Chapter 1

Preliminaries

The advantage of simple and clear language concepts
is that their implementation in a compiler
is easily and efficiently possible
without causing unexpected overhead.
–Leo Geissmann, PhD thesis

1.1 Overview

. . . of the Book

There are hundreds of programming languages in use, each with one or more translators. It is expected that the reader will be familar with several of them. The principal purpose of this book is to enable the reader to implement more translators. The reason to embark on such an implementation might be interest in the translation process, or the need for a programming language, either new or for which there is no readily available translator. One cannot, from the information in this book, construct compilers that are competitive with the products produced by mature engineering organizations – the devil is in the details and the details are not in this book.

The book contains many exercises, ranging in difficulty from easy to impossible. The serious student will read all the exercises and work many of them. There is at least a partial answer to each exercise in Appendix 5.

This first chapter in Volume 1 is devoted to preliminaries. Grammars are presented in Chapter 2. Formal semantics is presented in Chapter 3. Performance is discussed in Chapter 4. Advanced topics are presented in Chapter 5.

. . . of the Chapter

This chapter has three goals:

1. the informal presentation of programming language X,

2. some suggestions for extensions to X, and
3. the presentation of notation for sets and relations.

The reason to start with these topics is to get the reader going on a project and to prepare for the rest of the text. The pedagogical assumption is that one learns by doing. It is necessary to get started early so that the mechanics of doing things are no longer a hurdle when deeper concepts are presented.

The programming language X is presented in enough detail that the reader can write programs in it. Because an implementation is available, the reader can also run those programs. The language is relatively sparse — constituting a base language upon which to build. The idea is that students or researchers will start with a programming environment implementing a base language and then add, change or subtract features to suit their own special purposes. The environment is called Hyper. The Hyper environment is considerably more responsive than traditional compiler environments. During development of a program within Hyper, feedback is essentially instantaneous. The implementation environment, Hyper itself, takes no longer to change than any modest-sized conventional program.

The mathematics is relatively straightfoward. It allows substituting concise formalisms for the otherwise verbose English descriptions and diagrams. For some of the material this is just a matter of style but for other material it presents an opportunity for a deeper and more consistent treatment.

Section 1.2 is a descriptive presentation of the main constructs and concepts of the base language. Three features — macros, procedures and subprograms — are not implemented in the base system. The definitions of these features are given in Chapter 5.

Section 1.3 suggests projects. The reader is asked to name, design, and implement a programming language. For the student they are term projects. For the researcher they are hints on how to escape the uncomfortable embrace of some available compiler. Both kinds of users are encouraged to look for linguistic ideas in the examples in this section as well as in the scientific literature of their favorite application area. The project, like the exercises, is intended to be the vehicle for "learning by doing."

Section 1.4 presents some notation for describing and manipulating sets and relations. The notation is used throughout this text.

What you should have learned

The structure of the book and this chapter, what is expected of the reader, the pedagogical approach.

1.2 Informal Definition of X

... the art of language designing becomes

the art of designing a powerful and systematic language
primarily from those elements which can be processed elegantly by
a translator...
– *Edsger Dijkstra, foreword to Algol 60 Implementation*

The next two sections principally prepare the student for the term project. They can be skipped by those readers wanting to get right to the details of compiler writing.

Philosophy

One can observe the use of notation in other scientific disciplines, and admire the power that it brings to its users. What would the calculus be without the integral sign, or inorganic chemistry without reaction equations? It is true that scientific notation evolves. The mechanism is trial and error in the technical literature. What works survives. No committee decides. The inventors of notation are its users.

Computer languages, as a topic of discussion, do not invite calm and considered opinion. Perhaps we programmers are too close to them, or perhaps "language user community" is an inherently political concept. All agree that computer languages have provided an essential abstraction of the programming process. Most would like even higher levels of abstraction. But agreement on what new features are needed and what old features can safely be discarded is hard to achieve. One can observe that computer languages tend to grow, gathering constructs whenever the political climate is right. It is hardly ever time to throw things out; for every old feature there is a saving argument or a mass of entrenched use. The consequence is that widely used computer languages become clumsy. It is the intention of this book to return programming languages to their users by providing the users with the details of personal compiler implementation.

The language X is derived from the ideas presented by Edsger Dijkstra in *A Discipline of Programming*[**?**] and further developed by David Parnas in *A Generalized Control Structure and its Formal Definition*.[**?**] The central theme is simplicity. In fact X is so simple that it is hard for an experienced programmer to think of something interesting to write in it. Never mind: this book tells you how to add your favorite constructs. A second theme is elegance. If one is going to achieve Dijkstra's "compelling and deep logical beauty," one must forego much of the paraphernalia of conventional general purpose programming languages. This spartan attitude is most likely to succeed when the objective of the language is rigorously limited and the linguistic additions are few and well suited to meeting that objective. Here is the most important advice in this book – *don't let your ambition lead you to solving other people's problems*; at this stage your problems are enough by themseleves.

Informal Semantics

For the most part an experienced programmer will understand programs in X. If one writes:

```
a := 1;
```

one may infer that `a` is an integer-valued variable assigned the value 1. X is strongly typed, thus the type of `a` is constrained by the context to be integer. There is no other way to ascribe type to a variable in X. This assignment to `a` is apparently wasted, since the value is not subsequently used.

If Hyper is directed to run this program, it will present the result

```
a := 1;
```

and await further directions. Output is achieved in X by having Hyper report any wasted assignments. Thus, this simple program exhibits both typing of variables and output. One can now guess that the program:

```
a := b+1;        `this is a comment
```

will require an input value for `b`. Indeed, when this program is run in Hyper, the user will be asked

```
b := ?
```

Upon receiving an integer value, Hyper will present output for `a` as before.

