to fix
Firewall	91	17	2
NAT	97	39	0
6to4 tunnel	74	15	4

TABLE 1. Summary of the results on the three complex tasks

allows the equality operator "==", then generates a program that does not perform any equality check.

5. Discussion and Future Work

The results presented in section "Preliminary results" are an initial move towards the task of translating natural language instructions into HLDPPLs. An immediate next step is mostly quantitative in nature: increasing the number of example programs included in the prompt and developing additional tests to generate snippets and programs of increasing complexity. We would like to perform further experiments both with the baseline prompt (a large dump of all available resources) and with the technique of grammar prompting, which would allow us to better quantify and compare the efficacy of the two methods.

The issues mentioned at the end of section "Preliminary results", concerning constructs that are not valid for the grammar fragment and grammar rules that are unneeded, warrant additional attention. We believe they can be addressed effectively through an automated loop of additional queries that employ the LLM to analyse its answer, identify errors and suggest fixes, using a technique such as Tree of Thoughts [24].

Lucid requires a P4 harness with the basic structure of the data plane, together with annotations for where the functionality described in the high-level language should fit. In our tests, we only focused on the Lucid language itself; in the future, we would like to extend our framework, such that the LLM also generates the P4 template.

Information about Lucid, as well as about all the languages presented in section "High-level data plane programming languages", is very likely part of the dataset on which ChatGPT 4 and Gemini were trained. Our experiments show that grammar prompting yields better results than the naive baseline of relying solely on the LLM's existing knowledge, but such improvements may be possible solely due to the latent knowledge of the LLM. To explore this issue, we must experiment with new languages that are not part of the training dataset. We plan to design a minimalistic HLDPPL, producing all the relevant artifacts (description of the language, formal grammar, code examples) and apply the techniques presented here to evaluate code generation for this new language.

6. Conclusion

In order to address the difficulties of programming in P4 or NPL, several high-level data plane programming languages have been developed. As this

is an active field, neither of these have been standardized or widely-adopted. Thanks to their simple structure, narrow scope and short program length, we believe these are good candidates for the task of employing LLMs to generate languages for which no dataset is available. These code-generators would remove the burden of learning a new, possibly short-lived language and would allow network programmers to quickly develop proof-of-concept applications.

In section "Preliminary results" we presented preliminary results of using ChatGPT 4 and Gemini Ultra in order to generate code for the Lucid language, using the technique of grammar prompting. While Gemini Ultra failed to always produce a subset grammar (and thus grammatically correct code), ChatGPT-4 generated code that only needed minor fixes and could serve as a starting point for further development.

REFERENCES

[1] *E. O. Zaballa and Z. Zhou*, Graph-to-P4: A P4 boilerplate code generator for parse graphs, Symposium on Architectures for Networking and Communications Systems (ANCS), IEEE, 2019, pages 1–2.

[2] *A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever*, Reducing P4 language's voluminosity using higher-level constructs, Proceedings of the 5th International Workshop on P4 in Europe, 2022, pages 19–25.

[3] *C. Györgyi, S. Laki, and S. Schmid*, P4rrot: Generating P4 code for the application layer, ACM SIGCOMM Computer Communication Review, **53**(1), 2023, pages 30–37.

[4] *J. Sonchack, D. Loehr, J. Rexford, and D. Walker*, Lucid: A language for control in the data plane, Proceedings of the 2021 ACM SIGCOMM Conference, 2021, pages 731–747.

[5] *D. K. Loehr*, Lucid: A High-Level, Easy-To-Use Dataplane Programming Language, Princeton, NJ: Princeton University, 2024.

[6] *J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu*, Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs, Proceedings of the ACM SIGCOMM Annual Conference, 2020, pages 435–450.

[7] *P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al.*, P4: Programming protocol-independent packet processors, ACM SIGCOMM Computer Communication Review, **44**(3), 2014, pages 87–95.

[8] *Broadcom*, NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms, 2019, available at `https://nplang.org/`.

[9] *D. Loehr and D. Walker*, Safe, modular packet pipeline programming, Proceedings of the ACM on Programming Languages, **6**(POPL), 2022, pages 1–28.

[10] *F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth*, A survey on data plane programming with P4: Fundamentals, advances, and applied research, Journal of Network and Computer Applications, **212**, 2023, 103561.

[11] *M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al.*, Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374, 2021.

[12] *Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang*, Magicoder: Source code is all you need, arXiv preprint arXiv:2312.02120, 2023.

[13] *A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei et al.*, StarCoder 2 and The Stack v2: The Next Generation, arXiv preprint arXiv:2402.19173, 2024.

[14] *R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim et al.*, Starcoder: May the source be with you!, arXiv preprint arXiv:2305.06161, 2023.

[15] *D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li et al.*, DeepSeek-Coder: When the large language model meets programming–The rise of code intelligence, arXiv preprint arXiv:2401.14196, 2024.

[16] *M. Hogan, S. Landau-Feibish, M. Tahmasbi Arashloo, J. Rexford, D. Walker, and R. Harrison*, Elastic switch programming with p4all, Proceedings of the 19th ACM Workshop on Hot Topics in Networks, 2020, pages 168–174.

[17] *R. Shah, A. Shirke, A. Trehan, M. Vutukuru, and P. Kulkarni*, pcube: Primitives for network data plane programming, 2018 IEEE 26th International Conference on Network Protocols (ICNP), IEEE, 2018, pages 430–435.

[18] *B. Wang, Z. Wang, X. Wang, Y. Cao, R. A. Saurous, and Y. Kim*, Grammar prompting for domain-specific language generation with large language models, Advances in Neural Information Processing Systems, **36**, 2024.

[19] *R. Jain, W. Ni, and J. Sunshine*, Generating domain-specific programs for diagram authoring with large language models, Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, 2023, pages 70–71.

[20] *Gemini Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth et al.*, Gemini: A family of highly capable multimodal models, arXiv preprint arXiv:2312.11805, 2023.

[21] *M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J. B. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser et al.*, Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, arXiv preprint arXiv:2403.05530, 2024.

[22] *P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha*, A systematic survey of prompt engineering in large language models: Techniques and applications, arXiv preprint arXiv:2402.07927, 2024.

[23] *J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, and D. Zhou*, Chain-of-thought prompting elicits reasoning in large language models, Advances in Neural Information Processing Systems, **35**, 2022, pages 24824–24837.

[24] *S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan*, Tree of thoughts: Deliberate problem solving with large language models, Advances in Neural Information Processing Systems, **36**, 2024.

[25] *K. Ye, W. Ni, M. Krieger, D. Ma'ayan, J. Wise, J. Aldrich, J. Sunshine, and K. Crane*, Penrose: from mathematical notation to beautiful diagrams, ACM Transactions on Graphics (TOG), **39**(4), 2020, pages 144–1.

[26] *OpenAI et al.*, GPT-4 Technical Report, arXiv preprint arXiv:2303.08774, 2024.

[27] *M.-V. Dumitru, V.-A. Bădoiu, A. M. Gherghescu, and C. Raiciu*, Generating P4 Dataplanes Using LLMs, 2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR), IEEE, 2024, pages 31–36.