FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	P4 Language Server
Student:	Bc. Ondřej Kvapil
Supervisor:	Ing. Viktor Puš, Ph.D., MBA
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Familiarize yourself with the P4 language, program parsing with error recovery, and the domain of language servers.
Make a language server requirement analysis, including some unique aspects of P4: Packet header parsing constructs, support for various target architectures, and frequent use of compiler-specific or target-specific pragmas.
Design a P4 language server that provides autocompletion, go-to-definition and other common features, and can be integrated in VSCode.
Implement the language server and show its integration to VSCode.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 24 November 2022 in Prague.

Master’s thesis

P4 LANGUAGE SERVER

Bc. Ondřej Kvapil

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: Ing. Viktor Puš, Ph.D. MBA
2023-05-04

Citation of this thesis: Kvapil Ondřej. *P4 Language Server.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank Timothy Roberts and Adam Reynolds for their kindness and support of my internship during challenging times at Intel. Thanks to my supervisor, Viktor Puš, without whose enthusiasm I would never have applied for the internship. And, last but not least, thanks to my family and friends for making my life better than I deserve.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on 2023-05-04 . . . .

Abstract

The P4 language is used for configuring programmable network processors. Despite its popularity in the Software Defined Networking field, it suffers from a lack of modern developer tooling. In this thesis, we design and implement a language server for the P4 language, which provides support for lexical analysis, preprocessing, and parsing. On these foundations, we build support for autocompletion, error reporting, and navigation in the source code. The language server is implemented in the Rust language and integrated into the Visual Studio Code development environment.

Keywords language server protocol, language server, parsing, semantic analysis, P4, SDN, developer tools

Abstrakt

Jazyk P4 je používán pro konfiguraci programovatelných síťových procesorů. Navzdory své popularitě v odvětví Software Defined Networking ale zaostává co se podpory programátora týče. V této práci navrhujeme a implementujeme language server pro jazyk P4, který poskytuje podporu pro lexikální analýzu, preprocessing a syntaktickou analýzu. Na těchto základech stavíme automatické doplňování, reporting chyb a navigaci ve zdrojovém kódu. Language server je implementován v jazyce Rust a integrován do vývojového prostředí Visual Studio Code.

Klíčová slova language server protocol, language server, syntaktická analýza, sémantická analýza, P4, SDN, vývojářské nástroje

Summary

Introduction

We open with an overview of the networking setting that led to the development of the P4 language. After a brief survey of existing tooling, we decide to implement our own.

The P4 Language

Next, we delve into the details of P4 to better understand the commonalities and differences between it and conventional programming languages.

Language Server Architecture

We explore the language server protocol and existing LSP-compliant tools. We also survey architectural themes in conventional compilers and pluck a few ripe ideas for our own use later.

Design

In this chapter, we detail the design of our language server. We go over the key abstractions and algorithms, motivating our design decisions along the way.

Results

The final chapter closes with lessons learned, our implementation's strengths, and areas where it falls short. We conclude with a discussion of future work.

Acronyms

API application programming interface.

ASIC application-specific integrated circuit.

AST abstract syntax tree.

CFG context-free grammar.

CPU central processing unit.

CST concrete syntax tree.

DSL domain-specific language.

FPGA field-programmable gate array.

FSM finite state machine.

IR intermediate representation.

LLVM Low-Level Virtual Machine.

LSP Language Server Protocol.

LTO link-time optimization.

ONF Open Networking Foundation.

P4 Programming Protocol-independent Packet Processors.

PEG parsing expression grammar.

REPL read-eval-print loop.

SDN software-defined networking.

XML eXtensible Markup Language.

YACC Yet Another Compiler-Compiler.

Introduction

…in which we get to know the context that gave rise to the P4 language and the challenges we set out to overcome.

Programming Protocol-independent Packet Processors (P4) is a domain-specific language for programming network switches. Its release started a shift in the field of software-defined networking (SDN) which, up to that point, relied heavily on fixed-function hardware for high-performance networking applications. Similarly to how the C language became a de facto portable assembler, abstracting over the details of each microprocessor, P4 abstracts over the details of network processors by presenting a deeply customizable interface shared by networking software and hardware alike.

Unlike previous approaches in SDN, P4 does not have built-in support for common network protocols like TCP, IP, or Ethernet. Instead, it provides protocol-independent constructs that users can leverage in order to define arbitrary protocols and instruct a flexible network switch on how to handle them. This programmability lets a network engineer specify the configuration and packet processing steps of a router architecture independently of the underlying machinery.

The newfound flexibility of network processor programming that P4 enables does not get a chance to shine on traditional network hardware, which is built for a predetermined set of protocols and processing functions. The real power of P4 is unlocked by *programmable* network processors, which can be reconfigured to support novel protocols and forwarding setups. The commercial sector answered the call for such hardware, for example in Intel's Tofino line of chips.[1]

P4 became wildly popular in SDN since its introduction in 2014[1], sparking both research and commercial applications. Three years later, P4 underwent a major redesign[2], which simplified the syntax and removed special-purpose language constructs in favour of more general solutions.[2] The redesigned language is known as $P4_{16}$ [4] and its specification has since received more incremental updates.

A Growing Community

P4 gave rise to a varied ecosystem of commercial offerings of both hardware and software. It spawned several academic projects that investigated its semantics[5], allocation to heterogeneous hardware[6], open-source network testing[7], and sketch-based monitoring[8], among many others[9]. The Open Networking Foundation (ONF) maintains[10] an open-source reference implementation of a $P4_{16}$[3] com-

[1]`https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html`

[2]Specifically, features like counters and checksum units were replaced by `externs`, a universal construct for specifying additional hardware capabilities not explicitly covered by the core syntax. The language shrank from over 70 to less than 40 keywords[3].

[3]Although primarily developed for $P4_{16}$ since the revision of the language, it is also capable of migrating $P4_{14}$ programs to $P4_{16}$ or directly compiling them.

piler frontend and midend, along with several backends:

p4c-bm2-ss Targets a sample software switch for testing purposes.

p4c-dpdk Targets the DPDK software switch (SWX) pipeline[11].

p4c-ebpf Generates C code which can be compiled to eBPF and then loaded in the Linux kernel.

p4test A source-to-source P4 translator for testing, learning compiler internals and debugging.

p4c-graphs Generates visual representations of P4 programs.

p4c-ubpf Generates eBPF code that runs in user-space.

p4tools A platform for P4 test utilities, includes a test-case generator for P4 programs.

All of these components are open source. The frontend and midend of the reference compiler provide a foundation for hardware vendors to support P4 in their products and serve the community in resolving discrepancies between commercial compilers and the language specification.

Despite this growth, P4 has only mediocre support for real-time feedback to the programmer – the vast majority of open and commercial tools rely on compiler output to provide semantic insight into a P4 program[12], or, as evidenced by the open backends, are parts of the compiler proper. This is a problem, because the compiler was not designed for interactive use. Compilations of complex programs can take over an hour to complete. Long feedback loops hamper development. What's more, information provided by the compiler frontend is very limited – the compiler typically reports at most one error message with very little (if any) explanation. The lack of an integration between the compiler and development environments is an obstacle to new users and a neglected area of P4 developer experience. Even an ergonomic presentation of error and warning messages would be a large improvement.

We can do far better, as interactive editing support for conventional programming languages clearly demonstrates. Integrated developer environments provide many features that P4 programmers can only dream of, including autocompletion, documentation pop-ups, real-time diagnostics, code navigation, automatic formatting, and refactoring. These features are not only convenient, but also improve the quality of software by reducing the cognitive load on the programmer.

Language Servers to the Rescue

Over the last decade, *language servers* became a popular architecture[13] for providing semantically informed editing features in integrated developer environments and lightweight source code editors alike. Although examples of language server -like tools predate their standardization[14], a major milestone in their development was the introduction of the Language Server Protocol (LSP). The support of LSP in Visual Studio Code seeded an ever-growing environment of cross-platform tooling for a variety of programming, configuration, specification, and markup languages. Their success can be attributed in part to the investment by Microsoft. However, LSP as a standard has much technical merit.

Before the introduction of a common communication interface between language-specific tooling and a given source code editor, implementors had to create and maintain extensions for any number of different environments, often in several programming languages. For example, take an editor plugin providing smart completion and "go to definition" features for the Python programming language. Suppose the author aims to extend Vim, Emacs, and IntelliJ IDEA. They would therefore have to reimplement the given functionality in vimscript, Emacs Lisp, and a JVM-based language. Even though recent innovations in classical text editors[4] have partly moved away from using their own DSL's, the author is still burdened with three times the implementation work, as every development environment API is different.

[4]One such project is a fork and refactoring of Vi IMproved, *Neovim*: `https://github.com/neovim/neovim`.

The solution to this problem is to split editor extensions according to the client-server model. The major implementation work for domain-specific functionality resides on the server side, whereas the client is only a relatively thin wrapper that adapts a given editor's API. Implementors can build many thin clients for different development environments that all connect to the same *language server*. LSP's raison d'être is support of this model in Visual Studio Code.

Codenvy, Microsoft, and Red Hat collaborated on standardizing the protocol to enable other tools to benefit from a shared ecosystem[15, 16].

Combining the two

With these developments in mind, we would like to embark on the journey of supporting the P4 programmer in their development efforts. By building a language server, we aim to bridge the tooling gap and bring real-time feedback into the P4 programmer's development environment.

A note about authorship

This work concerns the P4 Analyzer project, designed and developed primarily by Timothy Roberts and me,[5] as a part of my internship at Intel. Tim was largely in charge of managing the integration into the Visual Studio Code editor, the server's compliance with the Language Server Protocol, and the networking aspects of the project.[6] I was responsible for the implementation of the analyzer core, which involved the lexing, preprocessing, and parsing of P4 code, designing the incremental computation architecture of the project, the interfaces the analyzer core exposes, and the abstractions used within.

To avoid problems of authorship, this thesis focuses on the analyzer core, which is an isolated module that I was largely in charge of. I can honestly claim that, at the time of writing, all the code in the `analyzer-core` crate is my own. On the other hand, P4 Analyzer is a large project with even larger ambitions. Tim and I were both involved in large design decisions.

[5] With recent help from Andrew Foot and Yusuf Adam.

[6] This involved unhealthy doses of both asynchronous code and Rust borrow checker errors, for which I can only offer sincere thanks and condolences. Tim somehow managed to stay optimistic and kind throughout. He rocks!

Chapter 1

The P4 Language

... in which we delve into the syntax and semantics of P4, explain its use cases, and discuss the differences to conventional programming languages.

The expressive power of a programming language arises from its strictures and not *from its affordances.*

– Robert Harper, 2019 [17]

One of the original motivations behind P4 was the need for restriction. While other domain-specific languages, such as Click[18], existed at the time, their embeddings within general-purpose programming languages made it difficult to analyze data dependencies crucial for scheduling parallel execution. The expressiveness of a language complicates its efficient compilation.

Rather than embedding P4 in an existing language, these considerations motivated a clean-slate design. However, to understand why is packet processing any different from tasks suited to general-purpose programming languages, we first need to familiarise ourselves with the switching architecture.

1.1 What's in a switch

[Packet switching is] *the routing and transferring of data by means of addressed packets so that a channel is occupied during the transmission of the packet only, and upon completion of the transmission the channel is made available for the transfer of other traffic.*

– Fiber optics standard dictionary [19]

In the following text, we choose to use *switch* to mean a general packet switching device. Such a device could be a dedicated piece of hardware or a software solution running on a general-purpose computer.

Traditionally, these devices were implemented by fixed-function hardware. Various networking protocols were built directly into the circuitry, which made these switches efficient, but inflexible. It is impossible to reconfigure a fixed-function application-specific integrated circuit (ASIC) to process a protocol it was not explicitly designed for in advance. If a new, backward-incompatible version of a given protocol emerges, or if a hardware error is found in the chip, the network administrators need to perform a costly hardware replacement in order to support or circumvent it, respectively. Moreover, fixed-function hardware design is a lengthy and resource-intensive process. It may take several years before an updated fixed-function chip hits the market.

The innovation of recent years is the introduction of *programmable* network processors, which can change the set of supported protocols on the fly. These are similar to FPGAs in their reconfigurability,

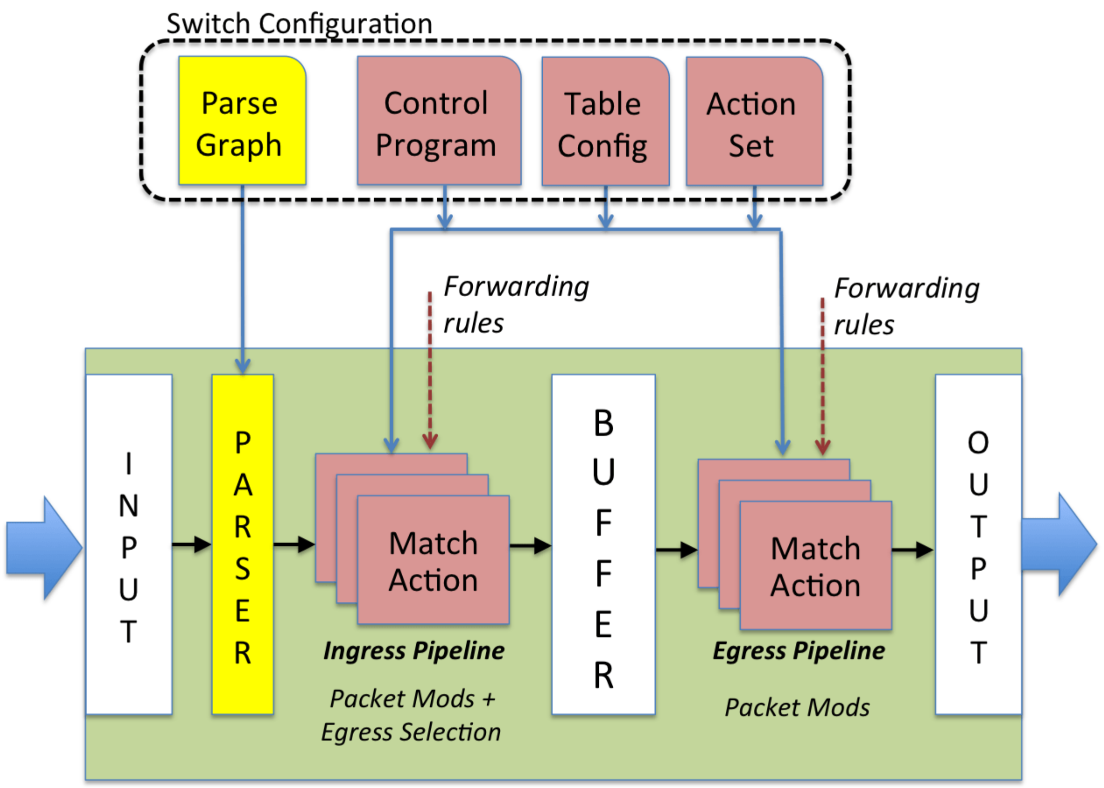


Figure 1.1 The original $P4_{14}$ abstract forwarding model, taken from [1].

but specialised to packet switching and routing, which makes them more efficient. A programmable network switch has typically no prior knowledge of networking protocols but contains efficient circuitry for parsing and pattern-matching in order to support arbitrary[1] protocols uploaded to the chip as microcode. While programmable networking hardware does incur a penalty for reconfigurability, when it comes to efficiency, it sits between fixed-function devices and completely general-purpose solutions.

Other implementations of packet switching are also common. The already mentioned field-programmable gate arrays can be programmed to simulate programmable or fixed-function networking hardware, and thus allow an even higher degree of flexibility. Naturally, FPGAs are less efficient than the circuits they simulate. Finally, there are software switches, programs for widespread processor architectures and operating systems, such as x86 and Linux. A software switch represents the peak of flexibility, programmability, and requires no special hardware other than what is already commonly present in conventional computers. Purely software-based solutions cannot compete in energy efficiency with any of the other approaches, but are often useful for testing and in small-scale networks.