The meaning of a program is its input/output mapping; the effect of running a program in Hyper is to display that input/output mapping. All input is collected before program execution and no output is reported until the completion of execution.[1]

The order of input requests and output reports is constrained to be consistent from run to run, but otherwise nondeterministic. The runtime system must supply sufficient information to make the input/output actions unambiguous — associating the name of the variable and its value is sufficient. The wise programmer will pick mnemonic names for input variables.

The state of a program is the set of variable-value pairs. The number of variables and the values of variables change during execution. The dynamic sequence of changes is, by intention, invisible to the user. The initial state is provided by *preactive input*; the final state is reflected as *postactive output*; what happens in between is the program's private business.[2]

For interactive programs, those that respond during execution to input from the user, there must be a richer concept of input and output, but even that does not imply that the programmer need know *everything* about the sequences of actions. Some extensions to X that implement programmer-controlled interaction are suggested in the exercises.

The comment convention is "backquote-to-end-of-line." The comment style used in this book is a combination of "banner" comments introducing major program parts, and, in addition, right-margin comments, aligned vertically for ease of reading.

[1]Batch input is provided by the subprogram facility. See Section 5.1.

[2]Highly optimizing compilers necessarily take the same viewpoint.

There is also a macro definition and expansion facility (using `#`) which is discussed in Section 5.3.

Statements

There are several kinds of statements in X. Where one statement can appear, a list statements joined by semicolons can appear. A statement list can be empty; a trailing semicolon never causes trouble.

name	*example*
assignment	`x, y := 1, z+2`
block	`be  y.`
	`    y := x + 3;`
	`    x := y*7`
	`eb`
selection	`if  x < y -> x := y-1`
	`::  x >= y ->`
	`fi`
iteration	`it`
	`    if  x < y -> x := x+7`
	`    ::  x >= y -> exit`
	`    fi`
	`ti`

Table 1.1: The Statements of X

Blocks, private and nonlocal variables, scope

A block contains a list of statements with its own state. The block is delimited by the pair `be-eb` (`be` means "begin block" and `eb` means "end block") and starts with an introduction of *private nomenclature*. The names introduced immediately after the `be` pertain only to the text bounded by `be-eb`. Consequently, any name appearing in a block and not explicitly listed in the nomenclature of that block is implicitly nonlocal (corresponding to the concept of name scope in conventional block-structured programming languages). Because a statement list can appear anywhere in the language that a statement can appear, the `be-eb` pair only needs to be used when introducing local nomenclature, rather than also for grouping statements.

Each private variable must be assigned a value before its value is used. The part of a block prior to an initializing assignment for a variable is called the *preactive region* for that variable. The part of the block subsequent to the last evaluation of a variable is called the *postactive region* for that variable. Between these regions is the *active region*. If there is a block nested within a block, and some name occurs in the nomenclature for both, the active region for the inner use of that name is subtracted from the active region for the outer use. Use of a name outside its active region refers to that same name in some containing block. In effect this rule turns the "uninitialized variable" into a block input mechanism and the "wasted assignment" into a block ouput mechanism.

The words "prior" and "subsequent" have more than one interpretation. The most satisfactory definition, presented in Section 5.2, depends on a flow analysis of the program. The one used here is textual order. One consequence of the preactive/postactive region concept is that a single name within a `be-eb` pair may represent two different variables, one private and one "somewhere out there." The user can avoid any appearance of ambiguity by chosing unique names for all variables, private or not.

Programs: outer list, implicit name introduction

A program in X is a list of statements. This is the *outer list.* Names not in the private nomenclature for any block are called *outer variables.* The meaning of a program is as if the outer list were embedded in a `be eb` pair and all outer variables listed in its nomenclature.

If a variable occurs in its preactive or postactive region, it refers, by the rules for variables in blocks, outside the block. The same is true for an outer variables; in this case they refer outside the program altogether, a kind of external block. It is these "external" occurences that cause interactive input and output.

For example, the program

```
x := 0;                     ` outer variable (level 0)
be x y.                     ` level 1 nomenclature for x and y
   y := x;                  ` x on level 0 and y on level 1
   be x.                    ` level 2 nomenclature for x
      x, y := y+1, 4;       ` x on level 2, y on level 1
      y := y+x;             ` ditto
   eb;                      ` end of level 2
   x := y;                  ` x on level 0, y on level 1
eb;                         ` end of level 1
output := x;                ` report result
```

will cause the output

```
output := 5;
```

There are, in fact, three different variables named `x` and one named `y` (and one variable named `output`).

Exercise

1. [1,1] Using colored pens or some other convenient device, identify all of the preactive, active and postactive regions for variable x and y in the preceding example. There are a total of $3 \times 4 = 12$ such regions.

Expressions and Type

X has two data types, *integer* and *boolean*. Arithmetic is exact. Any transformation of an expression that preserves its value, such as commutativity for addition, is permitted to the implementation.[3]

The arithmetic operators are unary negation (`-`), addition, subtraction, multiplication, division and remainder (`+ - * / //`). The boolean-valued operators are comparisons (`< <= = <> >= >`), logical or, logical and, and logical negation (`\/ /\ ~`).

The concept of an identifier is nearly universal in programming languages. It is a sequence of letters, and perhaps digits and underscore. An *operator identifer* is similar – a sequence of operator symbols. Thus `+`, `:=` and `->` are operator identifiers. This provides a rich set of names for extensions to X. Adjacent identifiers of the same kind must be separated by whitespace.

The type of an expression is derived from the type of its operands. It is a requirement of X, and its dialects, that constants of different types have different appearance. Thus if `2` is of type integer, one must use `2.0` or something else distinguishable from `2` for constants of type real. The type of a variable must be consistent with the values that are combined with it and assigned to it. For example, the assignment

```
x := y<1;
```

implies `x` is boolean and `y` is integer.

```
x := y;
y := z+1;
```

implies that all of `x`, `y` and `z` are integer.

```
x := y;
y := z;
```

only implies that all the variables have the same type, but the determination of what type is deferred until input is seen.[4]

Assignment

Assignment (including input) is the only way state is changed. The form of an assignment is a list of variables followed by the assignment operator followed by an equally long list of expressions. The meaning of an assignment is a state-transition that behaves as if all the right-hand side expressions were evaluated, and then each simultaneously assigned to its corresponding variable. In particular, the assignment

[3]When the underlying host system provides the capability (such as on the VAX), integer range overflow causes program abortion.

