Parsing Foundations

A Study of Data and Parsing

Unfinished Draft of First Edition

Parsing Foundations

A Study of Data and Parsing

Unfinished Draft of First Edition

Anonymous
Seattle, WA

Self Publishers Worldwide
Seattle San Francisco New York
London Paris Rome Beijing Barcelona

This book was typeset using LaTeX software.

Preface

Parsing is an important and well-established area within the field of software development and nearly every programmer can benefit from a solid understanding of parsing.

There is an enormous amount of information on the topic of parsing. However, it requires a serious investment of time (and often times some money) to identify, locate, acquire, organize, and understand the books, papers, and online resources that will help you gain a solid understanding of the relevant issues. This book was written to help the reader navigate this large body of information. There is also a significant amount of useful research which has been around for several decades but has been hidden in obscurity, and so a goal of this book is to present this research to a wider audience.

This book does not assume that the reader is an expert with set-builder notation[1] or category theory[2] or that the reader is familiar with specific mathematical symbols that have been used in previous papers on the subject. Thorough explanations are preferred over terse explanations.

Audience

This book can serve as an introduction to topics on data and parsing for the working software developer or student of computer science. The book covers foundational topics of parsing using concrete examples and common tools used in the field. In an academic setting, the book would serve well as a companion book for courses that only cover parsing as part of a larger topic, such as compiling.

[1] `https://en.wikipedia.org/wiki/Set-builder_notation`
[2] `https://en.wikipedia.org/wiki/Category_theory`
[3] `https://creativecommons.org/publicdomain/zero/1.0/legalcode`

Prior Work

Prior to the writing of this book there were no public domain books on parsing.

While there are quite a few papers and articles written for academic journals, these articles are typically written for a narrow audience by use of vocabulary and notation. Also, many important papers on parsing are not freely available; one must purchase a copy in order to read it, or have access to a very good library. Finally, while there are some good online resources such as the work of Federico Tomassetti et al, they are lost within a sea of information that is either outdated or poorly produced, or only covers a limited set of topics.

The following is a list of commercially published books that the author has been able to review.

1. Dick Grune, Ceriel J.H. Jacobs
   Parsing Techniques, A Practical Guide, Second Edition, 2008

   This is the best book focused on parsing for the general audience.

2. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
   Compilers: Principles, Techniques, and Tools, Second Edition, 2007

   This is a very popular book that covers lexing, parsing, and compilation. The chapters on lexing are particularly good, covering DFA construction. This book is commonly known as the "Dragon Book" due to the illustration on the cover.

3. S. Sippu, E. Soisalon-Soininen
   Parsing Theory, 1988

   This book describes parsing in terms of category theory and is appropriate for students who have taken graduate level classes in mathematics.

4. Nigel P. Chapman
   LR Parsing, Theory and Practice, 1987

   This book describes parsing in the language of mathematics and is appropriate for students who have taken undergraduate level classes in mathematics.

5. John-Paul Tremblay, Paul G. Sorenson
   The Theory and Practice of Compiler Writing, 1985

6. William A. Barrett, John D. Couch
   Compiler Construction: Theory and Practice, 1979

Terms and Symbols

Many of the ideas covered in this book originated in the 1950s and 1960s, near the dawn of the computer age. In the decades since, researchers and

educators have defined quite a few names for various concepts surrounding data and parsing. They have also adopted numerous symbols and notations as a convenient shorthand to write down their ideas. While it can be tedious to learn (and explain) these names and symbols, it is necessary if you want to be able to follow discussions or read original research papers on these topics. In some cases, words that you may be familiar with in common language may have a different definition in academic papers. This can be confusing if you are not already familiar with that term. Therefore, in this book, we will error on the side of explaining the academic terms and symbols when they are used. Sometimes it takes a number of definitions conspiring together to explain a larger concept, so bear with us if it seems like we are just defining words for no apparent reason.

Prerequisites

A knowledge of programming concepts is required.

Table of Contents

Chapter 1

Parsing Concepts

1.1 Strings

1.2 Languages

1.3 Grammars

context-free grammars are an ideal.
`https://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt.html`

1.3.1 Terminals

A terminal is an atomic element of a language.

