



Exporting Go

Robert Griesemer

GopherCon Singapore, 2017

Intro

- Go package
 - Namespace
 - Interface (export)
 - Import
- Implementation
 - **Export/import** (this talk)
 - Linker (not this talk)

A long, long time ago, somewhere in New Jersey...

Dennis Ritchie

Ken Thompson



PDP-11

Credits: <https://www.bell-labs.com/usr/dmr/www/picture.html>

An example

```
1 == 1
2 == 2 prime
3 == 3 prime
4 == 2^2
5 == 5 prime
6 == 2 * 3
7 == 7 prime
8 == 2^3
9 == 3^2
10 == 2 * 5
11 == 11 prime
12 == 2^2 * 3
...
996 == 2^2 * 3 * 83
997 == 997 prime
998 == 2 * 499
999 == 3^3 * 37
1000 == 2^3 * 5^3
```

Go

```
package main
import "fmt"

type List struct {
    Factor, Power int
    Link          *List
}

func Factor(x int) *List { ... } // returns the list of prime factors of x
func Print(l *List) { ... }      // prints the given list of prime factors

func main() {
    for i := 1; i <= 1000; i++ {
        fmt.Printf("%4d == ", i)
        Print(Factor(i))
        fmt.Printf("\n")
    }
}
```

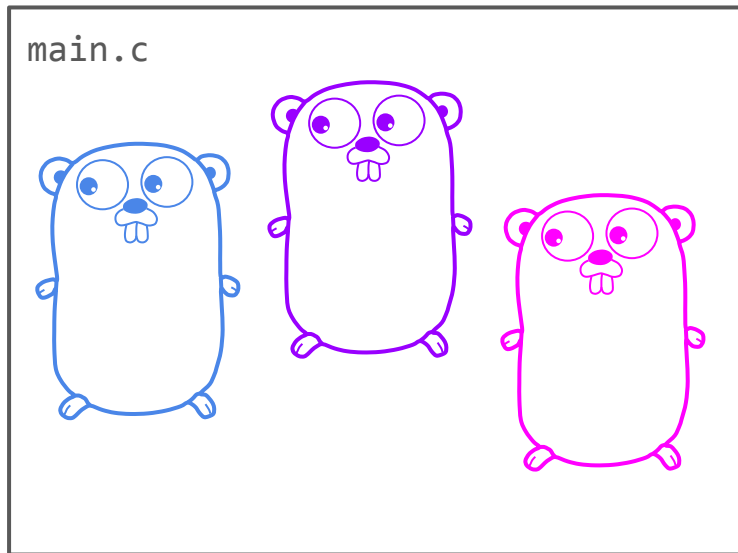
C

```
#include <stdio.h>
#include <stdlib.h>

struct List {
    int factor, power;
    struct List* link;
};

struct List* Factor(int x) { ... }
void Print(struct List* l) { ... }

int main() {
    for (int i = 1; i <= 1000; i++) {
        printf("%4d == ", i);
        Print(Factor(i));
        printf("\n");
    }
    return 0;
}
```



(To be accurate, 1970's C didn't permit inline declaration of `i` in the for-loop header. But it doesn't matter for the purpose of this illustration.)