[4]Dynamic type is a rare situation and can always be avoided by inserting some non-effect arithmetic.

```
x,y := y,x
```

interchanges the values of `x` and `y`. If a variable is repeated on the left of an assignment, the result is nondeterministic for the value of that variable.

Control: selection and iteration

There are two control constructs in X, selection (`if` – `fi`) and iteration (`it` – `ti`). Within `if` – `fi` brackets there is a set of alternatives. Each alternative starts with a guarding boolean expression. Each guard is followed by a statement list.

The meaning of a selection is program abortion if there are no true guards and otherwise it is the meaning of the statement list behind some true guard. It is good programming style to make the guards mutually exclusive.[5] *All* of the guards must be defined (no division by zero, etc.) for the selection to have meaning.

An iteration `it` S `ti` with statements S repeats statements S until `exit` is executed somewhere in S and outside any nested `it-ti` pair. In any case the iteration continues *to the end* of S before terminating.[6]

Guards: logical operators

The guards are logical expressions, combining relations over the expression values and the constants `true` and `false` with boolean operators. It is in the nature of X that guards cannot cause a change in the state of the program (except to initialize variables).

Subprograms and Procedures

Implementing procedures requires, or at least strongly suggests using a stack for local storage. The other features of HYPER can be implemented with a simple, static storage allocation scheme. To keep the presentation and examples simple, static allocation is used and procedures are, therefore, not implemented. See Section 5.1 for the changes needed to implement a run stack and procedures.

Functions

The boolean-to-integer type conversion function `b2i` is the only function that can be used in expressions. One can add other such functions by extending the implementation.

Examples

Suppose you wish to find out if three lengths can form a triangle. The underlying wisdom is that the shortest two sides must be long enough together to span the

[5]However, Dijkstra makes a case for non-determinism and overlapping guards.
[6]Unlike `break` and `return` in C.

longer side. The following program will report `triangle := true` if the input will form a triangle and `false` if not.

```
long,mid,short := side1,side2,side3;      ` input

it                                        ` sort
   if long < mid -> long,mid := mid,long
   ::  mid < short -> mid,short := short,mid
   ::  else exit
   fi
ti;

triangle := long < mid+short              ` output
```

The interactive input and output reflecting one case of the input/output mapping might appear:

```
side3 := 7;
side2 := 3;
side1 := 2;
triangle := false;
```

Suppose you wish to test whether an integer is prime. The idea is to examine the remainder after division by each feasible divisor. One can quit after trying everything up to the square root of the candidate value.

```
n := PrimeCandidate;                  ` input
trial := 2;                           ` smallest possible
it                                    ` look for candidates
   if n//trial = 0 -> exit            ` found divisor
   ::  trial*trial > n -> exit        ` enough, so quit
   ::  else trial := trial + 1        ` keep trying
   fi
ti;

NumberIsPrime := trial*trial > n;     ` output
```

In this case the input/output might appear as:

```
PrimeCandidate := 97;
NumberIsPrime := true;
```

Suppose you want to find a right triangle with integer-valued sides, two of which are given. The idea is to use the Pythagorean theorem and try all candidates. The value `0` signifies failure.

```
x, y := side1, side2;                   `input
z, result := 1, 0;
```

```
it
   if x*x = y*y + z*z
   \/ y*y = z*z + x*x
   \/ z*z = x*x + y*y -> result := z
   ::  else
   fi;

   if z < x+y -> z := z + 1
   ::  z >= x+y -> exit
   fi
ti;

side3 := result                         ` output
```

One could write a more efficient program. If x is greater than y one of the tests can be eliminated. Since neither x nor y is changing, x*x + y*y and x*x - y*y could be computed before beginning the interation. Furthermore, since (z+1)*(z+1) is 2*z + 1 greater than z*z, all of the multiplies in the loop can be eliminated. Doing all of these leads to a program that is about twice as long.

Exercises:

2. [20,50] Install Hyper.[7]

3. [20,50] Run each of the examples above in Hyper.

4. [20,50] Write the optimized program (suggested above) to find right triangles.

5. [20,50] Write another interesting program in x. Run it in Hyper.

Debugging

The lack of provision for intermediate output during execution is acceptable for working programs, or programs that have been proven correct. It is not practical, however, when the program behaves differently than expected. While one can always go back to the analysis to find an error, it is often expeditious to "open the bonnet" and do some troubleshooting. Traditional development systems provide output statements as well as debuggers for this purpose. Hyper provides postmortem examination of variables. One could add monitoring of assignments, break points, single-stepping and *in vivo* examination of the types, scopes and values of variables. Such tasks make acceptable, though slightly off-topic, term projects.

[7]See distribution package instructions.

When the user is also extending Hyper itself, standard debugging tools for the implementation language (presumably c) are essential.

1.3 Term Projects

Compiler projects are designed to satisfy certain constraints. The projects must exercise compiler-writing skills. They should be intellectually stimulating. They cannot require an unreasonable amount of time and effort, either for the student or for the instructor. They must not consume inordinate amounts of computer resources. They should have no commercial value.[8]

Where the details below do not seem to fit your desires, try to find a creative way to read the instructions so that they do.

There are a number of parts to a well-executed compiler project. Each takes time and care, and often compartmentalized knowledge and skills. It therefore is often constructive for students to work in teams of two or three on a single group project. Some extra effort is necessary to insure that each team member understands all the important lessons learned doing the project.