1.3.2 Nonterminals

Nonterminals are often referred to as variables.

1.3.3 Productions

A production is a rule that map a nonterminal to a sequence of terminals.

1.4 Tokens and their Sub-Grammars

Terminals are typically referred to as tokens in practice. Technically, a terminal is just a language element, while a token typically contains additional information about the context in which the element was recognized.

Tokens are typically defined by their own grammar based on regular expressions. This grammar is typically separate from the language grammar.

http://savage.net.au/Ron/html/graphviz2.marpa/Lexing.and.Parsing.Overview.html#Grammars_and_Sub-grammars

Modern parsing system are supporting Unicode with increasing frequency. Unicode is its own subgrammar.

Therefore, it is likely that there will be three layers of grammar involved in defining a language.

1.5 Parse Trees

1.5.1 Concrete Syntax Tree

A tree that accurately reflects the grammar that produced it.

1.5.2 Abstract Syntax Tree

This is a parse tree that is abstraction of a concrete (or grammar-based) parse tree in order to be independent of the grammar that produced it.

1.5.3 Document Parsing

```
parse_tree = parse(input)
```

1.5.4 Stream Parsing

```
for event in parse(input):
    handle_event(event)
```

For bottom up parsing, events are produced as wholly formed subtrees are recognized. For top down parsing, events are produced for creation of the root node, and for children of a previously parsed node.

1.5.5 Incremental Parsing

Parse trees are needed for incremental parsing:

```
new_parse_tree = reparse(old_parse_tree, text_delta)
```

https://channel9.msdn.com/Blogs/Seth-Juarez/Anders-Hejlsberg-on-Modern-Compiler-

1.5.6 Language Servers

https://microsoft.github.io/language-server-protocol/

1.6 Parsing Applications

There are important differences between parsing data files versus parsing computer languages.

1.7 Parsing Software

There are handwritten and generated parsers.

Chapter 2

Lexers

2.1 The Role of the Lexer

Lexemes and Tokens

Why perform lexing independently from parsing?

Wirth, in Compiler Construction, states

"A partitioning of the compilation process into as many parts as possible was the predominant technique until about 1980, because until then the available store was too small to accommodate the entire compiler"

"The design decision to decouple the lexer and parser has existed since at least Antlr 2 and likely made for separation of concerns and performance reasons. Where the lexer is driven by the parser, the lexer may have to re-lex portions of the input text many times in different contexts before the parser is able to satisfy a rule. The Antlr lexer does everything it can, and largely succeeds, in processing the entire input text in a single pass. Just different approaches taken with the same good intentions." `https://stackoverflow.com/questions/28873463/no-context-sensitivity-in-antlr4`

A pipeline supports separation of concerns and allows reuse of a lexer with other parsers that accept a stream of tokens.

`https://rosettacode.org/wiki/Compiler/lexical_analyzer`
`https://rosettacode.org/wiki/Compiler/Sample_programs`

2.1.1 Lexers, Tokenizers, and Scanners

The simplistic view is these are all the same. The more nuanced view is that there are distinctions between these words that can be helpful.

A *lexer* is software that takes data as input and outputs tokens, (i.e. it acts as a tokenizer) using a scanner to process the input data.

2.1.2 Scannerless Parsing

The concept of *scannerless parsing* was explored thoroughly by Salomon and Cormack in "Scannerless nslr(1) parsing of programming languages" July 1989 It may not be immediately clear why simply *not* using a scanner is an idea being discussed in the year 1989. If you read Knuth's 1965 paper on LR parsing, there is no mention of scanning. One would presume that the lack of a scanner was a normal thing. But strictly speaking, his paper was simply agnostic about the nature of the terminals. The parsing process he describes presumes the input string is composed of a pure sequence of atomic elements. There is no further discussion on the possible structure or origin of those elements.

In practice, however, a scanner is typically used because it allows the resulting grammar to be much simpler than a grammar in which every character is a terminal. A simpler grammar can often be parsed with a simpler parser, which is an essential benefit of using a scanner.

