the plang book (draft 0)

Erik Dominikus

2011-01-09

Notice of instability Both this document and the actual implementation may change without notice. In such case this document will no longer be correct. This document corresponds to Plang <version>.

Another notice This document contains too many errors.

ISBN X-XXX-XXXXX-X

Before you proceed

If I named this part Preface *instead, would you still read it?*

A programming language cannot be fully appreciated until it is used for doing something real for quite some time. Therefore this book presents real-world examples such that you get accustomed to Plang, and then as time goes by it will be a second nature. Sometimes the best way to learn something is to try it right away.

This book is large because it tries to contain as much information as possible to satisfy everybody that could possibly have something to do with Plang, such as programmers, managers, students, lecturers, computer scientists, software engineers, and other sorts of people. How you read this book depends on who you are. Unless you are everybody, you rarely need to read everything in this book. Therefore you can save time by not reading chapters that are not relevant to your goal.

My advice is to study not more than three full-text pages in one minute, and not more than seven new chapters per day.

Make sure you spend some time reading the table of contents to get a feel about where things are.

Part I is something that you would read when you have a modest amount of free time and are sick of mathematical details or computer codes. This part mostly consists of nonsensical humours, philosophies, and politics.

Part II and V are all wishes, empty talks, confusions, cajoleries, shattered dreams, unfulfilled ambitions. None of anything described there is actually implemented. I repeat: *none*. I was not under the influence of drugs but I don't know what I had in mind when I wrote them.

Part III is the most serious part. A lot of mathematics is going on there, but no joke. Do not read this unless you are prepared.

Part IV attempts to document the reality but it has gone quite outdated, worsened by haphazard updates that incorporated some wishes and empty talks.

Chapter 30 and 31 starting on page 105 and 106 respectively describe the standard data structures that come with Plang. The standard data structures are pair and dictionary. If you think that data structures mean linked list, hash table, and array, then you should read those chapters.

Chapter 32 starting on page 114 describes the Plang object system. One of its concern is implementing Plang objects in C.

Chapter 33 starting on page 118 describes how Plang can be used in functional style.

Parts

Chapters

Sections

List of Tables

List of Figures

List of Algorithms

Part I

Narrations

1
Some history

Remembrance of the olden days.

1.1 Programming languages with simple grammars

Across the years I had the chance to try various programming languages.

PostScript has a simple grammar in the sense that it has a small number of rules[1] in its Backus-Naur form. Lisp also has a simple grammar in the same sense.

The features of Fortran 77 which seems weird today can be understood if we think as someone back in 1977. Back then computers were huge, memory size was measured in kilobytes?, and storage was done with punch cards with (70?) columns. Since the form of the language is based on punch cards, it seems weird to people who do not know punch cards. Fortran 95 provided a free-form language.

I tried Bash.

I tried Sed.

Perl has a regular expression.

I tried PHP.

I tried Python.

I tried Prolog.

I tried Haskell.

(So what's the point of trying so many languages without actually mastering any?)

I tried Ruby. I got quite fascinated for quite some time.

I tried Assembly.

Forth had a chance to be liked. It has simple grammar although its gram-

[1]A rule is also known as a *production* in the theory of languages and automata.

mar is not exactly identical to that of PostScript. Its being untyped is understandable since the interpreter had to run on systems that are very limited according to the standards of today, even smaller than the embedded devices of today, and a type system would add overhead that is unacceptable for those small systems. As a rough estimation, the space overhead of Plang's object system is up to a few machine words for each object, and up to tens of machine words for each type. Nice features won't fit into a 8-bit microcontroller with only hundreds of bytes of memory.

Why didn't I just develop Forth?

1.2 Programming languages with interesting features

In my opinion the most prominent feature of Metafont is its ability to specify linear equations in the same way we write them mathematically, and it actually solves them as required. Polynomial interpolations are naturally expressible in the language. It is very suitable for specifying fonts which are defined by constraints. It allows us to think about *what* should be done instead of *how* it should be done. As a human myself I personally find that it is easier to think about what instead of how. Bloom's taxonomy of learning [cite?] describes this more detailedly.

1.3 Influence of other programming languages

The form of Plang might be identical to that of Joy. The grammar of Plang used to imitate PostScript. The parser was then rewritten to parse a grammar very similar to that of Joy.

All of them are concatenative and stack-oriented.

PostScript and Plang use late binding whereas Joy uses early binding. PostScript can also do early binding. PostScript can mix early binding and late binding.

1.4 Gratuitous creation

I simply just want to make my own programming language, and the curiosity had grown into a desire that got more irresistible as more time passed. After all, I'm a computer scientist, a programmer, and a mathematician, among other things. I don't know whether I really want to be a computer scientist, but when people think that I am a computer scientist, I want to make sure that I am indeed worthy enough to be called a computer scientist. I want to make sure that I am not called a computer scientist for nothing.

So I created it gratuitously. For me, it's just fun.

Early 2010, I tried writing an interpreter in Java, but somehow I ended up proliferating classes. I missed pointers, crashes, and memory leakages. A few weeks later I tried C++. However, in the end it was C that somehow felt more like home, whereas Java felt like a warehouse and C++ like a war trench; such comparison is just a nonsense, just for fun, of course. Unfortunately Assembly is not portable, but that is not to be worried because most personal computers I have encountered run on $x86$ with x greater than 6 (or at least something compatible with that). The x86 architecture itself is not a good thing: too few registers, some weird instructions, somewhat crippled by backward compatibility, among other things, but the uniformity is somehow good because it means less work for porting.

Then I created Plang. It used to be like a fully interpreted Forth with no option for compilation. No typing.

1.5 Departure from PostScript toward Joy

```
/whatsit { 1 2 add [ 3 4 5 ] { mul 1 add } each } def
"whatsit" [1 2 add [3 4 5] [mul 1 add] each] def
```

I decided to depart from PostScript (the first line) toward Joy (the second line). I realised that non-executable lists and executable lists do not have to be treated differently. Indeed there does not have to be anything like 'non-executable' lists. The PostScript behaviour of the left-bracket is mark and the right-bracket is list-until-mark. Brackets are now specials like braces. They used to be aliases for mark and list-until-mark. (Do we still need those PostScript operations?)

I also removed the difference PostScript made between names `/like-this` and strings `"like-this"`.

1.6 A brief history of computers and programming

Computers originated from mechanical engineering, electrical engineering, and then electronic engineering, with some die-hard hobbyists and undergraduate drop-outs interspersed among them. Then some mathematicians jumped into the bandwagon.

Programming languages became relevant since the invention of general-purpose processors that allow computers to be reprogrammed without physical rewiring. To program a computer is to tell it to do something.

At first there were machine codes. Then assembly language. Then macro assembler. Then procedural language. Block-structured language. But Lisp came in the late 1950s? When did Fortran came? In the late 1940s?

Processors are inherently procedural and sequential.

2 Considerable desiderations

Things that could be nice to have.

Several features come with several alternatives and consequences. In general, the addition of a feature makes the system more complex. What would be if we had it, what would be if we did not have it, must be thought.

2.1 Static binding, dynamic binding, or a combination of them

Suppose that you want to write a generic `min` function which takes two things and returns the smallest of them but you also want to be able to provide your own definition for the meaning of *smallest*. With dynamic binding you can write this:

```
# A B min M
"min"
[
        2 copy less-than
        [pop] [exch pop] ie
]
def

"less-than" [lt] def
3 5 min

"less-than" [length exch length exch lt] def
"hello" "goodbye" min
```

which demonstrates the redefinition of `less-than` at runtime. (I call this style *intrusive programming* or *invasive programming* because you invade subprograms by redefining their very components.) With static binding you have to pass the comparator as an argument like this:

```
# A B O min-with-order M
"min-with-order"
[
        3 copy exec
        exch pop
        [pop] [exch pop] ie
]
```

```
def

3 5 [lt] min-with-order
"hello" "goodbye" [length exch length exch lt] min-with-order
```

Dynamic binding allows redefining words for debugging or tracing without any modification to the source code of the inferior program and this can be done without restarting the program. Static binding requires recompilation and linking with the version of library with debugging information.

PostScript by default uses *late binding* but also provides *early binding* using the `bind` operator.

```
/woot { add } def
/willadd { 1 2 woot } bind def
/woot { sub } def
willadd =
1 2 woot = % will subtract
```

Static binding is simpler for compilers.

2.2 Unified numeric type

Unified numeric type means you can expect numbers to behave as in mathematics, as limited by available resources. In the unification all numbers are complex numbers. This is because the set of all complex numbers is *algebraically closed*.

Without unified numeric type you will encounter integer overflows when working with very large integers and you will also encounter domain errors when attempting to take the square root of a negative number, among other things.

When both unified numeric type and polymorphism-by-overloading are not implemented, there will be both `add` and `addf`. With unified numeric type there is only one `add` without having to implement overloading. This is convenient for script programmers. They might even expect this.

With unified numeric type the following fragment would produce `3.5`. Without unified numeric type it would produce an error since `add` expects two integers.

```
1 2.5 add
```

2.3 Proper type system

Allows static type checking and polymorphism-by-overloading.

2.4 Referential transparency by copy-on-write

The running program will see that all objects are passed by-value.

2.5 Macro-transformations

`function` inserts a prologue and an epilogue for named parametres.

```
PARAMETRES BODY function

"fun" [x y z] [x y mul z add] function

-->

[dictionary-new-empty begin
        # PARAMETRES
        "z" exch def
        "y" exch def
        "x" exch def
        # BODY
        x y mul z add
end]

1 2 3 fun --> 5
```

2.6 Flowchart programming, reactive programming, visual programming

Flowchart programming uses flowchart. This might be unpopular. Reactive programming might be natural for complex systems whose requirement specification is a list of expected behaviours. Visual programming is like how we make hardware: connecting several black boxes.

What the heck was I thinking? Do not take these too seriously.

2.7 Integer literal with arbitrary base

Binary, octal, hexadecimal, any base from 2 to 64.

2.8 String escape sequences

Enables putting quotes inside strings.

2.9 Dedicated boolean type

Addition of two new words: true and false. Gives somewhat stronger type checking.

2.10 Do we really need a virtual machine?

I could make virtual machine.

Hypothesis: Stack might cause problem. It could be bottleneck. It could hamper parallelism.

We can just interpret the AST or transform it to machine code.

2.11 Distributed memory manager

Can perform allocation in parallel.

The memory is divided into n regions of approximately equal size. Each region belongs to one and only one process. A process allocates memory from its own region. If its region is used up, it attempts to enlarge its region by shrinking others' region in a safe manner (a global lock must be obtained first of course).

2.12 Moveable memory

Memory region can be moved while program is running. Double indirection. Pointer: lock, unlock, move.

2.13 Apparently reversible computing

Restore a state that was serialised earlier.

2.14 Including other files

As a script grows larger, one may want to break it into several files.

```
FILE    include
```

where `FILE` is a path to an existing file. It is an error if the file does not exist. The path can be relative or absolute. Relative paths are relative to the directory that contains the including file.

The included file is executed on the same engine that executes the including file, but with different parser. C programmers can think it as including other files. Unlike C, there is no include search path. Actually the included files are executed. The effect is the same as if the included file was inserted at the point of inclusion, provided that inclusion is not nested too deeply.

Tips If your script includes many other files and the core library is loaded, then you can write something like the following instead of writing many lines of run-files.

```
[ FILE1 FILE2... FILEN ] { include } each
```

We considered the names execfile, include, runfile, run-file, execute-file.

Nested inclusion is allowed but the nesting depth is limited to 16. A warning is issued when the nesting depth reaches 8. Take care not to create a loop.

```
Warning: files are nesting quite deeply (N levels).
```

2.15 Compilation switches

-DTRACE_REFCNT enables tracing of reference counting. Every call to reference counting functions (`plang_obj_use` and `plang_obj_rel`) will be printed to standard error, along with the reference count of the operated object. This can produce lots of text.

-DTRACE_OBJECT=N enables tracing the creation and deletion of every object and limits the number of objects to N. Every call to `plang_obj_new` and `plang_obj_del` will be printed to standard error. At program termination, every alive objects is printed. These objects indicate leakage of reference count, typically due to forgetting to call `plang_obj_rel`, calling `plang_obj_use` too many times, or messing with reference counting. Defining this slows down object deletion, and thus slows down the interpreter. *This switch will change. Do not depend on it.*

-DLIMIT_NUMOBJ=N limits the number of objects to N. This switch can be used for simulating out-of-memory situations, but this does not limit the heap size. You might want to know about ulimit(3), getrlimit(2), setrlimit(2), or the Bash built-in named `ulimit`. If you are using Bash, you might want to run:

```
help ulimit
```

-DNDEBUG disables all assertions.

2.16 The parser for the modified BNF

R is the list of rules. Each rule is of the form $A \to B$ where A is exactly one nonterminal and B is an arbitrary non-empty sequence of terminals and nonterminals. The parser state is (S, N, T) which are stack, current nonterminal, and current token, respectively. Initially N is the starting nonterminal.

```
Function Parse(Nonterminal) returns a tree
foreach $(A \to B)$ in $R$ such that $A = N$ do
    foreach $y \in B$ do
        if $y$ is terminal and $y$ does not match suffix of input then
            unread $y$
            return Fail
        if $y$ is nonterminal and Parse($y$) fails then
            return Fail
        shift
    reduce
    return Succeed
Function Shift($T$)
Append $T$ to the end of parse tree.
Function Reduce($N$)
Replace token list with $N$.
```

Algorithm 1: A parser?

2.17 Recursive descent parsing

```
void            plang_parser2_init (Plang_Parser2* parser,
                        Plang_Stream* stream,
                        int recursion_depth_limit);
Plang_Errno     plang_parser2_parse_NT (Plang_Parser2* parser,
                        Plang_ST_Node** output);

NT
 -> program | node | real | integer | string | list | ...
PLANG_ERROR_PARSING
 -> *
PLANG_ERROR_STREAM_EXHAUSTED
 -> PLANG_ERROR_UNEXPECTED_END_OF_FILE
```

You call parse_NT where NT is a nonterminal until the procedure returns PLANG_ERROR_STREAM_EXHAUSTED.

2.18 Continuations, Landin's J operator, and McCartney's ambiguous operator

These sound mighty impressive but are not feasible to implement in Plang. I don't even know Landin's! Only heard of it. I guess I violated Rule 1: don't talk about what you don't do.

```
???     amb
```

2.19 Propositional logic formula bruteforcer

```
/formula { a b and c implies } def
formula [ /a /b /c ] { DOWHAT } each-satisfaction

FORMULA VARIABLES ACTION        each-satisfaction
```

Executes ACTION for each interpretation (giving truth values to variables) that satisfies FORMULA. Running time is in $O(2^n)$ where n is the number of variables. For one common computer of today, n of a hundred is already impractical. This problem is parallelisable.

2.20 Extension mechanism

The interpreter can be extended by loading native modules using load. The module format is whatever supported by binutils? Dynamic linker? Dynamic/runtime linking? The error code is an integer.

```
MODULE-PATH  load  -->  ERROR-CODE
"/usr/lib/up/module/plang-level-1.so" load
```

2.21 Input character decoder

UTF-8. Internally UCS-32.

The number of characters processed (not necessarily equal to the number of bytes processed because UTF-8 is a multibyte coding scheme), the number of lexemes produced.

The character decoder statistics: the number of bytes processed, the number of UTF-8 coding errors, the number of characters produced. What to do on a coding error? Abort with an error, or ignore and auto-resync with a warning?

2.22 Dictionary stack recursive look-up

PostScript has dictionary stack. The dictionary stack allows local variables. An issue is aliasing/shadowing. It seems that this will significantly complicates compilation.

Joy doesn't seem to have this. Life is simpler without this. So we abandon this, now that we have not yet implemented it?

The base dictionary is the bottommost dictionary in the dictionary stack. It contains predefined builtins. A user dictionary is pushed. It is a blank dictionary above the base dictionary.

The dictionary stack is a stack whose elements are dictionaries. Every time Plang is going to execute a word, it does a 'recursive look-up' on the dictionary stack. Recursively looking up a key k is looking for k, starting from the topmost dictionary until a definition is found.

2.23 C translator

The C translator naïvely transforms Plang source code into C code. Its primary purpose is for bug reproduction purposes: to ascertain whether a bug is indeed in the optimiser. We assume that since the C translator is simpler than the optimiser it is harder to make a bug in the translator. Therefore if there is a mismatch between the result of the optimised program and that of the translated program, then there is likely a bug in the optimiser.

2.24 Compiler

The compiler transforms virtual machine code to native machine code.

2.25 Machine-code optimiser

The machine-code optimiser performs machine-specific optimisation. Currently we plan to write one for x86.

Is this necessary?

Instruction elimination

```
        mov     eax, 1
        mov     eax, 2
        mov     ecx, 3
```

becomes

```
        mov     eax, 2
        mov     ecx, 3
```

2.26 Grammar of Plang-Lisp

Plang-Lisp is a dialect of Lisp and can interoperate with Plang. A caveat of the grammar is that comment cannot interrupt expression. The block comment delimiter is due to Scheme but here block comments cannot nest.

```
program ::= <EOF>
program ::= blank program
program ::= linecomment program
program ::= blockcomment program
program ::= expression program

expression ::= word
expression ::= string
expression ::= integer
expression ::= real
expression ::= list
expression ::= "'" expression
```

```
list ::= "(" listtail
listtail ::= ")"
listtail ::= expression listtail

linecomment ::= ";" linecommenttail

blockcomment ::= "#|" blockcommentbody
blockcommentbody ::= "|#"
blockcommentbody ::= anychar blockcommentbody

grey ::= '('
grey ::= ')'
grey ::= '#'
grey ::= '|'
grey ::= ';'
grey ::= "'"
grey ::= '"'
```

2.27 Example codes

```
(define 'f (function '(x y) '(+ x y)))
(define 'nat1 (function '(to) '(range 1 to)))
(define 'fac (function '(n) '(prod (nat1 n))))
(define 'fib (function '(n) '(
  ie (<= n 0)
    0
    (+ (fib (- n 1)) (fib (- n 2)))
)))
(define 'what (procedure '(x) '(display x)))
(what (f 3))
```

2.28 Indentation-mitigated Lisp

Only spaces, no tabs. No trailing spaces.