To support network configurations regardless of their physical implementation, the original P4 paper defines the target-independent *abstract forwarding model*, outlined in Figure 1.1. There is a key conceptual split in the architecture, common in packet processing in general, of the *data plane* and the *control plane*. We will meet these terms over and over, so let us define them here.

Data plane is the part of the switch responsible for packet processing, i.e. all the hard work of manipulating bits in the arriving packets and choosing where to send the packets next. It is the part that is programmable in P4, though it is unchanging in traditional packet processing approaches. The data plane is responsible for packet parsing, pattern-matching, and rewriting.

Control plane is the part of the switch responsible for configuration. It is not programmable in P4, but P4 programs implicitly define a software interface for it. The control plane is responsible for

[1]To some degree of complexity supported by the circuit.

uploading forwarding rules to the data plane, extracting data from special fixed-function devices (such as throughput counters), and reconfiguring the flexible parts of the P4 program at runtime. A control plane program could be a simple Python script or a C++ application.

Parallels to general-purpose computing

It is helpful to think of the data plane as an infrequently changing program running on a network chip.[2] It defines the protocols the switch can handle and the stages of packet processing the switch performs, but not the concrete forwarding rules, which can change at runtime.

In contrast, the control plane typically runs on a general-purpose computer and uses the data plane as a coprocessor. It uploads forwarding rules (e.g. which subnet to forward an arriving packet to) to coprocessor-specific memory. The interface between the control plane and the data plane depends entirely on the P4 program that the data plane runs, and relies on a hardware link between the two, such as PCI Express.

The abstract forwarding model assumes an end-to-end data plane pipeline split into *ingress* and *egress* parts. An arriving packet is first parsed to recognize the headers present therein. These headers then travel through the pipeline’s *match-action units*. Each match-action unit performs limited pattern-matching and rewriting on the parsed packet header – complex P4 code can map to a sequence of several match-action units. This part of the pipeline is partially configured at runtime by the control plane, usually by forwarding rules defined in software.

To relate match-action units to conventional computing, they are similar to cycles in a CPU pipeline. In fact, the entire abstract forwarding model can be thought of as the unrolling of a limited number of cycles of a CPU pipeline. This corresponds well to P4’s lack of looping constructs.

Packet headers (supplemental data placed at the beginning of a packet) are a central concept in packet processing, and therefore also in the P4 language. The abstract forwarding model assumes that the packet is split into two parts: the header and the payload. Only the header proceeds into the match-action pipeline.[3] The model assumes that the payload is handled separately by the device and is thus not available for pattern-matching.

Keeping it high-level

While the presented forwarding model gives a good overview, it is not the full picture. A notable omission is packet output. If the P4 program implements a switch, how does the packet exit the pipeline? How does it select the output port? And how does the switch filter out packets that should not be forwarded?

These questions touch on the issue of supporting use-cases and functionality too granular to be part of the abstract forwarding model. While they may seem like blatant oversights, these omissions of packet processing details are deliberate. P4 supports both output port selection and packet filtering via packet *metadata*. Intrinsic metadata is a target-dependent data structure that stores information about a given packet and is available to the P4 program at every stage of the pipeline, including parsing. Metadata may include the input port and other information provided by the device. Crucially, the metadata is typically mutable, and contains fields and flags for the output port, recirculation, packet dropping, and other features.

The final stage of the processing pipeline in programmable network hardware involves *deparsing*, i.e. serializing the packet header back into a bitstream. As we will see on page 20, deparsing is a case of a more general control mechanism that does not need special treatment in the abstract model.

The design philosophy of reliance on general-purpose constructs, such as metadata or `externs` (which we will see later), underpins much of the P4 language, and its effects can be seen in many facets of the $P4_{16}$ revision. It makes the language both simpler, as it avoids accumulating narrow use-case features, and more flexible, since it generalizes to platforms to which these concepts do not translate.

[2] Even though, as discussed, it could reside entirely in software.

[3] Although what part of the packet actually is “the header” is entirely up to the P4 program.

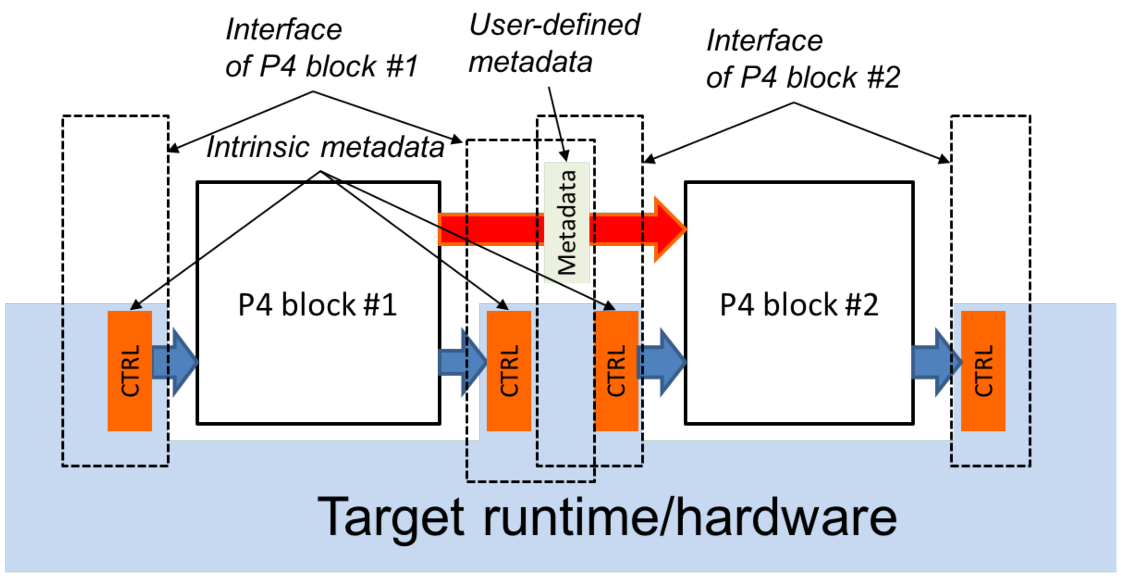


Figure 1.2 P4$_{16}$ program interfaces for an abstract architecture with two programmable blocks, taken from [4].

1.2 A tour of P4$_{16}$

This section largely mirrors the P4$_{16}$ language specification. It does not cover the entire language, but should serve as a decent introduction to its most important features, while attempting to draw some parallels to conventional programming languages and terms of general-purpose computing.

A P4 program specifies a mapping of vectors of bits – a bitvector endomorphism. Every P4 program terminates; the language has no looping constructs and no recursion, a compiler can thus determine the precise maximum runtime of a program statically.

P4 is therefore not a programming language for von Neumann architectures. Instead, its abstract model assumes the target machine to be some sort of network processor with programmable blocks embedded in a static pipeline. The overhaul into P4$_{16}$ slightly redefined the abstract forwarding model, as can be seen in Figure 1.2. This diagram focuses on the interfaces of the P4 program and points out metadata, which deserves a special mention.

Practical implementations of P4-configurable network hardware often need to track information not included in the packet headers. This is especially the case for devices with multiple pipelines, such as the one outlined in Figure 1.2, where the partial computation results from one pipeline need to travel to the next in order to continue processing. The nature of passing such metadata is heavily target-dependent, however. Some architectures may include explicit side-channels for metadata and reflect this in the interfaces they provide to user code, while other targets demand that the user includes metadata explicitly in the packet headers. In other words, the user metadata channel in P4$_{16}$'s abstract forwarding model is purely conceptual.

1.2.1 Syntax and semantics

P4$_{16}$ syntax is reminiscent of imperative programming languages in the C family. It uses prefix notation for typed bindings, braces for lexical scoping blocks, and semicolons to separate statements. Its expression syntax is very similar to C as well. However, instead of functions or procedures, the dominant top level constructs for executable code are control blocks and parsers, neither of which has a close relative in the world of conventional general-purpose computing.

The semantics of P4$_{16}$ is defined entirely in terms of abstract machines executing imperative code. A conforming compiler is free to rewrite the P4_{16} program as long as it maintains the observable behaviour of all abstract machines involved. Unfortunately, the specification gives no formal treatment of these machines; they are described only in natural language and pseudocode.

The authors of the specification acknowledge that undefined behaviour has been the cause of many problems in conventional languages and deliberately try to avoid it. However, several constructs in the language are still underspecified and users may run into undefined behaviour, for example by accessing the field of an invalid header.

Types

P4_{16} is a statically typed language with a type system designed for precise bit-level control of data layout. There are 9 primitive types:

- The `void` type. In a classic case of ignorance of basic type theory, the specification claims that `void` has no values, even though calls to methods returning `void` happily return without terminating execution. Clearly, `void` corresponds to the unit type with exactly one value, even though P4_{16} does not let the user name it.
- The `error` type, which is used to convey errors in a target-independent, compiler-managed way.
- The `string` type, which can be used only for compile-time constant string values.
- The `match_kind` type, which is used for describing the implementation of table lookups.
- `bool`, which represents Boolean values.
- `int`, which represents arbitrary-sized constant integer values, and only exists at compile time.
- Bit-strings of fixed width, denoted by `bit<>`.
- Fixed-width signed integers represented using two's complement `int<>`.
- Bit-strings of dynamically-computed width with a fixed maximum width `varbit<>`.

The inclusion of a boolean type that is semantically distinct from a bit-string or an integer frees the design to give bitwise operators (`&`, `|`, `^`) higher precedence than relation operators (<, >, <=, >=). The type system catches attempts to use bitwise operators on boolean values and logical operators on bit-strings.

Types can be combined to form *derived types* by means of 12 type constructors:

- `enum`
- `header`, conceptually a `struct` with an implicit bit indicating validity. The validity of a header is determined by a parser: a header is valid if the parser that loads it ends in the accepting state.
- header stacks
- `struct`, as known from C.
- `header_union`
- `tuple`
- type specialization
- `extern`
- `parser`

Code listing 1.1 An `extern` object specifying the interface to the target's checksum unit.

```
extern Checksum16 {
    Checksum16();                // constructor
    void clear();                // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

- `control`
- `package`
- `type`, which the specification does not list under derived types. It serves the role of Haskell's `newtype` construct, i.e. it introduces a new name for an existing type. In P4$_{16}$, this means that conversions between the new type and its right-hand side require explicit casts.

In a questionable decision to assert the design of compiler internals, the specification also includes synthesized set and function types, which we will not discuss here.

`extern` objects and functions

Before diving into the bulk of the syntactical forms that dominate user-written P4 code, we need to introduce the general concept of `extern`s.

`extern` objects and functions describe interfaces to facilities provided by the architecture, as well as certain built-in language constructs.[4] For example, the `extern` object in Listing 1.1 allows P4 code to utilize a fixed-function checksum unit provided by the target. The object specifies no implementation for the listed constructors and methods, rather, the target platform implements the asserted functionality intrinsically, e.g. by fixed-function circuitry.

To invoke a method of the checksum unit, the user needs first to *instantiate* the `extern` object.

Instantiation is the declaration of a type with a constructor. This syntactic form is shared among `extern` objects, `control` blocks, `parser`s, and `package`s alike. The effect of an instantiation is to allocate the corresponding object, binding it to the specified name.

The compiler is in charge of mapping instances of `extern`s to the target architecture (i.e. performing the allocation of resources on the target). If it does not find a mapping, either because the target does not support the `extern` or does not have the resources to fit all its instances, the compilation fails with an error.

An `extern` object may contain `abstract` methods. Methods with this modifier require the `extern` instantiation to provide an implementation for them.

`extern` objects and functions were already present in P4$_{14}$ version 1.1, but P4$_{16}$ fully embraced these constructs. This helped to both simplify and generalize the language and led to the elimination of several less general concepts. For example, P4$_{14}$ had dedicated syntax for checksum units.

Imports and the core library

Although not a language construct in its own right, the P4 core library is a set of common programming definitions distributed together with language tooling. Unlike preludes in certain programming languages, the core library is not automatically included in every P4 program and must be imported

[4]Such as the `packet_in` intrinsic in Listing 1.4

Code listing 1.2 A structured annotation on a table.

```
@MyAnnotation[1, 2, 3]
table myTable { /* ... */ }
```

explicitly with the preprocessor directive `#include <core.p4>`. P4$_{16}$ does not have a module system and relies solely on the C preprocessor (or a sufficiently capable subset thereof) for code reuse.

Another typical inclusion in a P4 source file is that of the target-specific *architecture description*. This is a library P4 file with at least a `package` declaration. The `package` construct is very similar to a `parser`, except that it has no body and no runtime behaviour. Instead, a top-level `package` instance matching the architecture's declaration serves as a conceptual entry point for the program, defining the used parser, control block, and `extern` instances by passing them to the `package` constructor.

L-values

P4$_{16}$ has a concept of *l-values*, reminiscent of C. An l-value is an expression that denotes a memory cell that can be assigned to. These include

- variables; identifiers of a base or derived type,
- fields of structs, headers, and header unions, and
- the results of bit-slice operators (e.g. `pkt[0:7]`).

Annotations

Almost all syntactical constructs in P4$_{16}$ can be modified by *annotations* (see Listing 1.2). These come in two flavours, structured and unstructured, depending on the restrictions imposed on their arguments. Unstructured annotations can take arbitrary token streams as their arguments, so long as the parentheses in these token streams are balanced.

Very few annotations are built into the language, so we will not discuss them in depth, but they provide an important avenue for target-specific extension of the language.

Functions and parametrized code

Functions in P4 come in several flavours. The first kind is known as *function declarations* and is perhaps the closest relative of procedures in conventional imperative programming languages. The major difference is that all parameters to a P4 function need to include a *direction*.

parameter direction is either `in`, `out`, or `inout`. `in` indicates a read-only parameter with a defined value, `out` indicates a read-write parameter whose value is initially undefined upon entering the function body, and `inout` indicates a read-write parameter with a defined value.

Only l-values can be passed as `out` and `inout` parameters. The execution of the function call can change the value of an `out` or `inout` parameter's corresponding l-value. Note that P4 functions can combine `out` parameters and return values.

Parameter directions are P4's way of introducing a limited form of references, themselves a principled approach to pointers. Parametrized P4 code follows a copy in / copy out calling convention, which makes functions easy to reason about and limits side effecting behaviour of `extern` methods. We will discuss this later on page 21.

Code listing 1.3 The conceptual model of the state of a P4 parser.

```
ParserModel {
    error       parseError;
    onPacketArrival(packet p) {
        ParserModel.parseError = error.NoError;
        goto start;
    }
}
```

Code listing 1.4 The intrinsic `extern` that facilitates data extraction.

```
extern packet_in {
    void extract<T>(out T headerLvalue);
    void extract<T>(out T varSizeHeader, in bit<32> varFieldSizeBits);
    T lookahead<T>();
    bit<32> length();  // May be unavailable in some architectures
    void advance(bit<32> bits);
}
```

Parsers

Another construct for parametrized code are *parser declarations.* A P4 parser describes a finite state machine whose job is to recognize valid packets and load their headers into the storage facilities of the target. Parser declarations define all the states of the FSM, except the implicit built-in `accept` and `reject` states. Each parser has to contain at least the initial `start` state. During parsing, the parser can manipulate local state in the form of variables and `extern` instantiations. These are defined directly in the parser declaration block, before listing any states, meaning that the type of this local context is statically known and independent of the state the parser may appear in at runtime. In other words, parser declarations follow lexical scoping.

States can introduce lexically-scoped local variables, but not additional `extern`s. Other statements like conditionals, assignments, or method calls are also allowed. The specification explicitly points out that specific target architectures can place restrictions on the set of constructs and operations a programmer can use within a parser.

$P4_{16}$ parser declarations unrestricted by the target architecture may appear very similar to conventional imperative code, and reminiscent of other function-like constructs the language offers. The crucial parser-only features are data extraction, pattern-matching, error handling, and state transitions.

Data extraction moves information from the input packet into memory available for pattern-matching further down the pipeline, typically registers or similar containers. To facilitate this operation, the P4 core library contains an intrinsic[5] `extern` definition called `packet_in` which represents incoming packets. This special `extern` cannot be manually instantiated, but it is instantiated implicitly for every parser parameter of type `packet_in`. This allows parser states to invoke the methods of the `extern`, shown in Listing 1.4.

Parser abstract machine

A parser semantically manipulates a `ParserModel` data structure.

The meaning of `accept` and `reject` states is architecture-dependent. For example, a rejected packet may be dropped or passed to the next block of the processing pipeline.

Parser transitions are analogous to `goto` statements or parameter-less tail calls.

[5]By an "intrinsic," we mean a language feature that somehow invokes a mechanism native to the host architecture, one that could not be defined in the language itself, or not as efficiently, and needs to be hard-coded in the compiler.

Code listing 1.5 An example of data extraction in a P4_{16} parser.

