8

A UNIFIED APPROACH TO DESIGN

Reference Materials

ANSI/IEEE X3.215–199x. Draft Proposed American National Standard for Information—Programming Languages—Forth. X3J14 dpANS–6—June 30, 1993.

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995.

Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003.

Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press. 1996.

Online Resources

www.gnu.org: GNU Foundation. *GForth Reference Manual*

www.Forth.org

8.1 INTRODUCTION TO DESIGN

Designing a large system is much different from designing a single object or function. The close relationship of object-oriented design methods and object-oriented programming languages, for example, is no mistake: programming languages, software engineering, and design have co-evolved throughout the history of computer science.

However, there are many other disciplines that use design principles and therefore other approaches. This chapter outlines the use of what has been learned in psychology about problem solving. Although there seems to be a dearth of texts in computer science about design, such books abound in the engineering sciences, architecture, and the trades. One approach

adopted in engineering is to treat design as problem-solving. This approach is not unknown in mathematics (Pólya 1957) and computer science (Simon 1969).

8.2 TWO ELEMENTS

Design of a large project requires a great deal of organization. Anecdotal evidence from industry indicates that a large project such as a compiler requires three to five years. Another piece of anecdotal evidence is that an experienced programmer produces 2.5 debugged lines of code per hour. With such a long development time, documentation and coordination are paramount concerns. Many documentation methods and approaches have been proposed but there is no silver bullet (Brooks 1975).

The approach here is based on concepts of problem solving as proposed by psychologists. The ideas are relatively simple and easy to document but they're still no silver bullet. The two elements are called *concept maps* and *schemata*.

8.2.1 Linguistic Issues

A *concept* is a word that names a class of objects, real or imagined. Concepts play a central role in epistemology, the theory of knowledge. Concepts are often thought to be the meanings of words or, perhaps better, the names of the meanings of words. Thus, a concept is a single word that can stand in for many different categories. A *category* is a set of objects. For example, the concept written "city" names a number of possible categories: "city" or "metropolis" or "town," but not "village." This idea of concept and of category is very important to us during design. The idea of *class* in object-oriented design is the same as our idea of category.

Concepts are essential to ordinary and scientific explanation: someone could not even recall something unless the concepts they hold now overlap the concepts they held previously. Concepts are also essential to categorizing the world; for example, recognizing a cow and classifying it as a mammal. Concepts are also compositional: concepts can be combined to form a virtual infinitude of complex categories, in such a way that someone can understand a novel combination. For example, "programming language" can be understood in terms of its constituents as a "language for programming machines." However, if one understands "language" but not "programming," then "programming language" is meaningless.

The set of all words that we can use in discussing a project is its *lexicon*. The lexicon can be used to develop the *vocabulary*. The vocabulary

is the set of pairs of a lexicon word and its *definition*. In order to understand design we must understand the concepts and relationships among them. The concepts and relationships come from the specification and this is almost always presented in a graphical form. We therefore must first undertake a linguistic analysis of the specification. Linguistic analysis provides us with the lexicon of the project and the vocabulary.

The above examples show that concepts can be related with one another. A well-known example in artificial intelligence and object-oriented programming is the "is-a" relationship. The "is-a" relationship is a variant of Aristotle's genus-species rules: "A tree is a graph."

So there you have it: concepts, relationships, lexicon, vocabulary. Concepts are related to other concepts based on stated relationships *in the context of the discussion at hand*. These relationships can be graphed as a DAG. Why DAGs? Because we cannot afford to have cycles in definitions!

8.2.2 Schemata

Concept maps portray factual knowledge, but such knowledge may not be static. The concept "bachelor" as an unmarried adult male is a fact of the definition and is unlikely to change. On the other hand, the concept of "democracy" is not so trivial. Concepts play an important role in the design because they set the range of assumed knowledge: the facts of the specification.

Design in computer science has a unique requirement. The product must always be a program written in a programming language. In order to do that, you must do the following:

1. Recognize the concepts in the requirements.
2. Determine the semantics of the problem in the *problem* vocabulary. This is the concept map.
3. Determine the semantics that must work in the programming language.
4. Determine the syntactic constructs to encode the semantics.

This is part of what makes programming hard: the translation of semantics to semantics, not syntax to syntax.

We present here one possible way to understand this process using a concept central to the psychological process of problem solving: the *schema* (sing.) or *schemata* (pl.). A schema has four interconnected parts:

1. A set of patterns that can be matched to circumstances
2. A set of logical constraints and performance criteria
3. A set of rules for planning
4. A set of rules for implementing

This idea actually originated in early artificial intelligence research in the 1960s. Rule-based programming systems such as CLIPS were developed based on the schemata concept. Concepts play a role in the first two items. We spend the rest of the chapter giving examples of these ideas.

A very simple example would be the following problem:

> Write a program in C to compute the sum of the first N cubes.

The linguistic analysis is easy: *C program* is obvious; *cubes* refer to *cubes of integers*. The phrase *compute the sum of* immediately indicates a loop of some sort. In this case, the "loop of some sort" is a schema.