It is one of the pleasant accidents of the field that fast compilers are within the abilities of beginning students. The essence of the accident is that heavy-duty algorithms take heavy-duty implementation efforts, computational resources, and understanding. But it happens that there are light-duty algorithms for the entire range of tasks, and they lead to compilers that are graceful and pleasant to use. And, often, fast-compiler building skills have application to other kinds of programming projects.

The actual code consists of several relatively independent modules. The Hyper distribution package comes set up to build and test each module separately, as well as for integrating the collection of modules and testing again. The student is advised to learn and use this build structure rather than just making changes and testing the whole.

The specific deliverables for a one-term compiler project are:

1. *Project Plan.* Any project needs planned milestones. The compiler project is on a tight schedule and is pushing the skill-set of the implementers — therefore it needs milestones all the more. Each of the tasks below needs a start date, a work estimate and a planned time of completion. A template and example of a project plan are given in Appendix B. The student should not be discouraged that the plan asks for detailed estimates just when one has the least concrete knowledge upon which to base them. That is in the nature of planning. Plans get updated as work progresses. Given an understandable plan, the instructor will advise changes early enough to avoid a serious mismatch of time and ambition.

[8]It is common for beginners to think they can get a start in business with a well-done compiler project. Making a *product* dooms the term project. The beginner cannot experiment much with compiler writing if time is taken up trying to meet commercial criteria.

2. *Source Language Description.* The input language must be defined. Typically the description is in terms of grammars and a user manual. The user manual defines the meaning of the language, either formally or informally, and provides the user with enough information to cope with the diagnostic messages that the compiler will ultimately have to generate. The size of the grammar is a major determinant of the reasonableness of the project: 60 grammatical rules is plenty. Typically a student will modify some grammar presented in this book rather than write one from "scratch."

3. *Front end.* Most programming languages have separate lexical and structural definitions. The scanner and parser together are often referred to as the *front-end* and the output of the front-end is referred to as the *intermediate language.* In the example implementation of a front end for X, everything is expressed as C code. In an industrial strength version, much of the front end could be expressed as tables (typically grammar tables).

    (a) The *scanner* is a module which is responsible for the lexical definitions. It reads the source text and passes it on as a sequence of tokens to the rest of the compiler. Typically a fast scanner is implemented in 200-500 lines of C code. Extending the given scanner typically involves only a small percentage of its code.

    (b) The *parser* is responsible for the structural definition. It must analyze the source program (as a sequence of tokens) and report what is found in a way that is convenient for the rest of the compiler. Typically the implementation of a fast parser requires 10-20 lines of very regular C code per grammar rule. Changing the given parser also requires 10-20 lines of code per change in the corresponding grammar.

4. *Back end.* The back end is responsible for taking the intermediate language and turning it into executable form for the target machine. The target machine could be an interpreter (either pre-existing or written by the compiler implementor) but for the compiler projects suggested in this book, *interpretive output is forbidden* since it allows the student to skip some essential compiler writing experiences. As in the case of the front end, much of the back end of a mature compiler would be expressed in tables. As before, such a structure here would only complicate the learning experience.

    The Hyper back end consists of three separately testable modules: a generator, symbol table mechanism and an emitter. (In some compilers the symbol table is considered to be part of the front end.) As in the front end, a small change in the language will require only small changes in these modules.

    (a) The *generator* receives the output of the parser and drives the symbol table and emitter. Changing the generator requires touching about 5 lines of code per grammar change.

(b) The *symbol table* records information about the variables. It should not require changes except to add kinds of attributes to the tables. The symbol table code contains an fairly intricate hashing mechanism which is probably more trouble to change than it is worth for the purposes of this book.

(c) The *emitter* formats and places the machine language form of a program. This is the ultimate goal of the compiler. If the target machine is to be changed (say from one manufacturer to another), the emitter may have to be largely rewritten (roughly 1000 lines of code). [9]

5. The *run time* puts the compiled program into execution and supplies the run-time services (input, output, standard subroutines, and the like). The Hyper compiler puts raw machine code into main memory. When the compiler is done, the run time module gets the hardware program counter pointed at the machine code and turns it loose.

   A more standard approach is for the compiler to write a file containing executable code, data and symbolic information. Other programs can later read one or more such files, collect the code, link it together, place it in memory, and start it into execution. The Hyper approach is simpler and faster, but it has the disadvantage of being able to run programs only embedded within Hyper.

   Separate execution of applications, running x programs from within a c program, and suspending Hyper itself, are discussed in Section 5.4.

6. *Report.* The final description of the project must be complete, but should also be brief. Giving differences from x rather than a definition from scratch is one way to reduce page count. The report must stand on its own, with sample output and tests, source code for essential and interesting changes, and a critique of what was easy or hard and why. A template for the final report is included in Appendix B.

Skeletal programs, documents, project environments, previously completed project reports and the like add significantly to the depth to which a project idea can be explored. These auxiliary materials also add to the instructional component of the project if they follow exemplary standards and show creative directions. It is expected the students will have access to such materials in the distribution package, or supplied by the instructor.

Project Suggestions

Anything that fits within the structure above, usually a subset of a popular programming language, is a suitable first compiler project. It is often a good strategy to build your project on top of a very simple language, rather than hack

[9] This is one place that tables are better than c code.

some industrial strength product down to size. The language X is provided as a reasonable starting point. Various changes and extensions are suggested.

1. —*Base Languages*— The language X, less macros, subprograms and procedures, is a base language. A subset of C, PASCAL, or ADA with approximately the same facilities can also be a base language. There are cosmetic differences. There are also differences in elegance which turn into a substantive matter when extensions are planned. In any case, the name L0 will be used in this section to denote any one of these minimal bases upon which to build a project; LK means the K$^{\text{th}}$ extension. All examples will be given in terms of X itself.