Explains the scanner is just an optimized parser: `https://softwareengineering.stackexchange.com/questions/337676/what-is-the-procedure-that-is-followed-when-w`

Explains that scannerless parsing doesn't have all the cool regex features like negation: `https://github.com/antlr/antlr4/issues/1527` Please, remove lexer from Antlr

2.2 Scanning

2.2.1 Input Data

2.2.2 Rules for Tokens

2.2.3 Literals

2.2.4 Regular Expressions

`https://regex101.com/` `https://news.ycombinator.com/item?id=23975041`

Using capture groups for acceptance allows non-captured expressions to indicate negation: `http://www.rexegg.com/regex-best-trick.html`

2.2.4.1 Regex DoS

`https://levelup.gitconnected.com/the-regular-expression-denial-of-service-redos-`
`https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/`

2.2.5 Pseudo-Tokens

2.2.6 Context-Free and Context-Aware Scanning

references: `https://www.umsec.umn.edu/publications/Context-Aware-Scanning-Parsing-E`
`http://tree-sitter.github.io/tree-sitter/creating-parsers#lexical-analysis`

2.3 Scanner Conflicts

When two different token rules match the same string, that is a *scanner conflict*.

2.3.1 Traditional Lexer Precedence

A *traditional lexer* resolves conflicts between tokens using *traditional lexical precedence* as defined by [Denny, Def 2.2.6] and repeated here.

Traditional lexers have been context-free lexers. They also resolve conflicts using the *longest match* rule first, followed by *positional priority* as described in the next two sections, respectively.

2.3.1.1 Longest Match Rule

2.3.1.2 Positional Priority

2.3.2 Longest Match Conflict

A longest match conflict occurs when a token chosen by a traditional lexer leads to an error state in the lexer or parser, whereas if a different token, such as a shorter one was chosen, the parse would succeed.

For example, 1..10

If 1 and 1. are valid tokens (for integers and floating point numbers), then these sequences are valid:

```
(1.) (.10)
(1) (..) (10)
```

Longest match favors the first sequence, but that token sequence is invalid to the parser, but the second sequence is valid.

The presence of this problem is a combination of:

- The traditional lexical and grammatical model cannot express this language without special handling.
- Trying to use or design a language that requires this special handling

For example, scannerless parsing with an LR parsing algorithm will be able to parse this string. Therefore, one might argue that the burden is on the parsing software to handle this issue. The counterargument is that the language designer can avoid this issue by being thoughtful of it during the design process and verifying that these conflicts do not exist using proper tools.

(Check to see whether lex's lookahead operator can solve this. this would work if the length used for longest match rule includes the lookahead string (seems unlikely but who knows))

Assuming you are using traditional scanning software, a strategy for handling longest match conflicts, which makes the grammar less portable between various parsing tools.

2.3.3 Context-Invasive Lexical Conflicts

A scanner conflict that only occurs because a token is active that is not associated with the current parsing context is given the name *context-invasive conflict* in this book.

2.3.3.1 An Example of an Invasive Conflict

2.3.4 Resolving Lexical Conflicts

2.3.4.1 Lexical Modes

Lexical modes are a tool for resolving lexical conflicts by creating separate sets of active tokens.

`https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md#lexical-modes`

2.3.4.2 Lexical Lookahead Operator

A lexical lookahead operator can specify a required pattern to follow a lexeme in order to match the token. This pattern is not included in the lexeme.

2.3.4.3 Lexical Predicates

A lexical predicate is arbitrary code that runs in order to determine whether to match the token.

2.3.4.4 Match Multiple Tokens

2.3.5 Academic History of Token Conflicts

2.3.5.1 Nawrocki

Token conflicts were first formalized by Nawrocki in 1991.

2.3.5.2 Keynes

The thick dot in some of his definitions are used two different ways:
universal quantified predicate (text p. 5, pdf p. 11)
`http://www.open-std.org/jtc1/sc22/open/n3187.pdf`

set comprehension term p. 20
`https://www.cs.ox.ac.uk/files/3389/PRG68.pdf`

2.3.5.3 Denny

Chapter 3

Parsing Algorithms

3.1 Left-to-Right Parsing

3.2 Left and Right Derivations

3.3 Top-down Parsing

This produces a left-most derivation. Top down mimics walking hierarchical data.