1. The pattern that we recognized was the need for the loop based on the statement of the problem.
2. There is an obvious logical requirement that deals with the counter of the loop. There is a subtle question here; do you see it?
3. There is some planning that must be done for a loop: in this case, N must be known to control the loop. Where does it come from? And what do we do with the output? The input requirement (for N) invokes several schemata: we could read it in from a file or read it on the command line or some other element. The output also invokes several possible schemata.
4. When all the planning is done, we can write the loop.

Did you consider the question in point 2? How big can N be? The summation of cubes runs as N^4. You can look that up in a mathematics handbook; the key point was to realize that there is a logical requirement on N. For a 32-bit signed-magnitude architecture, only $N < \sqrt[4]{2^{32}} = 2^8$ computes properly.

8.3 USING THESE IDEAS IN DESIGN

What does this have to do with compiler design? The design of the transformations is effectively schemata.

1. The input is a series of elements that have linguistic meaning.
2. The input must meet a required pattern. The input can be broken into a series of patterns through decomposition. The name of each schema comes from the grammar.
3. Each pattern, when recognized, causes a standard processing sequence to be invoked. The processing is recursive, but the number and content of each pattern schema are known. After imposing logical requirements and planning requirements (which is usually a decomposition of sorts), the transformation can be processed.

$$
\begin{aligned}
E &\rightarrow T + E \mid T - E \mid T \\
T &\rightarrow F * T \mid F / T \mid F\%T \mid F \\
F &\rightarrow I \in \text{int} \mid (E)
\end{aligned}
$$

Figure 8.1 The Old Faithful Grammar

How does this work in practice? Here's the famous Old Faithful grammar that recognizes integer arithmetic expressions (Figures 8.1 and 8.2). Figure 8.1 is really shorthand for the grammar shown in Figure 8.2. Recall from the discussion of grammars in Chapter 5 that the elements on the left-hand side are called nonterminals and any symbol that does not appear on the left-hand side is called a terminal. Why these terms are used will become apparent below.

By convention, E is called the sentential symbol: every possible string of symbols (called a sentence) made up of integers and the operations symbols must be constructible beginning with just the E. Although the form of the rules may appear strange, they can be converted to a program (see below).

8.3.1 Understanding the Semantics of Grammars

In order to make sense of this schematic form of programming, let's invent a simple interpreter that takes programs written as above and executes them. This exercise is crucial: it shows exactly what we must design to produce a compiler.

$$
\begin{aligned}
E &\rightarrow T + E \\
E &\rightarrow T - E \\
E &\rightarrow T \\
T &\rightarrow F * T \\
T &\rightarrow F / T \\
T &\rightarrow F\%T \\
T &\rightarrow F \\
F &\rightarrow \text{any integer} \\
F &\rightarrow (E)
\end{aligned}
$$

Figure 8.2 Unrolled Old Faithful Grammar

All the rules for a given nonterminal can be gathered into one C function by developing rules to follow when translating one to the other. In our case, we only need to go from grammars to programs; that you can go the other way is covered in the proof of the Chomsky Hierarchy theorem.

8.3.1.1 Reading Terminals

It is clear that the terminal symbols appear verbatim on the input. We can assume that there is a function called `read_symbol()` that reads the next well-formed symbol from the input if there is one; it returns `EOF` if there is no more input. It will be useful to have a function `next_symbol(string)` that returns *true* if the next input symbol is that string and *false* otherwise.

8.3.1.2 Dealing with Nonterminals

There are two ways we can deal with the nonterminals:

1. We can look on the input and determine what nonterminals the input might have come from. This process can be made very efficient, but it is not particularly intuitive. Such an approach is called *bottom up*.
2. We can start at the sentential symbol and *predict* what should be there. This is not as efficient because in general one has to search a tree (bottom up does not in general do this) although there are techniques to prevent this. This search technique is called *top down* or *recursive descent* when no backup can occur.

We will take the recursive descent approach because it leads to a more intuitive design process. Remember, you can't optimize a nonfunctioning program. We're working for understanding here, not speed.

Working for F upward, how would we program a C program to do F's functioning?

$$F \rightarrow \text{any integer}$$
$$F \rightarrow (E)$$

In keeping with the theme of schemata, what is the most general processing schema we can think about? The illustration here uses the *input–process–output* schema. Every algorithm you will ever write will have those elements. That's interesting, but what is the schema for a program? It has

- A name
- Some arguments represented by names and defined by types
- Some processing constructs using `if`, `while`, and so forth
- Some method of returning values or perhaps writing results

```
returntype programname(arguments) {
           Processing
           Return
}.
```

Figure 8.3 A Program Schema for Programs

We can see that the first two follow from the *input* requirement and the last one from the *output* requirement. The C schema for this is something like that shown in Figure 8.3.

How would we map the productions for F onto this schema? Clearly, F is the name, so let's call it `progF`. What is the input? That is not clear so we will wait on this. What does it return; it may not be clear but it needs to return a tree (Figure 8.4).