```
struct Result { Ethernet_h ethernet;  /* more fields omitted */ }
parser P(packet_in b, out Result r) {
    state start {
        b.extract(r.ethernet);
    }
}
```

Code listing 1.6 The interface of the `verify` built-in.

```
extern void verify(in bool condition, in error err);
```

An example of extraction of fixed-width data can be seen in Listing 1.5.

Data extraction and computation within parser states subsumes the stateful part of a finite state machine. Because isolated states would not be very useful on their own, P4 parsers can specify state transitions using the `transition` statement. The target of the transition can be either given statically or computed. A missing `transition` statement at the end of a state block implies a transition into the `reject` state.

`select` expressions

Another parser-specific construct facilitates the computation of transition targets by pattern-matching on extracted data. The `select` expression tries to match a value against a number of patterns and evaluates to a state, if successful. Otherwise, it triggers a runtime error with code `error.NoMatch`. The patterns of a `select` expression can contain integer literals, bit masks, ranges, don't-care values, tuples (when matching on multiple values simultaneously), or, on some architectures, *parser value sets* specified at runtime by the control plane.

While error-handling already happens implicitly in `select` expressions, the `verify` statement, defined in Listing 1.6, allows the programmer to ergonomically check arbitrary assertions about the parsed data. Passing `false` as the first argument to `verify` immediately transitions to the `reject` state, setting the parser error to the code given as the second argument. Otherwise, execution proceeds with the next statement.

Control blocks

While parsers execute at the very frontier of the pipeline, the bulk of packet processing happens in the previously mentioned match-action units. P4 has another parametrized construct for expressing entire sequences of pattern-matching and rewriting steps: `control` blocks. An example of one is shown in Listing 1.10.

A control block has a name and can take type and value parameters. The body of a control block begins with declarations of constants, variables, and instantiations. It follows with definitions of *`actions`*.

Code listing 1.7 A function declaration in P4_{16}.

```
bit<32> max(in bit<32> left, in bit<32> right) {
    return (left > right) ? left : right;
}
```

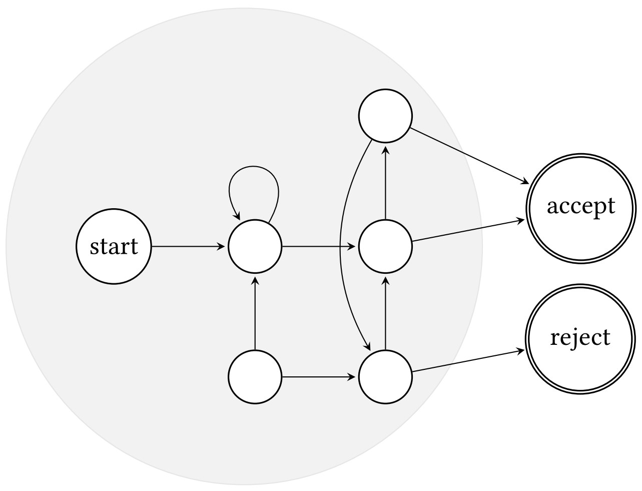


Figure 1.3 An abstract overview of a P4 parser. The states inside the grey circle are accessible to user code.

Code listing 1.8 Parser value sets, an advanced P4 feature for changing parser behaviour at runtime from the control plane.

```
struct vsk_t {
    @match(ternary)
    bit<16> port;
}
value_set<vsk_t>(4) pvs;
select (p.tcp.port) {
    pvs: runtime_defined_port;
    _: other_port;
}
```

Code listing 1.9 Parsers can instantiate and invoke other parsers as subroutines.

```
parser ipv4_parser(packet_in packet, out IPv4 ipv4) { /* ... */ }
parser main_parser(packet_in packet, out Headers h) {
    ipv4_parser() instance;

    state subroutine {
        instance.apply(packet, h.ipv4);
        // execution proceeds if the sub-parser
        // ends in accept state
        transition accept;
    }
}
```

Code listing 1.10 A P4$_{16}$ control block.

```
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }
    action set_nhop(bit<48> nhop_dmac, bit<32> nhop_ipv4, bit<9> port) {
        hdr.ethernet.dstAddr = nhop_dmac;
        hdr.ipv4.dstAddr = nhop_ipv4;
        standard_metadata.egress_spec = port;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }
    table ecmp_nhop {
        key = {
            meta.ecmp_select: exact;
        }
        actions = {
            drop;
            set_nhop;
        }
        size = 2;
    }
    apply {
        if (hdr.ipv4.isValid() && hdr.ipv4.ttl > 0) {
            ecmp_nhop.apply();
        } else {
            drop();
        }
    }
}
```

Code listing 1.11 A P4$_{16}$ action.

```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

Actions

A P4 action, seen in Listing 1.11, is a piece of code that directly manipulates packet headers. At a first glance, actions should be immediately familiar to imperative programmers, since they resemble functions with no return value. Their bodies are each comprised of a series of statements, which are executed sequentially,[6] with the restriction that actions cannot contain `table`, `control`, or `parser` applications.

Additionally, there are two important restrictions on action parameter lists:

1. Parameters with no direction must all come at the end of the parameter list. These directionless parameters indicate *action data*, and can either be provided by the program, just like `in` parameters, or by the control plane software.

2. Action parameters cannot have `extern` types.

Tables

The second major P4 construct that appears exclusively within control blocks are *tables*. Whereas an action specifies the operations performed on packet headers when pattern-matching succeeds, a table informs a match-action unit of the target how to perform the pattern-matching and what actions to invoke.

Analogy to functional programming

Functional programmers will note that tables and actions together seem like a low-level decomposition of pattern-matching from Haskell or Scala. This is a fairly accurate analogy, except that, whereas pattern-matching constructs in a programming language are typically fixed at compilation time, P4 tables are configurable at runtime by the control plane. Furthermore, P4's pattern-matching operates at the bit level and offers certain string-like facilities.

A `table` declaration is simultaneously an instantiation in the enclosing control block. The declaration lists various properties of the table, given as key-value pairs. It has at least the mandatory `key` and `actions` properties, which specify an expression used for computing the lookup key and the set of actions the table can invoke, respectively. A table declaration can optionally also include `default_action`, `entries`, and/or `size` properties, which specify the action invoked when no other actions match,[7] a predefined set of entries for the table, and the desired size for the table. Compilers can choose to extend this standard set of properties with additional target-specific key-value pairs.

The programmer can optionally declare table properties as `const`. This keyword ensures that the control plane cannot change the property's value at runtime. Somewhat confusingly, some properties are implicitly `const`, including the standard properties `key`, `actions`, and `size`. On these, the `const` keyword has no effect.

`key` The `key` table property specifies the scrutinee of pattern-matching. Grammatically, the key is a sequence of rows where each row contains the expression to match on, the kind of matching to perform, and a list of optional annotations.

The possible kinds of pattern-matching are `exact`, `ternary`, and `lpm`, all defined as part of the `match_kind` construct in the core library. A `match_kind` behaves like an enumeration, it lists a number of mutually-exclusive variants. The semantics of the `match_kind` variants is not given, it depends on the compiler and target architecture.

[6] At least from the perspective of the programmer.
[7] The `default_action` property defaults to the built-in `NoAction` when not specified.

Code listing 1.12 Use of the actions table property in P4 $_{16}$.

```
action a(in bit<32> x) { /* ... */ }
bit<32> z;
action b(inout bit<32> x, bit<8> data) { /* ... */ }
table t {
    actions = {
        // a;
        //  - illegal, x parameter must be bound
        a(5);  // binding a's parameter x to 5
        b(z);  // binding b's parameter x to z
        // b(z, 3);
        //       - illegal, cannot bind directionless data parameter
        // b();
        //  -- illegal, x parameter must be bound
        // a(table2.apply().hit ? 5 : 3);
        //   -------------- illegal, cannot apply a table here
    }
}
```

Key elements can be renamed with optional `@name` annotations to give them readable names in the control plane API.

`actions` The `actions` table property lists all the actions a table can invoke at runtime, separated by semicolons.

The specification mandates that action names of actions in the `actions` list have to be distinct. That is, two actions of the same name cannot appear in the `actions` list, even if they come from different scopes and could be disambiguated with fully qualified paths in other contexts.

An overview of the valid and illegal usages of the `actions` table property is given in Listing 1.12. The examples highlight the distinction of action parameters with and without direction. The directionless parameters, also known as *action data*, are used to communicate data between the control plane and the data plane at runtime, and therefore cannot be bound in the P4 program. Haskell enthusiasts may note that the distinction between compile time and runtime -bound values is somewhat reminiscent of second-class functions.

`default_action` The `default_action` table property specifies the action invoked when no other actions match. The value of this property is an expression that invokes the action in question, binding parameters with directions similarly to an `actions` list entry. It introduces an interesting redundancy in the P4$_{16}$ language, since the programmer needs to also include the default action in the `actions` list. Moreover, the default action and the entry in the list have to be syntactically identical, except for the directionless parameters. Since there is no action data for the default action at runtime, the `default_action` property has to specify them all. The directionless parameters are evaluated at compile time.

A table without a `default_action` property that does not match a given packet has no impact on packet processing, it is effectively skipped.

`entries` The optional `entries` property preconfigures a table's entries. This can serve to either initialize the table or, with the help of `const`, turn it into a static construct that cannot be modified on the control plane side.

`size` The optional `size` property indicates the number of table entries the table should support at runtime. It is a compile-time known integer. The interpretation of `size` depends largely on details of the target architecture, casting some doubt on why it was included in the core language. Some

■ **Code listing 1.13** The synthetic P4 code generated for a table.

```
enum action_list(T) {
    // the name of each enum variant is the name of an action
    action_1,
    action_2,
    ...
    action_n
}
struct apply_result(T) {
    bool hit;
    bool miss;
    action_list(T) action_run;
}
```

targets may require every table to specify a size, others may use it merely as a hint during dynamic resource allocation, others still may guarantee that a successfully compiled table can fit at least `size` entries. In general, none of these are the case.

Tables can specify additional properties outside of the base set provided by the $P4_{16}$ specification.[8]

Control block semantics

Invoking actions: actions can be invoked either implicitly or explicitly. Implicit invocations come from tables during match-action processing. Explicit invocations are calls from `control` blocks or other `actions`. In the explicit invocation case, directionless parameters follow `in` parameter semantics.

A table can be invoked explicitly by calling its `apply` method. This method is synthetic; it is generated by the compiler. Its return type is a synthetic `struct` shared with other actions in a given table. The compiler generates an `enum` and a `struct` for each table, both of which can be seen in Listing 1.13.

Here we see a rationale for the requirement that an action list cannot contain two actions with the same name. The restriction ensures that the corresponding enum variants never clash.

The fields in the `apply_result(T) struct` indicate whether the table hit or missed and which action was executed.[9] The `hit` and `miss` fields are mutually-exclusive, exactly one of them is set when the action returns.

The match-action pipeline abstract machine

Although the $P4_{16}$ specification does not define an abstract machine for the match-action pipeline, it does describe the semantics by analogy to imperative programs.

The body of a control block is executed serially, as a sequence of imperative statements. The syntax restricts the control block body such that the bulk of code within is limited to a special sub-block introduced with the `apply` keyword. The `apply` sub-block is the only place where tables can be explicitly invoked. The control can also invoke other controls or explicitly invoke actions by calling their `apply` methods, although it cannot invoke parsers.

The `apply` block can contain `return` and `exit` statements. A `return` statement immediately terminates execution of the control block and returns control to the caller. An `exit` statement terminates execution of the entire control block call chain.

A control block can invoke subroutines in the form of other controls during its execution. This requires prior instantiation of the sub-controls, either in the calling control block's body or prior to invoking a calling control block that is passed a control instance as a parameter. However, $P4_{16}$ offers

[8]The specification lists several motivated examples.
[9]`action_run` will be set to the variant corresponding to the table's `default_action` on a table miss.

Code listing 1.14 An example of direct type invocation.

```
control Callee(/* parameters omitted */) { /* ... */ }

control Caller(/* parameters omitted */) {
    apply {
        // Callee can be treated as an instance
        Callee.apply(/* arguments omitted */);
    }
}

// the Caller definition above desugars to the following

control Caller(/* parameters omitted */) {
    // local instance of Callee
    @name("Callee") Callee() Callee_inst;
    apply {
        // Callee_inst is simply applied
        Callee_inst.apply(/* arguments omitted */);
    }
}
```

Code listing 1.15 An example of a parser with constructor parameters.

```
// a parser with one constructor parameter
parser GenericParser(packet_in b, out Packet_header p)
                    (bool udpSupport) {
    // ...
}
...
// the instantiation specifies all constructor parameters
// topParser is a GenericParser where udpSupport = false
GenericParser(false) topParser;
```

syntax sugar for the common case of using a sub-control exactly once. This shortcut is known as *direct type invocation* and works even for controls with constructor parameters, discussed in the next section. An example can be seen in Listing 1.14.

Abstraction via constructor parametrization

Although parsers and control blocks take parameters, these only define the interface between P4 code and the target architecture. This makes abstraction difficult, as expressing control blocks or parsers with similar structure requires either code duplication or a reliance on preprocessor macros.[10] To address this problem, P4$_{16}$ supports *constructor parameter lists.* Constructor parameters are given in secondary, optional parameter lists, available to control blocks and parsers. Syntactically, they follow the regular parameters of the given construct. However, at the site of instantiation, the values of constructor parameters directly follow the type name.

All constructor parameters must be directionless. Instantiations must bind all constructor parameters to compile-time known values. Constructor parametrization offers a light form of templating that is sufficient for many use cases and saves the user from substituting generic parsers and control blocks by hand.

[10] Which would be a rather dreadful affair.

Code listing 1.16 An example of a P4$_{16}$ deparser.

```
control Deparser(inout headers hdr, packet_out packet) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        if (hdr.ipv4.isValid()) {
            if (hdr.ipv4.protocol == IPProtocol.UDP) {
                packet.emit(hdr.udp);
            }
            if (hdr.ipv4.protocol == IPProtocol.TCP) {
                packet.emit(hdr.tcp);
            }
        }
    }
}
```

Code listing 1.17 The `packet_out extern`.

```
extern packet_out {
    void emit<T>(in T data);
}
```

Deparsing

Deparsing is the inverse of parsing, i.e. the process of converting packet headers from their semantic representation back into a sequence of bytes. A careful reader will observe that the original P4$_{14}$ forwarding model from Figure 1.1 does not include this step. Indeed, a deparser is not necessary if the device does not rewrite packets. For example, certain architectures may rely solely on metadata to communicate information about packet forwarding from the data plane and possibly include a rewriting step outside the P4 model.

Nevertheless, deparsing is a common step for many network processors and P4$_{16}$ fully supports it. However, the language does not include any special syntax for deparsing, which speaks somewhat to its generality. P4$_{16}$ deparsers are expressed by means of control blocks combined with the special `packet_out extern`. A value of type `packet_out` denotes a buffer containing the packet to be sent out.

Listing 1.16 shows an example deparser. The `emit` method of the `packet_out extern` appends the given data to the packet buffer. The behaviour of a call to `emit` has different semantics, depending on the type of data being emitted.

Headers are emitted only if valid. If the header is invalid, the call to `emit` behaves like a no-op.

Header stacks have their elements emitted recursively.

Header unions and `structs` have their fields emitted recursively.

`enums` and `errors` cannot be serialized. It is illegal to invoke `emit` directly or indirectly (through implicit recursive behaviour) on these types.

Base types are emitted as-is, but only indirectly. It is illegal to explicitly invoke `emit` on a base type.