3.3.1 Predicitive Parsing

`https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm`

3.4 Bottom-up Parsing

This produces a right-most derivation. But LR might be able to produce LL (top-down) output: `https://cs.stackexchange.com/a/69996`

3.5 Parsing Families

"The primary difference between how LL and LR parsers operate is that an LL parser outputs a pre-order traversal of the parse tree and an LR parser outputs a post-order traversal."
`http://blog.reverberate.org/2013/07/ll-and-lr-parsing-demystified.html`

http://blog.reverberate.org/2013/09/ll-and-lr-in-context-why-parsing-tools.html

https://stackoverflow.com/questions/5975741/what-is-the-difference-between-ll-

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/100%20Bottom-Up%20Parsing.pdf

On why OCYacc generates a full LR(1) state machine: "The reason for this is that, in todays' computing environment, it may not necessarily be desirable to reduce the number of states in order to save space in the parsing tables." https://chaosinmotiondotblog.files.wordpress.com/2017/08/ocyacc-building-lr1-glr.pdf

3.6 The LL Parsing Family

3.7 The LR Parsing Family

LR is particularly difficult to implement manually.

3.8 CYK Parsing

3.9 Early Parsing

3.10 PEG Parsing

Parser Expression Grammars (PEGs) are a bit of a mix Using syntactic predicates can preclude using an lr or ll parser because of unbounded lookahead

3.11 Parser Combinators

"Efficient Parsing with parsing combinators" J Kurs et al.

Chapter 4

LL Parsing

4.1 LL Table Structure

4.2 LL Table Building

4.3 LL(1) Parsing

4.4 LL(k) Parsing

4.5 GLL Parsing

Afroozeh A., Izmaylova A. (2015) Faster, Practical GLL Parsing. In: Franke B. (eds) Compiler Construction. CC 2015. Lecture Notes in Computer Science, vol 9031. Springer, Berlin, Heidelberg

4.6 LL(*) Parsing

The parsing algorithm of ANTLR3.

4.7 ALL(*) Parsing

The parsing algorithm of ANTLR4. `https://www.antlr.org/papers/allstar-techre`
`pdf`

Chapter 5

LR Parsing

5.1 LR Table Structure

5.2 LR Table Building

5.2.1 LR Table Conflicts

5.2.2 Avoiding LR Table Conflicts

5.2.3 Resolving LR Table Conflicts

5.3 SLR(0) Parsing

5.4 Canonical LR(1) Parsing

Uncompressed, not large. "Large" is a carryover from a time when an LR table challenged memory resources of a typical computer. The difference now, between 100K and 1M is not much. In the 1980's, that could have meant much more.

On why OCYacc generates a full LR(1) state machine: "The reason for this is that, in todays' computing environment, it may not necessarily be desirable to reduce the number of states in order to save space in the parsing tables."
`https://chaosinmotiondotblog.files.wordpress.com/2017/08/ocyacc-building-pdf`

5.5 LALR Parsing

LALR is an optimization for reducing the size of the Canonical LR tables by having certain similar states share the entry in the parsing tables. However, some languages that are correctly parsed by CLR are not parsed correctly with

the smaller LALR table. Therefore, LALR is considered less powerful than CLR.

5.6 Minimal LR Parsing

It has been shown that there is a relatively simple method to determine which set of LR(1) states might cause a conflict if they share the same table entry.

A *Minimal LR Parser* only merges the states that are safe to merge. If the language requires separate states that LR(1) supports then those states will be left unmerged, but if the language is simpler and merging states does not create a conflict, then the states are merged. In this way, the parsing table is roughly sized according to the needs of the language.

This begs the question, why hasn't Minimal LR parsing been more popular?
`https://stackoverflow.com/a/42915777`

5.7 GLR Parsing

Ambiguity can be a PITA. Multiple parse trees are difficult to deal with. Tree-sitter uses GLR.