2. —L1— Replace data type `integer` in L0 with type `real`. The following example computes the base of natural logarithms e in such a langauge. The idea is to expand the continued fraction 2 1 2 1 1 4 1 1 6 1 1 8 . . . until the desired tolerance is reached.

   ```
   t := tolerance;                ` input
   i := 2.0;                      ` start at 2 1 1 4 1 1 ...
   ao, a := 2.0, 3.0;             ` numerator
   bo, b := 1.0, 1.0;             ` denominator
   it
      if abs(ao/bo - a/b) > tolerance ->
         ao, a := a, ao+a*i;      ` i
         bo, b := b, bo+b*i;
         ao, a := a, ao+a;        ` 1
         bo, b := b, bo+b;
         ao, a := a, ao+a;        ` 1
         bo, b := b, bo+b;
         i := i + 2.0
      ::  else exit
      fi
   ti;

   e := a/b                       ` output
   ```

   The limit of the series $1 - 1/3 + 1/5 - 1/7 + 1/9\ \ldots$ is $\pi/4$. Since the increments have alternating signs, the average of two successive values is more accurate than either.

   ```
   t := tolerance;                ` input
   i := 1.0;                      ` beginning of series
   x := 0.0;                      ` x & y are alternate values
   p := 0.0;                      ` initial approximation
   ```

```
it
   q := p;                      ` remember old approximation
   i, y := i+2.0, x+2.0/i;      ` add first term
   i, x := i+2.0, y-2.0/i;      ` subtract second term
   p := x+y;                    ` best approximation

   if p - q < t -> exit         ` quit iteration if good enough
   ::  else                     ` otherwise keep going
   fi
ti;

pi := p                         ` output
```

3. —L2— Extend L0 to include procedures.

4. —L3— Extend L0 to include type declarations.

5. —L4— Extend L0 to include pointers to any variable, including pointer variables.

6. —L5— Replace type integer in L0 with type complex with functions `Re` and `Im`. Experiment with the notation `|a|` for absolute value of `a`.

7. —L6— Replace type integer in L0 with some specific group element such as integers modulo n.

8. —L6=7— Do two or more of the above simultaneously (getting risky).

9. —EWD— Implement Dijkstra's language.[**?**]

10. —*Diagnostic compiler*— Arrange for superb diagnostics for LK. At a minimum each diagnostic should explain what caused the compiler trouble, where in the input the compiler detected the problem (maybe more than one place, e.g., a declaration and use of a variable do not agree), and where to look for advice on how to fix things up. The "weight" of the diagnostic should be proportional to the sublety of the problem. Easy problems should have concise diagnostics; tricky problems may elicit a soliloquy.

    Insure that the problem reported is the user's, not the compiler's (e.g., quit after the first diagnostic if necessary). Consider diagnostics in the style of "**O Master, I do not understand your divine intent**..."

11. —*Magic*— Build a compiler that *repairs* errors made by the user and then lowers its head and charges ahead into the input. Such systems have been built[10] but it is my opinion that they are not cost/effective when embedded in a very responsive environment.

[10]notably PL/C at Cornell

12. —*Extended precision*— On top of one of the Lk languages, implement an extension as follows:

    (a) —*Variable precision*— Let integers have arbitrary precision.

    (b) —*Fast variable precision*— Let integers have arbitrary precision *and* have target code execution speed as fast as for the base Lk subset. (Obviously where the base version would fail, the new version can afford to be slow without losing the race.)

    (c) — ∞ *and* Ω — Let integers have arbitrary precision and two new values: `inf` and `undef`. The value `inf` is signed. Be sure to state the rules of combination for operations accepting the new values.

13. —*Sets and relations*— Much of the neat formal mathematics that underlies compiler technology can be based on sets. Compiler students therefore often have a soft spot in their hearts for sets. To these students we dedicate the following projects.

    (a) —*Set of* `int`— Add a new type set (meaning finite set of integer) and operators for union, intersection, difference, membership, and contiguous sequence to L0.

    (b) —*Efficient compact sets*— Implement sets using bit-vectors to represent all small compact sets.

    (c) —*Using predicates*— Implement sets adding boolean operations, | for a *suchthat* separator, and predicate set definitions. E.g.
    $\mathtt{X} := \{\mathtt{y} \mid 1 < \mathtt{y} \wedge \mathtt{y} < 100 \wedge \mathtt{y} \neq 13\};$
    $\mathtt{X} := \mathtt{X} - \{2 * \mathtt{y} \mid \mathtt{y} \in 1..10000\}$

    (d) —*Infinite sets*— Implement sets according to one of the definitions above *and* extended precision arithmetic. Now the set `1..inf` is an acceptable computational data item. Add set complement to the language.

    (e) —*Sets of ?*— Replace set of integer with set of some other interesting kind of data item, such as real or character or string.

    (f) —*Sets of sets*— Extend the meaning of set to include sets of any valid data type in your language, including sets themselves. Add a powerset operator `^`, set union `||`, APL compression `||/`*`expr`*, etc. Test your implementation with the following program.

```
` Build a 3x3 square puzzle out of 2x2 v's.
puzzle := {00, 01, 02, 10, 11, 12, 20, 21, 22};
squares := {                          ` all 4 2x2 subsquares
   {00, 01, 10, 11}, {10, 11, 20, 21},
   {01, 02, 11, 12}, {11, 12, 21, 22}};
tiles := {                            ` all 12 2x1 tiles
   {00, 01}, {01, 02}, {10, 11}, {11, 12},
   {20, 21}, {21, 22}, {00, 10}, {10, 20},
   {01, 11}, {11, 21}, {02, 12}, {12, 22}};
t := {p||q | p in tiles /\ q in tiles /\ size(p||q) = 3};
vees := {p | p in t /\ q in squares /\ size(p&&q) = 3};
z, y := 2^vees, {};              ` potential solutions
it                               ` try one at a time
   if size(z) > 0 ->             ` more to do
      s := choice(z);            ` pick one trial
      z := z -- s;               ` remove it from z
      if size(s) = 3 /\ size(||/ s) = 9 ->
         y := y || s             ` put it into y
      ::  else                   ` if it is a solution
      fi
   ::  else exit                 ` no more trials
   fi
ti;
covers := y;                     ` output
```

Solutions following this style are possible for the 8 queens, the mutilated checkerboard of Golomb and the soma cube.[11]

(g) —*Relations*— A relation is a set of pairs. Add to one of the set languages a type "relation", cross product, transitivity-multiplication between relations, suffix operators `*` `+` for transitive completion, functions `Domain` and `Range`.