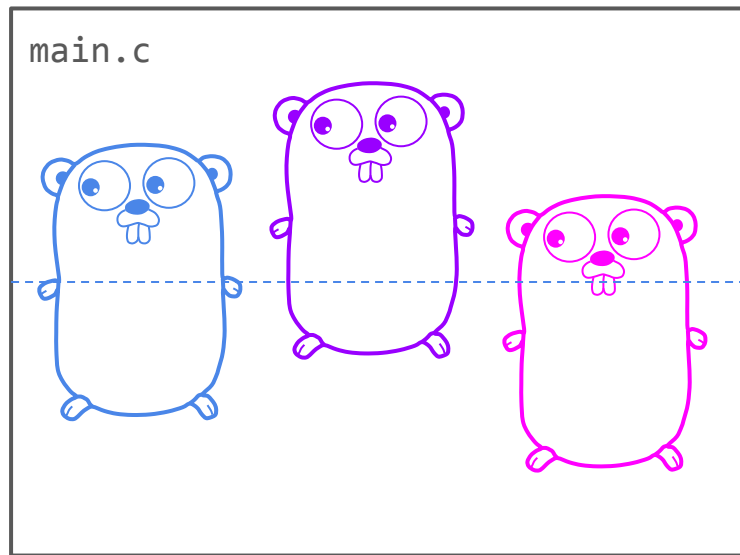
C

```
#include <stdio.h>
#include <stdlib.h>

struct List {
    int factor, power;
    struct List* link;
};

struct List* Factor(int x) { ... }
void Print(struct List* l) { ... }

int main() {
    for (int i = 1; i <= 1000; i++) {
        printf("%4d == ", i);
        Print(Factor(i));
        printf("\n");
    }
    return 0;
}
```



Using a library

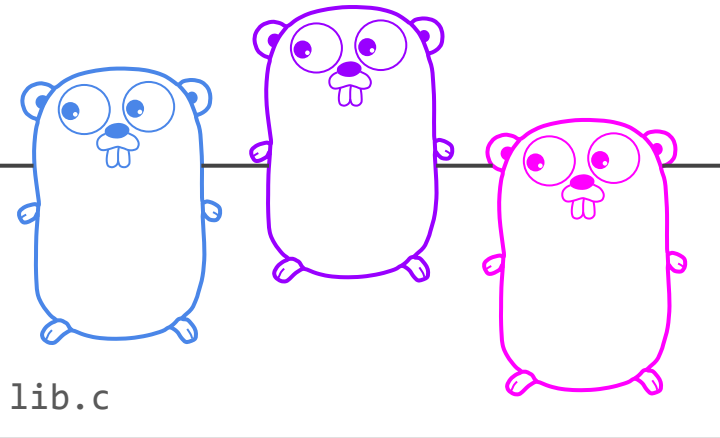
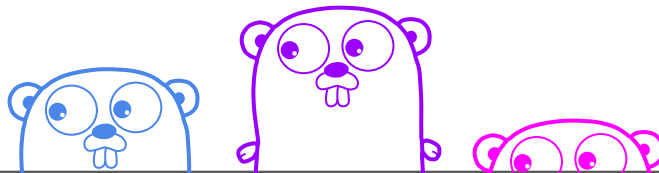
```
#include <stdio.h>
```

```
struct List {  
    int factor, power;  
    struct List* link;  
};
```

```
extern struct List* Factor(int x);  
extern void Print(struct List* l);
```

```
int main() {  
    for (int i = 1; i <= 1000; i++) {  
        printf("%4d == ", i);  
        Print(Factor(i));  
        printf("\n");  
    }  
    return 0;  
}
```

main.c



lib.c

Using a library, not so carefully

```
#include <stdio.h>
```

```
struct List {
```

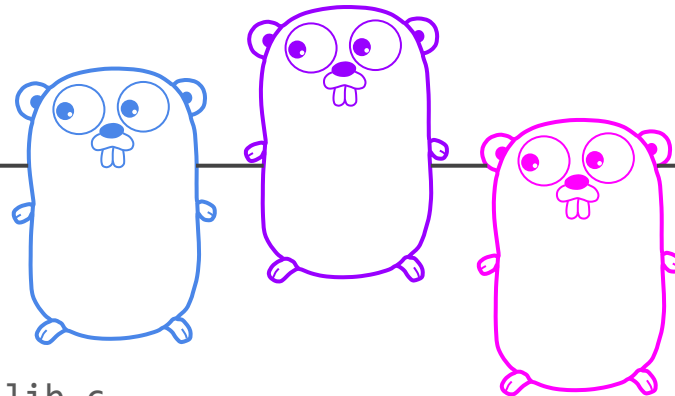
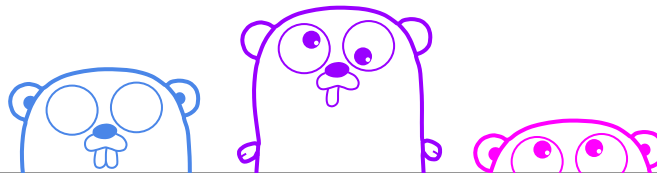
```
};
```

```
extern struct List* Factor(float x);
```

```
extern void Print(struct List l);
```

```
int main() {  
    for (int i = 1; i <= 1000; i++) {  
        printf("%4d == ", i);  
        Print(Factor(i));  
        printf("\n");  
    }  
    return 0;  
}
```