Chapter 6

Indentation-Sensitive Parsing

6.1 Indentation-Sensitive Languages

Indented Languages are not described by Context-Free Grammars

Python, YAML, and Haskell are indentation sensitive.

`http://trevorjim.com/parsing-not-solved/` `http://trevorjim.com/python-i`

6.2 Parsing Indented Blocks

6.2.1 Grammar Annotations

6.2.1.1 Grammar terms annotated with Counts

Indentation Sensitive Languages
2006 Leonhard Brunauer, Bernhard Muhlbacher
`https://www.google.com/search?q=LeonhardBrunauer+and+Bernhard+M%C3%BChlbacher.+Indentation+sensitive+languages`

The scheme allows a specific character to be matched a specified number of times.

For example, a tab character, represented by $\rightarrow$, can be repeated m times in a grammar with this syntax:
$C_{\rightarrow}^{m}$

This rule states the start variable expands to zero tabs followed by a statement:
$S \rightarrow C_{\rightarrow}^{0} Statement$

This rule states that the statement after an "if" condition has greater indentation:
$C_{\rightarrow}^{m} Statement \rightarrow C_{\rightarrow}^{m}$ "if" $Condition$ " : " $newline\, C_{\rightarrow}^{m+1} Statement$

```
if true:
    a=1
```

6.2.1.2 Relationship Annotated BNF

Principled Parsing for Indentation-Sensitive Languages 2013
`https://michaeldadams.org/papers/layout_parsing/LayoutParsing.pdf`

6.2.2 Explicit Indent Checking?

6.2.2.1 GLR Filtering

Layout-Sensitive Generalized Parsing
Erdweg S., Rendel T., Kästner C., Ostermann K. (2013) Layout-Sensitive Generalized Parsing. In: Czarnecki K., Hedin G. (eds) Software Language Engineering. SLE 2012. Lecture Notes in Computer Science, vol 7745. Springer, Berlin, Heidelberg

6.2.2.2 Data-Dependent Grammars

Ch. 3: "In this chapter we propose a parsing framework that embraces context information in its core."
`https://homepages.cwi.nl/~jurgenv/papers/PhDThesis_Ali_Afroozeh_and_Anastasia_Izmaylova.pdf`

Section 3.1:
Decls ::= **align** (**offside** Decl)*

| **ignore**('{' Decl(';' Decl)* '}')
This definition clearly and concisely specifies that either all declarations in the list are aligned, and each Decl is offsided with regard to its first token (first alternative), or indentation is ignored inside curly braces (second alternative).
align and offside keywords desugared to data-dependent expressions (section 3.3.5)

6.2.3 PEG Semantic Predicates

`https://gist.github.com/dmajda/04002578dd41ae8190fc`
Note that the 'I' provides an explicit token indicating that indentation is coming so this doesn't really work for arbitrary indentation like YAML.

6.2.4 Pseudo-Tokens and Pseudo-Grammars

6.2.4.1 Indented Blocks

INDENT means match pending spacing more than last indent and increases the indent level by one.

`indents.push(new_indent))`

DEDENT means pending spacing less than last indent. DEDENT represents the reduction of one level of current indentation. Multiple DEDENTs may appear in sequence.

`indents.pop()`

consumes no input

Parsing indentation requires context aware lexing. Otherwise, a dedent would always be active. Also, verbatim text would need a way to actively turn off the indentation tokenization somehow.

6.2.4.2 Explicit Current Indentation

SAMEDENT means match pending spaces equal to last indent.
Prior Work:
`https://stackoverflow.com/questions/11659095/parse-indentation-level-wit`

Implementation strategy:
If you use SAMEDENT and the parser has an adaptive lexer, you can limit indentation logic to just the tokenizing of specific tokens (INDENT/SAMEDENT/DEDEN only when they are active. If you wanted to combine a grammar with indent tokens with a grammar without indent tokens (as a sublanguage for example), then it makes the lexer easier if it knows, explicitly, when indentation is relevant, which makes it easier to write.