Define the *indentation level* of a line as the number of leading spaces in that line. If the indentation level increases then we insert a left parenthesis and push the indentation level to the stack. If the indentation level decreases then we insert a right parenthesis and pop an indentation level from the stack. A problem is multi-line string literal.

```
define 'f
  function '(x y) '
    + x y

define 'fac
  function '(n) '
    prod (range 1 n)

+
  length "what? me?
no
```

```
    problem"
  1

; should translate to: (+ length "what? me?\nno\n    problem" 1)
```

2.29 <was: do not read this chapter>

Random-access-memory sequential-execution instruction machine.

A program can be seen as a rewriting system.

Execution can be modelled as evaluation of functions.

In logic programming, execution can be modelled as finding logical values that satisfy the given constraints.

Memory M is an addressed collection of $|M|$ bytes where $|M|$ is the number of bytes in memory and a byte is an integer between 0 and 255 both inclusive. $M[0]$ is the first (the zeroth) byte. $M[|M|-1]$ is the last byte. Accessing (reading or writing) memory outside the address range is an error.

The operation set is the set of operations the machine can perform in one unit time. An *instruction* I is always in three-operand form like this, where f is the *operation* $f : B \times B \to B$ and B is the set of bytes.

$$I :\ M[a] \leftarrow f(M[b], M[c]) \tag{2.1}$$

or if we are to be parsimonious with notations

$$I :\ a \leftarrow f\ b, c \tag{2.2}$$

model the processors

2.30 Propositional logic

An expression is a propositional logic formula ϕ such as

$$\phi \equiv (a \wedge b) \vee c \tag{2.3}$$

and to evaluate the expression is to give truth values to variables such that the formula is satisfied.

2.31 Syntax-tree C API

```
typedef struct _Plang_ST_Node Plang_ST_Node;
struct _Plang_ST_Node
{
        Plang_Object            _;
        Plang_String*           type;
        Plang_Object*           value;
        Plang_ST_Node*          older;
        Plang_ST_Node*          younger;
        Plang_ST_Node*          heir;
        Plang_ST_Node*          parent;
};
```

The structure is designed for sequential depth-first traversal. In CST `value` is always either a string or null. In AST `value` is the *semantic value*. The type of the root node is always *program* and only the root node has that type.

Part II

Guides

3 Getting involved

Information for jumping on the bandwagon, and also jumping off it.

This is intended to be read by people who are interested in getting involved in the project.

Codebase language is C and x86 assembly.

Plang is hosted at GitHub (`http://github.com/edom/plang`) that provides Git repository, wiki, and issue tracker.

The repository contains source, examples, and documentation. Clone it like any one of these you feel most convenient:

```
git clone ssh://git@ssh.github.com:443/edom/plang.git
git clone ssh://git@github.com/edom/plang.git
git clone https://USER@github.com/edom/plang.git
```

where `USER` is your GitHub user name. For example, my GitHub user name is `edom`. In case you are not familiar with Git, you can read a tutorial at [?].

The wiki is used for announcements. It also serves as some kind of home page for the project. It provides links to further information. It is not for discussion and also not for documentation.

The issue tracker is used for reporting bugs and requesting features. It is also not the place for discussion.

There is also the Plang Developer Mailing List at Google Groups. It is the place for discussion.

Are you sure that you want to join? If yes, good. Start by cloning the repository. After copying thousands of lines of code from the Internet to the thing before you, the first thing you should do is to religiously believe that I have coded everything correctly in the same manner as you would have done, although soon you will find out that this is never true, but just pretend until you find out. Then, decide your goal, what you are going to do, what feature you would like to have, and how you are going to do it. Without goal you will waste your time. However, if you run into bugs, carefully look at your codes first before blaming mine.

3.1 Repository layout

On the root directory of the repository, you shall see the directories named `doc`, `examples`, `include`, and `src`. `doc` contains the documentation, all you need to know about Plang. `include` contains a C header file `plang.h` that is the public interface of Plang. `src` contains two directories: `libplang` and `plangi`, where `libplang` contains the source code of the library and `plangi` contains the source code of the interpreter.

How do you build Plang from source code?

I just list the things in my machine. I do not know whether building Plang will be successful on other machines with other versions of these tools and libraries.

- GCC 4.3.2
- Bash 3.2.39
- libc6 2.7-18lenny1
- libdl 2.7 (part of libc6 in Debian)
- libtool

Actually for convenience we can statically link with libtool.

You need LaTeX and some packages to compile the documentation. If you are using Debian, you can instal TeX Live (is that how you spell it?) and some packages (what are them?).

3.2 Porting

My machine happens to be running an installation of Debian 5 (Lenny). If you fail to compile Plang on your machine, I cannot do much. I do not bother to learn any sort of build system. The shell script does just fine. If the shell script fails, check whether your operating system is decent, and check whether you have satisfied all the requirements for building Plang.

You can port Plang to other platforms. A platform is an architecture plus an operating system. However, if you do so, you are responsible to maintain that port or to find someone capable of doing that. Do not spread false hope.

Plang requires that its underlying system support dynamic linking. This is done using `libltdl` that is a part of GNU Libtool.

- DLL does not support undefined symbols.

3.3 Getting disinvolved

4
Graph

Mathematically a graph G is a tuple (V, E) where V is the set of vertices and E is the set of edges.

Plang has three ways of representing a graph: *adjacency list*, *adjacency matrix*, *edge list*. Some graph operations are more efficient with some representations as some operations are more efficient with some data structures.

An *adjacency list* is a list of neighbours for each vertex. Formally an adjacency list is $[A_0, A_1, \ldots]$ where A_k is the *list of neighbours* of the vertex k. Actually in a directed graph, A_k is the list of all vertices that are *directly reachable from* k. Memory requirement is in $\Theta(v + av)$ where a is the average number of neighbours of a vertex and v is the number of vertices.

An *adjacency matrix* is a matrix A where A_{ij} is the weight of the edge (i, j). If $(i, j) \notin E$, then $A_{ij} = \infty$ (a special value like the largest signed 32-bit integer). It is ideal if the graph is not sparse, not too large, and the upper bound of the number of vertices is known beforehand. If the graph is sparse (has only few edges), then the matrix will also be sparse (will contain mostly zeroes) and will waste much space. If the upper bound of the number of vertices is not known, adding vertex later may resize the entire matrix, involving memory reallocation. Memory requirement is in $\Theta(v^2)$ where v is the number of vertices.

An *edge list* is $[(a, b), (c, d), \ldots]$. It is ideal for sparse graphs. It is ideal if you often add and/or remove edges from the graph. It can also represent graphs with infinitely many vertices but finitely many edges. However, it is not ideal for quickly counting the number of vertices in the graph. Memory requirement is in $\Theta(e)$ where e is the number of edges.

Among those three, adjacency matrix uses the most amount of memory.

The memory requirements of the various graph representations are summarised in the following table.

Vertices are numbered with 0, 1, 2, and so on.

Representation	Memory Requirement
Adjacency List	$\Theta(v + av)$ (from $\Theta(v)$ to $\Theta(v^2)$)
Adjacency Matrix	$\Theta(v^2)$
Edge List	$\Theta(e)$

Table 4.1: Memory requirement of graph representations

4.1 Creation

Start with an empty graph. Add edges.

```
NUMVERTICES graph-new-matrix GRAPH
graph-new GRAPH
graph-empty EMPTYGRAPH
GRAPH graph-is-empty YESNO
G V W graph-connect NEWGRAPH    # adds an edge
```

`graph-connect` connects the vertex v to w in the given graph G. That is, it creates the edge (v, w) in the graph. The graph is assumed to be an edge list.

4.2 Properties

```
GRAPH graph-num-vertices DEG
GRAPH graph-num-edges ORD
GRAPH graph-edges LISTOFPAIRS
GRAPH V vertex-indegree INDEG
GRAPH V vertex-outdegree OUTDEG
GRAPH graph-connected YESNO
GRAPH FROM TO graph-vertices-connected YESNO
```

4.3 Shortest Path

`shortest-path` computes the shortest path from v to w. Technically the path might be *a* shortest path instead of *the* shortest path. It uses an algorithm due to Dijkstra.

```
GRAPH FROM TO   shortest-path                  PATH
GRAPH FROM TO   shortest-path-weight           WEIGHT
GRAPH FROM      single-source-shortest-paths   PATHS
GRAPH           all-pairs-shortest-paths       PATHS
```

Floyd-Warshall algorithm.

4.4 Euler Path

```
GRAPH FROM      euler-path
```

5 Graphs

5.1 Representation

Usually we choose the representation which facilitates solving the problem.

5.1.1 Naïve

The naïve representation of a graph is a straightforward translation of the mathematical definition.

5.1.2 Indexed Relation

An improvement of the naïve representation. This allows fast lookup.

5.1.3 Adjacency List

5.1.4 Edge List

5.1.5 Adjacency Matrix

5.1.6 Incidence Matrix

5.2 Neighbourhood

The neighbourhood of a vertex is the set of all vertices which are adjacent from it.

The expansion of a neighbourhood is the union of it and the set of all vertices which are adjacent from that neighbourhood.

5.3 Connectedness and Component

A component of a graph is a maximal connected subgraph of that graph.

5.3.1 Neighbour Contagion

Imagine the spreading of a rumour. Rumour spreads in the neighbourhood, from houses to houses. A house is a vertex. The cloud of rumour expands from a house to other adjacent houses, until the rumour circulates in the entire neighbourhood. A house isolated from the neighbourhood will not hear about the rumour. If you don't like rumour, you can think of contagious plague instead.

Select a vertex. The neighbourhood initially contains one vertex.

Expand its neighbourhood until it can't expand more. Select an incident edge.

If its neighbourhood is the entire graph, then the graph is connected. Otherwise, the graph is not connected and has several components. A neighbourhood is a component. Repeating this algorithm until there is no vertex left gives the graph components.

This can be done using breadth-first enumeration.

Using naïve representation, the space complexity of this algorithm is $\Theta(|V|)$ and the time complexity is $\Theta(|E| \log |V|)$? The complexity of the representation isn't included.

5.4 Connectedness

An undirected graph is connected if and only if every pair of different vertices in it are connected.

Determining the connectedness of a graph involves computing the transitive closure of (what?).

5.4.1 Connectedness of Vertex Pair

A pair of vertices is connected if and only if there is a path which starts at one of those and ends at the other.

5.5 Cycle Detection

If in a graph $|E| \geq |V|$, then it is cyclic. The converse doesn't always hold.

Select a vertex.

Expand the neighbourhood until impossible. If at any one time, the intersection between the current fringe and the current neighbourhood isn't empty, then the graph is cyclic.

Repeat all steps above until there is no vertex left.

5.6 Shortest Path

Every subpath of a shortest path is also a shortest path. If you know the shortest path from Jakarta to Bogor, and you also know you must pass Depok in your way, then that path contains the shortest path from Jakarta to Depok and the shortest path from Depok to Bogor. Formally, if P is the shortest path from v_1 to v_4, p is a subpath of P, and p connects v_2 and v_3, then p is the shortest path from v_2 to v_3.

The converse doesn't always hold. For example, if you know the shortest path from Surabaya to Ujung Kulon and you know the shortest path from Ujung Kulon to Semarang, concatenating them gives you a more farther path than the shortest path from Surabaya to Semarang.

The algorithms here work on directed graphs.

5.6.1 Single-source Multiple-destination Shortest Path

If all edge weights are always nonnegative, then forming larger paths always increase the distance between the ends of the path. Dijkstra's algorithm assumes nonnegative edge weights and exploits that monotonicity. That monotonicity also implies that a loop can never be a shortest path. Violating this assumption leads into an infinite loop. However, negative-weighted edges can be used to mean isolation.

If there is a path from a to b and it turns out that there is another shorther path from a to b, replace the path.

At first, $d(a, x) = \infty$. $d(x, x) = 0$.

For each ? in fringe: if $D(a, b) + d(b, c) < D(a, c)$, update $d(a, c)$ and reinsert c into the queue.

There is initially one neighbourhood. This contains the source vertex.

Expand the neighbourhood.

5.6.2 Multiple-source Multiple-destination Shortest Path

Floyd-Warshall algorithm also assumes nonnegative edge weights. The difference from Dijkstra's algorithm is Floyd-Warshall expands $|V|$ neighbourhoods each iteration whereas Dijkstra's algorithm expands 1 neighbourhood each iteration.

5.7 Spanning Tree

5.7.1 Breadth-first Enumeration

5.7.2 Depth-first Enumeration

5.8 Minimum Spanning Tree

5.8.1 Prim's Algorithm

5.8.2 Kruskal's Algorithm

5.9 Maximum Flow

Belman-Ford algorithm.

5.9.1 Maximum Flow between Two Vertices

5.10 Euler Tour

An Euler tour of a graph is a tour which visits each edge exactly once and ends at the starting vertex. A tour is another word for traversal.

5.11 Shortest Euler Tour

5.12 Hamilton Tour

5.13 Knight's Tour

5.14 Travelling Salesman Problem

Shortest Hamilton Tour.

5.15 Computing In-Degree

5.15.1 Edge List

```
For each edge:
    Let $(u, v)$ be the edge.
    Increment $D(v)$.
    The time complexity is $\Theta(|E|)$.
```

5.15.2 Reverse Adjacency List

Simply figure out the size of the set of neighbours of each vertex.

Time complexity is $\Theta(|V|)$ assuming that figuring out the size of the set of neighbours of a vertex is $\Theta(1)$.

5.15.3 Adjacency Matrix Representation

Simply sum a column and you get the in-degree of the corresponding vertex.

Time complexity is $\Theta(|V|^2)$.

5.16 Topological Sorting

5.16.1 Naïve

Topological sorting attempts to find a possible ordering of the vertices of a directed acyclic graph whose edges represent precedence. $(v, w) \in E$ means v must be done before w. The graph represents a partial order.

Line up the vertices. The order doesn't matter here.

For each pair of vertices, swap their position if they violates order. A pair of vertices v and w violates order if and only if $(v, w) \in E$ but v doesn't come before w.

The worst-case time complexity is $\Theta(|V|^2)$.

Repeat until there is no violation.

5.16.2 Variation of Naïve?

For each pair of vertices v and w, if $(v, w) \in E$ but v doesn't come before w, move v to the last place and move w to the place v occupied earlier.

5.16.3 Greedy: Repeatedly Choose Minimum

S will be an ordered sequence. It is initially empty.

Select a minimum, say m, from the set V. A minimum is a vertex with zero in-degree.

Remove m from V.

Remove all edges originating from m from E.

Append m to S.

Repeat from selecting a minimum until no vertex remains.

5.16.4 In-Degrees

Edge list representation is assumed.

S will be the answer. It is initially empty.

Let $D(v)$ be the in-degree of vertex v.

Choose any v such that $D(v) = 0$. ($\Theta(|V|)$ assuming array is used)

Append v to S.

Decrement $D(v)$ by 1.

For every vertex w adjacent from v, decrement $D(w)$ by 1. ($\Theta(f)$; f is the neighbourhood factor; a rough estimate for f is $|E|/|V|$.)

Repeat from choosing until no vertex is left. ($|V|$ times)

$D(v) = -1$ means v has been visited.

Time complexity $\Theta(|V|f)$. Roughly $\Theta(|E|)$ or $\Theta(|V| + |E|)$?.

5.16.5 Graph Transformation?

Imagine t is on top, l is bottom left, r is bottom right. For each three-friend (t, l), (t, r), (l, r), delete (t, r).

5.16.6 What?

Choose any vertex with zero in-degree. For each vertex with zero in-degree, recurse. If there's no such vertex at all, the graph is cyclic.

Label the current vertex v as $L(v) = 1$.

5.17 Graph Colouring

5.18 Constraint Graph

$(v, w) \in E$ means v and w must differ.

6 Graph visualisation

Chapter 4 describes the representation of graphs.

A *visualisation* here is a drawing in two-dimensional Euclidean space. The system should be deterministic. The same input should always produce the same output. The declarativeness of Metafont is a respectable example. The actual drawing is not done by Plang, but by GTK, GDK, PostScript, or other systems.

There are many ways to draw a graph. Every nonempty graph has infinitely many visualisations but only some will satisfy typical users. Whether a visualisation will satisfy users depends on what they want. Users who want to draw fancy graphs will need full control of positioning and rendering. Users with modest expectations might just want to see some circles and lines.

For every shape users might want, there should be a reasonable default.

GraphViz? ACM SIGGRAPH? Graph drawing algorithms? Spring model?█

6.1 Restrictions and limitations

Every graph we will be talking about in this chapter is a simple directed unweighted graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. *Simple* means that the graph cannot contain any loop. A loop is an edge with the same source and destination.

6.2 Representation of graphs

The representation captures the *connections* and that's all. The representation does not need to know anything about how it will be drawn. Keep in mind that the representation has to be simple and minimal since it might come from other programs. Vertices are numbered from 0.

```
[ NUMBER-OF-VERTICES  LIST-OF-EDGES ]
```

For example, the graph $G = (V, E)$ with

$$V = \{0, 1, 2, 3, 4\}$$
$$E = \{(0, 1), (1, 2), (2, 3)\}$$

is represented as

```
[5  [0 1  1 2  2 3]]
```

6.3 Graph drawing algorithm

The graph drawing algorithm takes four things:

- a graph representation (section 6.2, p. 29)
- a vertex positioner (chapter 7, p. 31)
- an edge renderer (chapter 8, p. 33)
- a vertex renderer (chapter 9, p. 34)

The algorithm is simply as follows:

1. Call the vertex positioner for each vertex.
2. Call the edge renderer for each edge.
3. Call the vertex renderer for each vertex.