main.c



lib.c

Using a header file

```
#include <stdio.h>
#include "lib.h"
```

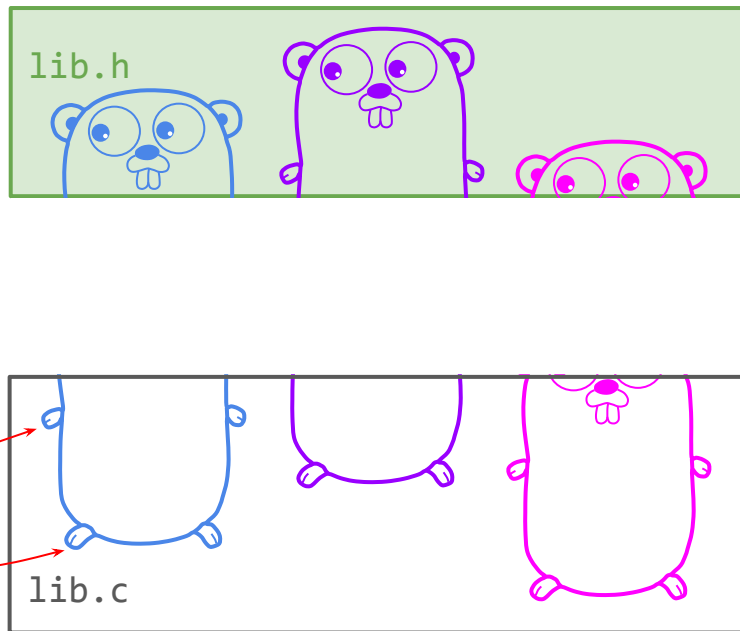
```
int main() {
    for (int i = 1; i <= 1000; i++) {
        printf("%4d == ", i);
        Print(Factor(i));
        printf("\n");
    }
    return 0;
}
```