The recursive behaviour of `emit` always follows the order of declarations in the P4 source code.

Calling convention

In order to simplify reasoning about function-like constructs, P4$_{16}$ follows the *call by copy in / copy out* calling convention. The specification describes the exact steps a conforming implementation needs to go follow in general[11] to evaluate a function call expression.

1. *Arguments are evaluated from left to right as they appear in the function call expression.*
2. *If a parameter has a default value and no corresponding argument is supplied, the default value is used as an argument.*
3. *For each `out` and `inout` argument the corresponding l-value is saved (so it cannot be changed by the evaluation of the following arguments). This is important if the argument contains indexing operations into a header stack.*
4. *The value of each argument is saved into a temporary.*
5. *The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this "generated" function call can be implemented using call-by-reference if supported by the architecture.*
6. *On function return, the temporaries that correspond to `out` or `inout` arguments are copied in order from left to right into the l-values saved in Step 3.*

– *P4$_{16}$ Language Specification v1.2.3 — p4.org* [4]

The calling convention gives calls[12] a semantics that ensures arguments cannot alias one another and impure functions cannot hold references to P4 code. This design is ultimately motivated by the need to control the side effects of `extern` functions and methods, which are arbitrarily powerful. The copy in / copy out calling convention lets compilers reason about programs in the presence of `extern`s.

The P4 abstract machine

Throughout this chapter, we have occasionally referred to some values as "evaluated at compile time" or "compile time -bound." A reader familiar with staged programming, dependent types, or C++ template instantiation may worry for the compiler performance that unrestricted compile-time evaluation entails. Fortunately, the P4$_{16}$ specification has a rigorous definition for these terms and clarifies the extent to which a P4 compiler must evaluate a program during translation.

Compile-time known values are generally constants and stateless constructs that themselves only depend on other compile-time known values. Since P4 has no loops, no recursion, and no complex metaprogramming facilities, the set of compile-time known values is finite and can be evaluated in a single pass.

A P4 program is evaluated in two stages, statically at compile time and dynamically at runtime. The notion of compile-time evaluation in P4 is closely related to resource allocation of the target device. Static evaluation proceeds in the order that declarations appear in the source file, beginning at the top level and recursing into lexical scopes. This pass resolves compile-time known values and determines the full set of stateful objects that need to be allocated in order to accommodate the program on the target device.

[11] Compiler optimizations could, in specific cases, eliminate some of these steps, provided prior analysis determines such optimizations correct.
[12] Of all function-like constructs.

Control plane API

A P4 compiler generates methods for interacting with data plane constructs that can be in any way controlled or configured at runtime. These are:

- value sets
- tables
- keys
- actions
- `extern` instances

To disambiguate references from the control plane to these entities, $P4_{16}$ requires that each such entity has a unique fully qualified name. Additionally, control block and parser instances also need unique names, because they contain constructs from the above list.

The $P4_{16}$ specification lays out in detail what names are given to various syntax forms, but these technicalities are not relevant for the purposes of this thesis. Generally, the control plane name corresponds to the accessor syntax of the construct in P4 code. For example, a slice of the four least significant bits of the bit string `s` (`s[3:0]`) is named `s[3:0]` in the control plane API. The only thing to note here is that in cases where a name is not given straightforwardly, the compiler requires a `@name` annotation to be attached to the construct. `@name` can also be used to rename other constructs. The second annotation controlling naming is `@hidden`, which can be used to hide constructs from the control plane API. Hidden constructs are not subject to the unique name requirement.

Dynamic evaluation

A P4 program defines the parsers, control blocks, `externs`, and other constructs that make up the data plane. However, it is up to the target architecture to decide how and when are all these execution blocks evaluated at runtime. Packet processing systems often operate concurrently, so the $P4_{16}$ specification lays out a concurrency model for the data plane.

Without `externs`, the semantics of concurrent executions is trivial[13]: every parser and control block is executed in a separate thread with only thread-local storage. Therefore, concurrent executions cannot interfere, regardless of their interleaving.

`externs` pose a challenge, however, as their instances are shared between concurrent executions. The use of `externs` can thus lead to data races. To combat this, $P4_{16}$ demands that executions of method calls on `extern` instances are atomic. Moreover, users can place the `@atomic` annotation on arbitrary blocks of code to make their execution atomic as well. The specification mandates that a compiler which cannot guarantee the requested atomicity must reject the program.

[13] Although still very informal, as the concurrency model is only described in English and without much care to define every referenced term.

Chapter 2

Language Server Architecture

…in which we introduce language servers in general and the Language Server Protocol in particular, outline their relationship to other classes of language tooling, and look at high-level architectural decisions that their use-cases imply.

Language servers have a lot in common with compilers. Like compilers, they have to build up a semantic model of a program and provide useful diagnostics for invalid or suspicious input along the way. Unlike a compiler, a language server needs to maintain the semantic model over the course of an editing session. Rather than operating in batch mode, a language server runs continuously and is expected to provide feedback within milliseconds.

However, language servers need neither to produce compilation artifacts nor to optimise the programs they process. Instead, they are effectively special-purpose query engines. Their output is a data structure optimised for fast querying, such as finding references, definitions, or providing context-sensitive completion suggestions. From that point of view, it could seem like a language server is merely a compiler frontend with very little backend logic. Unfortunately, this is not the case.

The requirement for real-time feedback to the developer is the primary constraint on a language server's design. For all but the most basic languages and features, instant feedback requires an incremental computation approach and management of state that persists across updates to source files. The second most important consideration, and one to a large extent not shared with compilers, is resilience to errors. While a compiler generally expects well-formed input, a language server deals with all sorts of intermediate states of a document, including files with many syntactic errors, invalid encodings, or unsaved buffers outside the filesystem.

To make matters worse, while compiler frontend implementations are often guided by a language specification,[1] the space of invalid programs is unconstrained. Developers of language servers have to guess what intermediate states a program goes through during development and how to respond to them. Generating meaningful semantic models from invalid input is a challenging task, but doing so is often crucial for developers. For example, smart auto-completion in statically typed languages is expected to provide type-correct suggestions, even as the document being edited is type-incorrect, and often semantically or even syntactically invalid.

In this chapter, we explore the status quo of language servers, how they differ from conventional batch-processing compilers, how this gap may narrow in the future, and what makes building interactive language tooling difficult.

[1] Even if, usually, an informal one.

2.1 The fruits of semantic support

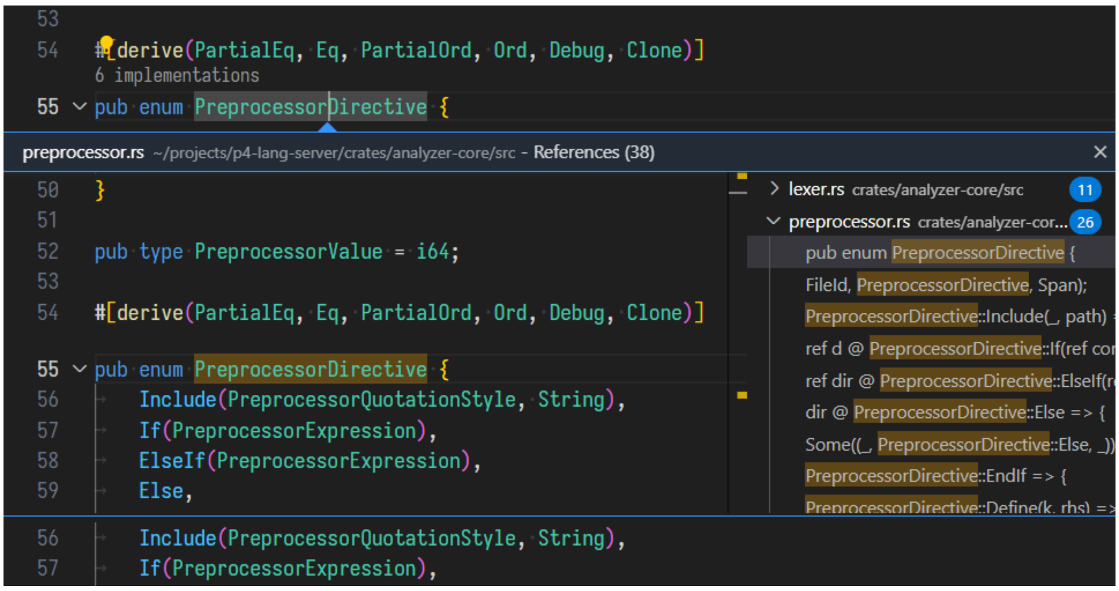
The largest language servers conforming to LSP offer a wide variety of features. The vast majority of the API surface is optional, however. Upon establishing a connection, the client and server exchange information about their respective capabilities, establishing a subset of LSP they both support.

The functionality of LSP comes in two main flavours: code comprehension and coding features. The former subsumes utilities which ease reading and navigating through code, such as Hover (where the editor displays details about an object under the pointer), Go to Definition, Find References, etc. Utilities like diagnostics, auto-completion, or code actions are more relevant to the programmer at the time they are authoring code and belong in the latter category.

Next, we will delve into both categories and take a closer look at what they offer.

2.1.1 Code comprehension in LSP

Code comprehension functionality takes up the majority of LSP's API surface and has been growing during the protocol's evolution. At the time of writing, the latest stable LSP release is version 3.17. The central features, already present in the initial specification of the protocol, are Go to Definition, Find References, Document Highlight, Document Link, Hover, Code Lens, and Document Symbols.



```
#[derive(PartialEq, Eq, PartialOrd, Ord, Debug, Clone)]
6 implementations
pub enum PreprocessorDirective {
```

preprocessor.rs ~/projects/p4-lang-server/crates/analyzer-core/src - References (38)

```
}

pub type PreprocessorValue = i64;

#[derive(PartialEq, Eq, PartialOrd, Ord, Debug, Clone)]

pub enum PreprocessorDirective {
    Include(PreprocessorQuotationStyle, String),
    If(PreprocessorExpression),
    ElseIf(PreprocessorExpression),
    Else,
```

```
> lexer.rs crates/analyzer-core/src 11
v preprocessor.rs crates/analyzer-cor... 26
    pub enum PreprocessorDirective {
    FileId, PreprocessorDirective, Span);
    PreprocessorDirective::Include(_, path)
    ref d @ PreprocessorDirective::If(ref co
    ref dir @ PreprocessorDirective::ElseIf(r
    dir @ PreprocessorDirective::Else => {
    Some((_, PreprocessorDirective::Else, _))
    PreprocessorDirective::EndIf => {
    PreprocessorDirective::Define(k, rhs) =>
```

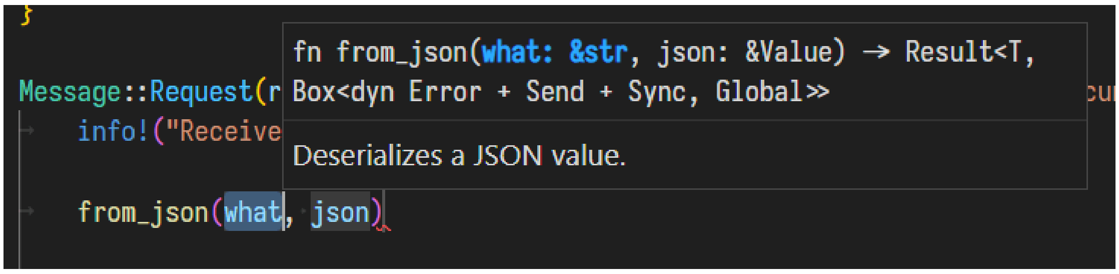
```
    Include(PreprocessorQuotationStyle, String),
    If(PreprocessorExpression),
```

Figure 2.1 Find References in Visual Studio Code via rust-analyzer.

Go to Definition and Find References are present in some of the oldest code comprehension tools, dating back at least to the Unix utility `ctags`[20]. These let the editor jump from a symbol's use-site to its definition and vice versa, just as their names imply. LSP's Document Highlight request does not provide syntax highlighting, rather, it serves to visually assist the programmer with locating references of a given symbol without having to explicitly invoke the Find References feature. Document Highlight could be merged with Find References functionality, but the LSP maintainers choose to keep them separate and allow Document Highlight to report imprecise ("fuzzy") matches. A response to the Document Link request lists the hyperlinks embedded in the document. Document Symbols provides a potentially hierarchical overview of the symbols of a document, which serves the "outline" feature of modern editors: a tree overview of a program's constructs, such as modules, classes, fields, and methods. The Hover feature provides additional contextual information when navigating code. It is typically implemented

1. Go to Declaration
2. Go to Definition (original)
3. Go to Type Definition
4. Go to Implementation
5. Find References (original)
6. Prepare Call Hierarchy
7. Call Hierarchy Incoming Calls
8. Call Hierarchy Outgoing Calls
9. Prepare Type Hierarchy
10. Type Hierarchy Super Types
11. Type Hierarchy Sub Types
12. Document Highlight (original)
13. Document Link (original)
14. Document Link Resolve (original)
15. Hover (original)
16. Code Lens (original)
17. Code Lens Refresh
18. Folding Range
19. Selection Range
20. Document Symbols (original)
21. Semantic Tokens
22. Inlay Hint
23. Inlay Hint Resolve
24. Inlay Hint Refresh
25. Document Color

Figure 2.2 Code comprehension -related requests in LSP 3.17.



```
}
Message::Request(r
    info!("Receive

    from_json(what, json)
```

```
fn from_json(what: &str, json: &Value) → Result<T,
Box<dyn Error + Send + Sync, Global>>
```

Deserializes a JSON value.

Figure 2.3 Signature help in VS Code for Rust shows a pop-up with documentation as well as the signature of the callee, highlighting the parameter under cursor.

by the client rendering a pop-up box of documentation for a given symbol. Finally, Code Lens is a versatile editor feature for displaying additional information at a given position in a document, such as the number of references of a type or the code metrics of a procedure[21]. It can trigger an action when activated, which is used by some servers to run tests or open a Find References dialog.

2.1.2 Coding features in LSP

A shorter but no less important range of API calls supports the developer right when they are authoring code. The main features are auto-completion, signature help, formatting, and symbol renaming.

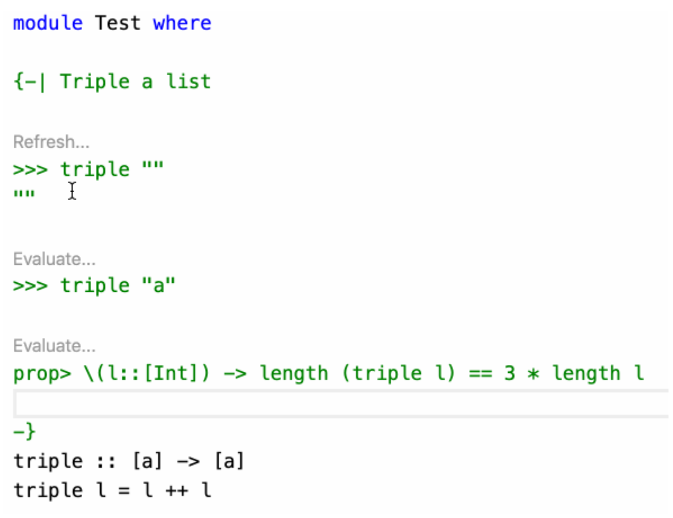
Auto-completion offers to fill in code as the programmer is typing, supports ranking results based on their relevance, on both the server and the client side, and can include a "quick info" description for each option. Signature help shows parameter name and type information when calling a procedure, method, or function. Formatting allows a language server to rewrite a document upon request, for example to conform to a particular code style. The LSP formatting functionality can format either the entire document, a selected range, or reactively an arbitrary part of the document as the user types.

1. Inline Value
2. Inline Value Refresh
3. Moniker
4. Completion Proposals (original)
5. Completion Item Resolve (original)
6. Publish Diagnostics
7. Pull Diagnostics
8. Signature Help (original)
9. Code Action
10. Code Action Resolve
11. Color Presentation
12. Formatting (original)
13. Range Formatting (original)
14. On type Formatting (original)
15. Rename (original)
16. Prepare Rename
17. Linked Editing Range

Figure 2.4 Coding language features in LSP 3.17.