14. —*Matrices*— Replace type integer in Lk with type `matrix` meaning $m \times n$ matrices; `[[1,2],[3,4]]` is a typical constant integer matrix. Implement a `submatrix` function; 1×1 matrices are type-correct for matrix elements; $1 \times n$ and $m \times 1$ are column and row vectors. Implement inner product and cross product.

    (a) —`float`— Let the matrix elements be of type float. For $n \times n$ matrices, `1/A` is the inverse of `A`; `det(A)` is the determinant of `A`.

    (b) —`complex`— Let the matrix elements be of type complex. Implement functions `Re` and `Im`.

    (c) —`boolean`— Let the matrix elements be type boolean. Implement the operators `|` `&` `~` .

15. —*Strings*— Strings of characters can be manipulated as arguments and results of library packages, or via built-in operators of a language. The principal implementation decision has to do with how string storage is managed. Some languages (PL/1, C, Pascal) require that each string variable have a fixed upper bound on the length of string that can be

[11]The problems are often discussed in the Mathematical Recreations department of Scientific American magazine.

stored in it. Other languages such as AWK, SNOBOL and XPL have either no limit or a large universal limit. In analogy to the variable precision arithmetic discussed in an early project suggestion, type string here will mean the unlimited length variety. String constant delimiters and (perhaps) character delimiters are needed. There are a number of approaches.

(a) —PL/1 *strings*— Implement the PL/1 string functions `substr`, `||`, and `unspec` as an extension to LK.

(b) —*Rosin operators*— Implement the operators `length`, `cat`, `before`, `after`, `[`*expr*`]`, `[`*expr*`..`*expr*`]`.[12] One might write:

```
x := "abc";
y := x after (x before "b");
x[2] := 'a';
y[0..4] := x[1..2] cat "def";
```

A subscript expression extracts a character as an integer value; a subscript range extracts a substring. Operators `before` and `after` give substrings. All of the operators except `cat` can be used on the left and right of the assignment operator. Given strings a, b and c, some obvious relations between the operators are:

```
((a cat b) after a) = b
(a cat b)[0..length(a)−1] = a
(a cat b)[length(a)..]  = b
((a cat b) cat c) = (a cat (b cat c))
(a cat "") = a
```

Why isn't the following also an invariant?

```
((a cat b) before b) = a
```

16. —HITEX— TEX is a macro language which describes complex ways to glue together little rectangles to make bigger rectangles, and so on up to page size. Design and implement a high-level language HITEX to do some subset of TEX-like things. You might consider one of the string languages above as a starting point. Your target language most likely will be TEX itself.

17. —*Static Analysis*— Static analysis is the process of (mechanical) examination of program text to discover properties of the program. Often the properties of interest are pathological constructs, surely indicative of a programming error. The UNIX program `lint` is a popular but sometimes inadequate program of this type. The idea of this exercise is to abandon the generation of code altogether in favor of implementing some useful analysis tools.

[12] The operators `before` and `after` were suggested to the designers of PL/1 by Bob Rosin. They were not adopted.

(a) —*Static trace*— Implement a static analyzer for LK that prints the set of possible values for each variable after each assignment to it. Such sets may be infinite. Refer to the languages including `set` above for hints.

(b) —*Aliasing(1)*— Do the above exercise after adding a pointer modifier to the types of LK.

(c) —*Anomalies*— Implement a static analyzer for LK that detects *all* side-effect anomalies and access anomalies. The assignment

```
y:=(z:=1)+(z:=2);
```

is a typical side-effect anomaly leaving `z` undefined if the order of evaluation or operands is undefined (as it is in C). Two assignments in a row `y:=3; y:=4;` cause an access anomaly since the first value of `y` is (wastefully) discarded.

(d) —*Aliasing(2)*— Do the previous exercise for a version of LK containing pointers.

(e) —*Flow analysis*— Implement a static analyzer for LK that will (sometimes? usually? always?) detect dead code and infinite loops.

18. —*Pretty printing*— The UNIX programs **cb** and **indent** are elaborate examples of existing pretty printers. Implement a translator that, for an input program in LK, produces an equivalent program obeying some formatting rules. At the least, properly indent the whole program. The spacing of operators may also indicate hierarcy (e.g. `x + y*z`). A pretty printer should be idempotent. There are some variants:

(a) —*Heavy-handed beauty*— For a language that allows multiple declarations (e.g. `int x, y;`) separate each variable into a private declaration followed by a dummy right-margin comment:

`int` x;   / ∗ *describe x please* ∗ /
`int` y;   / ∗ *describe y please* ∗ /

Don't destroy any existing comments in the process.

(b) —*Very pretty printing*— Make TEX or LATEX output. Cause reserved identifiers to be set in boldface, variables in italics, and whatever other pretty typography suits you. For example, the line of C:

```
    if (x==1) x = x +1; else x=x+x*3/x;
```

might turn into the TEX text:

```
    {\bf if} $(x==1)$ $x = x +1$; {\bf else} $x=x+x*3/x$;
```

which, when processed by TEX becomes:

if $(x == 1)$ $x = x + 1$; **else** $x = x + x * 3/x$;

(c) —*Industrial strength beauty*— Instead of Lk, pretty print some widely used language. Your program need not respond gracefully to input errors but must handle the most grotesque uses of the language. The hardest problems are dealing with large comments, long strings and excessive nesting. A stylish and constructive refusal is a permissable response to an unreasonable request.