6.2.4.3 Implicit Current Indentation

Same indentation is implied in the grammar, I.E. indent checking is always on.
`https://docs.python.org/3/reference/grammar.html`

```
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
```

note that stmt does not refer to indentation (no SAMEDENT)

If you can't use SAMEDENT because the tokenizor cannot (or it's too difficult to) keep enough context, then indentation logic will need to be pervasive across the tokenizer, activating after every newline. if you need to suspend indentation, you'll need to program that into the tokenizer for that situation (opt-out)

6.3 Parsing Flow-Style Lists

6.3.1 Grammar

First field needs to be defined differently (without samedent) than remaining fields

`MARK_INDENT means indents.push(cur col), consume no input`

Use MARK_INDENT after LIST_SEPARATOR

6.4 Parsing Blank Lines in Indented Blocks

We need to swap tokens. Adding psuedo tokens is not helpful. Stream based token swapping solves this.

Chapter 7

Constructing a Parser

You can hand write a parser or use a parser generator to automatically generate a parser PEG is typically partially handwritten and partially generated.

- Prototype with custom tokenizer
- LR(1) Grammar with no token conflicts
- Validate with various Parser Generators
- Rewrite as top down
- Fuzz test for both, validating equivalency

7.1 Handwritten Parsers

7.2 Parser Generators

Reality of Parser Generators

- Limited language support Not necessarily an issue if you are just trying to prove out your grammar
- Multiple approaches to grammar syntax and expressibility
- No standard lexing strategy
- No standard parse trees or eventing (sax)
- No standard code integration strategy with the grammar (see PEG.js example for indents)
- So many to evaluate

Selected Parser Generators

- LALR Parsing
  - Flex/Bison
  - Jison
- ALL Parsing
  - ANTLR4
- GLR
  - Tree-Sitter
- PEG
  - PEG.js

Chapter 8

A Model Data Parser

8.1 Design Choices

- Grammar driven dynamic Lexing
- Token-specific custom lexing
- Emit multiple tokens per scan
- Minimal LR(1) Parsing
- Layered Grammar
  - map: map-text
  - list: list-text

Appendix A

Data

A.1 Motivation

This book covers the topic of data because a parser operates on data, and there are important issues to be aware of with a common form of data called Unicode. These issues are listed at the beginning of this chapter in order to motivate the reader to explore these topics further in detail. The rest of the chapter is written to give the reader a full understanding of these issues, from the ground up, so that the reader can be confident that they have dealt with them appropriately when they arise in the context of parsing or any other context.

The first issue is that there are some symbols in the Unicode standard which have multiple definitions. This can lead to potential problems if this issue is not accounted for properly. For example, it requires a strategy for determining whether two names, for example, are the same. Another set of issues arise because it may be unclear what one means by the *length* of a sequences of symbols.

These issues with Unicode are typically either not well known or not understood by both the users and developers of parsing software. When the strategy for handling these questions is left undocumented there is risk that software will behave in an unexpected way, potentially leading to serious consequences.

A.2 Symbols

The signs, sounds, marks, and words humans use to represent ideas are called *symbols*. For example, the twenty six letters of the English alphabet are symbols representing sounds or parts of words. Words are also symbols for the ideas they represent. The numerals 0 through 9 are symbols for numbers. As long as something represents, or *symbolizes* something else, it is a symbol. We use symbols as sort of "real world" currency to remember or communicate something meaningful to ourselves or others.

A.2.1 Interpreting Symbols

Wikipedia defines *data* as any sequence of symbols for which "a specific act of interpretation" gives those symbols meaning[1]. In other words, the essential nature of data is that is in a *sequence*, which means that the symbols occur one after another, and that there is a particular meaning associated with the ordering and choice of the symbols in the data.

A.2.2 Manipulating Symbols

Computers are well known for their ability to manipulate data. A computer can read and *execute*, or follow, instructions as represented by a sequence of symbols. These instructions can, in turn, direct the manipulation of symbols stored in the computer's memory. It would be confusing to think of a computer working with the kind of visual symbols that humans typically think of like a + (plus) sign or the numeral 2, because they do not actually work directly with those kinds of symbols. The symbols that a computer can read, follow, and manipulate are quite limited. In fact at the lowest levels, there are only two symbols. Inside the computer, these symbols exist as electrical states inside the transistors and wires of the computer. In an electric model, we think of these states as *off* and *on* and by convention these states symbolize the numbers 0 and 1, respectively.