Finally, symbol renaming performs a context-sensitive workspace-wide rename of a given symbol.

Later LSP revisions added high-level features not universally applicable to all programming languages. For instance, version 3.16 added *linked editing*, which some conforming implementations use to update opening and closing XML tags seamlessly without the user specifically triggering a rename action. Version 3.17 introduced *type hierarchy* requests, relevant only to programming languages with subtyping. On the other hand, more general facets of the protocol see creative use in unintended contexts. One example is the use of lenses and special comments in the Haskell Language Server project[22] to provide REPL-like functionality. Another is the LT$_E$X Visual Studio Code extension[23], which provides spell and grammar checking in Markdown and LaTeX documents, as well as in programming language comments. Even though LSP has no built-in support for extracting the comments of a document or for spell checking in general, LT$_E$X achieves this with a combination of non-LSP APIs and by leveraging LSP diagnostics and code actions to provide suggested spellings.



```
module Test where

{-| Triple a list

Refresh...
>>> triple ""
""

Evaluate...
>>> triple "a"

Evaluate...
prop> \(l::[Int]) -> length (triple l) == 3 * length l

-}
triple :: [a] -> [a]
triple l = l ++ l
```

Figure 2.5 The Haskell Language Server project supports in-editor expression evaluation in comments prefixed with `>>>` and checking QuickCheck properties in comments starting with `prop>`.

Language servers for popular technologies

Many of the most widely used programming languages have corresponding language server implementations. The table below presents VS Code extensions and programming languages in order of their install count and popularity, respectively. The number of extension installations is tracked by the VS Code marketplace (`https://marketplace.visualstudio.com/search?target=VSCode&category=Programming%20Languages&sortBy=Installs`).

Rank	VS Code extension	Popular languages, according to StackOverflow[24]
1	Python	Python
2	C/C++	Java
3	Java	C#
4	C#	C/C++
5	Go	PHP
6	PHP	PowerShell
7	PowerShell	Go
8	Dart	Rust
9	Ruby	Dart
10	Rust	Ruby

We have chosen to exclude some technologies from this comparison. JavaScript, TypeScript, CSS, and HTML support is built into VS Code and therefore does not show up in VS Code marketplace statistics for installed language extensions. SQL, while popular, has too many dialects to present a faithful picture through the lens of extension installations alone. These technologies were in turn filtered out from StackOverflow's list of most popular languages to highlight the differences in ranking.

2.2 Lessons from the compiler world

We have previously established how closely does the task of implementing language servers relate to writing compilers. Let us expand on the possible architectural similarities of the two kinds of language tools in this section.

2.2.1 The pipeline

Traditional compiler architectures build around a pipeline approach. The compiler begins with a frontend, typically composed of a lexer[2], a parser, and a step of semantic analysis. The second major part is the backend, a combination of an optimiser and a code generator. The lexer ingests bytes of text, turning them into tokens for the parser. The parser matches the tokens against the productions of a given language's grammar, producing abstract syntax trees. Semantic analysis then verifies that the parsed program adheres to the language's semantic restrictions. For example, semantic analysis ensures that variables can only be referenced after their declaration or definition, `break` statements can only appear in the bodies of loops, and the program follows typing rules.

[2]Although the lexer/parser distinction is maintained, partly for historical reasons, in many modern compilers, the jobs of lexers and parsers are conceptually identical. These compositional algorithms transform input of one type (usually a string) into output of another, verifying certain properties along the way. Concretely, they match the input against some grammar.[3]

[3]A keen reader will note that without restrictions on the *class* of grammars, this statement makes the description no more concrete.

Is it the frontend's responsibility to identify and report user errors, terminating the compiler pipeline as soon as it identifies invalid input. This is a practical choice, since later stages of the compiler can assume the program valid and not worry about possible errors. Furthermore, computation of the backend stages would be wasted on invalid input anyway, as the compiler could give no guarantees about the produced executable form.[4]

One important step that typically happens on the frontend/backend boundary is *lowering*. This process turns the AST obtained and validated by previous stages into the compiler's *intermediate representation (IR)*. The IR represents a simplified language devoid of syntactical sugar and various programmer-facing niceties. For example, various types of conditional expressions are usually represented by a single IR instruction.[5] Similarly, different forms of loops lower to a few canonical translations. Freedoms in the source-level code style are eliminated to make reasoning about the program further down the pipeline easier. Nested expressions are flattened using short-lived local variables, often imposing an evaluation order. Overall, intermediate representations tend to be semantically closer to the target architecture.

After lowering, the backend takes over. The optimizer, if it is involved in the compilation, iteratively transforms the IR in a series of analysis and rewriting steps that attempt to minimise certain cost functions.[6] Optimization can sometimes cross the boundaries of source-level code. That is, certain IR programs do not have a corresponding source program, because the lowering phase is not a surjective mapping. This is a necessary freedom for eliminating overhead in implementations of high-level programming languages, but makes mapping issues in the final executable more difficult.[7]

Finally, the pipeline terminates in a code generation phase which emits the final executable form. This step requires rewriting the IR program in order to fit the target constraints. The amount of work the compiler needs to do in this last stage depends on the semantic differences between the structure of the intermediate representation and the target language. It could be as simple as writing the IR to an output file, if the intermediate and target languages perfectly match. For conventional processor targets, however, code generation necessitates at least instruction selection and register allocation, as well as maintaining some level of conformance to standard calling conventions.

Where the pipeline falls short

In recent years, the feedback loop from writing code to executing it has gotten shorter and shorter. The case for early feedback is simple: programmers want results as soon as possible. Moreover, since the programmer typically uses the compiler in an online fashion, changes to the source files tend to be small and local. Compiling small changes in the source program often only requires small changes to the output. With proper caching, most information can be reused from previous runs of the compiler.

With semantic language support receiving more and more attention, the overlap between compilers and interactive language tooling is only getting clearer. Both classes of programs now need to maintain and incrementally update databases of semantic information about the codebase in order to respond quickly to user requests. With semantic information at hand, it is only natural for both classes to integrate tightly with editors and development environments to provide features reliant on high-level information. Both compilers and language servers should be resilient to errors in the user input and continue processing as far as is practical, for example to report semantic errors even in the presence of syntactical issues. Just as with optimisers and other components useful across many frontends, reimplementing a lot of complicated functionality is tiresome and unnecessary.

Unfortunately, the well-established pipeline approach to compiler construction is, despite its many innovations, difficult to adapt to the modern incremental workloads. The interfaces between stages do not share a principled, universal structure, requiring each phase to implement its own variant of caching

[4]Whether that is machine code in a platform-specific executable, bytecode for a virtual machine, source code in the case of transpilers, or something else entirely.

[5]Which may in fact conceptually be an instruction, or a node of the IR graph.

[6]The actual cost functions may vary based on what steps the optimizer takes. For example, the metrics used for inlining can differ from heuristics consulted for loop-invariant code motion.

[7]Which is of major concern for debugging, and consequently one of the primary reasons why native executable debuggers tend to operate better on programs compiled with fewer optimizations.

and cache-invalidation. Running phases concurrently for independent sections of the input program can be problematic, because older pipeline-based compilers often rely on global variables.[8]

Conventional incremental compilers choose a granularity of input file, compilation unit, or module, and often run several pipelines in parallel, culminating in a sequential step that combines the constituent products into the final executable. This approach achieves very short compilation and recompilation times for many programs, but tends to produce suboptimal code, since the optimiser can only see a subset of the code and is limited in attempting whole-program optimisation and cross-module inlining. Nowadays, linkers (the usual last step in the production of executables) counter this downside with link-time optimization (LTO), trading linking time for better quality code. Running many pipelines in parallel also tends to increase the size of the intermediate compilation artifacts, because dead code elimination cannot safely remove unused definitions, possibly referenced by other compilation units.

2.2.2 The pipeline as a sequence of queries

With these new developments and performance constraints in mind, recent compiler construction techniques put incremental computation at the centre of their design. For example, Roslyn,[9] a collection of compilers and analysers for C# and Visual Basic, builds heavily on incremental computation. A Roslyn compiler takes centre stage in a number of interactive and batch applications, maintaining a semantic model of the codebase. Higher-level tools then query and update this model via several APIs designed for static analysis, code refactoring, and other use-cases[25]. The entire stack is optimised for interactive use. For example, the parsers utilise a combination of persistent "green" abstract syntax trees and transient "red" trees. The second, ephemeral type is built on-demand, optimised for querying, and discarded with edits[26], while the "green" tree serves as the source of truth.

As more and more language tools interact with and rely on the compiler, compiler authors find themselves adapting the pipelined architecture to emit additional intermediate artifacts. If this evolution happens organically over long periods of time and without a methodical approach, compiler codebases can degenerate into clouds of complex, intertwined code.

A possible remedy is to adapt the pipeline architecture in ways that make it simple to incrementalise and memoize its stages. The key insight is that the pipeline is conceptually a collection of data-dependent queries. If we express the compiler in the language of a general query engine, caching and incremental computation features come for free. Propagating changes through the data dependency graph is simple, and this change processing subsumes cache invalidation. The interfaces between compiler queries effectively become high-level APIs, with little extra work. These interfaces are shared with other applications, making integration simpler and less error-prone than in many traditional pipeline architectures, which maintain separate sets of internal and external interfaces.

A functional approach

Even though the traditional pipeline and its query-based reimagination are conceptually close, their implementations are vastly different. Typical compilers mutate global state during the course of a compilation. If any stages produce additional data, they populate side channels in the form of global maps and tables for further passes to use. This can lead to subtle compiler bugs when compiler passes are added, removed, or reordered. Mutable state is also an obstacle to parallelism and makes it difficult to reason about where exactly in the pipeline does one intermediate form change into another.

On the other hand, the query-based approach is inherently functional. Compiler passes are ordered by their data dependencies and the amount of available parallelism is limited only by the number of independent queries at a given stage. A query-based compiler should avoid using mutable state, so that the query engine can correctly propagate changes and update memoized functions.[10]

[8] This is no fault of the architecture as a whole, but it is an important practical consideration.

[9] This product is officially called *The .NET Compiler Platform SDK* but it is arguably better known under the Roslyn codename.

[10] Some hidden mutability may still be beneficial for performance optimisations. For example, a hidden mutable map can serve as a backing store for string interning. Naturally, the programmer needs to be vigilant around any such places in the implementation

Queries in the wild

Although query-based approaches promise many benefits, their adoption in large compilers seems scarce. Large compilers and compiler frameworks, including LLVM and p4c, still build on the traditional pipeline model.

One exception is The Rust compiler. Although it was not originally built around a query system, one has been retrofitted into it. Major parts of the pipeline between `rustc`'s high-level IR and LLVM IR are now implemented as incremental interdependent queries[27].

Smaller compiler projects have ventured further into the query-based realm of data dependencies and memoization. Examples can be seen in Sixten,[11] Sixty,[12] and Eclair.[13] Olle Fredriksson developed a dedicated library for query-based build systems dubbed `rock`[14] that all three projects build on.

and verify that the introduced side-effects do not compromise correctness of the entire system.

[11] `https://github.com/ollef/sixten`

[12] `https://github.com/ollef/sixty`

[13] `https://github.com/luc-tielen/eclair-lang`

[14] `https://github.com/ollef/rock`

Chapter 3
Design

The high-level architecture of the P4 Analyzer project marks a departure from conventional language server designs in that it primarily targets WebAssembly and aims to run entirely within the Visual Studio Code editor. This decision makes installation simpler for the end-user, cross-platform support easier for the developers, and security policy conformance trivial for any security teams involved.

The main mode of operation is thus as follows: the language server runs in a WebAssembly worker of the P4 Analyzer VS Code extension. The extension itself defines a simple TextMate[28] grammar specification and serves as a thin client for the server. The main bulk of LSP functionality is delegated to the editor. VS Code forwards edits to open files to the language server, which updates its model of the workspace. When VS Code asks for completions, hover, diagnostics, or other features, the analyzer recomputes necessary information on-demand and responds appropriately.

In addition to the WebAssembly executable, the P4 Analyzer project also compiles to a native binary that executes in a standalone process and communicates with an arbitrary LSP-compliant client over a socket. However, the standalone language server requires support for certain features related to filesystem functionality that fall outside the protocol specification. We will discuss these later.

3.1 The P4 Analyzer pipeline

The first step in our pipeline is lexical analysis. Somewhat unconventionally, our lexer produces tokens even for the preprocessor (i.e. it analyses preprocessor directives). Our preprocessor then operates at the lexeme level, rather than running separately as the first step. This requires a reimplementation of the preprocessor, which is already necessitated by fault-tolerance and WebAssembly support requirements anyway. On the upside, a custom preprocessor simplifies tracking of source positions, which are crucial for accurate diagnostics.

3.1.1 Lexical analysis

The P4$_{16}$ specification defines a YACC/Bison grammar for the language. However, this grammar has several flaws.

For example, it reuses the `parserTypeDeclaration` nonterminal in `parserDeclarations` but imposes extra restrictions: a parser declaration may not be generic. This requires checking the child production outside the grammar specification.

However, the primary issue is that the grammar design does not maintain a clean separation between a parser and a lexer and requires these two components to collaborate.

The grammar is actually ambiguous, so the lexer and the parser must collaborate for parsing the language. In particular, the lexer must be able to distinguish two kinds of identifiers:

- *Type names previously introduced (`TYPE_IDENTIFIER` tokens)*
- *Regular identifiers (`IDENTIFIER` token)*

The parser has to use a symbol table to indicate to the lexer how to parse subsequent appearances of identifiers.

– *P4$_{16}$ Language Specification v1.2.3 — p4.org* [4]

The specification goes on to show an example where the lexer output depends on the parser state and mentions that the presented grammar "*has been heavily influenced by limitations of the Bison parser generator tool.*"

The tight coupling between the lexer and the parser, as well as the decision to remain in the confines of an outdated parser generator despite its many drawbacks, are in our opinion examples of poor design for a language born in the twenty-first century. We have elected not to follow this ambiguous grammar specification in the P4 Analyzer project and instead build a pipeline that is tolerant to invalid input to the fullest extent possible, while accepting the same language.

Our lexer's task is to convert the input string into a stream of lexemes. The lexer is a standalone finite state machine independent of any later stages in the pipeline. It has a secondary output for reporting diagnostics, but this side channel is write-only.

Error tolerance

Error tolerance at the lexer level means proceeding with lexeme stream generation despite nonsensical input. We emit a special error token whenever such input is encountered. Additionally, the lexer validates numeric constants, which can specify width, base, and signedness. These properties could be out of bounds for a given literal. In these cases, the lexer should still produce a valid token while logging an error-level diagnostic. The server can then report the diagnostic to the user once lexing completes.

3.1.2 The preprocessor

P4$_{16}$ requires support for a preprocessor, very similar to the C preprocessor, directly in the specification. However, it does not ask implementors to support the entirety of `cpp`. Notably, only simple, parameterless macros are allowed. This is already enough to necessitate running the preprocessor before starting the parser, however. Consider the code in Listing 3.1. These examples show how grammatically invalid code may become valid and vice versa, based only on the right-hand sides of preprocessor macros.

An important consideration for a correct implementation of preprocessor directives is their context-sensitive nature. Expressions for conditional inclusion in directives `#if`, `#elif`, and `#ifdef` are themselves subject to macro substitution and thus have to be kept in plain text or lexeme form until their evaluation.

One more thing to note here is the mechanism of document inclusion. Before analysing a P4$_{16}$ source file (at least to some degree), the full extent of its dependencies is unknown and arbitrary. The language has no module system and imposes no restriction on the paths a source file can include. This poses a challenge for lexeme-level preprocessors, as a file needs to be lexed before it can be included. To deal with this, a correct implementation should collect the paths a source file can depend on, lex their contents, and include their lexemes in the preprocessed lexeme stream. This is of particular note in our implementation, as the collection of dependencies reports this dependency set to the editor to set up filesystem-level watches. Subsequent edits to the dependencies, or even to the dependency set itself, can be processed incrementally.

Error tolerance

Error tolerance in the preprocessor means reporting errors and warnings about malformed input to the user while continuing to interpret directives in the input stream on a best-effort basis. Mistakes in preprocessor directives come in several flavours.