As you can see, all edges are drawn first before any vertex is drawn.

Each step is customisable. For example, customising the first step allows you to draw arbitrary shapes, customising the second step allows you to draw fancy edges with arrow-heads, and customising the third step allows you to draw boxes as vertices, among other things.

6.4 Possible improvements

In the future we should add the capability for directed graphs, loops, vertex labelling, edge labelling, incremental graph drawing, graph diffing (comparison by highlighting differences).

7

Vertex positioners

The input of a vertex positioner is an integer v which is the index of the vertex. The output is a position (x, y). Typically a vertex positioner is *parametrised* and thus has to be *instantiated* first before passed to the graph drawing algorithm in §6.3 (p. 30).

7.1 Rectangular

symbol	description
s	rectangle span in metres
s_x	rectangle width in metres
s_y	rectangle height in metres
n	rectangle span in grids
n_x	rectangle column count
n_y	rectangle row count
g	spacing (grid span in metres)
g_x	horizontal spacing (grid width in metres)
g_y	vertical spacing (grid height in metres)
k	vertex number
j	column number
j_k	column number of vertex k
i	row number
i_k	row number of vertex k
x_k	horizontal position of vertex k
y_k	vertical position of vertex k

Table 7.1: Symbols used in rectangular vertex positioner description

(x_0, y_0) is the bottom-left corner of the rectangle. Therefore vertex 0 is at the bottom-left corner of the rectangle. In Figure 7.1, $k \in V$ where $V = \{0, 1, 2, \ldots, v - 1\}$ where v is the number of vertices.

$$n_x g_x = s_x \tag{7.1}$$
$$n_y g_y = s_y \tag{7.2}$$
$$x_k = x_0 + j_k g_x \tag{7.3}$$
$$y_k = y_0 + i_k g_y \tag{7.4}$$
$$i_k = k \div n_x \tag{7.5}$$
$$j_k = k \bmod n_x \tag{7.6}$$
$$g_x > 0 \tag{7.7}$$
$$g_y > 0 \tag{7.8}$$
$$n_x > 0 \tag{7.9}$$
$$n_y > 0 \tag{7.10}$$
$$s_x > 0 \tag{7.11}$$
$$s_y > 0 \tag{7.12}$$

Figure 7.1: Constraints of rectangular vertex positioner

c	circle centre
r	default radius
θ	angle

Table 7.2: Symbols used in circular vertex positioner description

7.2 Circular

$$x_k = c_x + r_k \cos\theta_k \tag{7.13}$$
$$y_k = c_y + r_k \sin\theta_k \tag{7.14}$$
$$\theta_k = \theta_0 + \frac{2\pi k}{v} \tag{7.15}$$

7.3 Spoke

Exactly one vertex is at the centre. This vertex is the *axle*. The other vertices are around this axle. Spoke without axle is circular.

7.4 Hierarchial

8
Edge renderers

The input of an edge renderer is (x_1, y_1) and (x_2, y_2). No output is expected. The edge renderer is called for its side effects.

8.1 Default edge renderer

The default edge renderer draw a black line segment, with 4 pt thickness from (x_1, y_1) to (x_2, y_2). The thickness and colour are customisable.

9 Vertex renderers

The input is (x, y). No output is expected. The vertex renderer is called for its side effects.

9.1 Default vertex renderer

Let C be a circle whose centre is (x, y) and radius is 8 pt. The default vertex renderer fills C with white, and then strokes the circumference of C with black pen, 2 pt thick. The thickness, stroke colour, and fill colour are customisable.

9.2 Complete examples

9.2.1 Plain

```
<
        /vertices       5
        /edges          [ 1 2  2 3  3 5 ]
>
graph-draw
```

9.2.2 Arbitrary positioning

9.2.3 Colours

Part III

Specification

Part IV
Implementation

Part V

Do not read this part

10
Optimisation

"Premature optimisation is the root of all evil."
Donald Ervin Knuth (where?)

I strongly agree with Knuth. *Premature* optimisation indeed can cause a significant waste of time because it invites programmers to think of whatever clever tricks they have up their sleeves. This is worsened by the opinion that programmers take pride in being clever. Therefore before optimising anything you should ensure that:

1. Optimisation is truly necessary. You have a comparison that this one is slow, for example comparing with a C program that does the *same* thing.
2. You are not optimising by hand. That is, you are not refraining from writing *what* you want instead of telling the computer *how* to do it just because the beautiful code happens to be slow and the ugly code happens to be fast. Let the computer transform your beautiful code to fast code. That is what compilers are for anyway. That also means that you should write a really good compiler instead of optimising by hand.
3. You know what needs to be optimised. You know the bottleneck. You know the part of the program where the processor spends most of its time. You know the *hot spots*. Use the profiler to find them.
4. You have some sort of plan.

10.1 Introduction

Some idioms are common. For example programmers writing the following Plang code

```
1 10 range [puts] each
```

would write a for-loop instead when they are programming in C.

10.2 Things that make Plang slow

Maybe memory allocation? Or calls to glues?

In the worst case adding two integers requires 5 calls to the memory manager. This worst case is typical as most integer objects are short-lived.

1. Allocate the first integer.
2. Allocate the second integer.
3. Call the glue.
4. Allocate an integer for the result.
5. Free the first integer.
6. Free the second integer.

Maybe dictionary access? Each word requires finding one key in dictionary. Such traversal typically is an act of following a few pointers. Maybe this is not cache-friendly.

10.3 Collecting statistics

Sometimes we want to know what optimisations are performed how many times.

11

Examples of optimising AST transformations

11.1 Inlining

The content of the definition is copied. This assumes that the inlined word is never redefined.

```
"muladd" [mul add] def

3 4 5 muladd

(program
        (integer 3)
        (integer 4)
        (integer 5)
        (word "muladd")
)

(program
        (integer 3)
        (integer 4)
        (integer 5)
        (word "mul")
        (word "add")
)
```

11.2 Gradual loop unrolling

Will give up after n tries?

```
1 2 3 4 5 6 [mul add] 2 times

(program
        (integer 1)
        (integer 2)
        (integer 3)
        (integer 4)
        (integer 5)
        (integer 6)
        (list
                (word "mul")
                (word "add")
        )
        (integer 2)
        (word "times")
)

(program
        (integer 1)
        (integer 2)
        (integer 3)
        (integer 4)
        (integer 5)
        (integer 6)
        (word "mul")
        (word "add")
        (list
                (word "mul")
                (word "add")
        )
        (integer 1)
        (word "times")
)
```

```
(program
        (integer 1)
        (integer 2)
        (integer 3)
        (integer 4)
        (integer 5)
        (integer 6)
        (word "mul")
        (word "add")
        (word "mul")
        (word "add")
        (list
                (word "mul")
                (word "add")
        )
        (integer 0)
        (word "times")
)

(program
        (integer 1)
        (integer 2)
        (integer 3)
        (integer 4)
        (integer 5)
        (integer 6)
        (word "mul")
        (word "add")
        (word "mul")
        (word "add")
)
```

11.3 Eager evaluation

This assumes that + and * are never redefined. Other functions can also be evaluated eagerly provided that you know how to call them.

```
(integer a) (integer b) (word "+") -> (integer (+ a b))
(integer a) (integer b) (word "*") -> (integer (* a b))
```

Needs several repeated applications.

```
1 2 + 3 4 * +

(program
        (integer 1)
        (integer 2)
        (word "+")
        (integer 3)
        (integer 4)
        (word "*")
        (word "+")
)

(program
        (integer 3)
        (integer 3)
        (word "*")
        (integer 4)
        (word "+")
)

(program
        (integer 9)
        (integer 4)
        (word "+")
)

(program
        (integer 13)
)
```

11.4 Generation of threaded code

```
1 2 pee puts

(program
        (integer 1)
        (integer 2)
        (word "pee")
        (word "puts")
)
```

If the compiler finds out that the Plang function `pee` binds to the C function `cork` with signature `fff` then it can transform the AST to the following:

```
(program
        (push-single 1)
        (push-single 2)
        (call "cork")
        (cdecl-clean 8)
        (retrieve-single)
        (single-to-string)
        (call "puts")
        (call "free")
        (cdecl-clean 4)
)
```

`retrieve-*` copies the return value to the stack according to GNU C calling convention. On x86, `retrieve-single` pops `st0` as a single-precision floating-point integer to the x86 stack.

```
sub     esp, byte 4
fstp    dword [esp]
```

Therefore the generated code is:

```
main:
        push    dword 0x3f800000        ; IEEE 754 single 1.0
        push    dword 0x3f900000        ; IEEE 754 single 2.0
        call    cork
        add     esp, byte 8
        sub     esp, byte 4
        fstp    dword [esp]
        call    single_to_string
        call    puts
        call    free
        add     esp, byte 4
```

12
Virtual machine

12.1 Model

The virtual machine has

- an instruction pointer,
- an operand stack (*the* stack),
- a dictionary stack,
- a call stack,
- a string pool.

The instruction pointer contains the address of the buffer that contains the operation codes.

The string pool consists of a dictionary and a table. The dictionary maps strings to integers. The table maps integers to strings. Integers are allocated consecutively starting from 0. The purpose of the string pool is to deduplicate strings. The table is also known as the *atom table*?

13
Translator

The translator can be thought as non-optimising compiler. It simply transforms abstract syntax trees into virtual machine codes.

Example: the following abstract syntax tree:

```
1 2 add
"hello" length
[ 1 2 ]
[ 1 ] exec
```

becomes

```
:main
        push.int        1
        push.int        2
        lookup          0       #       "add"
        call
        push.str        1       #       "hello"
        lookup          2       #       "length"
        call
        push.addr       3       #       :gensym0
        push.addr       4       #       :gensym1
        call
        quit

:gensym0
        push.int        1
        push.int        2
        ret

:gensym1
        push.int        1
        ret
```

Example:

```
1 10 range spill
```

becomes

```
        range 1 10
        spill
```

14 Optimiser

14.1 General optimisation algorithm

1. Sequentially apply each pass.
2. Repeat until the code does not change or the maximum number of passes is exceeded.

Several optimisation passes are implemented in this optimiser.

Make sure that the optimisation does not change the meaning of the program. You must be careful about integer overflows. Write all your assumptions explicitly.

14.2 Strength reduction

Multiplication by power of 2 becomes left shift. Division by power of 2 becomes right shift.

14.3 Dictionary-entry binding

If the optimiser can prove that a word always references the same dictionary entry throughout in the entire life of that word then it can *bind* that word. The only thing that complicates the proof (often making it impossible) is eval.

14.4 Eager evaluation of pure functions

A *pure* function is a function in the mathematical sense. Formally a subroutine is said to be pure iff it:

1. always returns the same value for the same input and
2. can be called as many times as we like without making any difference.

Therefore it cannot have any side effect. A program containing pure functions can be optimised by evaluating them at compilation time. Eager evaluation of pure function includes *constant expression elimination*.

For example if `add` is a pure function $a(x, y) = x + y$ then

```
        pushi   1
        pushi   2
        pushi   3
        call    <mul>
        call    <add>
```

can be optimised to

```
        pushi   1
        pushi   6
        call    <add>
```

which is then subjected to further optimisations in other passes.

Part VI

Mini-thesis draft

15
Introduction

The most carefully written chapter that is also the least carefully read.

(TODO find the citation—tis one is from anarchist FAQ) quotation [being] a handy thing to have about, saving one the trouble of thinking for oneself.

Alan Alexander Milne (1882–1956)

[1] views a programming language as a notation.

Throughout the book, we view a programming language as a notation. The user of a notation must make a mental investment in learning before gaining any advantage. If the investment is greater than the return in mental leverage for the production of correct programs, the language fails. The mental leverage results from an increased ability by the programmer to master the details of a computation. For the investment to be low, there must be an inherent simplicity in the language, uniformly applied rules, and few special cases.

For good mental leverage, the language must provide powerful abstraction tools that allow the user to subdue the clamor of complexity and master a large program. It is essential to remember that the most important function of a programming language is to communicate the algorithm to other programmers. The language is constructed for the convenience of humans, not machines. [1, p. xviii]

15.1 Many faces of Plang

These phrases summarise various aspects of Plang:

- programs acting as data
- data acting as programs
- a representation for anything
- a glue for using C libraries together
- a way of interactively trying C libraries

- a program for manipulating trees and executing them
- a playground for computer scientists and programmers
- an agglomeration of unfulfilled ambitions and shattered dreams
- a stack-oriented interpreted programming language with a simple form
- a desperate attempt of an undergraduate student looking for personal identity
- something like XML, configuration file, rc file, but executable and without tags

Plang shall help you whatever you want to do with a computer.

Plang is built on top of C with the intention of integrating various C libraries. There is currently no intention of replacing C although such possibility is not precluded.

15.2 Plang as a programming language

Plang started as a programming language.

Plang can be used in imperative style, procedural style, structural style, object-oriented style, functional style, declarative style, or a mix of styles, anything you can think of. It is just a matter of creating a convenient abstraction.

Plang is a stack-oriented concatenative interpreted programming language that stole much from Joy and PostScript.

At the time of this writing Plang was available only for GNU/Linux on x86 because the interpreter and the libraries were written in C and a little x86 Assembly.

Chapter 18 starting on page 58 contains a hands-on introduction with several example codes, describing how to use Plang as a calculator.

15.3 Separation of form and meaning

Mathematics separated content (semantics, meaning) and presentation (form, appearance) and provides several meaningful rules which allow us to manipulate form without actually needing to associating any meaning to that form. The rules have meanings but the manipulations do not have to be meaningful. Giving meaning to something is hard? Mathematics enables the scientific revolution because it allows humans to overcome their intuitions which often get into dead ends or the wrong way. (Or mathematics didn't have anything to do with scientific revolution?)

For example previously we thought that $+$ means the addition of two numbers. Upon knowing abstract algebra we see that $+$ is just a binary relation with certain properties. We have generalised our thinking. By generalisation we separate meaning and form further by giving a wider meaning to the same symbols and also inventing new symbols.

15.4 The cause of bugs

Unless there is hardware failure, a computer never fails to do what programmers *told* it to do. Unfortunately sometimes what programmers told it to do is not what they *want* it to do. That miscommunication between human and computer is the cause of bugs. What people write is sometimes not what they mean and they do not realise that it might be interpreted differently until they find something they did not expect, such as errors, crashes, hate mails, newspaper editorials, lawsuits, injunctions, or other feedback. A good feedback is important for finding the bug.

If I may, I'd like to coin the rather unhelpful abbreviation WYWINWYW (what you write is not what you want).

15.5 Preventing bugs

There are several possible approaches:

1. Do not give any meaning to programs.
2. Make the form of the program reflect the meaning of the program.
3. Make it easy for people to write what they mean.
4. Make it hard for people to write what they don't mean.

Software engineering is all about eliminating those bugs.

People do not understand who they are talking with.

The reason for this miscommunication is that the same word has different meaning for different people. People have different experience, attach different meaning to various words, and therefore have different expectations about how a word is used and what it means.

Thought as internal mental state is abstract whereas speech and writing are concrete. People cannot read the thought of others but can hear the speech of or read the writing of others. Communication is bidirectional. The transmitter concretises thought into language by speaking, writing, or other means. This concretisation is then abstracted again in the receiver. Miscommunication occurs when the concretisation and abstraction are not mutual inverses. The transmitter's concretisation might be imperfect. The receiver's abstraction might be imperfect. The transmission medium might be imperfect.

15.6 The difficulty of finding one's own bugs

When we strongly believe that what we write is correct (that is, we believe that what we mean is indeed what we write), we become significantly biased, and thus lose objectivity to find our own bugs.

We are easily biased because we tend to put more meaning to what we write than what is actually there. Maybe that is just how the brain works.

Unlike people who will scan and skim through your writing miss your point, and misunderstand your argument while believing that they understand, computer will read every of your writing, bit by bit, byte by byte, character by character.

16
Literature study

Comparison of similar several programming languages.

16.1 Plang and Joy

Joy was created by Manfred von Thun. The form of Joy is the source of the form of Plang. The primary similarity between Joy and Plang is:

1. Concatenative in the sense that programs are formed by concatenating words [cite?].
2. Stack-oriented in the sense that the results of operations are put in the stack and both of them have words that explicitly manipulate the stack [cite?].

The primary differences between Joy and Plang is:

1. Joy is a pure functional language [2] whereas Plang started as a procedural language which can be used in functional style, and therefore this makes Plang an impure functional language.
2. Joy uses static binding in the sense that it is not possible to redefine a word while the program runs whereas Plang uses dynamic binding in the sense opposite to that.

Guile by GNU is similar in spirit.

In my opinion Joy has a more desirable grammar than PostScript.

16.2 Plang and PostScript

Although PostScript is usually known as a page description language, it is indeed a programming language, as demonstrated by [?].

The primary similarity:

1. Both of them are interpreted.
2. Both of them are stack-oriented.

The primary difference:

1. Plang was not specifically designed for describing pages, although Plang should be able to be used like PostScript due to the high degree of similarity.

16.3 Plang and Factor

17
Interpreter

The thing that makes introductory programming courses somewhat more convenient.

The interpreter is a command-line program that executes Plang source files. It has two modes: *interactive* and *batch*.

17.1 Invocation

Command-line arguments are read from left to right, in the same order they are written. Two adjacent arguments are separated by space.

```
plangi -h
plangi -v
plangi (-lLIBNAME | PATH)* [-- SCRIPTARGS]
```

-h Print a brief reminder of these command-line arguments and then quit.

-v Print the version of the interpreter and then quit.

-lLIBNAME (A hyphen followed by the lower-case letter L. Note that there cannot be any space in this argument.)

This argument tells the interpreter to load the dynamic library whose name without extension is `LIBNAME`. The dynamic library is supposed to be a Plang dynamic library, that is a usual dynamic library but with some special symbols that are specific to Plang. The library is searched in the *library search path*.

See examples below if your libraries are not in standard directories.