In terms of our schema model, we have pattern matched things we know from the problem to the schema. Let's focus on the *Processing* part. It is clear that there can only be two correct answers to F's processing: either we find an integer or we find a left parenthesis. Anything else is an error—or is it? What about an expression such as $(1 + 2)$? This causes us to consider a different issue: that of *look-ahead*. Put that issue on a list of issues to be solved.

There is another issue for F. If F has a left parenthesis, how does it process the nonterminal E? The rule we want to have is that each nonterminal is a function and therefore recognizing a string requires *mutual recursion* among several functions. The thought pattern is "If F sees a left parenthesis, then call `progE` with no arguments. When `progE` returns, there must be a right parenthesis on the input."

8.3.2 Designing a Postfix Printer Program

To illustrate how to use the grammar to design, let's develop a program that prints an expression in Polish postfix. There are two issues: (1) how

```
tree progF( ??? ) {
     tree Result
     Processing
     return Result
}
```

Figure 8.4 Schema for `progF`

```
match expression {
      case pattern: code
      ⋮
      default: code
}
```

Figure 8.5 `Match` Schema

to translate the grammar form above into a C program and (2) what is the starting point?

Let's invent a new program element, called `match`, that syntactically looks like a C `switch` statement. The general form is shown in Figure 8.5.

Furthermore, let's assume that someone has actually implemented the `match` command in C. This is not that far fetched because functional languages of the ML-based variety all have such a facility. In order to organize this we turn each *nonterminal* into a function name, for example, *E* into `progE` (Figure 8.6). Each function has one argument, `Tree`.

Everything in this program except the `match-case` are regular C. Try your hand at completing the functions `printT` and `printF` before considering the hand calculator program.

To review, the use of the `match-case` construct encodes the schema. The `case` construct is a pattern. Because the program is so simple, we did not have a separate criterion, planning, or implementation phase. However, the `printT` and `printE` can be thought of as planning steps. The `putchar` function is clearly an implementation step. Also, there is no knowledge to apply to the schema.

```
void printE( tree Tree ) {
     match Tree
     case T+E: printT(T); printE(E); putchar('+'); return;
     case T-E: printT(T); printE(E); putchar('-'); return;
     case T: printT(T); return;
     default: abort();
     }
}
```

Figure 8.6 Program to Print *E* Recognized Terms

```
int progE( tree Tree ) {
    match Tree
    case T+E: return (progT(T)+progE(E))
    case T-E: return (progT(T)-progE(E))
    case T : return progT(T)
    default : abort()
    }
}

int progT( tree Tree ) {
    match Tree
    case F*T: return (progF(F)*progT(T))
    case F/T: return (progF(F)/progT(T))
    case F%T: return (progF(F)%progT(T))
    case F : return progF(F)
    default : abort()
    }
}
```

Figure 8.7 Complete Design for Print Program

CASE 48. MATCH-CASE SEMANTICS

Write a one- or two-page paper that outlines *exactly* how the `match-case` statement should operate.

8.3.3 Calculator

One of the classic examples of this form of programming is the program that acts like a hand calculator. This example appears in the so-called "Dragon Book" of Aho, Sethi, and Ullman (1986); it is called the "Dragon Book" because of the distinctive cover featuring a dragon.

In the hand calculator program, we take in any expression that is syntactically correct by our grammar and this expression is a tree. The purpose is to compute the integer result. You should know from your data structures experiences that the expressions matched by Old Faithful grammar can be represented in a tree (see Figures 8.7 and 8.8). **Note**: What are the logical requirements for correct computation based on the limits of representation? Can you check these *a priori?* How would you find out the limits have been violated?

```
int progF( tree Tree ) {
    match Tree
    case I: return atoi(I)
    case(E): return E
    default : abort()
    }
}
```

Figure 8.8 Complete Design for Print Program (continued)

8.4 DESIGN RECAP

The design of a compiler follows from the material in Chapters 5 to 8. The principles are

1. The input text is converted to *tokens* that capture the type of grammar terminal.

```
void progE( tree Tree ) {
     match Tree
     case T+E: progT(T); progE(E); print("+")
     case T-E: progT(T); progE(E); print("-")
     case T : progT(T)
     default : abort()
     }
}

void progT( tree Tree ) {
     match Tree
     case F*T: progF(F); progT(T); print("*")
     case F/T: progF(F); progT(T); print("/")
     case F%T: progF(F); progT(T); print("%")
     case F : progF(F)
     default : abort()
     }
}
```

Figure 8.9 Complete Design for Gforth Program

```
void progF( tree Tree ) {
     match Tree
     case I: printf(I)
     case(E): progE(E)
     default : abort()
     }
}
```

Figure 8.10 Complete Design for Gforth Program (continued)

2. The tokens are converted by the parser into a tree that captures the deep structure of the program.
3. The trees can be processed, sometimes multiple times, to produce structures that describe the sequence of operations.
4. The final structure can be converted to the program run by a computer, or, in our case, Gforth.

As a final example, let's convert the calculator program into Gforth code (Figures 8.9 and 8.10).