Code listing 3.1 $P4_{16}$ preprocessor example

```
#define op +
// #define op 2

#define paren )

header h {
    bit<1> field;
}

control pipe(inout h hdr) {
    Checksum16() ck;
    apply {
        // arithmetic expression could be invalid
        h.field = 1 op 3;
        // a parse without prior macro substitution would fail
        ck.clear( paren;
        // this would parse correctly, but macro substitution
        // will reveal a parse error
        ck.update(op);
    }
}
```

The directive itself may be malformed, either due to a typo in its name or a problem in some of its arguments. The former case will simply be lexed as an unrecognized directive and reported as such. It is possible to suggest fixes for common typos to the user. A problem in the directive's argument or arguments needs to be resolved based on its meaning. For example, an `#include` directive could point to a non-existent file, the preprocessor should then report this error and proceed as if the file were not included. This is likely to lead to further errors down the road, but without knowledge of the referenced file's contents, it is the best a preprocessor can do.

Another class of errors is semantic and context-sensitive in nature: a directive may be used in the incorrect context or missing where it is expected. For example, a user may forget to add an `#endif` directive, or include more than one `#else` directive for a condition. Unfortunately, guessing the user's intention when faced with any syntactic or semantic problems in the input is a tall order. No guarantees of optimality can be given, as is often the case with similar heuristics. In the duplicate `#else` problem, the preprocessor could be reasonably expected to either skip over the first `#else`'s body, the second `#else`'s body, or assume either of the directives was inserted by accident and pretend it is not a part of the input stream. We choose to skip the second `#else`'s body in our design, but other strategies are equally valid.

3.1.3 The parser

The next natural step in the pipeline is the act of finding the productions of a $P4_{16}$ grammar that match the preprocessed input program; parsing. While the steps up to this point are fairly simplistic and efficient, parsing is a resource-intensive process. A language server is expected to provide real-time feedback to the developer, including auto-completion suggestions updated with every keypress. Low latency is crucial to the end-user and the parser lies on every critical path from user input to high-level results shown in the editor's interface. At the same time, a typical $P4_{16}$ program is likely to consist of a long prefix that does not change between edits and a user-maintained suffix that changes frequently. This is because a $P4_{16}$ program usually begins with `#include` directives referencing platform-specific files with constants, error codes, `extern` definitions and other shared code. These constraints and conditions are a very good fit for the field of *incremental parsing*.

An incremental parser aims to reuse previously computed information about the input in response to small perturbations. Our parser specifically builds on incremental packrat parsing[29], which places few constraints on grammar design and is easy to implement in an extensible manner.

Packrat parsing[30] is a linear time algorithm for recognizing productions of a parsing expression grammar (PEG). It relies heavily on memoization to avoid costly backtracking, at the expense of memory overhead. Dubroy et al. augment the packrat memoization table to support incrementality. The result is a parser that is at once simple, general, incremental, and efficient.

Our packrat parser conceptually handles parsing expression grammars[31], a class of unambiguous grammars for context-free languages. PEGs are syntactically similar to context-free grammars. However, the choice operator in CFGs is ambiguous, whereas PEGs use ordered choice, which greedily attempts to match alternatives in order. The right-hand side of a parsing expression grammar rule can also contain predicates, which attempt to match without consuming input. Predicates are useful for positive and negative lookahead.

$$
\begin{array}{rll}
e ::= & \varepsilon & \text{(empty string)} \\
 & \mid \mathtt{t} & \text{(terminal)} \\
 & \mid e_1 e_2 & \text{(sequence)} \\
 & \mid e_1 | e_2 & \text{(ordered choice)} \\
 & \mid e^* & \text{(zero or more)} \\
 & \mid e^+ & \text{(one or more)} \\
 & \mid e? & \text{(zero or one)} \\
 & \mid \& e & \text{(positive lookahead)} \\
 & \mid !e & \text{(negative lookahead)} \\
\mathtt{t} \in & \text{tokens} &
\end{array}
$$

Figure 3.1 Syntax of parsing expression grammars.

The syntax of typical parsing expression grammars can be seen in Figure 3.1. Our grammars are slightly simpler: we implement neither positive lookahead nor the + and ? operators. Positive lookahead can be simulated by nesting two negative lookaheads, and the + and ? operators can desugar to combinations of general repetition, ordered choice, and the empty string. These decisions were made to simplify the parser implementation, but in turn complicate the grammar with verbose and repetitive definitions. It remains to be seen whether they survive future refactorings. A further restriction is that all grammar rules must contain at most one level of nesting, i.e. the grammar must be given in a normal form where the immediate subtrees of a rule's right-hand side are all non-terminals.

Our design differentiates between a generic parser library and a parser built on it. Grammars are defined using a small DSL implemented with Rust's declarative macros. An example can be seen in Listing 3.2. The `grammar!` macro expands to a data structure representing the grammar itself. This structure can be passed to a smart constructor, which validates the grammar[1] and returns a parser. The implementation interprets the grammar DSL at runtime.

If future testing and development necessitate optimization of the parser, there is room to build a parser compiler for the DSL and generate a more efficient solution from the same grammar. Procedural Rust macros[2] could take care of integrating the parser compiler into the build process.

[1]Ensuring all referenced non-terminals are in fact defined, and that `start` is present.

[2]While declarative macros can only perform a very restricted set of rewriting operations on token trees, procedural macros can run arbitrary Rust code during expansion.

Code listing 3.2 Example grammar in our DSL.

```
grammar! {
    // The initial non-terminal is called ‘start‘
    start => p4program;
    // The postfix ‘rep‘ operator corresponds to Kleene star
    ws => whitespace rep;
    // Non-terminals can expand to terminals by wrapping the terminal in
    // parentheses
    whitespace => (Token::Whitespace);

    // The right hand side can also be a sequence separated by commas
    p4program => ws, top_level_decls, ws;
    // ...or a choice separated by pipes
    top_level_decls =>
        top_level_decls_rep | top_level_decls_end | nothing;
    top_level_decls_rep => top_level_decl, ws, top_level_decls;
    top_level_decls_end => (Token::Semicolon);

    direction => dir_in | dir_out | dir_inout;
    // Rules can also match tokens against an arbitrary Rust pattern,
    // which is useful for identifying soft keywords
    dir_in    => { Token::Identifier(i) if i == "in" };
    dir_out   => { Token::Identifier(i) if i == "out" };
    dir_inout => { Token::Identifier(i) if i == "inout" };
}
```

Incremental updates

Since the parser is required to process incremental updates to the input sequence, it is not simply a function from a sequence of tokens to a parse tree. Rather, the parser takes a reference to a read-write lock of the input. It defines an `apply_edit` method that acquires a write lock of the input sequence, applies the change, invalidates relevant entries in the memoization table, and releases the lock.

To initiate parsing, the user invokes the `parse` method, which acquires a read lock on the input for the duration of parsing.

Error reporting

Error handling in parsing is far more nuanced than in any of the previous steps, which is not surprising, considering the relative complexity of the languages that the individual steps recognize and process. A good parser should attempt to provide as much feedback as possible to the user, even when faced with unexpected tokens. It is not enough to simply stop at the first error, and it is not enough to be imprecise about the locations of problems in the source file. Both of these considerations pose some challenges.

The desire to continue parsing malformed input to provide feedback to the developer has a long history[32]. Error recovery has been studied at length in parsing expression grammars as well[33, 34, 35, 36, 37]. The PEG case is interesting, because a packrat parser relies on failures to guide choice selection. By the time a parsing failure propagates to the starting non-terminal, the information about the context that led to it is lost.

Specifically, when encountering unexpected input, a packrat parser unwinds the stack to a “calling” ordered choice operator, and attempts to parse the next alternative. The next alternative is likely the wrong choice, however. The recently failed alternative should have matched, but encountered invalid input. Thus, many other alternatives may fail before an error eventually propagates to the user. The end result is that the programmer receives an unhelpful error message that could potentially come from a position many tokens before the actual problem’s origin. This specific challenge has a popular practical solution in the form of the *farthest failure heuristic*[38]. It is based on the observation that the alternative that should have matched will probably process the longest prefix of the token stream.

While the farthest failure heuristic addresses the location problem in many practical situations, it is not a general solution. Worse, the imprecise error reporting of PEGs has other implications as well. Notably, a common consideration for error messages is the suggestion of expected tokens to the user, to provide a rudimentary selection of possible fixes. However, the compounding failures of PEG choice operators grow the set of expected tokens, which makes the suggestions in error messages irrelevant.

Intuitively, the problem with PEG error reporting is that non-terminals deeper down the parse tree have no way of distinguishing between an error in the input that will ultimately cause the overall parse to fail, and an error that can be recovered from in an ordered choice operator higher up.

To address this problem, Maidl et al. conservatively extended the PEG formalism with *error labels*[34]. Error labels semantically[3] stand for errors that a grammar rule can raise when encountering unexpected input. This differentiates errors caused by nonsensical input from benign parse failures, and thus solves the problem of accurate error reporting in parsing expression grammars.[4] The extension comprises several components:

- The parsing expression grammar is extended with a finite set of labels L.
- A special failure label `fail`, `fail` $\notin L$, is also added to the grammar. This label indicates a benign failure that can be caught by the ordered choice operator.
- The grammar of parsing expression grammar right-hand sides is enriched with the `throw` operator, which takes a label $l \in L$ as an argument. An error thrown by `throw` cannot be caught by ordered choice and thus indicates a parse error that should be immediately reported to the user.
- Rules are modified to include instances of the `throw` operator.

The error label extension is reminiscent of exception handling in ordinary programming languages, and indeed was originally modelled after it, complete with an extension of ordered choice playing (the role of) `catch`[35]. However, the `catch`-like mechanism is not necessary for error recovery, so we will not discuss it further.

Error labels returned by the parsing process can be mapped to readable error messages. Because parsing terminates early, the set of expected tokens is kept accurate. In addition, having a single, obvious point of failure also makes location tracking trivial.

Tracking source locations

Tracking precise source locations is a common requirement for compilers and is arguably even more important for language servers. Our lexer provides the precise span of source code for every token and our preprocessor keeps track of where a token came from during file inclusion. It is important to track the entire "inclusion path" for every token, which consists of segments of file identifiers and spans, since a single file can be included multiple times from multiple locations, with or without detours through other files.

The consideration for the parser here is twofold, namely, to continue to track the source locations of tokens and grammar productions in the format provided by the preprocessor, and to maintain incrementality while doing so.

The first consideration requires some care in pattern-matching of tokens – the grammar DSL is not intended to pattern-match on token inclusion paths. Additionally, an important question is how to assign inclusion paths to grammar productions. This is trivial for terminals, lookahead, and alterations. The interesting case is how to handle sequences and repetitions. While a grammar production could, in theory, store the set of all inclusion paths that it depends on, doing so is wasteful and not very helpful. Instead, we can inspect the inclusion paths of the first and the last token in the sequence or repetition.

[3]In the sense that we are adding semantic information to the process that analyzes syntax.
[4]At the expense of extensive manual annotations of the grammar. See [37] for a possible remedy.

If these two paths match (meaning they differ at most in the last span), we can simply use this path, adjusting the final span. This loses information about tokens from the middle of the sequence, but note that subtrees will still include it. If the two paths do not match, we can take their longest common suffix, again adjusting the last span.[5]

To maintain incrementality, the parser needs to reuse previous parse results in response to small changes of the input. Using actual token spans would be problematic, since a single character change could invalidate the spans of potentially all tokens in the file. Instead, the parser needs to work with the relative notion of token count. Every CST node stores the number of tokens that it spans (the number of tokens in its subtree). A traversal of the CST can then reconstruct the absolute offset of each node without ever explicitly storing it.

One more thing to note here is that the units of offsets in the CST, and therefore of the reconstructed spans, are individual tokens. These need to be converted to spans in the source text before being reported to the user.

3.1.4 Abstract syntax trees

The presented packrat parser produces a *concrete syntax tree (CST)*. A concrete syntax tree is a full-fidelity representation of a grammar production, meaning it includes all tokens and non-terminals, including comments, whitespace, and intermediate non-terminals introduced to circumvent the constraints of the grammar DSL. Preserving full-fidelity syntax trees is important to maintain the parser's incremental performance and avoid expensive backtracking, but CSTs include a lot of state that is unimportant to further processing and analysis steps.

Moreover, the shape and types of these trees are determined by the structure of the grammar. That is, nodes in a CST are alterations, repetitions, sequences, etc. Analysis code would instead prefer a typed API to access the semantic, abstract nodes in the syntax tree, without interference from trivia tokens and non-terminals.

Tree abstractions

These use cases are covered by AST translation. The `ast` module provides three layers of abstraction over CSTs: `GreenNode`s, `SyntaxNode`s, and `AstNode`s.

A `GreenNode` wraps a successful parsing result (an `ExistingMatch<P4GrammarRules, Token>`) in a reference-counted cell, to simplify reuse. It provides a `children` method that returns an iterator over the children of a node. This iterator is a heap-allocated `dyn` type to smooth over differences in the backing CST node.[6] The children are `GreenNode`s themselves, wrapped transparently by the iterator. `GreenNode`s form immutable, functional trees. They are cheap to clone (thanks to reference counting) and can be structurally shared, although not between threads.[7]

A `SyntaxNode` is a zipper data structure that maintains a parent pointer (and therefore the path to the root of the tree) along with the absolute offset in tokens. It again provides a `children` method with an iterator that yields `SyntaxNode`s. These zippers are useful for traversing `GreenNode`s. A `SyntaxNode` also carries a reference to the grammar definition and keeps track of the non-terminal that it is in, by storing it alongside the parent pointer. This is important, as neither the `GreenNode` nor the CST provide this information.

Finally, an AST node is a typed wrapper around a `SyntaxNode`. Unlike the previous two types, it is not a single `struct`. Instead, there is a different type for each AST node, but all of them share a common interface via the `AstNode` trait. This trait provides methods to cast a `SyntaxNode` to a particular `AstNode` type, to access the backing `SyntaxNode`, and to intuit the node's offset and span.

[5]Note that our implementation is incomplete in this regard and presently only reports token spans local to the parsed file.

[6]Future extension and optimisation efforts may decide to change the CST representation to avoid such indirections in tree traversals.

[7]This could be amended by using atomic reference counting cells instead.

Code listing 3.3 The signature of the `ast_node!` macro.

```
macro_rules! ast_node {
    ($non_terminal:ident $(, methods: $($method:ident),+)?) => {
        paste! {
            // see later listings for the body of this macro
        }
    };
}
```

Code listing 3.4 The main body of the `ast_node!` macro generates newtypes for `SyntaxNode`s and implements `AstNode` for them.

```
#[derive(Debug, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct [<$non_terminal:camel>] {
    syntax: SyntaxNode,
}

impl AstNode for [<$non_terminal:camel>] {
    fn can_cast(node: &SyntaxNode) -> bool {
        node.kind() == P4GrammarRules::$non_terminal
    }

    fn cast(node: SyntaxNode) -> Option<Self> {
        if node.kind() == P4GrammarRules::$non_terminal {
            Some(Self { syntax: node })
        } else {
            None
        }
    }

    fn syntax(&self) -> &SyntaxNode { &self.syntax }
}

// continues with optional methods
```

Many AST nodes share the binary representation with their underlying `SyntaxNode`s, meaning they only have a single field of the `SyntaxNode` type. Additionally, the implementations of the `AstNode` trait are often structurally similar. To avoid code duplication and simplify maintenance, the `ast` module provides the `ast_node!` macro, through which most AST nodes are defined.

The signature and top level of the macro are shown in Listing 3.3. Note the inclusion of the `paste!` block: the `paste` crate[8] provides means for converting the case of identifiers. We use this throughout the `ast_node!` macro to generate the `struct` name from the name of the non-terminal. Rust types are conventionally written in camel case while methods, as well as our non-terminals, have snake case names.