PATH Executes the file whose path is `PATH`. If that path contains unusual characters such as spaces, <, >, `|`, `*`, and/or `?`, among other things meaningful to the shell you are using, then you will have to *quote* it. In this case for the Bourne shell it is usually sufficient to put an apostrophe (`'`) before the path and an apostrophe after the path. Refer to the documentation of the shell you are using for how to quote an argument. If the path is `-` (a hyphen) then the interpreter reads standard input instead. To execute a file named `-` in the current working directory, specify `./-` instead.

-- That is two hyphens. Indicates that all arguments after this double-hyphen are *script arguments*. The double-hyphen itself is not a script argument. All script arguments are passed to the script as a list named `ARGS`. You must use the double-hyphen if you want to pass arguments to your script. Otherwise the interpreter will treat them as paths of script files to be executed instead of script arguments. To execute a file named `--`, specify `./--` instead.

17.2 Example invocations

```
LD_LIBRARY_PATH=. ./plangi -lcore
```

```
LD_LIBRARY_PATH=. ./plangi -lcore dumpargs.p -- one two three
```

The following example loads the arithmetic library and the core library respectively in that order, executes the file `mydefs.p`, executes standard input, loads the C bindings, and then executes the file `saybye.p`:

```
./plangi -lcore -larith mydefs.p - -lc saybye.p
```

17.3 Help system

The interpreter comes with a help system. Entering `help` starts this system. The word `apropos` takes one string and searches it in the description of help items. You use `apropos` when you know what you want to do but don't know the word that does it. The word `whatis` takes one name and fetches the help entry for it. You use `whatis` when you know a word but don't know what it does. For example, `/integer apropos` looks for everything which contains the string 'integer', and `/add whatis` looks for the description of `add`.

The interpreter might be linked with GNU Readline and GNU History. How about BSD Editline if you don't want GPL?

17.4 Interactive mode

It also allows interactive programming with Plang. It also allows debugging Plang scripts. In interactive mode, given program fragments, it executes them, prints the stack after execution, and waits for the next fragment. When the interpreter is ready for the next fragment, it displays the following prompt and waits for you to enter something.

```
>
```

The interpreter stops executing a file when it encounters the end-of-file and then it executes the next file if there is any. In UNIX, when the interpreter is reading from standard input, you can press Ctrl-D to write an end-of-file. If there is no more file, the interpreter quits. To quit the interpreter regardless of whether there is any next file, you can enter:

```
quit
```

This document assumes that you are using the Bourne shell or a shell compatible with that shell, like Bash (Bourne-again shell).

18
Using Plang as a calculator

A hands-on introduction, since math homeworks will still be abound in the foreseeable future.

This describes how to use the interactive interpreter as a programmable postfix calculator. This starts with simple examples, and builds up complex things as it goes. Make sure that you have some time while reading this because you will try some examples.

18.1 Introduction to postfix notation

Plang as calculator is mostly the same as the usual scientific calculator. The difference is that in Plang you *don't* write things like this:

```
1 + 2
```

but you write things like this instead:

```
1 2 add
```

In the first of the two fragments above, `1` and `2` are the *operands* of the *operator* +.

The advantage of postfix notation (this manner of writing operators after the operands) is that the order of operation is clear and therefore it does not need to be specified with parentheses for example. The order of evaluation is the same as how you write them (from left to right). Moreover parsing postfix expressions is simpler than parsing parenthesis-laden infix expressions.

18.2 Starting the interpreter

Now is the time for some action: `cd` to the directory that contains the file `plangi`, and then enter the command line below into your terminal to start the interpreter.

```
./plangi -lcore -lsome_math
```

18.3 Quitting the interpreter

In UNIX, Ctrl-D (that is holding Ctrl down and then pressing D) will send an *end-of-file* to the interpreter. The interpreter quits when it encounters an end-of-file. This is the preferred way of quitting the interactive interpreter.

18.4 Addition

To compute the sum of one and two, enter the following code and the interpreter will answer with three.

```
1 2 add
```

To add three to the result of the computation in the previous example, enter the following code and the interpreter will answer with six.

```
3 add
```

Also try using `sub`, `mul`, and `div` in place of `add`.

18.5 The operand stack

At this point you should wonder how the interpreter remembers the result of previous operations. Now try entering this fragment into the interpreter.

```
5 4
```

The interpreter responds with `6 5 4`. At this point you may have correctly inferred that the interpreter maintains a *stack*. Operations operate on the values on the top of the stack. After the execution of each fragment, the interpreter prints the contents of the entire stack. Try entering this:

```
3 add mul add
```

and the interpreter answers with 41, which is the result of adding 3 and 4, multiplying 5 and the result, and then adding the result and 6, as expected. Note that the top of the stack is the right side of the text. This is the convention used by the interpreter.

18.6 Other functions

This is a list of some rather commonly-known mathematical functions you can try for now. Some things you have to be aware of are:

- The trigonometric functions accept angles in *radians*. (In case you forget, 2π radians are equal to 360 degrees.)
- `log` is the *natural* logarithm. (In case you forget, the base of the natural logarithm is e, that is approximately 2.71828.)

- An integer may be silently converted to an equivalent floating-point integer where required, but a floating-point integer is never implicitly converted to an integer.

These functions take one number to produce one number.

```
exp log log2 log10 sin cos tan csc sec cot
asin acos atan acsc asec acot
sinh cosh tanh csch sech coth
asinh acosh atanh acsch asech acoth
```

These functions take two numbers to produce one number.

```
add sub mul div mod
shl shr pow
hypot
```

(Where is the reference?)

Most computers today conform to IEEE 754, a standard for floating-point arithmetic. However, you should expect floating-point arithmetics to be *inexact*. If you are a scholar in computer science, you may want to read the paper *What Every Computer Scientist Should Know about Floating-point Numbers* by (who? David Goldberg?).

18.7 When errors occur

Plang is *case-sensitive*. It distinguishes capital letters and small letters. For example, in the following fragment, unless `ADD` has been defined, the interpreter will complain with an error that such name is undefined.

```
1 2 ADD
Error: /undefined.
```

The error happens after the fragment has been partially executed. As a consequence there are some leftovers on the stack.

Dividing by zero is also an error.

```
1 0 div
Error: /division-by-zero
```

In the usual integer arithmetics, division by zero is best left undefined since defining it will raise contradictions. However, in IEEE 754 arithmetics, dividing a nonzero number by zero yields infinity whose sign is determined by the sign of the product of those two numbers, dividing zero by zero yields NaN (really?).

```
1.0 0.0 div
```

Another exception, also in IEEE 754 arithmetics, taking the logarithm of zero yields Not-a-Number. This value may indicate that the computation is ill-defined.

```
0 log
```

18.8 Defining your own functions

Of course you can define your own functions in a programmable calculator or it won't be called 'programmable'; the calculator doesn't get its name for nothing. The functions you define correctly will work just as fine as the functions that come with Plang.

Let's say we want to define a custom function named `muladd` like this:

```
"muladd" [mul add] def
```

The word 'def' is not a keyword. It is the name of a function which takes a name and a procedure and associates that name with that procedure. It modifies the interpreter's dictionary. The word 'def' is not limited to defining something to functions. It can establish new relationships or modify existing relationships. This means you can redefine 'def', but that is discouraged since that affects the expected behaviour of the interpreter.

Theoretically you can also write '/1 0 def' which indeed redefines the number '1' to mean 0, but that is surely discouraged since that may lead to great confusion. In Plang, conceptually 2 is the name of a small program which pushes the integer whose value is two onto the top of the stack. That is possible because in Plang, numbers don't have special syntax; the lexical analyser treats identifiers and numbers the same, that is as black tokens. To understand that, you have to understand how the interpreter parses its input, which is beyond the scope of this book. You can safely skip this paragraph. You can try.

```
/1 0 def
1 1 add
# The interpreter answers with 0.
```

Perhaps you want to specify angles in degrees. You can convert angles from degrees to radians and then feed the converted values to the functions which have been provided by the system. Now you have some trigonometric functions which accept degrees as their inputs. You can think of 'sindeg' as a big box which contains two boxes: 'degtorad' and 'sin', in which the output of 'degtorad' is fed to the input of 'sin'. By that, we have demonstrated software componentry.

```
/degtorad { 2 pi mul 360 div mul } def
/sindeg { degtorad sin } def
/cosdeg { degtorad cos } def
```

How does it work? What happens when you the interpreter encounters a name of a function you have previously defined? What if the function hasn't been defined yet? Can you redefine a number? (No.)

Write a function 'inc' which increments the number at the top of the stack by 1. Example: '1 inc' gives '2'.

18.9 Giving names to variables

$f(x, y) = x^2 + y^2$.

The variables x and y will be created when execution enters f, and destroyed when execution exits f. The function left-brace actually pushes an empty dictionary to the dictionary stack. The function right-brace pops a dictionary from the dictionary stack. This comes with an overhead, so if you are absolutely certain that you won't ever modify the dictionary in your procedure, you can use 'defwhat' instead of 'def'?

```
/f {
    /y var
    /x var
    x x mul  y y mul  add
} def
```

18.10 Piecewise functions

```
/f {
    /x var
    x 0 eq { 1 } { x sin x div } if-else
} def
```

18.11 The *case* function

Theoretically, we could have made sin take two inputs: a number and a unit name, but a mathematician deals with angles in radians, so '30 /deg sin' and '2 pi mul /rad sin' is out of the question, let alone '1 /grad sin'. However, if you really want, 'sinu' (sine with units) is defined.

```
/uadapt {
  case
      /deg /degtorad
      /rad { }
      /grad /gradtorad
  esac
} def
/sinu { uadapt exec sin } def
/cosu { uadapt exec cos } def
180 /deg sinu
pi /rad sinu
1 /grad sinu # ???
```

18.12 Stored programs

Some functions are used more often than others. Some functions are long. We write them once, and reuse them until the end of time.

Source codes.

19
Architecture

Describes what happens to your code.

Let's say that you have written some code and saved it in a text file.

The parser reads characters and gives a syntax tree. Chapter 20 starting at page 66 defines the grammar. The grammar is the transformation from characters into the syntax tree. Chapter [?] defines the transformations from concrete syntax trees into abstract syntax trees. The syntax trees can be further transformed by the optimisation process. The syntax trees can be translated to machine code or executed by the syntax tree interpreter.

The approach used for transformation is *cascading transformation* because it is too complicated to directly transform the source to machine code. The approach used for optimisation is *multipass optimisation*.

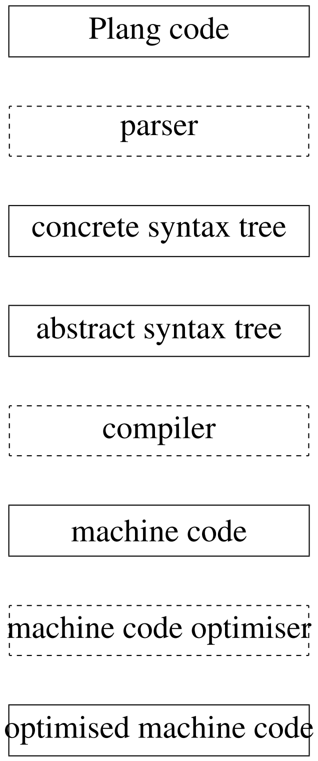

Figure 19.1: Optimisation scheme

20 Grammar

The specification of the form of a Plang program.

20.1 The main idea of Plang grammar

A program is almost just as simple as an alternation of words and blanks. Alternatively if you prefer to emphasise the words over the blanks then you can say that a program is a blank-delimited sequence of words. Put simply, a program is almost word, blank, word, blank, word, blank, and so on, like:

```
word word word word word word word ...
```

I said *almost* because there are indeed many other things, such as comments, literals, and others, beside words and blanks.

20.2 The manner of specification

This section describes how to read the next section.

The grammar is context-free, sequential, and easily-satisfied, where sequential means that the rules are sequentially tried from top to bottom and easily-satisfied means that if a nonterminal is successfully matched then all other possibilities are ignored. Such oddities were crafted in order to eliminate ambiguity by making the order of the rules unmistakably important and preventing alternative trees. In the example below, if the rule $A \to B$ matches then all other rules whose left-hand side is A (the rules $A \to C$ and $A \to D$) are ignored.

```
A ::= B
A ::= C
A ::= D
```

20.3 Grammar of Plang

This section presents the Plang grammar with implied sequentiality and easily-satisfiedness as explained in §20.2. You do not have to read the accompanying narration if the description in boxed monospaced text is already clear for you.

The starting nonterminal is *program* which is a sequence of top-level nodes which are children of the root node which is the node for the program nonterminal itself. `<END>` is a special symbol that matches the end of input.

```
program:
    <END>
    node program
```

There are at least eight kinds of *top-level nodes* of the concrete syntax tree which is described in more detail later in chapter 24 starting at page 81.

```
node:
    real
    fraction
    integer
    list
    string
    blockcomment
    linecomment
    word
    blank
```

A *real-number literal* can have a sign, an integral part, a fractional part, and an exponent specifier, but only some combinations of them are legal. The value of the real-number literal `i.fEp` is $i.f \times 10^p$ where i is the integral part, $.f$ is the fractional part, and p is the exponent. If the exponent specifier is not present then $p = 0$. The fractional part is a radix point followed by one or more decimal digits. The exponent specifier is the letter E followed by an integer.

```
real:
    intpart fracpart expspec
    intpart fracpart
    intpart expspec

intpart:
    sign decimals
    sign
    decimals

fracpart:
    '.' decimals

expspec:
    'E' integer
    'e' integer

fraction:
    integer '/' integer
```

An *integer literal* is an optional sign followed by an uninterrupted sequence of one or more decimal digits. If the integer literal has a sign then there cannot be any intervening character between the sign and the first digit. In particular there cannot be any white space between the sign and the digits. The nonterminal `decimals` matches a string of decimal digits.

```
integer:
    signed-integer
    unsigned-integer

signed-integer:
    sign unsigned-integer

unsigned-integer:
    "0x" hexadecimals
    "0X" hexadecimals
    decimals

hexadecimals:
    decimal hexadecimals
    decimal
    hexadecimal-letter hexadecimals
    hexadecimal-letter

decimals:
    decimal decimals
    decimal
```

A *list* (of tokens), also known as a *quoted program*, *delayed program*, or *stored program*, is a program surrounded by brackets. Without this rule the language would be regular. In case you forget, a regular language is a language that can be recognised by a finite-state automaton.

```
list:
    '[' ']'
    '[' program ']'

listhead:
    '[' program
    '['

listtail:
    ']'
    program ']'
```

A *string literal* can contain almost anything, including newlines and control characters. The only thing a string literal cannot contain is the quote character itself which is already used to mark the ends of that literal.

```
string:
    '"' stringtail

stringtail:
    '"'
    anychar stringtail
```

The rule for a *word* is very permissive; almost anything that does not fit anywhere else is a word. Other programming languages might use the term *identifier* for what we call *word* here but the grammar for those identifiers is more restrictive than the grammar for these words. On the other hand a *blank* is a sequence of white spaces. It can be seen below that blanks and words have remarkably similar rules.

```
word:
    black word
    black

blank:
    white blank
    white
```

There are two kinds of comments: *line comments* and *block comments*. A line comment ends with a line terminator. There are three recognised line terminator sequences (four if you count the end-of-input although technically it is not a sequence). A block comment can containing anything but a right parenthesis. Block comments do not nest. The nearest right parenthesis after the current position terminates the block comment.

```
linecomment:
    '#' linecommenttail
    '#'

linecommenttail:
    <CR> <LF>
    <CR>
    <LF>
    anychar linecommenttail

blockcomment:
    '(' blockcommentbody

blockcommentbody:
    ')'
    anychar blockcommentbody
```

The rest of this section describes the *character classes* which are just one step away from terminals: a `sign` is either plus or minus, a `decimal` is a decimal digit, an `anychar` is any character, a `white` is any white space, a `grey` is any character that has special meaning, and a `black` is any character that is neither white nor grey.

```
sign:
    '+'
    '-'

decimal:
    '0'
    '1'
    '2'
    '3'
    '4'
    '5'
    '6'
    '7'
    '8'
    '9'

hexadecimal-letter:
    'A'
    'B'
    'C'
    'D'
    'E'
    'F'
    'a'
    'b'
    'c'
    'd'
    'e'
    'f'

anychar:
    <any character>

white:
    <HT (ASCII 9)>
    <LF (ASCII 10)>
    <VT (ASCII 11)>
    <FF (ASCII 12)>
    <CR (ASCII 13)>
    <SP (ASCII 32)>

grey:
    '#'
    '('
    ')'
    '['
    ']'

black:
    <any character not white not grey>
```

20.4 Example of concrete syntax tree

The following program

```
1 2 add
```

gives rise to the following concrete syntax tree

```
program
  integer "1"
  program
    blank " "
    program
      integer "2"
      program
        blank " "
        program
          word "add"
          program
            <END>
```

21 Parsing

Characters in, concrete syntax trees out.

21.1 The goal and purpose of parsing

The goal of parsing an input text with respect to a grammar is to construct a concrete syntax tree from that input text with respect to the grammar. The purpose of that concrete syntax tree is to be transformed into abstract syntax trees.

21.2 Introduction to stream parsing

The parser is a *stream parser* which means that characters are taken one by one to build a *token* and once a character is taken there is no way to give it back. Furthermore for simplicity we also prohibit the parser from buffering and therefore the parser cannot backtrack.

Immediately after a character is read, the parser either:

- *appends* that character to the token, or
- *emits* the token and then starts a new token begun by that character.

Appending a character to a token can change the type of that token. For example when the parser encounters a point[1] while building an integer literal, the token being built becomes an incomplete real-number literal.