main.c

lib.h

private parts accessible

lib.c

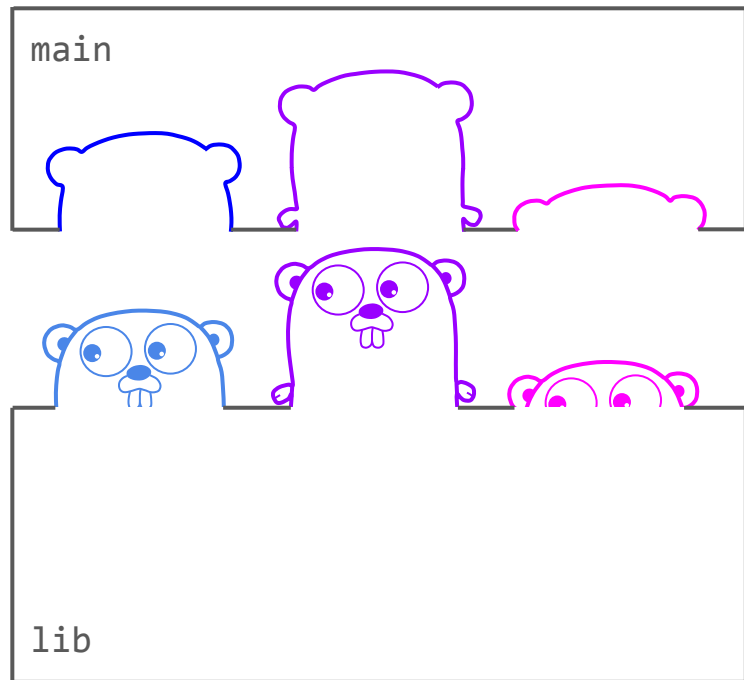


Header files issues

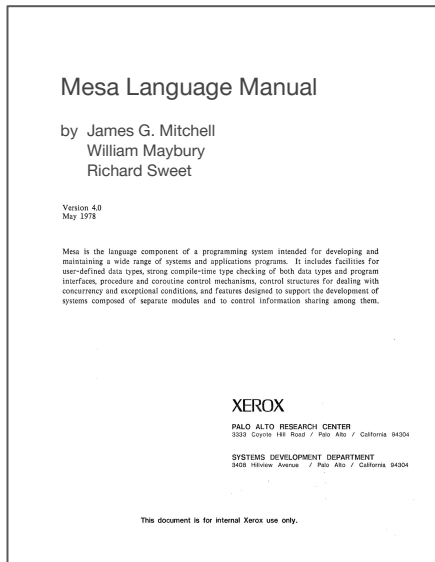
- Include guards (#ifdef, etc.)
- Duplication of definitions
- Information leakage
- Repeated processing (includes of includes)
- Size

What we really want

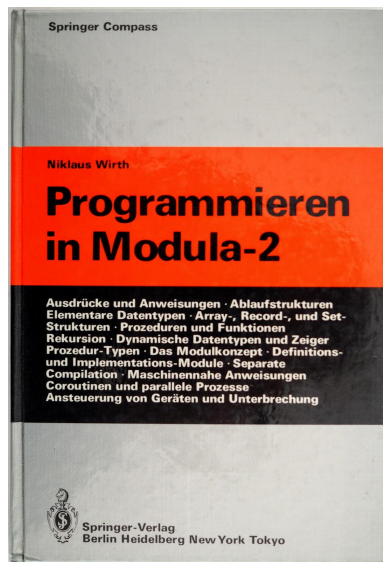
- Dedicated language feature
- No boilerplate
- Less redundancy
- Information hiding
- Efficient implementation
- Self-contained package interface