The body of the macro defines the `struct` and implements the `AstNode` trait for it. To generate documentation comments, the macro's body gives AST node `struct`s the `doc` attribute. The line can be seen below, it was omitted from Listing 3.4 to avoid confusing the lexer:

`#[doc = "AST node for [‘P4GrammarRules::" $non_terminal "‘]."]`

Finally, the invocation of `ast_node!` can optionally contain a list of methods to implement on the `struct` itself, outside the scope of the `AstNode` trait. The implementation of this functionality can be

[8]`https://crates.io/crates/paste`

Code listing 3.5 The optional methods section of the `ast_node!` macro's body.

```
$(impl [<$non_terminal:camel>] {
    $(
        #[doc = "Fetch the `" $method "` child of this node."]
        pub fn $method(
            &self
        ) -> impl Iterator<Item = [<$method:camel>]> {
            fn go(
                node: SyntaxNode
            ) -> Box<dyn Iterator<Item = SyntaxNode>> {
                match node.trivia_class() {
                    TriviaClass::SkipNodeAndChildren =>
                        Box::new(std::iter::empty()),
                    TriviaClass::SkipNodeOnly =>
                        Box::new(node.children().flat_map(go)),
                    TriviaClass::Keep =>
                        Box::new(std::iter::once(node)),
                }
            }

            self.syntax()
                .children()
                .flat_map(go)
                .filter_map([<$method:camel>]::cast)
        }
    )+
})?
```

seen in Listing 3.5. These generated methods provide type-safe access to children while transparently handling both the skipping of trivia nodes and invalid underlying CSTs. Each returns an iterator without a guarantee of exactly how many children will be iterated over, smoothing over issues of arity. It is up to the users of this API to detect and report such errors.

The entire AST translation layer was inspired by the internals of rust-analyzer,[9] an LSP-compliant language server for Rust. While rust-analyzer's syntax tree manipulations work with a different underlying representation and rely on a recursive descent parser, the high-level interfaces, the split between `GreenNode`s, `SyntaxNode`s, and `AstNode`s, as well as the heavy reliance on code generation, are all motivated directly by rust-analyzer's developer documentation. We would like to thank the rust-analyzer developers for maintaining a comprehensive high-level documentation of their approach and including known alternative approaches, e.g. from Roslyn and IntelliJ, that other developers can learn from.

Identifying trivia nodes

The reader will observe that we have carelessly included a previously undiscussed type in the `ast_node!` macro's body: the `TriviaClass` enum. Listing 3.5 makes it quite clear that the `TriviaClass` type determines which subtrees to iterate over when invoking a generated method of an `AstNode` type.

Our grammar DSL has a few extra features. The author of a grammar can identify the trivia non-terminals by including *annotations* in the grammar definition. Listing 3.6 shows how to use these annotations. By default, every non-terminal has a `TriviaClass` of `Keep`, which means it will be enumerated by an `AstNode`'s iterator.

It is worth noting that the macro for defining the P4 grammar is also the source of truth for the set of non-terminals: it simultaneously defines the `P4GrammarRules` enum. This is very useful for developers,

[9] `https://rust-analyzer.github.io/`

Code listing 3.6 An example of trivia annotations and doc comments in the grammar DSL.

```
grammar! {
    @SkipNodeAndChildren at_symbol close_paren;
    @SkipNodeOnly maybe_direction parameter_comma;

    /// Semantic non-terminal that marks an identifier as a definition.
    ///
    /// For example, in ‘parser MyParser<T>(inout T x) { }‘,
    /// ‘MyParser‘, ‘T‘, and ‘x‘ are all definitions,
    /// and possible targets for go-to definition.
    definition => ident;

    // other non-terminals omitted
}
```

since it avoids code duplication and enables features like go-to definition or find references right in the grammar definition. Using an `enum` to represent non-terminals is also a requirement for exhaustiveness checking, in case any later steps in the pipeline require it. The grammar DSL forwards documentation comments above productions to the variants of `P4GrammarRules`.

3.2 Query-based memoization

In section 2.2.2, we have touched on the query-based approach in certain modern compilers, and some of its implications regarding scalability and parallelism for large incremental systems. Motivated by the query system in the Rust compiler and other incremental computation projects, some of the developers behind rust-analyzer and the Rust compiler proper have developed an incremental computation library called Salsa.

Salsa lets the Rust programmer structure their program as a graph of interdependent query-like computations. All the user needs to do is set up data structures representing the inputs to the system and the intermediate products of its computation. Equipped with a set of library-defined macros and attributes, the user can then annotate functions in their program such that they are automatically memoized in a Salsa database.

Salsa macros transform user-provided `structs` into simple numerical identifiers. Each field of a `struct` gets a method which takes as a parameter a reference to the Salsa database. Calling this method extracts the object’s associated data from the database, either by cloning or by returning a reference. This choice is, too, governed by a macro.

Apart from Salsa-tracked intermediary values and inputs to the system, Salsa also supports automatic interning of (using the same type transformation as for other values) and side-channels for extra output. These side channels are conceptually monoids, a Salsa computation can push new values to the side channel. One can query the database for the entire log of values for a given invocation of a Salsa computation. Since computations are keyed by their inputs, the inputs have to be provided in this case as well.

Salsa’s monoids are known as “accumulators.” P4 Analyzer uses them for diagnostic output. The type of diagnostics is shared among the lexer, the preprocessor, and the parser. Each of these steps is driven by a Salsa computation, itself driven by a request to extract its data from the database (computing on-demand when necessary). The user can then extract the diagnostics for any of these steps.

Salsa implements a pull model. Each tracked computation (a function annotated with a special macro) informs the system of the dependencies on other computations or inputs. The library keeps note of these dependency edges. Naturally, for dependency tracking to work correctly, Salsa computations must depend solely on their inputs. In general, they should avoid any side effects.

All these functionalities are packaged up in the `Analyzer` type provided by the `analyzer-core` crate. This is the single point of contact between the analyzer core and the surrounding machinery for LSP integration. The core knows nothing of the protocol used to communicate with clients.

3.2.1 Integrating with the file system

A major simplification in the P4 Analyzer design compared to rust-analyzer is that we rely on a single source of truth for the contents of buffers and files on-disk alike. All of these are provided by the editor.

Updates to open files are already handled by LSP, but an `#include` directive pointing to an unopened file would pose a problem. To work around this, we rely on VS Code's workspace API, which allows an extension to watch for changes to unopened files.

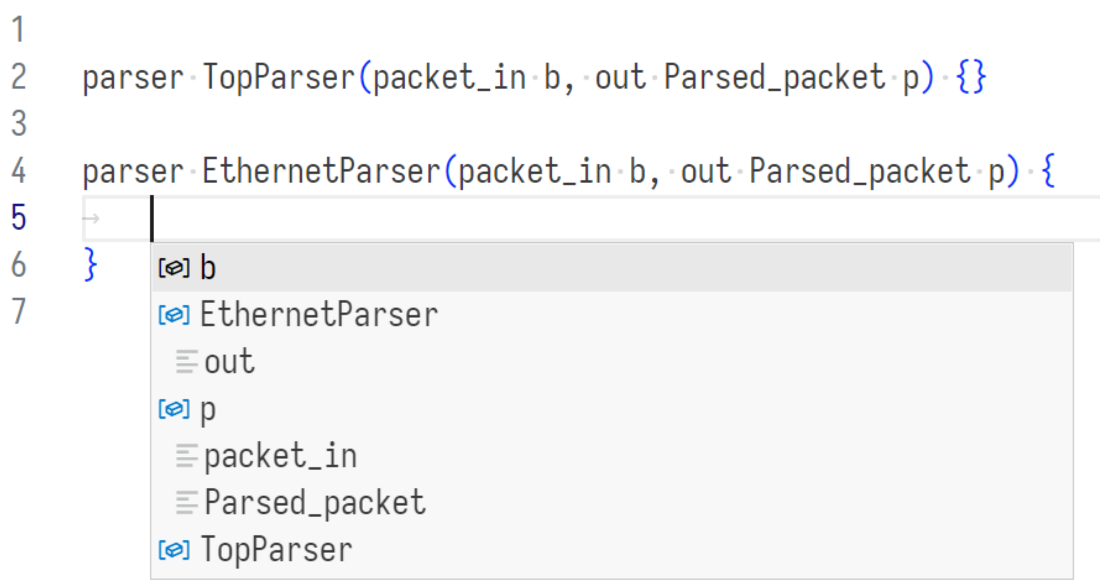
The reason why P4 Analyzer does not follow this approach is historical: the addition of the workspace API to VS Code is a recent development that has not yet made it to other editors.

Chapter 4

Results

Our implementation lays the foundation for the $P4_{16}$ language server and verifies the feasibility of the design decisions made in chapter 3. We have implemented the P4 Analyzer pipeline, which consists of the lexer, the preprocessor, a generic parser library, and a proof-of-concept grammar for $P4_{16}$. P4 Analyzer integrates tightly with Visual Studio Code and other LSP clients, providing autocompletion, diagnostics, and limited go-to-definition functionality. The server is resilient to common errors in user input. The project's integration with VS Code ensures that the core analyzer is restarted if it encounters fatal errors.

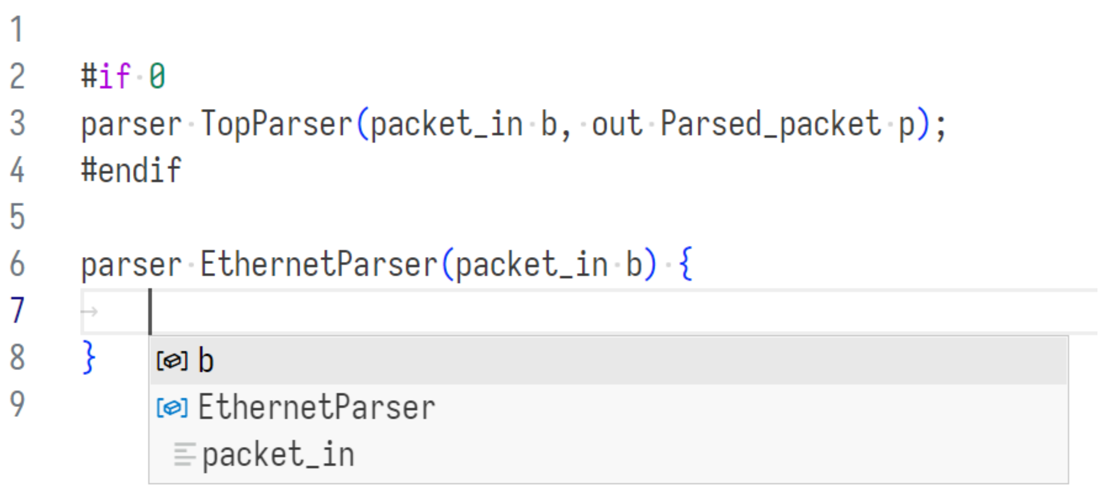
4.1 Overview of implemented features



```

parser TopParser(packet_in b, out Parsed_packet p) {}

parser EthernetParser(packet_in b, out Parsed_packet p) {

}

```

Figure 4.1 Autocompletion in Visual Studio Code. The blue items are suggested from definitions recognised by the parser, while grey items are simply identifiers in the preprocessed token stream.

The lexer and preprocessor parts of the pipeline have been implemented quite thoroughly, with good error recovery and feature sets on par with the design presented in chapter 3. Lexer and preprocessor



```
#if 0
parser TopParser(packet_in b, out Parsed_packet p);
#endif

parser EthernetParser(packet_in b) {
    
}
```

Figure 4.2 The set of suggested identifiers depends on preprocessor directives.

error channels propagate to the LSP client with understandable error messages. The preprocessor, in particular, attempts to recover from missing or superfluous directives, and suggests possible fixes to the user. It also detects circular file inclusion problems.

One feature that is currently lacking is the precise tracking of source locations via the "inclusion path" mechanism. This requires some changes in the preprocessor, but not a significant refactor. One more concern is the evaluation of complex conditions for conditional inclusion directives. If the condition contains references to macros outside the `defined` construct, these macros could, depending on their definition, change the syntactic meaning of the condition, and therefore influence evaluation. A simple fix for this issue is to store conditions as strings and parse them on-demand after performing macro expansion, at the expense of some computational overhead. However, such use cases are rare, so this is not considered a priority for the project.

The parser, on the other hand, is only a proof-of-concept. While the test suite indicates that the library handles edits correctly, integrating this custom approach to incrementality with the Salsa library is not straightforward. Currently, the parser reparses preprocessed token streams from scratch. Parsing is still memoized as a Salsa query, so it happens on-demand and only after edits. Error labels, discussed in the parser design section, have not been implemented yet, and the parser still reports at most one error to the user.

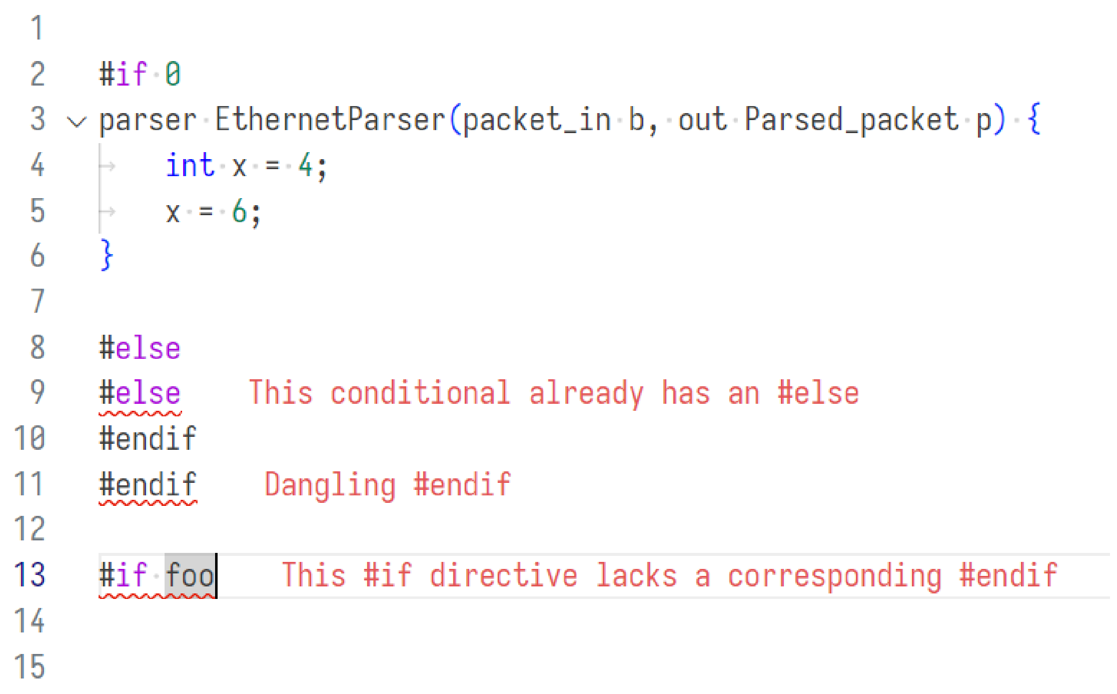
While we deem the abstract syntax tree API rather robust, owning to its success in the rust-analyzer project, its scope and usefulness hinges on a well-designed grammar for the P4$_{16}$ language. The current grammar is very lacking in this regard. The go-to definition functionality relies on the semantic `definition` non-terminal and its correct use in the grammar. Currently, it does not respect lexical scoping – the grammar requires a larger overhaul, possibly supported by extensions to the supported PEG constructs, in order to meaningfully recognise scoped syntactic constructs.

4.2 Benchmarking

An important consideration for an interactive developer tool is its performance. Our test suite includes lexer and parser throughput measurements in the form of micro-benchmarks based on the `criterion` library.[1]

Table 4.1 shows the results of the throughput benchmarks. Some context is necessary to understand these measurements.

[1] `https://crates.io/crates/criterion`



```

#if 0
parser EthernetParser(packet_in b, out Parsed_packet p) {
    int x = 4;
    x = 6;
}

#else
#else    This conditional already has an #else
#endif
#endif    Dangling #endif

#if foo    This #if directive lacks a corresponding #endif


```

Figure 4.3 The preprocessor continues to produce output even in the presence of several errors.