21.3 Mathematical description of stream parsing

Let Σ be the set of all bytes. Let the input x be an element of Σ^* (that is to say that the input is a byte string). Let S be the set of all possible states of the parser. Let t be the transition function of the parser. This function describes how the parser state changes when the parser reads a character or encounters an end-of-input marker. The signature of this function is

$$t : S \times \Gamma \mapsto S \tag{21.1}$$

[1]ASCII 46 decimal (2E hexadecimal), also known as dot, period, or full-stop.

where

$$\Gamma = \Sigma \cup \{\omega\} \tag{21.2}$$

is the augmented alphabet and ω is the end-of-input marker.

The *parsing function* of the parser describes the state of the parser after the parser has completely read the input. The parsing function is defined as

$$t^*(s, x) = t^*(t(s, a(x)), b(x)) \tag{21.3}$$

where a is the beheader and b is the betailer

$$a(\varepsilon) = \omega \tag{21.4}$$

$$b(\varepsilon) = \varepsilon \tag{21.5}$$

$$a(st) = s \text{ where } s \in \Sigma \text{ and } t \in \Sigma^* \tag{21.6}$$

$$b(st) = t \text{ where } s \in \Sigma \text{ and } t \in \Sigma^* \tag{21.7}$$

where ε is the empty string and $\cdot$ denotes string concatenation, and thus:

$$a : \Sigma^* \mapsto \Gamma \tag{21.8}$$

$$b : \Sigma^* \mapsto \Sigma^* \tag{21.9}$$

$$t^* : S \times \Gamma^* \mapsto S \tag{21.10}$$

When the input terminates, the parser must eventually terminate. The parser terminates when it reaches a *fixed point*, that is when:

$$t(s, \omega) = s \tag{21.11}$$

which causes $t^*(s, \omega) = s$. That is to say that both the input is exhausted and the state will not change anymore. It is possible for the state to change after the input has been exhausted.

The concrete syntax tree for input x will be contained somewhere in the parser state $t^*(s_0, x)$ where s_0 is the initial state of the parser.

21.4 The parser state

The parser state is a tuple (T, k, p) where T is the concrete syntax tree, k is the type (kind) of the partial token, and p is the string of the partial token.

21.5 The role of parser generator

A parser generator transforms a grammar to a corresponding transition function. The grammar is a way of specifying the transition function.

21.6 Introduction to distributed parsing

In distributed parsing, the input is split arbitrarily into fragments, processed independently, and then merged to give one concrete syntax tree.

Why would one want distributed parsing?

1. Does the input of the parser ever become significantly large, such as one million bytes?
2. If such limit is passed, does the parser still run in acceptable time?

In my machine compiling the Linux kernel source tree with all-no configuration takes about ten minutes? Where is the bottleneck? Is it in the parser? The linker? The optimiser? The operating system which spawns processes?

A drawback of distributed parsing is that parsing error cannot be determined until all trees have been merged.

21.7 Prerequisites of distributed parsing

Let $P(x)$ be the concrete syntax tree that results from parsing the string $x \in L(G)$ according to the unambiguous context-free grammar G. It can be seen that P is not defined for any $x \notin L(G)$. $L(G) \subseteq \Sigma^*$.

Informally the parser for the grammar G is said to admit distributed parsing if the input can be split arbitrarily into fragments, each parser can work alone with its fragment, and then the output of each parser can be merged reasonably efficiently. Formally the grammar G is said to *admit distributed parsing* if the following two functions exist for G.

1. P^* (a conservative extension of P from $L(G)$ to Σ^*) is a function whose domain is Σ^* with the restriction that $P^*(x) = P(x)$ for all $x \in L(G)$. This is to say that P^* has to be defined for all inputs, including strings that are illegal for P (the strings outside $L(G)$) but P^* must be P for inputs that can be handled by P.
2. M (a merging function) is a function such that:

$$\forall s \in \Sigma^* \ \ \forall t \in \Sigma^* \quad P^*(st) = M(P^*(s), P^*(t)) \tag{21.12}$$

and M must be associative:

$$M(a, M(b, c)) = M(M(a, b), c) \tag{21.13}$$

If we define $C(s, t) = st$ to be the result of concatenating s and t, then we can write in function notation $P^*(C(s, t)) = M(P^*(s), P^*(t))$.

Indeed it can be generalised: the characteristics of tasks that admit distributed processing is: can be split arbitrarily, processed independently, and merged reasonably efficiently.

If $P^*(st) = M(P^*(s), P^*(t))$ then we can show that $P^*(stu) = M(P^*(s), M(P^*(t),$
We can show that $P^*(stuv) = M(P^*(st), P^*(uv)) = M(M(P^*(s), P^*(t)), M(P^*(u), P^*$

21.8 Time complexity of distributed parsing?

If an input of size s is split into n fragments of size $s_1, \ldots, s_n$, then it can be merged with $O(\lg n)$ times of calling M. If we assume that splitting can be done in $O(f(n, s))$ time and parsing can be done in $O(g(s_k))$ time and merging a tree of size a and another of size b can be done in $O(h(a, b))$ time then... what?

21.9 Continuing parsing in spite of errors

In interactive mode the interpreter should stay alive when fed with wrong input. The prerequisite of such robustness is that the parser must have a well-defined state after encountering unexpected input. Even in non-interactive mode the parser must still be robust in the sense that it cannot just crash when fed with unexpected input.

When a character is read and there is no legal action for the partial token, the character is called the *offending character* and the partial token is either *emitted* or *discarded*. The token is emitted if replacing the offending character with a space would indeed have emitted the token; otherwise the token is discarded. Then a new partial token is allocated, and the offending character (not the space) is appended to this new token. A parsing error is raised. Then parsing can proceed as if no error had happened.

21.10 Example of parsing error

When the input is

```
1.2blah 3
```

After reading `1.2`, its partial token is an incomplete real-number literal. To complete the real-number literal, the parser expects a blank. When it encounters `b`, it returns an error number that indicates a parsing error, emits the token `1.2`, and starts another token `b`, this time of type *word*. It then continues scanning `lah`, forming the partial token `blah`. It then encounters a space and thus emits the token `blah` that is a word, and then the engine would try to execute that word.

For example, in the following fragment, the interpreter issues an parsing error, but gives 8 on the stack.

```
3 5 1.2add
```

21.11 Example of parsing

The program in Figure 21.1 becomes the CST in Figure 21.2 which then becomes the abstract syntax tree in Figure 21.3. In the conversion from CST to AST there might be "integer literal too large" error since the AST might use fixed-width integers. The optimiser works with abstract syntax trees and consists of AST transformations.

```
# b
12 "abc" [a  [ b ]c] a
(

hooray

)
# banzai
```

Figure 21.1: An example of a program

```
(program
        (debug-file 0 "/tmp/test.p" 1295166708)
        (debug-line 0 1 " b"
                (line-comment " b")
                (eol "\n")
        )
        (debug-line 0 2 "12 \"abc\" [a  [ b ]c] a"
                (integer "12")
                (blank " ")
                (string "abc")
                (blank " ")
                (list
                        (word "a")
                        (blank "  ")
                        (list
                                (blank " ")
                                (word "b")
                                (blank " ")
                        )
                        (word "c")
                )
                (blank " ")
                (word "a")
                (eol "\n")
        )
        (debug-line 0 3 "(\n\nhooray\n\n)"
                (block-comment "\n\nhooray\n\n")
                (eol "\n")
        )
        (debug-line 0 8 "# banzai"
                (line-comment " banzai")
                (eol "\n")
        )
)
```

Figure 21.2: The CST of the program in Figure 21.1

```
(program
        (debug-file "/tmp/test.p" 1295166708)
        (debug-line 1 " b")
        (debug-line 2 "12 \"abc\" [a  [ b ]c] a"
                (integer 12)
                (string "abc")
                (list
                        (word "a")
                        (list (word "b"))
                        (word "c")
                )
                (word "a")
        )
)
```

Figure 21.3: The AST of the CST in Figure 21.2

22
CST transformation examples

22.1 Prefixing definition

Suppose that Fred wants `def` to be in front and wants it to automatically recognise an identifier.

```
define name [content] -> "name" [content] def
```

```
(word "define") (word NAME) (list CONTENT)
-> (string NAME) (list CONTENT) (word "def")
```

22.2 Extracting documenting comments

```
#[
[fubarstify fubarstifies the calvroxified volcrax,
causing its calcrux to dwelven in its ksamedorn.] function
[dajo The string to be frobnicated.] param
[Frobciborfnicatenated dajo.] return
[frobnix frobby defrob] see-also
[2011-01-16] date
[Fred] author
#]

"fubarstify" [[frobnicate] 7 times] def
```

23 C API for parsing

23.1 Usage

The parser might need to be called several times before actually emitting any token.

Parsing can start as soon as a character is available. Parsing is done in one pass. No need to unget any input. No need to store any input beside the current character.

The *parser state* is encapsulated in a Plang_Parser structure. If you are not developing the parser then you do not need to know what is inside. The state consists of the path of the file that is being parsed, the current line number, the current column number, the number of bytes read so far, the current depth, and the depth limit. Position information is used for error reporting. Unfortunately errors are detected late in the program. The reported error might not always be the root cause of error, and therefore error reporting becomes less useful as a program gets more complicated. The programmer simply has to program correctly in the first place.

```
void    plang_parser_init (Plang_Parser* state);
```

The Plang_Parser structure is usually allocated as an automatic local variable, in C parlance. That means that you are expected to write like this:

```
        Plang_Parser    state;

        plang_parser_init(&state);
```

and then use the parser only in the procedure where you defined that state, or in procedures you call from that procedure. Nothing really prevents you from allocating memory on the heap, provided that you can make sure that it is freed later, but C programmers usually prefer to allocate variables on the stack instead of on the heap.

To feed a character to the parser, you use plang_parser_feed. PLANG_-AGAIN means the parser is still scanning and a token is not yet available.

PLANG_OK means a token is available. Other return value means error.

```
Plang_Errno     plang_parser_feed (Plang_Parser* state, int ch,
                          Plang_Token** out);
```

Note that you still need to feed the end-of-file (in C this is `EOF`) to the parser, or it might fail to emit the last token. Therefore the preferred way of looping the parser to read a block is like this:

```
for (;;) {
        int ch = getchar();
        Plang_Token* token;
        Plang_Errno er = plang_parser_feed(&state, ch, &token);

        ...

        if (ch == EOF) { break; }
}
```

23.2 Current implementation

The lexer is about two hundred lines of hand-written C code. The grammar is context-free but almost regular. It would indeed be regular if it were not for the nested braces.

A character is a byte. Unicode adoption and UTF-8 encoding should be easily implementable.

23.3 Character coding

Byte stream is an abstraction of source. Character stream decodes bytes from byte stream into characters. This abstraction allows us to treat terminals, files in local disk, files in another machine, strings in memory, and network sockets, among all possible IO sources, in the same way so that we can read and execute programs from those sources in the same way.

Decoder deals with character encodings. It transforms byte sequences to 32-bit Unicode code-points. This allows writing source code in various world scripts, not only in Latin. Currently there is only the decoder for 7-bit US-ASCII. However, we'll eventually support UTF-8. Unicode. ISO 10646.

The parser should be a conforming UTF-8 consumer. It should reject illegal sequences.

24 Syntax trees

Concrete syntax trees and abstract syntax trees.

There are two stages of syntax trees: CST (concrete syntax tree) and AST (abstract syntax tree). CST retains enough information so that it can be transformed back to the original source code. AST is closer to the intention of the programmer and also is more compact since blanks and comments are discarded.

Both the CST and the AST are structurally similar. The title of this chapter is *Syntax trees* because the API used to manipulate CST and AST is the same.

24.1 Dynastic trees

A dynastic tree is either a dynastic tree node or null (the empty dynastic tree, symbolised $\emptyset$). A dynastic tree node is $V = (c, h, k)$ where c is *content*, h is heir, and k is closest younger sibling. Older siblings are drawn more to the left. h is a dynastic tree. k is a dynastic tree. Define these accessor functions:

$$C(V) = c \tag{24.1}$$

$$H(V) = h \tag{24.2}$$

$$K(V) = n \tag{24.3}$$

A tree node is (v, w, e, s, n). The other components are named after the compass directions: n, e, s, w means north, east, south, and west, respectively.

- w is the closest older sibling, the node on the immediate left side of the current node, both having the same parent.
- e is the closest younger sibling, the node on the immediate right side of the current node, both having the same parent.
- s is the first child of the current node. The first child is the leftmost child. Although in historical dynasties heirs to the throne are not always the first child, the first child is indeed always the heir in dynasty trees.
- n is there only to facilitate navigating the tree. The root node is the only node with null parent.

24.2 Line-oriented debugging information

```
(debug-file ID PATH TIME)
(debug-line ID NUMBER TEXT CHILDREN)
```

`ID` is an integer which is used as unique identifier. `PATH` is the absolute path of the source file. `TIME` is the Unix time when the source file was parsed to generate this tree. Unix time is the number of seconds since Unix epoch (1970-01-01 00:00 UTC). This time is used to detect whether the source file is newer than debugging information. `NUMBER` is the line number. `TEXT` is the line as string without line terminator. `CHILDREN` are the actual parsing result. Debugging information is inserted as early as the CST is being built. *Stripping* discards debugging information. When debugging information is discarded, `CHILDREN` are all that remain.

25 Semantics

The terminology and notation used in describing the semantics.

25.1 The goal of semantics

The specification of semantics aims to define the meaning of a program. It is necessary but not sufficient to prove program correctness, and also for correct optimisation. Part of the semantics is deliberately undefined to provide room for future improvements.

> The problem of defining the semantics is very difficult. The goal, of course, is to specify the meaning of every syntactically valid construct so that its behavior can be understood unambiguously. [1, p. 91]

> We might argue that a number system based on 16 would be preferable to one based on 10, but until we evolve three more fingers on each hand, it runs counter to everyday experience. [1, p. 518]

> Probably the greatest contribution to semantic complexity comes from side effects. [. . .] Side effects are the root of the problems with aliasing. [1, pp. 518–519]

There are three approaches to the formal definition of semantics: operational, denotational, and axiomatic [1, p. 92].

25.2 Aspects of Plang semantics

There are many aspects of Plang semantics.

1. What happens to the stack after a word is executed?
2. Which parts of the program are executed before others?

Chapter 26. Some subsystems do not have pure functional semantics but have procedural semantics. Chapter 27. Functional semantics is a subset of procedural semantics. Everything that has functional semantics also has procedural semantics but not the other way. Does not talk about side effects and complexity.

In functional semantics there is "no concept of sequential time"[1].

25.3 List notation

(The list defined here has nothing to do with the implementation of Plang lists. The list here is used to define the stack later in this chapter.)

For the sake of describing semantics, a list is an integer-indexed collection. Formally a *list* L is a function from N to Ω

$$L : N \mapsto \Omega \tag{25.1}$$

where N is the set of indices of the list and Ω is the universal set[2], where N is a contiguous subset of the set of natural numbers starting from zero.

$$N = \{0, 1, 2, \ldots, n\} \tag{25.2}$$

A list can be written more detailedly (spelled out) as

$$L = [x_0, \ldots, x_{n-1}] \tag{25.3}$$

to mean

$$\begin{aligned} L(0) &= x_0 \\ L(1) &= x_1 \\ &\vdots \\ L(n-2) &= x_{n-2} \\ L(n-1) &= x_{n-1} \end{aligned}$$

where n is the *size* of the list and $L(k)$ is the element whose index in L is k. (Do not say 'kth element' as counting here does not begin at one.)

25.4 List concatenation

The *concatenation* of $A = [a_0, \ldots, a_{m-1}]$ and $B = [b_0, \ldots, b_{n-1}]$ is

$$AB = [a_0, \ldots, a_{m-1}, b_0, \ldots, b_{n-1}] \tag{25.4}$$

Do not confuse list-concatenation notation with array-access notation. In array-access notation $A[x]$ means the element at index x of the array A. In list-concatenation notation $A[x]$ means $[a_0, \ldots, a_{m-1}, x]$ which is the concatenation of A and the one-element list $[x]$. Here we rarely if ever use the array-access notation so if you encounter a capital letter followed by an opening square bracket then it likely means list concatenation.

[1]phrase borrowed from Manfred von Thun (cite?)

[2]Yes, I know, Russell's paradox. Shut eyes and cross fingers.

25.5 Stack, list, and stack operations

A *stack* is a list with the following *stack operations*. In the following, S is a list, x is anything, y is anything, and $\mathbb{L}_m$ is the set of all lists whose size is at least m. The stack grows to the right. The top of the stack is on the right.

$$\text{push} : \mathbb{L}_0 \times \Omega \to \mathbb{L}_1 \qquad \text{push}(S, x) = S[x] \tag{25.5}$$
$$\text{pop} : \mathbb{L}_1 \to \mathbb{L}_0 \qquad \text{pop}(S[x]) = S \tag{25.6}$$
$$\text{dup} : \mathbb{L}_1 \to \mathbb{L}_2 \qquad \text{dup}(S[x]) = S[x, x] \tag{25.7}$$
$$\text{exch} : \mathbb{L}_2 \to \mathbb{L}_2 \qquad \text{exch}(S[x, y]) = S[y, x] \tag{25.8}$$
$$\text{top} : \mathbb{L}_1 \to \Omega \qquad \text{top}(S[x]) = x \tag{25.9}$$

Therefore for example pop and dup are undefined for the empty stack, and exch is undefined for all stacks containing less than two elements.

25.6 Suffix-rewriting and stack-effect notation

To suffix-rewrite a list is to rewrite its suffix. Formally iff $L = AB$ then A is a prefix of L and B is a suffix of L. Effects of executing words are described in terms of suffix-rewriting. The stack-effect notation summarises the inputs and outputs of a word. Do not confuse stack-effect notation with the function-signature notation.

Iff executing the fragment f rewrites the stack from PA to PB for all P then we write:

$$f : A \to B \tag{25.10}$$

which means: if the stack is PA then executing the fragment f will rewrite the stack to PB where P here is the *maximal irrelevant prefix*, that is the part of stack that does not affect the fragment and is not affected by the fragment. P is to say that there might be other things deeper in the stack. P also says that words operate the top of the stack.