A.3 Binary Data

A.3.1 Bits

A numerical symbol which is either 0 or 1 is known as a *binary digit*, or *bit* for short. At the most fundamental level, bits are the letters of a very simple alphabet that both humans and computers can manipulate in an equivalent way. For example, both humans and computers are capable of numerically adding together the symbols 1 and 0 to produce the symbol 1, although they go about it in different ways. It is through these common definitions and ways of processing symbols that computers can perform computations that are relevant to humans.

A.3.2 Binary Codes

Since computers are only able to manipulate sequences of bits, we have devised an incredibly rich variety of ways, known as *codes*[2] to map the information available to us into sequences of ones and zeros. For example, *decimal*[3] numbers, which are the standard numbers we use from day to day, can be converted to a

[1] `https://en.wikipedia.org/wiki/Data_(computing)`
[2] `https://en.wikipedia.org/wiki/Code`
[3] `https://en.wikipedia.org/wiki/Decimal`

binary number composed of binary digits, using a *binary code* as seen in Table 1.1.

Decimal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table A.1: Map of Decimal Digits to Binary Digits

A.3.3 Bytes and Octets

A *byte* is typically 8 bits. A byte also represents the smallest unit of a computer's memory that is *addressable* by location. Computer systems store and retrieve bytes of data in specific locations of the computer's memory and these locations are identified using a number, called a *memory address*. There are examples of old computers that had more or less bits but it is standard now to assume a byte has 8 bits. In some contexts, such as with networking standards, the term *octet* may be used instead of byte if there is a concern that the definition of a byte is ambiguous.

A.3.4 Base-N Number Systems

People typically communicate using numbers which are split into digits based on the number ten. This is called the *decimal numeral system* When a digit exceeds nine, another digit to the left represents ten times the current digit. Further digits can be added to represent higher numbers based on powers of ten. Numbers in the decimal number system are also called *base-10* numbers.

With binary numbers however, when a digit exceeds *one*, another digit is added to the left that represents *twice* the current digit. Binary numbers are based on powers of two and are called *base-2* numbers.

In addition to these number systems, two other systems are commonly used with computer systems, called *hexadecimal* and *octal* numbers which are base-16 and base-8 numbers respectively.

A.3.5 Octal Notation

Octal is a numerical encoding that allows the use of 8 different symbols for each digit, 0 through 7. Each symbol encodes 3 bits of a binary number.

A.3.6 Hexadecimal Notation

Hexadecimal is a numerical encoding that allows the use of 16 different symbols for each digit, 0 through 9 and A through F. Lowercase letters are also used but mixing upper and lowercase letters is uncommon. Each symbol encodes 4 bits, or a nibble, of a binary number. Hexadecimal numbers are convenient because you can always represent one byte with only two digits.

Decimal	Binary	Octal	Hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	18	10
17	10001	19	11
18	10010	20	12
19	10011	21	13

Table A.2: Numbers in Various Numeric Notations

A.4 Character Encodings

Characters are symbols that represent sounds or parts of words.

A.4.1 ASCII

An early standard character encoding was ASCII, which stands for American Standard Code for Information Interchange.

A.4.2 Terminal Control Characters

A computer terminal is composed of a display monitor for displaying characters to humans and a keyboard to send characters to the computer. These characters are displayed in rows and columns.

Control characters were invented in order to control terminal operations using character codes, such as sending a character code to go to the next line. Character codes are still supported, although many of them are outdated.

A.4.3 Unicode and UTF-8

The Unicode® Standard is a universal character encoding standard for all written characters and text.

A.4.3.1 Codepoints

A.4.3.2 Graphemes

A.4.3.3 Unicode Normalization

Unicode Normalization is a process used to resolve ambiguities in the Unicode Standard.

A.5 Summary of Chapter A

Index