19. —*Applications*— There are many more possible projects than can be proposed in these pages. The best of them will be unexpected, a result of someones deep knowledge of a computational problem, perhaps from science in general, perhaps from some other source. The following list is meant to be evocative, to inspire the reader to that unpredictable creative leap from which our legacy of knowledge always comes:

    - —*Predicate calculus, logic programs,* Prolog— Given an existentially quantified predicate, say $\exists x.B(x)$, one method of proof is to instantiate the variable with a constant, say a, and prove $B(a)$ instead. Any general method of choosing a and checking $B(a)$ is a programming system with predicate calculus as its language. The output, in this case, is the value a. Consider researching and implementing a language in this class.
    - —*Differential and integral calculus*— Only a few of the interesting solutions to differential and integral equations can be expressed in closed form; the rest must be numerically approximated. Consider a language designed for some such class of problems. Instead of implementing solutions in an array data structure, one might invent type `mesh` with operators `N, E, S, W` and maybe `NE, SE, SW, NW` to move around within it. A mesh might be constant sized or have a varying granularity. A problem might be to solve some set of equations with a given boundary condition, or track an oscillation over the mesh, or something else. The output could be graphic.
    - —*Organic and inorganic chemistry*— Chemistry combines discrete conditions (CH_4) with continuous conditions (energy, position, orientation). Consider implementing a language where chemical entities have data types and invariants of the discipline are built into the compiler.
    - —*Knitting and Weaving*— The preparation of cloth is highly algorithmic — the Jaquard loom was one of the first "computers." Research the current technology in describing cloth (a magazine on knitting is a good place to start) and implement a language to describe cloth-construction. The output could be a picture of the product or instructions to a cloth-making machine.
    - —*Massively parallel targets*— For any compiler one can consider a new target machine. Parallel architectures are an interesting kind of target.

- —`termcap`— Any UNIX system contains a long list of terminal characteristic descriptions. Given such a description one should be able to emulate the terminal on any reasonable hardware. Build a termcap compiler and use the second window of HYPER to emulate the resulting terminal. One should feel free to redefine the termcap language into a more elegant form.
- —*Register-transfer level machine descriptions*— It is often important to describe computers before they are built. Such a description would be compiled into a machine interpreter which would then execute its own machine language in the second window of HYPER.

Wild flights of fancy are too risky for the beginning student, but may be the only reason for a more seasoned investigator to attempt to master HYPER.

1.4 Formalisms

Real programmers do not use Greek letters.

The theory of compilers can be expressed in terms of logic, sets and relations. The reader is expected to be familiar with these concepts. This section establishes a uniform notation for the mathematics of this book, some of which is standard and some of which appears only here. The notation used in this book is summarized and tabulated in Appendix A. Many useful relations between the notational conventions are also given there.

When a symbol or formula is defined in terms of previously established material, the form

$$\textit{new} \overset{\text{def}}{=} \textit{old}$$

is used. Otherwise '=' means equality for whatever type of operands it has.

Propositions and Predicates

The operands for the propositional calculus are given, in increasing binding hierarchy, in Table 1.2.

iff	means	logical equivalence
$\Leftrightarrow$	means	logical equivalence
$\Rightarrow$	means	logical implication
$\vee$	means	logical or
$\wedge$	means	logical and
$\neg$	means	logical negation

Table 1.2: Propositional Operators

The extension of the propositional calculus to the predicate calculus introduces the two quantifiers '$\forall$' (read as 'for all') and '$\exists$' (read as 'there exists'). They are like declarations in programming languages in that they introduce new variables of a specific type for a part of a logical expression. The form of application is given by example in Table 1.3. The scope of the bound variables is the predicate to the right of the separating dot '.'. Parentheses are used only if the scope of operators or quantifiers is unclear. If the universe of quantification, or type, is obvious from the context, it is implicit. Otherwise an explicit set membership will be included in the predicate.

$$\forall x.\ Prime(x) \Leftrightarrow x \in integer \wedge x > 1 \wedge \neg\exists y.\ y > 1 \wedge y < x \wedge y \text{ divides } x$$

$$\exists x.\ x \in real \wedge x < 10.0$$

Table 1.3: Examples of Predicates

Sets

Definitions for sets: Let A and B be sets, P be a predicate, and U be the universe of values.

$\{\}$		empty set
U		universe
$\{1, 2, 3\}$		explicit finite set
$a \in A$		set membership, a is in A
$\{x \mid P(x)\}$		set of x such that $P(x)$
$A = B$	$\overset{\text{def}}{=} x \in A \Leftrightarrow x \in B$	set equivalence
$A \subseteq B$	$\overset{\text{def}}{=} x \in A \Rightarrow x \in B$	subset
$A \subset B$	$\overset{\text{def}}{=} A \subseteq B \wedge A \neq B$	proper subset
$A \cup B$	$\overset{\text{def}}{=} \{x \mid x \in A \vee x \in B\}$	union
$A \cap B$	$\overset{\text{def}}{=} \{x \mid x \in A \wedge x \in B\}$	intersection
$\overline{A}$	$\overset{\text{def}}{=} \{x \mid x \in U \wedge x \notin A\}$	complement w.r.t. U
$A - B$	$\overset{\text{def}}{=} A \cap \overline{B}$	set difference
2^A	$\overset{\text{def}}{=} \{B \mid B \subseteq A\}$	powerset
$\max(A)$		maximum element (for ordered A)
$\min(A)$		minimum element (for ordered A)
$\text{choice}(A)$		nondeterministic choice
$\text{size}(A)$		size of set A

Table 1.4: Set and Operator Definitions

The definitions for max and min apply only if U is ordered. The function "choice" is undefined if it is applied to an empty set; it is some value from the set otherwise. The universe is typically countably infinite.