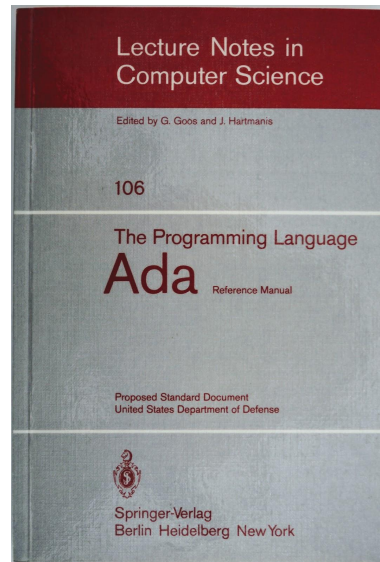
Languages pioneering modularization



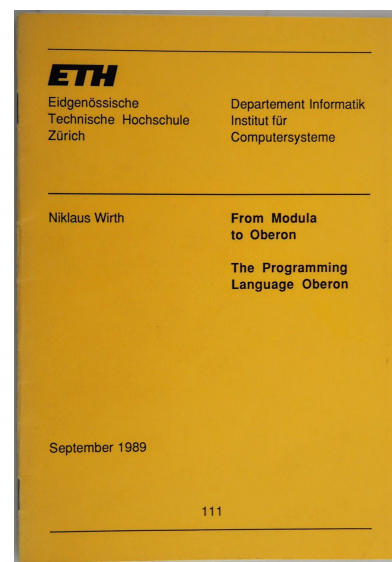
1976 Modula
Mesa



1978 Modula-2

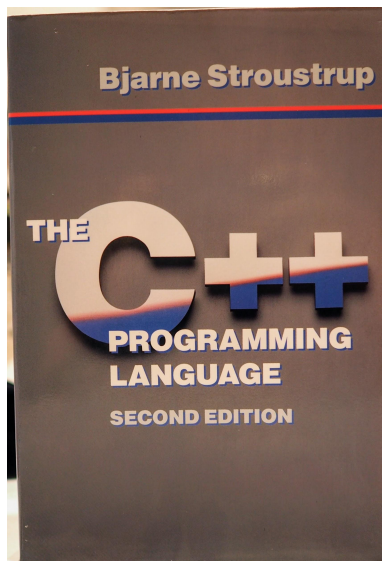


1980 Ada
Modula-3



1986 Oberon

Conspicuously absent ...



1983 C++

Today

- Language support for modularization is commonplace
 - Java/Scala, C#, Swift, Rust, etc. (statically typed)
 - JavaScript (ES6), Python, Ruby, etc. (dynamically typed)
- Similar concepts
- Different implementations
 - e.g., object files vs class files

Implementation

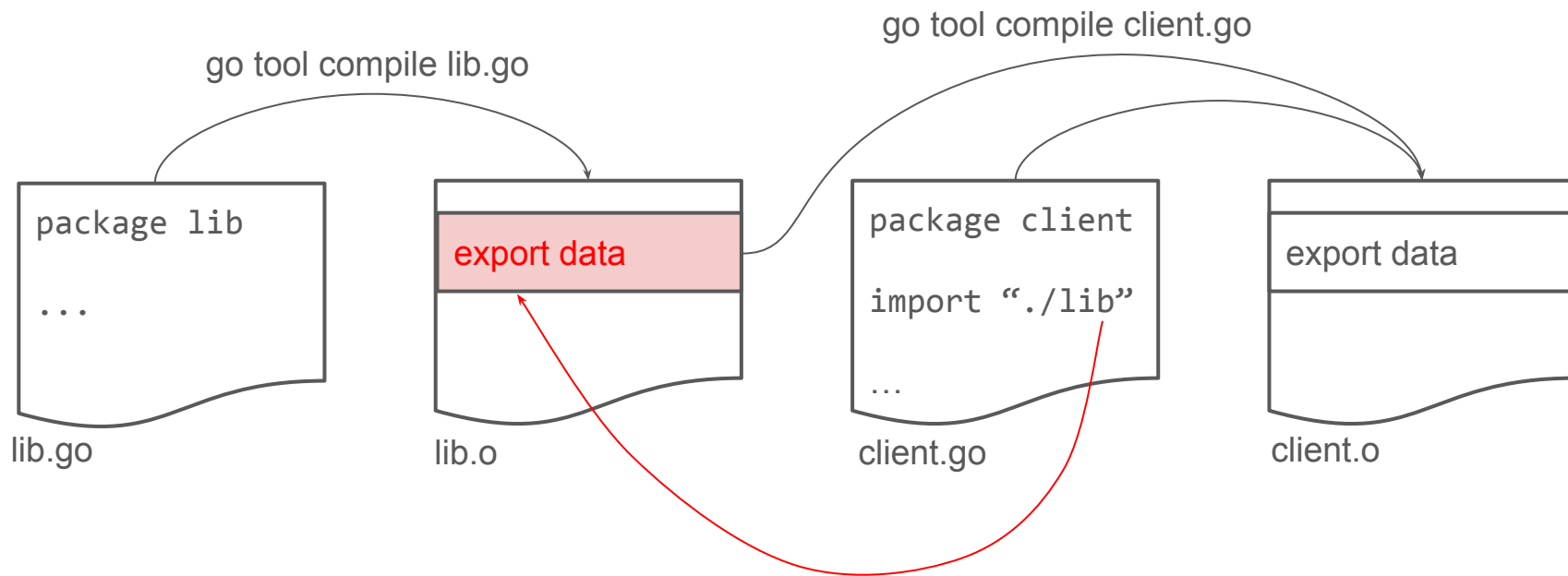
What does a compiler need to compile an import declaration?