For example the word divmod rewrites the stack from $P[7, 2]$ to $P[3, 1]$ for all P since dividing 7 by 2 gives 3 with a remainder of 1. In this case we write:

$$\textsf{divmod} : [7, 2] \to [3, 1] \tag{25.11}$$

25.7 Alternative notation for lambda expression

We will just call this the *anonymous function notation*. The left side is the equivalent lambda expression. The right side is the anonymous function notation. Throughout the semantics specification we will write things like the right side.

$$(\lambda x_1 \ldots \lambda x_n \,.\, y) = (x_1, \ldots, x_n \mapsto y) \tag{25.12}$$

Example:

$$(\lambda a\ \lambda b \,.\, a + b) = (a, b \mapsto a + b) \tag{25.13}$$

is the function that takes two inputs (a and b) and computes their sum.

26
Functional semantics

Let S be the syntactic domain, the set of concrete syntax trees. Let $\mathcal{F}[x]$ denote the functional semantics of x where $x \in S$. The semantic function is $\mathcal{F} : S \to \mathbb{L}$ where $\mathbb{L}$ is the set of all (heterogeneous) stacks of Plang objects.

$$\forall a\, \forall b \quad \mathcal{F}[(\text{program } a\ b)] = \mathcal{F}[a]\mathcal{F}[b] \tag{26.1}$$

The following says that blanks are ignored.

$$\forall a\, \forall b \quad \mathcal{F}[(\text{program (blank } a)\ b)] = \mathcal{F}[b] \tag{26.2}$$

$$\tag{26.3}$$

$$\mathcal{F}[(\text{integer } m)] = n \text{ where } m \text{ is a numeral of } n \text{ where } n \in \mathbb{Z} \tag{26.4}$$

$$\mathcal{F}[(+\ a\ b)] = a + b \text{ where } a, b \in \mathbb{C} \tag{26.5}$$

$$\mathcal{F}[(\text{program } a\ b)] = \mathcal{F}[a]\mathcal{F}[b] \tag{26.6}$$

The execution of a program can be explained as mathematical substitution. Executing a program is simplifying an expression.

26.1 Stack effect

One of the most interesting for casual programmers is the stack-effect semantics of words which is written like

$$\mathcal{S}(n, L) = M \tag{26.7}$$

where n is a *node* of the concrete syntax tree. A node is a word, a blank, a literal, a list, and so on. The above notation is read as "the atom a rewrites the stack from L to M".

For example

$$\forall P \in \mathbb{L}\ \forall a \in \mathbb{C}\ \forall b \in \mathbb{C} \quad \mathcal{S}((\text{word } \mathsf{add}), P[a, b]) = P[a + b] \tag{26.8}$$

will often be written with implicitly quantified P as

$$\forall a \in \mathbb{C}\ \forall b \in \mathbb{C} \quad \mathcal{S}((\text{word } \mathsf{add}), P[a,b]) = P[a+b] \tag{26.9}$$

where P is accepted to be universally quantified over the set of all stacks. The above means that the word add so replaces two topmost items of the stack with their sum. Since the above notation is cumbersome we will use the following notation instead which is more convenient, in which we omit both P and the quantifier.

$$a\ b\ \mathsf{add} \rightarrow a + b \text{ where } a, b \in \mathbb{C} \tag{26.10}$$

which uses complex-number arithmetics.

If b is a blank then

$$\mathcal{S}(abc, L) = \mathcal{S}(c, \mathcal{S}(a, L)) \tag{26.11}$$

26.2 Fragment concatenation

The functional semantics of fragment concatenation is: iff $f_1 : C_1A_0 \rightarrow A_1$ and $f_2 : C_2A_1 \rightarrow A_2$ then

$$f_1\ f_2 : C_2C_1A_0 \rightarrow A_2 \tag{26.12}$$

where C_1 is empty list. This is almost similar to function composition.

For example:

$$\mathsf{add\ mul} : [4, 1, 2] \rightarrow [12] \tag{26.13}$$

since:

$$\mathsf{add} : [1, 2] \rightarrow [3] \tag{26.14}$$
$$\mathsf{mul} : [4, 3] \rightarrow [12] \tag{26.15}$$
$$f_1 = \mathsf{add} \tag{26.16}$$
$$f_2 = \mathsf{mul} \tag{26.17}$$
$$A_0 = [1, 2] \tag{26.18}$$
$$A_1 = [3] \tag{26.19}$$
$$C_2 = [4] \tag{26.20}$$
$$A_2 = [12] \tag{26.21}$$

The functional semantics of fragment concatenation can be generalised to: iff $f_k : C_kA_{k-1} \rightarrow A_k$ for each k from 1 to n then

$$f_1\ \ldots\ f_k : C_n \ldots C_1A_0 \rightarrow A_n \tag{26.22}$$

where C_1 is empty list.

26.3 Denotational semantics

We write $\|f\|$ to mean the denotational semantics of f.

Axiom: Let f be a fragment and $g_1, \ldots, g_m$ be functions. Iff $f : [x_1, \ldots, x_n] \to [g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)]$ then $\|f\| = (g_1, \ldots, g_m)$.

Example: prove that $\|\text{dup add}\|$ is $(x \mapsto 2x)$ where $x \in \mathbb{Z}$.

$$\text{dup} : [x] \to [x, x] \tag{26.23}$$

$$\text{add} : [x, x] \to [x + x] \tag{26.24}$$

$$\text{dup add} : [x] \to [x + x] \tag{26.25}$$

$$x + x = 2x \text{ since } x \in \mathbb{Z} \tag{26.26}$$

$$\text{dup add} : [x] \to [2x] \tag{26.27}$$

26.4 Rewriting systems

The subsystem consisting of all referentially transparent functions of Plang has rewriting-system semantics.

assume that the words are not redefined

assume no error

can use monads for sequencing

A *rewriting system* M is a tuple (A, C) where A is the *alphabet* and C is the *codex*. The codex is set of *rules*. Formally $C \subseteq A^* \times A^*$ is a relation. A *rule* is written like $L \to R$. A can be infinite. C can be infinite.

(M is the first letter of mechanism.)

('Reduction' is a bit misnomer since nothing prevents the rule from creating a more complex program. 'Transformation' would be more proper for this. However we usually choose the rules that make the program simpler. Hence the name reduction.)

26.4.1 Expression

An *expression* in M is an element of A^*.

Iff E_1 and E_2 are expressions then $E_1 E_2$ is their *concatenation*.

26.4.2 Direct reduction

For all $(H, L, R, T) \in (A^*)^4$ we write $HLT \to_M HRT$ (read: M *directly reduces* the expression HLT to the expression HRT) iff the rule $L \to R$ is in C.

We write $HLT \rightarrow_M^n HRT$ iff what? M reduces a string HLT to HRT iff what?

a string S is reducible in M iff there exists H, L, R, T such that $S = HLT$ and M reduces HLT to HRT

26.4.3 Ambiguity

M is *ambiguous* iff there exist expressions X, Y, Z such that $X \rightarrow_M^* Y$ and $X \rightarrow_M^* Z$ and $Y \neq Z$.

The following illustrates an ambiguous rewriting system:

$C = \{ab \rightarrow c, b \rightarrow c\}$

$ab \rightarrow_M c$ but also $ab \rightarrow_M ac$

26.4.4 Values and irreducibility

A *value* is an expression that is not reducible.

26.4.5 Evaluation

to evaluate an expression is to repeatedly reduce it until it is a value

26.4.6 Termination

a program fails to terminate (is non-terminating) iff repeatedly reducing it will never give a value

property 1

rule application is associative

which rule is applied first does not matter

example

rule $a\ b$ add $\rightarrow c$ where c is the result of signed 32-bit two's-complement integer addition of a and b

how about $a\ b$ mul $\rightarrow c$?

$$\begin{aligned} 1\ 2\ 3\ \mathsf{mul}\ \mathsf{add} &\rightarrow 1\ 6\ \mathsf{add} && (2\ 3\ \mathsf{mul} \rightarrow 6) \\ &\rightarrow 7 \end{aligned}$$

problem:

inherently static

not dynamic

26.4.7 Relationship with function application

Let $f(L) = R$

$HLT \to HRT$

26.4.8 Concatenation means function composition

Let $E \sim f$ and $F \sim g$. Then $EF \sim g \circ f$.

We mean $(f \circ g)(x) = f(g(x))$.

26.5 Enforcing referential transparency

The problem is that passing object by reference is not referentially transparent. The problem of forcing to pass object by value is frequent deep-copying

Copy-on-write seems to solve the problem or work around the problem quite well until it is bypassed by low-level memory primitives.

27 Procedural semantics

If a is a fragment and b is a fragment then $a\ b$ is the concatenation of a and b. Executing $a\ b$ means executing a first and then executing b. To make two programs be executed in sequence, simply concatenate them.

Procedural semantics does not always hold when the optimiser is used.

$f : A \to B$ means A is popped and then B is pushed.

28
Error handling in C

Errors are everywhere.

28.1 What is an exception?

For our purposes here, we define an *exception condition* as:

> A condition that prevents the completion of the operation that detects it, that cannot be resolved within the local context of the operation, and that must be brought to the attention of the operation's invoker.

The action of bringing the condition to the invoker's attention is called *raising* the exception. The corresponding action by the invoker is called *handling* the exception.

Generally, once an exception condition is raised, it must be handled; otherwise, the program is in error. Some languages provide default actions for conditions that are not handled by the program. [1, p. 445]

28.2 What is an error?

An error can be:

1. a mistake,
2. a situation we do not expect (or expect to not happen),
3. something that is not right,
4. something that is not meant to be.

One possible cause: the assumption of the programmer is violated. The hard thing is the programmer assumes implicitly and he/she himself/herself is not even aware that he/she is making assumption. He/she simply takes it for granted.

An unhandled error is a symptom that the program is wrong or incomplete.

28.3 Possible responses

When an error occurs, execution cannot proceed in the usual way.

Errors are found while the program is running. Can we fix the error without stopping the program?

Should the program be immediately halted, or is there still some hope of performing some corrective action so that the program can continue as if the error had never happened?

There are some possible responses to an error:

1. Abort. This is the response of choice for fatal errors where it is unsafe to continue execution. This response is usually inconvenient to users. A program that aborts easily is not robust.
2. Perform corrective action if possible, and then Retry the operation.
3. Ignore. This is generally not safe and will cause subsequent errors later. Handling an error is *not as simple as silencing that error*. Silencing an error by suppressing all error messages merely creates an illusion of having no error. Silencing is more dangerous than not handling it at all because silencing can hide bugs.
4. Let caller handle error.

In Unix, there are also signals. Some signals are used to indicate errors asynchronously.

The user should be clearly informed. The program should not try to handle the error silently.

We can classify errors into two classes: those that require user intervention and those that don't.

A possible different classification: those that are recoverable and those that are not.

If the program has to check for error every time it calls a subroutine that can err, then soon the program will be cluttered by error-checking conditionals.

If the error happens while the program is holding some resources, then the program might have to release those resources.

For a concrete example let's throw in some code. The following fragment reads a file name from the user, and then tries to open that file and then close it immediately.

```
gets O_RDONLY open close
```

When the requested file does not exist, `open` fails for a reason that should be obvious, assuming that `open` never attempts to create any new file.

Proper error handling is necessary but not sufficient for reliability.

28.4 Practically there will always be errors

If we take the following three statements to be axioms:

1. Programmers practically always have to make simplifying assumptions.
2. Every assumption can always be violated.
3. Violation of such assumption causes error.

then we can conclude that there will practically always be errors.

I tend to believe that it is more practical to assume that error cannot be completely eliminated. As a consequence it would be more consistent for me to try to live with errors than to try eradicating it completely. By living with errors I do not mean to blindly accept those errors as they are, but I mean that we should minimise the severity of the errors so that not all hope is lost whenever there is an error. Anyway, robust in some sense means being able to still work correctly despite errors (to an extent of course); robust does not mean not being able of erring.

One way to make errors somewhat less stressful (or, in a positive tone, more comfortable) is by providing helpful error messages.

But computers don't seem to be willing to cooperate, but instead seem to insist that wrong is wrong.

Defence in depth.

28.5 Possible error handling in C

There are several choices, from the most preferable to the least preferable:

1. non-local exit using setjmp and longjmp in conjunction with error handler stack
2. forcing every function to return a code
3. special return values (discouraged)
4. global errno variable (very discouraged)

28.6 Non-local exits

```
#include        <setjmp.h>

void fun (Plang_Engine* engine) {

        jmp_buf b;

        if (setjmp(b) != 0) {
                /* Exception. */
                return;
        }

        /* Normal. */
        if (error) { /* hypothetical */
                plang_engine_throw (engine, PLANG_ERROR_?);
        }

}
```

A Plang string whose size is sizeof(jmp_buf).

28.7 The Plang_Errno enumeration values

Let us define that a function is *risky* iff it can err. Examples of risky functions are input-output functions. Functions that use memory allocations are risky. Almost all functions in the source code are risky.

In the source code, every risky function returns a value of type `Plang_Errno` (which is actually an integral type) to indicate the kind of error that occurred. This is very much like the `errno` of the C standard library but with return value instead of global variable.

The caller of a function is responsible for handling all errors that might arise when that function is executed. If the caller is not fit to be the responsible one, then the caller of the caller might be. Therefore an error can propagate through the call stack.

Every error must be properly handled or Plang will crash.

An errno `er` indicates an error if and only if `(er & PLANG_ERROR)` is nonzero.

28.8 The ubiquitous CHECK macro for checking errors

Handling errors soon gets annoyingly ubiquitous. Thus `CHECK` was born, for the lack of a better name. You can try `grep`ping for it to see the ubiquity of error-checking. `CHECK` is a preprocessor macro that expands to something like:

```
do {
        if (er & PLANG_ERROR) { goto end; }
} while(0)
```

Consequently if you use it, then you must:

- have a variable whose name is `er` and type is `Plang_Errno`,
- have a label named `end`,
- end the macro invocation with a semi-colon,

or a compilation error occurs. The following is an example usage of that macro.

```
Plang_Errno     er = PLANG_OK;

        ...

/* There are many lines like this in the codebase. */
er = plang_<FUNCTION>(<ARGUMENTS>); CHECK;

        ...

end:
/* You put cleanup codes here as needed. */
return er;
```

28.9 List of constants and descriptions

The complete list of error number constants is in the file include/plang.h. The description of those constants is in the file src/libplang/plang.c

28.10 Giving helpful error messages

An error message should inform the programmer at least about *where* the interpreter was when the error occurred and *what* kind of error occurred. The reported place is not necessarily where the actual error is. It might not be the root cause of the error, but that is better than nothing. It might also suggest the possible causes and the possible corrections to the programmer, but these are hard to do correctly, so we leave it out entirely instead of giving misleading suggestions. The error message format is:

```
<FILE>:<LINE>:<COLUMN>: <KIND><CODE>: <MESSAGE>
<WHERE>
```

where the *kind* is either E for error or W for warning, and the *code* is the error number as a 4-digit hexadecimal number.

`<MESSAGE>` should be concise (that is brief but not obscure).

Example

```
example.p:1:20: E6011: key not found in dictionary (near 'get')
< /a 1 /b 2 > /c get pop "hello" puts
                   ^
```

`plang_describe` takes an error number and returns a pointer to a static constant null-terminated string that is the official description of the error.

28.11 Out-of-memory errors

In my life as a programmer, I find that out-of-memory error is the most difficult error to handle correctly, because it can happen almost anywhere and anytime since memory allocation is almost anywhere and anytime in an interpreter.

29 Input and output

29.1 Introduction

> Input and output are generally found to be among the least satisfactory aspects of a programming language. This is probably because the clean abstract view of the world presented by a programming language must meet the practical compromises of the real world. In the early days of high-level languages, the compiler generated code that interacted directly with the input-output hardware; as a result, the peculiarities of the hardware were reflected directly in the language. [1, p. 263]

Input-output is very important and ubiquitous. It enables data transfer into and out of your program. Every non-trivial program needs it. Basically it is the passing of bytes between your program and its surroundings.

Several input-output subsystems are described in this document. Your choice depends on your needs.

29.2 POSIX low-level input-output

```
PATH    FLAGS                        open           FD
PATH    FLAGS   MODE                 creat          FD
FD                                   close          RET
FD      BUFFER                       read           COUNT
FD      BUFFER                       write          COUNT
FD      OFFSET  DIR                  lseek          POS
```

29.2.1 Constants

```
SSIZE_MAX
```

29.2.2 open

29.2.3 close

29.2.4 read

29.2.5 write

29.2.6 lseek

29.2.7 More information

- GNU C Library Info document (`info libc`).
- The manual pages of individual functions (such as `man 2 open`).

29.3 C unformatted input-output

Most of the functions here simply call the C function with the same name.

getchar	read a byte from standard input stream
putchar	write a byte to standard output stream
gets	read a line from standard input stream and discard the line-feed
puts	write a string to standard output stream and write a line-feed
print	write a string to standard output stream
EOF	the end-of-file marker

Table 29.1: C unformatted input-output functions

When the end-of-file or an error is encountered, getchar returns an integer constant. The constant is the same thing that EOF returns. EOF in C is a preprocessor macro, but EOF in Plang is just a function like getchar and others.

```
                getchar         BYTE
BYTE            putchar         RETVAL
                gets            LINE
STRING          puts            RETVAL
                EOF             THEINTEGER
```

Example:

```
getchar EOF equal
        "end of file"
        "not yet end of file"
ie puts
```

Line terminators gets discards the line terminator whereas puts prints an additional line terminator.

Return values Both putchar and puts do return an integer that is negative iff an error occurs. As far as I know, C programmers usually ignore the return values of those functions, although they might do it unknowingly. However, Plang has no way of ignoring the return value of a function, but if you are not in the mood of checking errors, you can always just pop it like this:

```
65 putchar pop
"hello" puts pop
```

Now you might want to contemplate why you ignored those return values. Ignoring the return value of a function is often a programming error anyway.