Benchmark	Min	**Mean**	Max
`lexer baseline`	4.95 ms	5.00 ms	5.06 ms
`lexer timing`	130.46 ms	132.41 ms	134.61 ms
`parser baseline`	0.31 ms	0.31 ms	0.31 ms
`parser timing`	187.33 ms	189.38 ms	191.63 ms

Table 4.1 Lexer and parser timings, statistics of 100 samples.

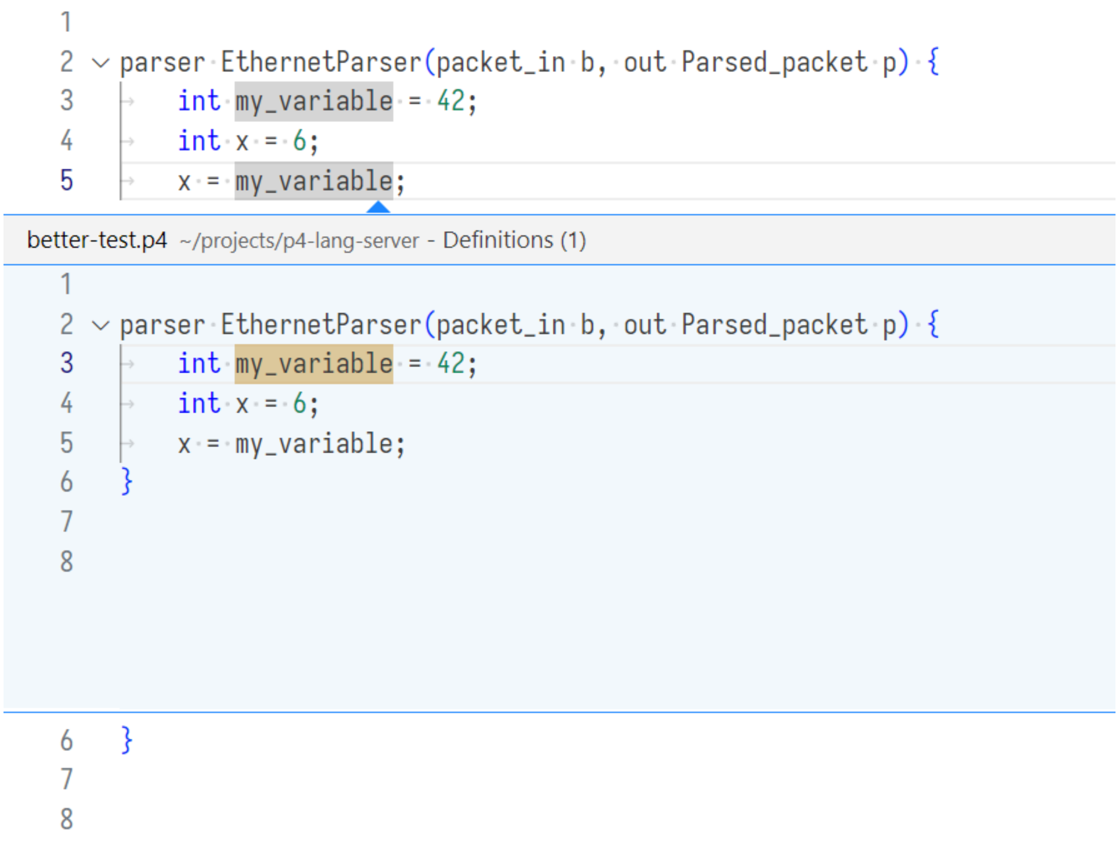
Lexer baseline This benchmark measures only the overhead of copying the input and converting it from a UTF-8 string into a vector of individual codepoints. It does not involve the lexer at all.

Lexer timing This benchmark runs our lexer on example P4$_{16}$ code adapted from the specification[4]. This is roughly 200 lines of code, concatenated with itself 1, 000 times. The total measurement is the time it takes to lex this input (200, 000 lines, about 4.8 megabytes when packed as UTF-8).

Parser baseline The parser baseline measures the overhead of constructing a parser and initializing it with input. It is measured on the same input string and grammar as the **parser timing** benchmark. It copies the input string into a vector (similarly to the lexer baseline), then back into a UTF-8 string. This benchmark does not invoke the parser.

Parser timing This benchmark measures the time it takes to parse a long arithmetic expression in a simple grammar, shown in Listing 4.1. The expression is a randomly generated sequence of integers, with addition and subtraction operations interspersed throughout. The input string has only 100, 000 characters. Note that input generation is not a part of the benchmark and that the benchmark harness verifies that the expression parses correctly before initiating the measurement.

As the measurements show, the lexer is fast with throughput well over one million lines of P4, or over 30 megabytes per second. The parser is significantly slower: when operating on character tokens



```
parser EthernetParser(packet_in b, out Parsed_packet p) {
    int my_variable = 42;
    int x = 6;
    x = my_variable;
```

better-test.p4 ~/projects/p4-lang-server - Definitions (1)

```
parser EthernetParser(packet_in b, out Parsed_packet p) {
    int my_variable = 42;
    int x = 6;
    x = my_variable;
}
```

```
}
```

Figure 4.4 Rudimentary support for go-to definition is also available.

(Unicode code points), it can only parse roughly 0.5 megabytes per second.[2]

4.3 The open-source project

P4 Analyzer was open-sourced on April 18, 2023. The project is available on GitHub under the umbrella of the P4 language organization.[3] It is permissively licensed under the terms of the Apache 2.0 license. Despite its fairly quiet release, it gathered significant attention both internally within Intel and from the P4 community.

We have discovered that in 2023, a research group at McMaster University has independently started a very similar project. Their P4$_{16}$ language server is also developed in Rust and is available from the ACE Design organization's GitHub.[4] We have since gotten in touch with the researchers working on it and are looking for ways to collaborate.

4.4 Future work

As is clear from our results so far, there is much more work left to be done. After finalising the parser, future development efforts should aim to cover more of the API surface of LSP and provide useful fea-

[2]Assuming a packed representation, to relate to the given lexer measurements, even though, in this case, the parser works with a vector of 4-byte characters instead.
[3]`https://github.com/p4lang/p4analyzer`
[4]`https://github.com/ace-design/p4-lsp`

Code listing 4.1 The grammar used in the parser benchmark.

```
let grammar = grammar! {
    start => num, expr_chain;
    expr_chain => expr_choice rep;
    expr_choice => add_chain | sub_chain;
    add_chain => plus, num;
    sub_chain => minus, num;
    plus => "+";
    minus => "-";
    num => digit, many_digits;
    many_digits => digit rep;
    digit => n0 | n1 | n2 | n3 | n4 | n5 | n6 | n7 | n8 | n9;
    n0 => "0";
    n1 => "1";
    // etc.
};
```

tures. In particular, we would like to develop lints[5] and code navigation features built on the AST abstraction framework. Rust makes usage of iterators ergonomic and efficient. We remain optimistic that the architecture of P4 Analyzer presented in this thesis will serve future extensions well.

[5]Soft warnings about code style and possible bugs.

Conclusion

We have designed a complex and incremental query-based language server architecture for the P4$_{16}$ language, including the lexing, preprocessing, and incremental parsing steps. We have implemented a proof-of-concept language server based on this architecture in Rust and integrated it with Visual Studio Code using the Language Server Protocol. The language server provides lexer and parser -based autocompletion, diagnostics for lexer, preprocessor, and parser errors, as well as go-to definition functionality for a subset of the P4$_{16}$ language.

The code base is open to extensions and based on innovative designs from modern compilers and language tooling. All implemented features are backend-agnostic.

Work in progress

Appendix A
Sample appendix

Whatever doesn’t belong to the main part should be put here.

Bibliography

1. BOSSHART, Pat; DALY, Dan; GIBB, Glen; IZZARD, Martin; MCKEOWN, Nick; REXFORD, Jennifer; SCHLESINGER, Cole; TALAYCO, Dan; VAHDAT, Amin; VARGHESE, George, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*. 2014, vol. 44, no. 3, pp. 87–95.
2. BUDIU, Mihai; DODD, Chris. The P416 Programming Language. *SIGOPS Oper. Syst. Rev.* 2017, vol. 51, no. 1, pp. 5–14. ISSN 0163-5980. Available from DOI: `10.1145/3139645.3139648`.
3. CONSORTIUM, The P4 Language. *$P4_{16}$ Language Specification v1.0.0 — p4.org* [online]. 2017. [visited on 2023-03-06]. Available from: `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-p4-language-evolution--comparison-to-previous-versions-p4-v10v11`.
4. CONSORTIUM, The P4 Language. *$P4_{16}$ Language Specification v1.2.3 — p4.org* [online]. 2022. [visited on 2023-03-06]. Available from: `https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html`.
5. DOENGES, Ryan; ARASHLOO, Mina Tahmasbi; BAUTISTA, Santiago; CHANG, Alexander; NI, Newton; PARKINSON, Samwise; PETERSON, Rudy; SOLKO-BRESLIN, Alaia; XU, Amanda; FOSTER, Nate. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages*. 2021, vol. 5, no. POPL.
6. SULTANA, Nik; SONCHACK, John; GIESEN, Hans; PEDISICH, Isaac; HAN, Zhaoyang; SHYAMKUMAR, Nishanth; BURAD, Shivani; DEHON, André; LOO, Boon Thau. Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In: *NSDI*. 2021, vol. 21, pp. 571–592.
7. ANTICHI, Gianni; SHAHBAZ, Muhammad; GENG, Yilong; ZILBERMAN, Noa; COVINGTON, Adam; BRUYERE, Marc; MCKEOWN, Nick; FEAMSTER, Nick; FELDERMAN, Bob; BLOTT, Michaela, et al. OSNT: Open source network tester. *IEEE Network*. 2014, vol. 28, no. 5, pp. 6–12.
8. NAMKUNG, Hun; LIU, Zaoxing; KIM, Daehyeok; SEKAR, Vyas; STEENKISTE, Peter; LIU, G; LI, A; CANEL, C; PHILIP, AA; WARE, R, et al. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 743–759.
9. LIATIFIS, Athanasios; SARIGIANNIDIS, Panagiotis; ARGYRIOU, Vasileios; LAGKAS, Thomas. Advancing SDN from OpenFlow to P4: A Survey. *ACM Computing Surveys*. 2023, vol. 55, no. 9, pp. 1–37.
10. *Open Networking Foundation – p4c; P4 Language Consortium — p4.org* [online]. [visited on 2023-03-07]. Available from: `https://p4.org/products/p4c/`.
11. *9. DPDK Release 20.11 2014; Data Plane Development Kit 23.03.0-rc1 documentation — doc.dpdk.org* [online]. [visited on 2023-03-06]. Available from: `https://doc.dpdk.org/guides/rel_notes/release_20_11.html`.

12. *Intel – P4 Insight; P4 Language Consortium — p4.org* [online]. [visited on 2023-03-07]. Available from: `https://p4.org/products/intel-p4-insight/`.
13. BARROS, Djonathan; PELDSZUS, Sven; ASSUNÇÃO, Wesley KG; BERGER, Thorsten. Editing support for software languages: implementation practices in language server protocols. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems.* 2022, pp. 232–243.
14. BOUR, Frédéric; REFIS, Thomas; SCHERER, Gabriel. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages.* 2018, vol. 2, no. ICFP, pp. 1–15.
15. HANDY, Alex. *Codenvy, Microsoft and Red Hat collaborate on Language Server Protocol - SD Times — sdtimes.com* [online]. [visited on 2023-03-07]. Available from: `https://sdtimes.com/che/codenvy-microsoft-red-hat-collaborate-language-server-protocol/`.
16. KRILL, Paul. *Microsoft-backed Language Server Protocol strives for language, tools interoperability — infoworld.com* [online]. [N.d.]. [visited on 2023-03-07]. Available from: `https://www.infoworld.com/article/3088698/microsoft-backed-langauge-server-protocol-strives-for-language-tools-interoperability.html`.
17. HARPER, Robert. *Practical Foundations for Programming Languages [1/4] – Robert Harper – OPLSS 2019* [online]. 2019-06-19. [visited on 2023-03-07]. Available from: `https://youtu.be/8cXl2Tfhy_Q?t=2169`.
18. KOHLER, Eddie; MORRIS, Robert; CHEN, Benjie; JANNOTTI, John; KAASHOEK, M Frans. The Click modular router. *ACM Transactions on Computer Systems (TOCS).* 2000, vol. 18, no. 3, pp. 263–297.
19. WEIK, Martin. *Fiber optics standard dictionary.* Springer Science & Business Media, 2012.
20. HIEBERT, Darren. *Exuberant Ctags — ctags.sourceforge.net* [online]. [visited on 2023-03-14]. Available from: `https://ctags.sourceforge.net/`.
21. KENCANA, Gilang Heru; SALEH, Akuwan; DARWITO, Haryadi Amran; RACHMADI, R Rizki; SARI, Elsa Mayang. Comparison of maintainability index measurement from Microsoft Codelens and line of code. In: *2020 7th International Conference on Electrical Engineering, Computer Sciences and Informatics (EECSI).* IEEE, 2020, pp. 235–239.
22. *Haskell Language Server – GitHub.com* [online]. [visited on 2023-03-15]. Available from: `https://github.com/haskell/haskell-language-server`.
23. *LT$_E$X* [online]. [visited on 2023-03-15]. Available from: `https://valentjn.github.io/ltex/faq.html#whats-the-difference-between-vscode-ltex-ltex-ls-and-languagetool`.
24. *Stack Overflow Developer Survey 2022* [online]. 2022-06-22. [visited on 2023-03-16]. Available from: `https://survey.stackoverflow.co/2022`.
25. [online]. [visited on 2023-04-04]. Available from: `https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model?source=recommendations`.
26. [online]. 2012-06-08. [visited on 2023-04-04]. Available from: `https://ericlippert.com/2012/06/08/red-green-trees/`.
27. TEAM, Rust compiler. *Rust Compiler Development Guide* [online]. [visited on 2023-04-04]. Available from: `https://rustc-dev-guide.rust-lang.org/overview.html#queries`.
28. GRAY, James Edward. *TextMate: Power Editing for the Mac.* Raleigh, NC: Pragmatic Programmers, 2007.
29. DUBROY, Patrick; WARTH, Alessandro. Incremental packrat parsing. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering.* 2017, pp. 14–25.
30. FORD, Bryan. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *ACM SIGPLAN Notices.* 2002, vol. 37, no. 9, pp. 36–47.

31. FORD, Bryan. Parsing expression grammars: a recognition-based syntactic foundation. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2004, pp. 111–122.

32. GRAHAM, Susan L; RHODES, Steven P. Practical syntactic error recovery. *Communications of the ACM*. 1975, vol. 18, no. 11, pp. 639–650.

33. REDZIEJOWSKI, Roman R. Mouse: from parsing expressions to a practical parser. In: *Concurrency specification and programming workshop*. 2009.

34. MAIDL, André Murbach; MASCARENHAS, Fabio; IERUSALIMSCHY, Roberto. Exception handling for error reporting in parsing expression grammars. In: *Programming Languages: 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3-4, 2013. Proceedings 17*. Springer, 2013, pp. 1–15.

35. MEDEIROS, Sérgio de; MASCARENHAS, Fabio. A parsing machine for parsing expression grammars with labeled failures. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 2016, pp. 1960–1967.

36. MEDEIROS, Sérgio de; MASCARENHAS, Fabio. Syntax error recovery in parsing expression grammars. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1195–1202.

37. MEDEIROS, Sérgio Queiroz de; JUNIOR, Gilney de Azevedo Alvez; MASCARENHAS, Fabio. Automatic syntax error reporting and recovery in parsing expression grammars. *Science of Computer Programming*. 2020, vol. 187, p. 102373.

38. FORD, Bryan. *Packrat parsing: a practical linear-time algorithm with backtracking*. 2002. PhD thesis. Massachusetts Institute of Technology.

Contents of the enclosed medium

```
readme.txt ........................ stručný popis obsahu média
exe ........................ adresář se spustitelnou formou implementace
src
  impl ........................ zdrojové kódy implementace
  thesis ........................ zdrojová forma práce ve formátu LaTeX
text ........................ text práce
  thesis.pdf ........................ text práce ve formátu PDF
```