- Description of imported library interface for type-checking
- Internal representation of that interface

Internal representation forms a graph, similar to a syntax tree.

- May contain additional implementation-specific information
- May be a DAG, or have cycles

Compiling a package



lib.o before Go 1.7

```
go object darwin amd64 go1.5 X:none
```

```
$$
```

```
package lib
```

```
    import runtime "runtime"
```

```
    import fmt "fmt"
```

```
    type @"".List struct { Factor int; Power int; Link *@"".List }
```

```
    func @"".Factor (@"".xÂ·2 int) (? *@"".List)
```

```
    func @"".Print (@"".lÂ·1 *@"".List "esc:0x1")
```

```
    func @"".init ()
```

```
$$
```

```
!
```

```
...
```

Issues with textual export data

- Not really Go code
- Go syntax redundant
- References may be long identifiers
- More complicated than necessary to read and write
- But: Human-readability is big plus (for compiler writer)

lib.o after Go 1.7

```
go object darwin amd64 devel +86f5f7fdfa Tue May 9 18:35:13 2017 +0000 X:framepointer
```

```
----
```

```
$$B
```

```
version 5
```

```
^@^B^A^Elib^@^E^M^?
```

```
^M^@    Users^Egri^WGoogle Drive^OGopherSg^Ego1^Klib.go^GList^@^U^F^B^KFactor^B\
```

```
^@^@    Power^B^@^B^GLink^W<^@^@    ^R^@^B^B^Ax^@^@^A^W<^@ <    Print^\  
^@^B^W<^A1^@^Mesc:0x1^@    ^?^B^A^]<autogenerated>^Ginit^@^@^@K^H^K^@^A^@
```

```
$$
```