29.3.1 No buffer overflow but perhaps out-of-memory

The gets in Plang is somewhat safer than its C counterpart because the gets in Plang is invulnerable to buffer overflow. This invulnerability is due to the way Plang represents strings. Plang always checks the size of a buffer before filling it. The gets in Plang does not use the gets in C. However, it might be possible to mount a denial-of-service attack by passing a line so long that the interpreter uses a large amount of memory so the operating system begins to swap and slow down.

29.3.2 getchar

29.3.3 putchar

29.3.4 gets

29.3.5 puts

29.3.6 EOF

29.4 Reckless input-output

The functions here are called *reckless* because they work without much consideration. They do not care about errors. They are not made with security consideration in mind. They are made solely for convenience. These are usually used by programmers who do not need to care much, like those who are creating prototypes, proof of concepts, throw-away scripts, and such. Never use these in situations where correctness or security is important.

slurp	read file into memory
vomit	write buffer into file

Table 29.2: Reckless input-output functions

29.4.1 slurp

slurp reads an entire file into memory. A bad thing will happen when the file is very large.

29.4.2 vomit

vomit writes the content of a buffer into a file. A bad thing will happen when the storage medium runs out of space.

29.5 Lazy list approach

The lazy list approach is a way of doing *sequential* input-output. A *byte stream* is an infinite list of integers. Reading an element that has not yet been read will cause the program to *block* until that element is available.

There is also the *lazy array* approach that is pretty much the same as the lazy list approach, but admits a more efficient random access.

29.6 Bus input-output

Similar to message-passing.

Interprocess communication.

Ideal for broadcast.

Synchronous notification system.

```
BUSNAME          iobus-open              BUS
BUS              iobus-close
BUS              iobus-write
BUS              iobus-read
BUS              iobus-lock
BUS              iobus-unlock
BUS              iobus-wait
BUSNAME PROC     with-iobus
```

29.7 Channel input-output

The channel input-output subsystem is provides abstract sequential input-output. It is a specialisation of the bus input-output where every bus is restricted to exactly two parties. A channel is a dedicated bidirectional line of communication.

```
PATH                    ioch-open-file          CHANNEL
SIZE                    ioch-open-memory        CHANNEL
HOST PORT               ioch-open-tcp           CHANNEL
CHANNEL                 ioch-can-read           BOOLEAN
CHANNEL                 ioch-can-write          BOOLEAN
CHANNEL                 ioch-read               STATUS
CHANNEL                 ioch-write              STATUS
CHANFROM CHANTO         ioch-pass               BYTE
CHANFROM CHANTO         ioch-passthrough        STATUS
CHANNEL                 ioch-wait
CHANNEL                 ioch-read-buffer        BUFFER
CHANNEL                 ioch-write-buffer       BUFFER
```

```
IOCH-CLOSED
IOCH-AGAIN
IOCH-DONE
IOCH-NOT-READY
```

29.7.1 Memory input-output channel

29.8 Reading and Writing Objects

Every object which can be marshalled and then demarshalled can be written and then read.

29.9 Concepts

29.10 Internal and External Representation

A token has an internal representation and an external representation. The external representation is a string.

29.11 Standard Streams

A stream of token. Serial or sequential input, one follows another in an orderly fashion.

When you are using the interpreter interactively, the standard input is whatever you type into the interpreter, and the standard output is whatever you see on screen. You can see that as you type, the characters you typed are *echoed* to the standard output.

Somehow we use terminology which gives the impression of our program's being literate, that is able to read and write.

`read` reads one token from standard input.

`read` reads the external representation of one token from standard input and converts it into an internal representation.

`write` writes one token to standard output.

`write` computes the external representation of a token and writes it to the standard output.

30
Pairs

A pair is two things.

A list is made of pairs, as in Lisp. Thus a list is just a special case of pairs. You can think like this:

$$[1\ 2\ 3] \sim (1, (2, (3, \text{nil})))$$

Most pair and list functions are stolen from Lisp and Haskell, so the style is rather functional.

```
FIRST SECOND    pair            PAIR
PAIR            unpair          FIRST SECOND
PAIR            first           FIRST
PAIR            second          SECOND

LIST            head            HEAD
LIST            tail            TAIL
```

Difference between tail and second tail happily accepts nulls whereas second barfs on them.

30.1 nil: the empty list

`nil` is the null pointer. It also acts as the empty list. Since there's one and only one null pointer, there is one and only one empty list. It's not *a* null pointer; it's *the* null pointer. It's not *an* empty list; it's *the* empty list. Well, at least conceptually. If that bothers you, just drop the articles altogether: it's null pointer, it's empty list.

30.2 Versatility of pairs

If time is not constrained then pair can be any data structure.

Every C struct can be represented as a pair but somehow we still create C structs. The C struct represents a *layout*: how to group a bunch of bytes in memory. The Plang pair represents an aggreggation: a mere grouping of several things.

31 Dictionaries

This describes how Plang maps strings to objects.

31.1 Interface

A dictionary maps keys to values. A key is a string. In the following, D is a dictionary, K is a key, V is a value, V is the old value if there's any or null otherwise.

```
        K       V       def
D       K               get             V
D       K       V       put             V'

< K1 V1 ... Kn Vn >     -->     D
```

The following example creates a new dictionary that maps the string 'a' to the integer 1:

```
< /a 1 >
```

You can also write:

```
< "a" 1 >
```

You can make a dictionary with angle brackets (< and >) like this:

```
< KEY1 VAL1 KEY2 VAL2 ... KEYN VALN >
```

The following creates a new empty dictionary.

```
< >
```

The following creates a dictionary with two entries. This dictionary maps 'a' to 1 and 'b' to 2.

```
< /a 1 /b 2 >
```

31.2 Current implementation

Plang uses a 256-way prefix tree to map a string to an object. One reason for 256 is that the size of a byte is 8 bits (therefore a byte can have $2^8 = 256$

possible combinations).

The source code is entirely contained in src/libplang/dictionary.c.

Each node of an n-way tree can have up to n children, but not more.

In a prefix tree, two strings that have a common prefix will share a common path from the root to somewhere in the tree. The depth of this shared path is the length of the longest common prefix. The dictionary is most efficient when it contains many keys that share common prefixes, such as $\{aaa, aab, aac\}$ and least efficient when it contains keys whose longest common prefix is empty, such as $\{aaa, baa, caa\}$. (The string p is a common prefix of the string a and the string b if and only if there exist a' and b' such that $a = pa'$ and $b = pb'$.)

The depth of the tree is the length of the longest key.

31.2.1 Time complexity

The time for finding a key is proportional to the length of the key. Finding a key of length n in the tree requires $\Theta(n)$ time.

31.2.2 Space complexity

31.3 What the heck are these?

31.3.1 Association arrays

This is even simpler to code. The author had to work under great deadline pressure, so he had to choose what could be coded pretty damn quickly instead of what would have been ideal.

Once we have a hash table, a hash dictionary is very easy to implement that we would always use a hash dictionary instead of hash table.

The first implementation of hash tables is pretty damn quick. Growth, shrinkage, and collision were not considered. When there is collision, the hash table simply returns an error.

Claustrophobicity. Also known as *one minus load factor*. If the hash table is this full, then it enlarges itself. *Claustrophilicity*. If the hash table is this empty, then it shrinks itself. *Statistics*. With the assumption that addition and removal are random, the hash table uses statistics to feel when to grow and when to shrink.

Hash tables are good if the entries can be hashed quickly, the number of entries is small, the dictionary never grows too much, ordering is not required.

Iterator performance for sparse large tables can be improved by *skip lists*.

31.3.2 Association lists

An *association list* is a list of key-value pairs. It is $L = [(k_1, v_1), \ldots, (k_n, v_n)]$. A key-value pair is (k, v) where k is the *key* and v is the *value*. A linear search is performed whenever someone has to do something with the dictionary. Let n be the size of the association list. Worst-case time complexity of searching is in $\Theta(n)$. Time complexity of putting an entry is always in $\Theta(n)$ if you care about ensuring that there are no duplicate entries.

Performance can be improved somewhat if the access is predictable. If a pair is accessed, move it to the front of the list.

31.3.3 Association arrays

An association array is the same as an association list but uses fixed-size arrays instead of linked lists.

The same move-to-front can also be done to association arrays.

31.3.4 Hash tables

A hash table is backed by an array. The size of that array is the *capacity* of the hash table. Every element of the array is called a *bucket*. A bucket is a pointer to an object. If that pointer is null, then the bucket is said to be *empty*. Otherwise it is said to be *filled*. The *fullness* of a hash table is its number of filled buckets divided by its capacity.

A hash function here is a function that takes an object and gives a hash. The hash here is treated as an unsigned 32-bit integer. Let $h(x)$ be the hash of the object x. Let c be the capacity of the table. The index of x in the array will be $h(x) \bmod c$. You can also call that the *bucket number* of x. The hash function can involve additions, multiplications, bit shifting, exclusive disjunctions, anything that can be computed quickly. where x is the address of the object. The most significant bits of the address usually do not vary much. The address is treated as an unsigned 32-bit integer. Address is usually aligned (divisible by a power of two). In that case, several least significant bits of the address will be zeroes. The capacity of the table should be relatively prime to the smallest typical alignment.

Since the hash function is a function after all, it has to satisfy

$$x = y \implies h(x) = h(y) \tag{31.1}$$

but it does not have to satisfy the converse. When it does not satisfy the converse of the above implication, a *hash collision* occurs. According to the pigeon hole principle, hash collision is unavoidable if the domain is larger than the codomain.

Using memory address as hash If an object and another have equal addresses, then they are equal. The converse does not hold when the equality between two objects can be redefined. Using memory as hash is simple but can be incorrect.

After all hash tables are not so easy to make.

Performance measurement. Benchmarking. Testing. If the output looks clearly correlated with the input, then the hash function is bad.

We use *closed hashing*. On a hash collision, the table tries to put an object in the next bucket.

```
typedef struct _Plang_Hash_Table_Entry
{
        void* object;
}
Plang_Hash_Table_Entry;

typedef struct _Plang_Hash_Table
{
        Plang_Object _;
        Plang_Hash_Table_Entry* array;
        size_t capacity;
        size_t size;
        /* Profiling information. */
        size_t num_collisions;
        size_t num_rebuildings;
}
Plang_Hash_Table;

Plang_Errno plang_hash_table_new (size_t capacity);
```

31.4 Dictionary stack

```
current-dictionary      CURDIC
```

A dictionary stack allows local scoped definitions. It is currently not implemented.

plang_hash_table_new
plang_hash_table_put
plang_hash_table_get
plang_hash_table_ensure_capacity
plang_hash_table_set_capacity

Table 31.1: Hash table functions

31.5 Other usages

31.5.1 Using a dictionary as a set

A dictionary can be used as a set without requiring any modification. Because the dictionary is backed by a tree, you get a set whose elements are automatically ordered according to their natural ordering. The worst case time complexity of determining whether a thing is in a tree is in $\Theta(d)$ where d is the depth of the tree.

Let S be a set that is backed by the dictionary D. To put an element e into S, map the key e in D to any arbitrary value (for example, a null value). To check whether e is in S, check whether the key e exists in D. To remove an element e from S, remove the key e from D.

```
LIST            list-to-set     SET
                empty-set       EMPTYSET
SET     ELEM    set-add
SET     ELEM    set-remove

-- functional flavour of set-add; create new set

SET     ELEM    set-with        SET2

-- functional flavour of set-remove; create new set

SET     ELEM    set-without     SET2
SET             set-clear       SET2
SET             set-is-empty    EMPTY
SET             set-size        SIZE

/set-add { nil put } def
/set-remove { dictionary-remove } def
/set-is-empty { dictionary-is-empty } def
/set-size { keys length } def
```

31.5.2 Using a dictionary as a bag

Pretty much the same as the previous section, but here the value is an integer instead of always null. To put an element e into the set S, increment the value whose key is e in D. If the mapping does not already exist, the value is set to 1. To remove an element e from S, decrement the value whose key is e in D. As soon as the value reaches zero, its key is removed from D.

31.6 Considerations

31.6.1 Possible backing data structures

I know three ways of making a dictionary. From the one I prefer most to the one I prefer least, they are:

1. prefix trees,
2. association lists,
3. hash tables.

There are some data structures which can form a dictionary, such as association lists, hash tables, and prefix trees. Association list is simple and can use unordered keys but its performance becomes unacceptable as the dictionary grows. Hash table offers amortised constant time look-up but requires a hash function and scrambles the keys. It is recommended that an implementation uses a tree to implement the dictionary because tree has a predictable logarithmic time complexity and allows ordered traversal of keys. However, tree requires that the keys be totally ordered.

- Is creation of new dictionaries frequent? Yes. Fragments may create a dictionary for the purpose of storing temporary variables, and these fragments may be recursive.
- Is disposal of old dictionaries frequent? Yes. There may be many short-lived recursive fragments.
- Is insertion frequent? The dictionary may need to be resized.
- Is deletion frequent?
- How many entries are there in a usual dictionary? Dictionaries with tens of entries are usual for storing temporary variables. The root dictionary may contain hundreds of entries. Dictionaries with thousands of entries may arise when Plang is used for general data exchange.
- The running code should never have to care about how a dictionary is implemented.
- Engineering prefers predictability to best-case performance. Engineering prefers a unamortised logarithmic time complexity rather than an amortised constant time complexity.

31.6.2 Predictable performance degradation

Trees offer a more predictable performance degradation than hash tables do as dictionaries grow. Using a tree, the program slows down steadily and memory consumption rises steadily. Using a hash table, the program pauses as the hash table is resized, and pauses again as the hash table is resized again.

31.6.3 Ordering of keys in tree-based dictionaries

31.6.4 Simplicity

The mathematical analysis of trees is simpler than that of hash tables. In particular, the analysis of trees does not involve amortisation and probability.

Trees also degrade gracefully. It slows down gradually, unlike hash tables that might need resizing that occurs rarely but has large penalty. In short, trees are simpler, and thus is preferable. In engineering, predictability is more desirable than arbitrariness.

Moreover, in order to utilise hash table well, one has to think of a good hash function. Usually the hash function is computed modulo the capacity of the table, and modular arithmetic suggests that the capacity of the table be a prime number.

If we can just choose any mediocre hash function, hash tables are easier to implement than trees. Hash table iterators are also easier to implement than tree iterators.

Association lists Association lists are even simpler although its performance is unacceptable for large dictionaries.

31.6.5 Sparsity

A disadvantage of using trees is that the dictionary becomes quite sparse. In typical situations, most children are null, and thus sparse dictionaries waste a large amount of space, although this is hidden by the large amount of memory of today's machines.

One tree node takes up 256 machine words plus a little, regardless of how full it is, regardless of how many children it actually has. If you have a dictionary that contains only one key that is 16 bytes long, then the dictionary takes 256 machine words plus some more, of which about 16 are really used and the rest is zeroed.

Sparsity can be mitigated by reducing the branching factor, but a smaller branching factor causes a greater number of pointer accesses, and performs allocations in smaller chunks. This can give the memory manager a hard time. This can make life hard for the memory manager of the operating system and the cache controller and memory management unit of the processor.

31.7 Mathematical treatment

A dictionary D is a set of ordered pairs with the following restrictions:

$$D \subseteq K \times V \tag{31.2}$$

$$\forall k_1 \, \forall k_2 \, [k_1 = k_2 \wedge (k_1, v) \in D \implies (k_1, v_1) = (k_2, v_2)] \tag{31.3}$$

where K is the set of all byte strings and V is the set of all Plang objects. (31.3) is the restriction that the same key cannot be simultaneously paired with two different values.

Usually a dictionary is finite.

We say that the dictionary D *contains* the key k iff

$$\exists v \;\; (k, v) \in D. \tag{31.4}$$

We write $D[k] = v$ iff $(k, v) \in D$.

$$\begin{aligned} \mathrm{Get}(D, k) &= v \text{ iff } (k, v) \in D\text{; Undefined Key Error otherwise} \\ \mathrm{Put}(D, k, v) &= D - \{(k, w) | w \in V\} \cup \{(k, v)\} \\ \mathrm{Remove}(D, k) &= D - \{(k, v) | v \in V\} \end{aligned}$$

where K is the set of keys, V is the set of values, In other words, $D[k]$ means ‘the value associated with the key k in dictionary D’.

32 Object system

32.1 History and evolution of the object system

Before the advent of the object system every item in the stack was simply an `int`. There was no type checking at all. All sorts of things were just cast to `int` and pushed onto the stack. As a result the interpreter would faithfully crash when instructed to go wild like adding pointer and integer and then dereferencing the result as pointer to byte.

At first the object system had one and only one purpose: to prevent crash that was caused by the obvious blunder above. The simplest thing I knew of was *tagging* which meant that every item on the stack came with a number indicating its type. Which number meant which type was `#define`d in a header file. Therefore before adding two objects, the interpreter would make sure first that both objects are integers by checking the tags. It was not worthy enough to be called an object system.

The crash did not go away. The interpreter still crashed due to memory management issues like double-freeing and buffer-overrunning which then dominated the crash scene. On other occassions whenever it did not crash it leaked memory. By then it was apparent that the object system had to deal with memory management. Thus as time goes by the object system also began to provide memory management. I chose *reference counting* because it was simple enough for me as I was not technically capable of implementing garbage collection back then.

Then Plang briefly imitated Ruby in the sense that every instance had its own method table. There were about ten function pointers in one such table and the table kept growing. Ten function pointers occupied 40 bytes on an x86 so an integer that usually took 4 bytes then took 44 bytes instead. Most of those bytes were indeed zeroes which meant that the methods were not implemented and space was wasted.

I felt that it was too wasteful so I moved the entire method table was into a *type descriptor* structure. The method table in each instance was replaced by a pointer to that type descriptor. In this way every object of the same type had the same method table. Now the overhead of an object becomes only one machine word plus one dereferencing of pointer. Thus on an x86 an integer that usually took 4 bytes then took 8 bytes when wrapped as a Plang object. Then Plang became something like a primordial Java.