Exercises

6. [1,1] Is size(2^A) $= 2^{\text{size}(A)}$?
7. [1,1] Is $\max(A) \geq \text{choice}(A)$?
8. [1,1] Is $\max(A) \leq \max(A \cup B)$?
9. [1,1] Is $(A - B) \cap B = \{\}$?
10. [1,1] Does $A \neq \{\} \Rightarrow \text{choice}(A) \in A$?
11. [1,1] Does $\text{size}(A) \neq 0 \Rightarrow \text{size}(A - \{\text{choice}(A)\}) = \text{size}(A) - 1$?

Ordered Pairs

Definitions for pairs: Let A and B be sets, $a \in A$, $b \in B$.

$\langle a, b\rangle$		ordered pair
$A \times B$	$\stackrel{\text{def}}{=} \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$	cross product
$\text{size}(A \times B)$	$= \text{size}(A) \times \text{size}(B)$	

Table 1.5: Ordered Pairs

Sequences

Definitions for sequences: Let α, β be sequences, A be a set, and B be a set of sequences. Sequences are built up of ordered pairs. That is, the expression $\langle a, \langle b, \lambda\rangle\rangle$ is a sequence of length 2 and written $\langle a, b, \lambda\rangle$ or ab. The notation A^* means sequences of elements taken from A. The inverse, $B^{1/*}$, is the elements which make up the sequences in B. The importance of sequences is that we regard a language as a set of sequences of words. If A is a dictionary, then A^* is everything you can say (meaningful or not).

λ		empty sequence
$\text{length}(\lambda)$	$\stackrel{\text{def}}{=} 0$	length of empty sequence
$\alpha\beta$		catenation of sequences
$\alpha\lambda = \lambda\alpha$	$\stackrel{\text{def}}{=} \alpha$	catenation of λ
$\text{length}(\alpha\beta)$	$\stackrel{\text{def}}{=} \text{length}(\alpha) + \text{length}(\beta)$	length of sequence
A^0	$\stackrel{\text{def}}{=} \{\lambda\}$	
A^n	$\stackrel{\text{def}}{=} A \times A^{n-1}$	fixed-length sequences
A^*	$\stackrel{\text{def}}{=} \bigcup_{i=0}^{\infty} A^i$	all finite sequences
A^+	$\stackrel{\text{def}}{=} \bigcup_{i=1}^{\infty} A^i$	non-empty finite sequences
$B^{1/*}$	$\stackrel{\text{def}}{=} \{a \mid \alpha a \beta \in B \wedge \text{length(a)} = 1\}$	elements in sequences in B

Table 1.6: Notation for Sequences

Exercises

12. Is $(V^*)^{1/*} = V$?
13. What is $\{\}^*$?
14. What is $\{\}^+$?
15. What is size($\{\text{a}\}^7$)?
16. What is size(A) $\times$ size(B)?
17. Define length (formally).

Relations

Definitions for relations: A relation is a set of ordered pairs. Let A be a set, R and S be relations and U be the universe of values.

$\mathcal{D}(R)$	$\overset{\text{def}}{=} \{x \mid \exists y.\ \langle x, y\rangle \in R\}$	domain
$\mathcal{R}(R)$	$\overset{\text{def}}{=} \{y \mid \exists x.\ \langle x, y\rangle \in R\}$	range
$A \lhd R$	$\overset{\text{def}}{=} (A \times U) \cap R$	domain restriction
$R \rhd A$	$\overset{\text{def}}{=} R \cap (U \times A)$	range restriction
$S \circ R$	$\overset{\text{def}}{=} \{\langle x, z\rangle \mid \exists y.\ \langle x, y\rangle \in R \wedge \langle y, z\rangle \in S\}$	relational composition
R^0	$\overset{\text{def}}{=} \{\langle x, x\rangle \mid x \in U\}$	identity relation
R^{-1}	$\overset{\text{def}}{=} \{\langle y, x\rangle \mid \langle x, y\rangle \in R\}$	inverse
R^n	$\overset{\text{def}}{=} R \circ R^{n-1}$	transitive extension
R^*	$\overset{\text{def}}{=} \bigcup_{i=0}^{\infty} R^i$	zero or more steps
R^+	$\overset{\text{def}}{=} \bigcup_{i=1}^{\infty} R^i$	one or more steps
$R(x)$	$\overset{\text{def}}{=} \text{choice}(\mathcal{R}(\{x\} \lhd R))$	function application

Table 1.7: Relations

When a relation is also a function, it may be applied in the normal way to an argument. The superscript '*' has two reasonable interpretations. Since any relation is also a set, it could have been used to signify a sequence. In fact it is used instead for the reflexive transitive completion of the relation; a matter of iterating on relational composition. The collection of these operators allows one to build and examine relations.

Exercises

18. [1,1] Let P be the universe of living people, M be all men, W be all women. $P = M \cup W$. Let $Married \subseteq M \times W$.

    (a) What is $\mathcal{R}(Married)$ and $\mathcal{D}(Married)$?

(b) What do you call x where size($\mathcal{R}(\{x\} \lhd Married)$) > 1?

(c) What do you call $M - \mathcal{D}(Married)$?

(d) Suppose $x \in M$. What is $Married(x)$?

19. [1,1] Continuing the previous exercise, let $Mother \subseteq P \times W$ and $Father \subseteq P \times M$.

    (a) What is $Mother^{-1} \cup Father^{-1}$?

    (b) What is $\overline{\mathcal{D}(Mother \cup Father)}$?

    (c) What is $Mother \cap Father$?

    (d) What is $Mother^2$?

20. [1,1] Suppose P, M and W refer to the living and the dead. Describe Adam and Eve.

Bibliography