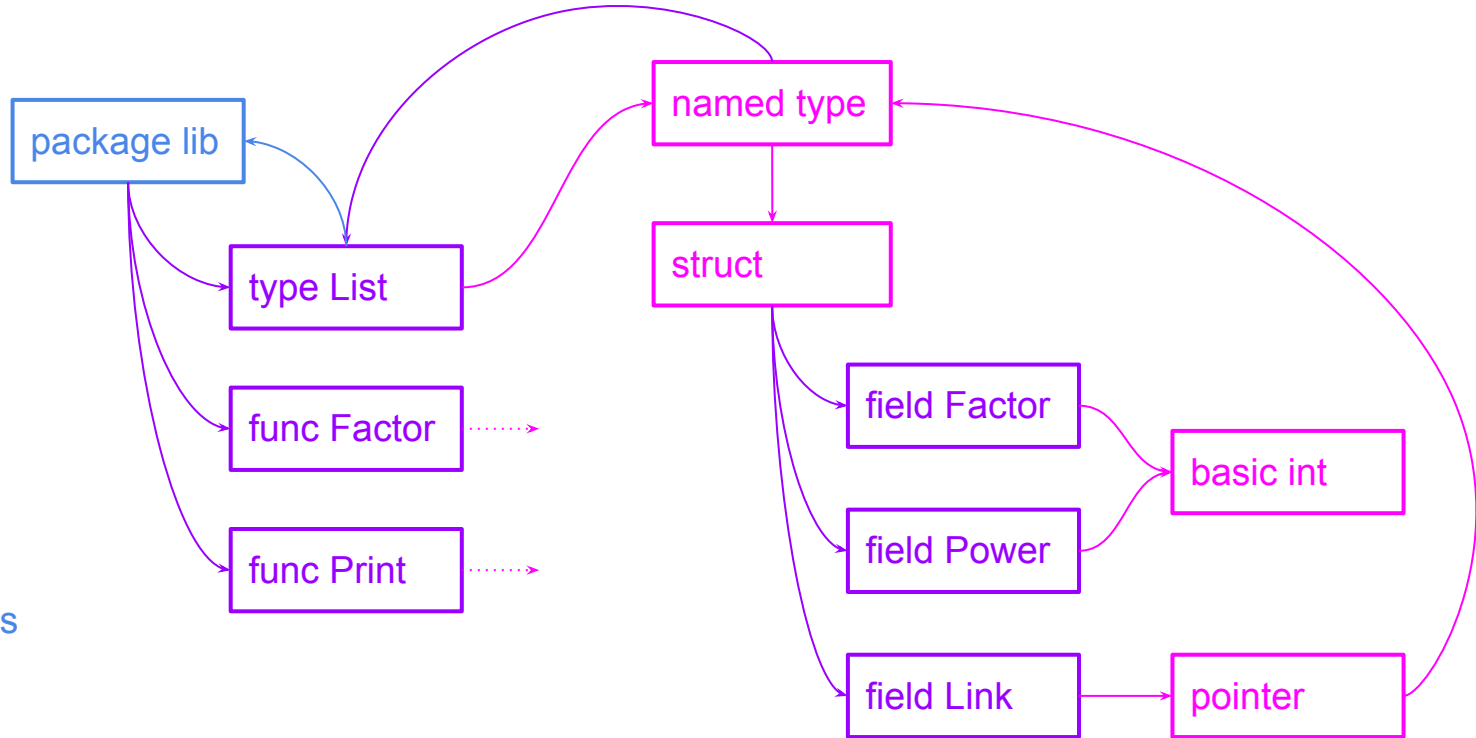
```
!
```

```
...
```

Advantages of binary export data

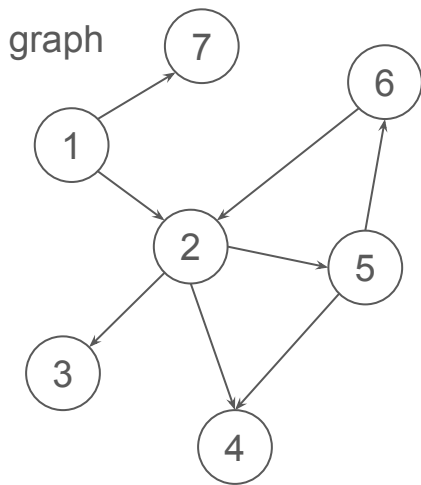
- Easier and faster to write, read
- Format can be very compact
- More easily extensible
- But: Binary format is not human-readable anymore

Exported objects of package lib



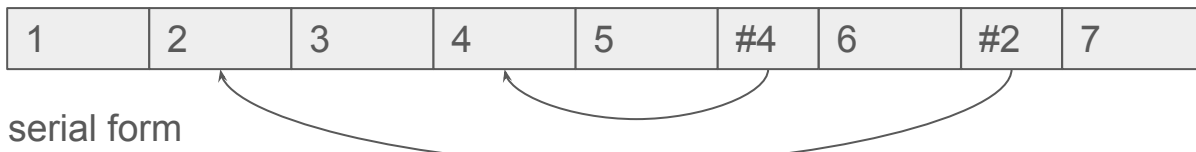
Packages
Objects
Types

Exporting means serializing a graph

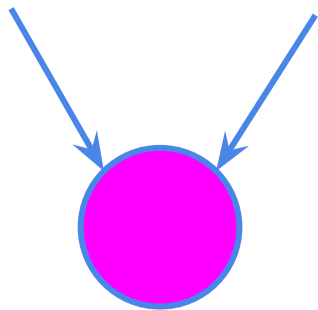


Algorithm:

- Traverse graph recursively
- Write nodes (contents) in order of traversal
- If a node was seen before, write reference instead



Excursion: DAGs in nature



Serialization algorithm

```
var seenBefore = map[Node]int{nil: 0}
```

```
// map of nodes already written out
```

```
func writeNode(n Node) {
    if index, ok := seenBefore[n]; ok {
        writeInt(index)
        return
    }
    seenBefore[n] = len(seenBefore)
    writeInt(-tag(n))
    writeContents(n)
}
```

```
// index >= 0
```

```
// -tag(n) < 0
```

```
// may call writeNode recursively
```