Not all news was good though for type-checking was not free lunch. The source code becomes cluttered by error checking codes made of conditional gotos. The other bad news was even a simple operation like adding two integers involved five memory-management-related calls.

32.2 Anatomy of an object

```
typedef struct {

        const char*     name;
        size_t          instance_size;

        /* method table */

        void (*del) (void*);
        /* ... */

} Plang_Type;

typedef struct {

        Plang_Type*     type;

        /* instance variables */

        int             refcnt;

} Plang_Object;
```

Inheritance is quite limited. Typically the inheritance tree consists of only two levels, and there are only a few classes. By convention, the root class is Object, although the object system itself does not have the concept of classes, let alone root class. Most other classes are children of Object, and might not always be designed to be subclassed.

32.3 Allocators, initialisers, and destructors

Every object has an *allocator*, an *initialiser*, and a *destructor*.

An allocator is a function named like `plang_X_new`. It is responsible for allocating memory. Typically it uses `malloc`, but nothing prevents you from creating your own allocator, such as a pool allocator.

An initialiser is a function named like `plang_X_init`. It is responsible

plang_object_new
plang_object_delete
plang_object_freeze
plang_object_frozen
plang_object_use
plang_object_release
plang_object_copy_shallow
plang_object_copy_deep
plang_object_equal
plang_object_hash

Table 32.1: Object system functions

for filling the fields of the object with default values. The initialiser is also used for unmarshalling as it is supposed to know how to interpret a series of bytes that is the result of marshalling an object of that kind.

The destructor is a function named like `plang_X_del` and releases resources used by the object, but not the object itself. However, `plang_obj_del` does free the memory of the deleted object, and the object system assumes that this function is the only function that is allowed to free an object.

Those functions must have signatures like these:

```
Plang_Errno     plang_X_new (void** result);
Plang_Errno     plang_X_init (void* object, void* marshal);
Plang_Errno     plang_X_del (void* object);
```

Note that `marshal` can be null. If it is so, then the initialiser is not unmarshalling, but creating a new object.

32.4 Equality comparison

Equality can be redefined. The default equality comparison procedure is to compare the memory addresses.

```
Plang_Errno plang_object_equal(void* a, void* b, bool* eq);
```

32.5 Object space

The object space is a dictionary of all live objects. A key is an object address. A value is a Plang object. By default the object space is disabled. The object space incurs performance penalty (both time and space) because it causes the interpreter to intercept every object creation and destruction. Since objects are created every now and then, penalty could be high. The object space can be arbitrarily enabled and disabled at runtime. If the object space is not enabled, objects will return an empty dictionary. The object space is intended for debugging, not for production. If you enable the object space, get the dictionary, and disable the object space, you will get a filled dictionary.

```
                object-space-enabled          BOOL
BOOL            object-space-enabled!
                objects                       SPACE
```

Example The following fragment approximately counts the number of live objects if the object space is enabled.

```
objects length
```

33 The functional style

Function (in the mathematical sense) is the central theme of functional programming. (Function application or composition?) Some examples of functional languages I have ever touched are ML, Haskell, Lisp, and Scheme. Lisp and Scheme are impure functional languages. Haskell is a pure lazy functional language.

The program

```
f
```

can be thought as applying the function f to the stack.

Plang itself can be classified as a lazy impure functional language. Impure functional languages allow mixing the usual procedural style.

Plang has something corresponding to lambda expressions in most functional languages, but in Plang the parametres are implicit.

Higher-order functions. Dyadic (two-argument) functions.

map zip fold/accumulate/compress/inject compose (the name ‘fold’ is preferred)

name fetch : executable

function

/f /g compose –> g f

```
1 [1 2 3 4 5] [mul] fold
0 [1 2 3 4 5] [add] fold
```

Lisp	Plang
(lambda (x) (+ x 1))	[1 add]
(lambda (x) (+ 1 x))	[1 exch add]

Table 33.1: Some Lisp lambda-expressions and their equivalent Plang fragments

fold actually takes an iterable, so it can take things more general than a list as long as it is iterable.

nat1 the first natural numbers; counting starts from 1. nat0 is similar but counting starts from 0 instead. They don't form a list; constant-space.

Naïve. Unguarded argument. Negative or nonintegral argument causes infinite loop.

33.1 Frustratingly simple examples

Factorial and Fibonacci.

```
"factorial" [
    0 eq 1 [ dup 1 sub factorial mul ] ie
] def
```

Factorial can be defined as a tail-recursive function.

When you program, you should seek for beauty, simplicity, and straightforwardness of translation.

Factorial can be concisely defined. The Plang version is a rather straightforward translation of the mathematical definition. This is very desirable. This specifies *what*, now *how*. Somewhat declarative, isn't it?

The mathematical definition.

$$\begin{aligned} 0! &:= 1 \\ n! &:= \prod_{k=1}^{n} k = 1 \times 2 \times 3 \times \ldots \times n \\ &= \prod_{k=n}^{1} k \end{aligned}$$

The Plang version.

```
"factorial" [nat1 prod] def
"factorial" [1 range prod] def
```

That reads

`def`ine `factorial` as

the `prod`uct of `nat`ural numbers starting from `1`

Very dense, good, concise, isn't it? All unnecessary details are left out, aren't they?

33.2 ?

ui> 5 factorial 120

Syntactically (although this isn't what happens in the interpreter, the effect is equivalent), '5 factorial' gets expanded to '5 nat1 /mul fold'. Then '5 nat1' becomes '-an iterable- /mul fold', which is equivalent to '(1 2 3 4 5) /mul fold', which is equivalent to '1 2 mul 3 mul 4 mul 5 mul', so the result is 120.

33.3 Another example

arg = exch def

```
####### /sorted-insert {
    /ordered? arg # comparator
    /list arg
    /new arg
    new list head ordered?
        { new list cons }
        { new list tail { ordered? } sorted-insert }
    if
} def #######

/bisect {
...
} def

/merge {
...
} def

/sort {
    /list arg
    list empty?
        { list }
        { list bisect sort exch sort exch merge }
    ife
} def
```

Off-the-shelf sorting algorithms widely circulated on the Internet.

34 Memory management

34.1 Reference counting

Every object has a reference count. The initial value of the count is *one*. The caller of the constructor owns the initial reference. Reference-counting is not thread safe. Every object shall be used only by one thread.

34.2 Mark-and-sweep concurrent generational garbage collection

Objects in programming environments exhibit high infant mortality. Recent objects tend to be short-lived. (cite?)

Iff an object is *unreachable* then that object can be garbage-collected.

If the memory is exhausted then full garbage collection is performed. If after full garbage collection the memory is still exhausted then an out-of-memory error occurs. Full garbage collection might exhibit the *embarrassing pause*. The garbage collector also needs space to operate. What if the garbage collector itself runs out of memory? Who watch the watchers? Full GC might worsen the memory exhaustion.

On exit the garbage collector frees all objects in an unspecified order.

Every time an object is created it is also registered to the garbage collector.

34.3 Implementation costs

The addition of these variables to the `Plang_Object` structure:

```
        /*
         * The collector assumes that these variables
         * exclusively belong to it.
         * If you are not the collector
         * then do not ever touch these.
         */

        int             mark;           /* mark-and-sweep GC */
        int             generation;     /* generational GC */
```

The addition of this method to every type:

```
void*   gc_fringe (void*, int);
```

The addition of a mutex lock:

The requirement for proper synchronisation:

35 Serialisation and persistence

Serialisation means transforming objects to a string that contains enough information so that the string can losslessly be transformed back to the objects anywhere anywhen without needing any other information.

An object may reference other objects. References can be cyclic.

Marshalling means writing objects to a *stream*. Unmarshalling means reading them in the same order they were marshalled. If the object is large then this might take a long time. Sequential input-output is used.

The stream is byte-order-agnostic.

Deduplication.

Typically a marshaller recursively calls the marshaller for every field.

Unmarshalling requires a *type system*. The type system has a dictionary that maps names to type descriptors. Types must be registered first before they can be used.

Requirement of maintenance of reference: Let there be two objects A and B. Iff A and B have the same address then A and B must have the same UID.

35.1 Stream header

```
MAGIC VERSION
```

`MAGIC` is the fixed string `Plang-Marshal-Stream`. `VERSION` is a string denoting the version of Plang which created this stream.

```
Plang-Marshal-Stream 0a 2010-11-17
```

35.2 Object

```
ID TYPE VERSION CONTENT
```

`ID`, `TLEN`, and `VERSION` are variable-length integers. `ID` only needs to be unique in the same stream. `ID` must start from zero and must be consecutive. `TYPE` is a string. If type is empty, then this entry is a reference to an earlier object. `VERSION` is a variable-length integer for backward combatability. `CONTENT` is object-specific information which is understood only by the unmarshaller for this type.

Example:

```
00 07 Integer 01 23 45 67
01 07 Integer 89 ab cd ef
02 06 String 05 Hello
```

Example:

```
00 04 Pair 01 02
01 07 Integer 00 00 00 01
02 07 Integer 00 00 00 02
03 06 Single 3f 80 00 00
04 0a Dictionary 03 09 first-key 05 0a second-key 06 09 third-key 07
05 07 Integer 00 00 00 01
06 07 Integer 00 00 00 02
07 07 Integer 00 00 00 03
08 07 Integer 00 00 00 04
```

Example: marshalling the same object three times:

```
00 07 Integer 01 23 45 67
01 00 00
02 06 Single 3f 80 00 00
03 00 00
04 00 02
05 00 02
```

It is an error for a reference to refer to another reference.

35.3 Encoding of strings

```
LENGTH CONTENT
```

`LENGTH` is a variable-length integer which is the exact length of `CONTENT`▮

35.4 Decoding of variable-length integers

1. Initialise S to 0.
2. Read 8 bits into I.
3. $S \leftarrow ((S << 7) \vee (I \wedge 127))$.
4. If $I \wedge 128 \neq 0$ then go to step 2.

```
usual                   VLI
01000000                01000000
10000000                10000001 00000000
10000000 00000000       10000010 10000000 00000000
```

35.5 Constructing a stream

Stream remembers the last object identifier and also maps identifiers to live objects (their addresses maybe?). Pointer swizzling.

35.6 Reading a stream

Pointer unswizzling. An object is allocated but not initialised. The unmarshaller initialises it.

35.7 Unmarshallable error

Some objects cannot be marshalled.

35.8 Invalid stream error

Part VII

Appendices

A
Russell's paradox

Set theory used to be founded on first-order logic. We say that something is in a set iff that something satisfies a predicate.

$$A = \{x|p(x)\} \iff (\forall x \;\; x \in A \iff p(x)) \tag{A.1}$$

Nothing prevents us from theoretically asserting that there exists A such that $A = \{x|x \notin x\}$ although practically we never write that. The consequence of that assertion is

$$\begin{aligned}
\forall x \;\; (x \in A &\iff p(x)) && \text{definition (A.1)} && \text{(A.2)}\\
\forall x \;\; (x \in A &\iff x \notin x) && \text{since } p(x) = x \notin x \text{ by definition (A.1)} && \text{(A.3)}\\
A \in A &\iff A \notin A && \text{instantiation by setting } x = A && \text{(A.4)}\\
p(A) &\iff \neg p(A) && && \text{(A.5)}\\
&\text{false} && && \text{(A.6)}
\end{aligned}$$

Therefore, we can deduce that the axioms of set theory lead to falsity. Yet with this paradox, set theory still worked so well. It is simple, intuitive, practical, ubiquitous. Set theory worked because people did not take $A = \{x|x \notin x\}$ seriously, at least until Bertrand Russell.

What if we directly confront the paradox by adding the following axiom to set theory? This is more like prohibiting the mathematician rather than fixing the theory, if anything needs fixing at all.

$$\neg\exists A \;\; A = \{x|x \notin x\} \tag{A.7}$$

It is akin to the following dialogue between two computer programmers:

A: Hey, I fed this input to your program and it crashed.

B: So don't do that.

B

Integer division

The existence of a dedicated chapter for integer division indicates that it can be quite problematic.

B.1 The philosophy of integer division

Suppose that I have exactly n pebbles and I want to as evenly as possible give away[1] all those pebbles to exactly d persons. I do this by giving a pebble to person 1, then giving another pebble to person 2, and then giving another pebble to person 3, and so on until person d, and then shouting, "Round!" After shouting I repeat the whole of the previous paragraph all over again until I run out of pebbles without taking back what I have already given them.

Suppose that I happen to end this activity in silence, that is without a shout. It means there are integer p and integer x such that each of the persons from 1 to p has $x + 1$ pebbles and each of the persons from $p + 1$ to n has x pebbles, hence at least one person has gotten less pebbles than someone else and therefore I am not giving the pebbles evenly, and thus in such case of unevenness I take back one pebble from each persons from p to 1 so that each person involved in this activity has x pebbles.

When I run out of pebbles, how many times I shouted "Round!" is the *quotient* and the number of pebbles I took back is the *remainder*. However the above story assumes that all numbers are positive integers.

The division of the dividend n by the divisor d gives the quotient q and the remainder r. Define integer division as follows for all $n, d, q, r \in \mathbb{N}$, $d \neq 0$:

$$n : d = (q, r) \iff \left(\left(n - \underbrace{d + \ldots + d}_{q} = r \right) \land (0 \leq r < d) \right) \tag{B.1}$$

[1]By *give away* I mean that I am not a recipient myself.

B.2 Integer division with negative arguments

This one section exists to answer the question "What is the mathematically correct result of integer division when at least one operand is negative?"

B.2.1 Several ways of extending

When we extend the operands from $\mathbb{N}$ to $\mathbb{Z}$ there are several options:

1. Let it be undefined. This is the choice of ANSI C?
2. Round the quotient toward zero. This is the choice of GCC.
3. Force the modulus to be positive by changing $(0 \leq r < d)$ to $(0 \leq r < |d|)$.
4. Change $(0 \leq r < d)$ to $(0 \leq |r| < |d|)$.

Let us extend both n and d from $\mathbb{N}$ to $\mathbb{Z}$. Observe the pattern of the table. Let $D(n,d) = (q,r)$. We can generalise the pattern in the table to: The sign of the remainder is the sign of the divisor.

$$D(n,d) = (q,r) \iff D(-n,-d) = (q,-r) \tag{B.2}$$

dividend	forced positive remainder		round quotient toward zero	
-3	$-1 \times 3 + 0$	$+1 \times -3 - 0$	$-1 \times 3 - 0$	$+1 \times -3 - 0$
-2	$-1 \times 3 + 1$	$-1 \times -3 - 1$	$-0 \times 3 - 2$	$+0 \times -3 - 2$
-1	$-1 \times 3 + 2$	$-1 \times -3 - 2$	$-0 \times 3 - 1$	$+0 \times -3 - 1$
$+0$	$+0 \times 3 + 0$	$-0 \times -3 - 0$	$+0 \times 3 + 0$	$-0 \times -3 + 0$
$+1$	$+0 \times 3 + 1$	$-1 \times -3 - 2$	$+0 \times 3 + 1$	$-0 \times -3 + 1$
$+2$	$+0 \times 3 + 2$	$-1 \times -3 - 1$	$+0 \times 3 + 2$	$-0 \times -3 + 2$
$+3$	$+1 \times 3 + 0$	$-1 \times -3 - 0$	$+1 \times 3 + 0$	$-1 \times -3 + 0$

(The sign of zero is only my guess. Zero is of course neither positive nor negative.)

Table B.1: Some integer divisions

The following C program was used to determine the behaviour of GCC as written in the documentation of glibc (section 'Integer Arithmetic').

```
#include        <stdio.h>

int main (void)
{
        int n;
        int d;
        int q;
        int r;
        d = -3;
        for (n = -3; n <= 3; ++n)
        {
                q = n / d;
                r = n % d;
                (void) printf("%2d = %2d x %2d + %2d\n", n, q, d, r);
        }
        d = 3;
        for (n = -3; n <= 3; ++n)
        {
                q = n / d;
                r = n % d;
                (void) printf("%2d = %2d x %2d + %2d\n", n, q, d, r);
        }
        return 0;
}
```

B.2.2 Algorithm? What algorithm?

To compile the above program, save it into a file named a.c and:

```
cc -O0 -Wall -ansi -pedantic -o a a.c
```

```
Input: the dividend $n \in \mathbb{N}$ and the divisor $d \in \mathbb{N}$ where $d \neq 0$
Data: the quotient $q$ and the remainder $r$
Output: the result $(q, r)$
$q \leftarrow 0$
$r \leftarrow n$
while $r \geq d$ do
    $r \leftarrow r - d$
    $q \leftarrow q + 1$
```

Algorithm 2: Repeated-subtraction integer division

The following algorithm performs repeated subtraction to divide n by d. The loop invariant is

$$qd + r = n \tag{B.3}$$

Those quantities are defined to satisfy the following constraints:

$$n, q, d, r \in \mathbb{Z} \tag{B.4}$$

$$d \neq 0 \tag{B.5}$$

$$n = qd + r \tag{B.6}$$

$$0 \leq r < d \tag{B.7}$$

If any of those constraints is not satisfiable then we say that the result of the division is undefined.

B.3 Handling division errors

Division by zero. Division overflow (dividing INT_MIN by -1 on a two's-complement machine whose quotient is as wide as the dividend).

IEEE 754 arithmetic returns special value: positive infinity, negative infinity, or not-a-number, depending on the operands.

x86 raises an exception. In my Linux box that exception is caught by the kernel and sends a SIGFPE to the user program. The default action of that signal is to abort the program.

In Java and Ruby a runtime exception is thrown. The exception is an ordinary exception that can be caught and handled in the ordinary way.

Scheme has the fraction data type.

Bibliography

[1] Michael Marcotty and Henry F. Ledgard. *Programming language landscape*. Science Research Associates Inc., second edition, 1985.

[2] M. von Thun. A short overview of Joy. Available from the author, 1994.

Index