Deserialization algorithm

```
var seenBefore = []Node{nil: 0}
```

```
// list of nodes already read in
```

```
func readNode(n Node) {
```

```
    i := readInt()
```

```
    if i >= 0 {
```

```
        return seenBefore[i]
```

```
    }
```

```
    // i < 0
```

```
    n := newNode(-i)
```

```
    seenBefore = append(seenBefore, n)
```

```
    readContents(n)
```

```
// may call ReadNode recursively
```

```
}
```

Textual export

entity	data to encode	encoding
package lib	packageTag lib ""	package lib
type List	typeTag namedTag List package lib	type @"".List
struct type	structTag len(fields) Factor type int Power type int Link pointerTag type List	struct { Factor int Power int Link *List }
func Factor	funcTag Factor package lib	func @"".Factor
...		...

⇐ 72 bytes

Binary export

entity	data to encode	encoding
package lib	packageTag lib ""	-1 -3 l i b 0
type List	typeTag namedTag List package lib	-3 -7 -4 L i s t 0
struct type	structTag len(fields) Factor type int Power type int Link pointerTag type List	-11 3 -6 F a c t o r 1 -5 P o w e r 1 -4 L i n k -12 30
func Factor	funcTag Factor package lib	-5 3 0
...		

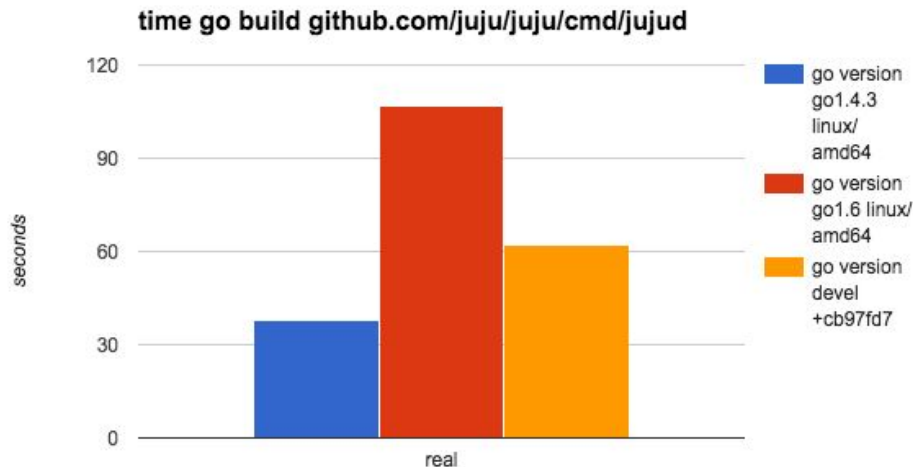
⇐ 38 bytes
~50% of text

Export data sizes after initial commit ae2f54a77

Package	old	new	new/old
archive/tar.....	13875	3883	28%
archive/zip.....	19464	5046	26%
bufio.....	7733	2222	29%
bytes.....	10342	3347	32%
cmd/addr2line.....	242	26	11%
cmd/api.....	39305	10368	26%
cmd/asm/internal/arch.....	27732	7939	29%
...			
unicode/utf16.....	1055	148	14%
unicode/utf8.....	1118	513	46%
vendor/golang.org/x/net/http2/hpack.....	8905	2636	30%
All packages	3518505	1017774	29%

Good news everyone, since gri's switched to making binary export/import the default, and a few followups from khr the time to build jujud compared to 1.4.3 is now solidly below 2x.

Dave Cheney, 2016-04-28




Future work

- Very large imports
 - Export data must always be read sequentially
 - Inefficient if we only use small part of it
 - Not uncommon with very large exported protobufs
- Possible solutions
 - Stop reading once we have what we need
 - Provide indexed access to export data

Final observations

- Module/package support is “must-have” language feature
- Import/export mechanism makes packages work
- Cheaper workarounds fail, eventually
 - C #includes
- Good implementation requires significant engineering



If the abstraction is right,
it's worth paying its price.



Thank you!