Go Compiler for COMP-520

Vincent Foley-Bourgon

Sable Lab
McGill University

November 2014

Agenda

- COMP-520
- Go
- My implementation
  - Lexer gotchas
  - Parser gotchas
- Recap

Questions welcome during presentation

COMP-520

- Introduction to compilers
- Project-oriented
- Being updated
- One possible project: a compiler for Go

COMP-520

- Introduction to compilers
- Project-oriented
- Being updated
- One possible project: a compiler for Go
- Super fun, you should take it!

Go

Go

- Created by Unix old-timers (Ken Thompson, Rob Pike) who happen to work at Google
- Helps with issues they see at Google (e.g. complexity, compilation times)
- Imperative with some OO concepts
  - Methods and interfaces
  - No classes or inheritance
- Focus on concurrency (goroutines and channels)
- GC
- Simple, easy to remember semantics
- Open source

Why Go for a compilers class?

- Language is simple
- Detailed online specification
- Encompasses all the classical compiler phases
- Allows students to work with a language that is quickly growing in popularity

Current work

My compiler

- Explore the implementation of Go
- Pin-point the tricky parts
- Find a good subset
  - Useful for writing programs
  - Covered by important compiler topics
  - Limit implementation drudgery

Tools

- Language: OCaml 4.02
- Lexer generator: ocamllex (ships with OCaml)
- Parser generator: Menhir (LR(1), separate from OCaml)

Why OCaml?

- Good lexer and parser generators
- Algebraic data types are ideal to create ASTs and other IRs
- Pattern matching is great for acting upon AST
- I like it!

Lexer

Lexer

- Written with ocamllex
- $\sim$270 lines of code
- Go spec gives all the necessary details
- One tricky part: automatic semi-colon insertion

Semi-colons

What you write

```
package main
import (
        "fmt"
        "math"
)

func main() {
        x := math.Sqrt(18)
        fmt.Println(x)
}
```

What the parser expects

Semi-colons

What you write

```
package main
import (
        "fmt"
        "math"
)

func main() {
        x := math.Sqrt(18)
        fmt.Println(x)
}
```

What the parser expects

```
package main;
import (
        "fmt";
        "math";
);

func main() {
        x := math.Sqrt(18);
        fmt.Println(x);
};
```

Semi-colons

When the input is broken into tokens, ***a semicolon is automatically inserted into the token stream*** *at the end of a non-blank line if the line's final token is*

- *an identifier*
- *a literal*
- *one of the keywords* `break`, `continue`, `fallthrough`, *or* `return`
- *one of the operators and delimiters* `++`, `--`, `)`, `]`, *or* `}`

Solution

```
rule next_token = parse
(* ... *)
| "break" { T_break }

| '\n' { next_token lexbuf }
```

Solution

```
rule next_token = parse
(* ... *)
| "break" { yield lexbuf T_break }

| '\n' { if needs_semicolon lexbuf then
          yield lexbuf T_semi_colon
        else
          next_token lexbuf
      }
```

Solution

```
rule next_token = parse
(* ... *)
| "break" { yield lexbuf T_break }

| '\n' { if needs_semicolon lexbuf then
          yield lexbuf T_semi_colon
        else
          next_token lexbuf
      }
| "//" { line_comment lexbuf }


and line_comment = parse
| '\n' { if needs_semicolon lexbuf then
          yield lexbuf T_semi_colon
        else
          next_token lexbuf
      }
| _ { line_comment lexbuf }
```

Pause philosophique

Is Go lexically a regular language?

Lexer

Supports most of the Go specification

- Unicode characters are not allowed in identifiers
- No unicode support in char and string literals
- Don't support second semi-colon insertion rule

```
func () int { return 42; }
```

Parser

Parser & AST

- Parser written with Menhir
- Parser: ∼600 lines of code (incomplete)
- AST: ∼200 lines of code
- Some constructs are particularily tricky!

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments
```

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments


func substr(str string, start int, length int)
    // named arguments, long form
```

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments


func substr(str string, start int, length int)
    // named arguments, long form


func substr(str string, start, length int)
    // named arguments, short form
```

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments


func substr(str string, start int, length int)
    // named arguments, long form


func substr(str string, start, length int)
    // named arguments, short form


func substr(string, start, length int)
    // Three parameters of type int
```

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments


func substr(str string, start int, length int)
    // named arguments, long form


func substr(str string, start, length int)
    // named arguments, short form


func substr(string, start, length int)
    // Three parameters of type int


func substr(str string, start int, int)
    // Syntax error
```

Tricky construct #1: function parameters

```
func substr(string, int, int)
    // unnamed arguments
```

```
func substr(string, start, length int)
    // Three parameters of type int
```

Tricky construct #1: function parameters

How to figure out named and unnamed parameter?

- Read list of either `type` or `identifier type`
- Process list to see if all `type` or at least one `identifier type`
- Generate the correct AST nodes (i.e. `ParamUnnamed(type)` or `ParamNamed(id, type)`)

Tricky construct #1: function parameters

How to figure out named and unnamed parameter?

- Read list of either `type` or `identifier type`
- Process list to see if all `type` or at least one `identifier type`
- Generate the correct AST nodes (i.e. `ParamUnnamed(type)` or `ParamNamed(id, type)`)

Only named parameters for project.

Tricky construct #2: Calls, conversions and built-ins

From the Go FAQ:

> *[...] Second, the language has been designed to be easy to analyze and* ***can be parsed without a symbol table****.*

Tricky construct #2: Calls, conversions and built-ins

Type conversions in Go look like function calls:

```
int(3.2)                // type conversion
fib(24)                 // function call
```

Tricky construct #2: Calls, conversions and built-ins

Type conversions in Go look like function calls:

```
int(3.2)                // type conversion
fib(24)                 // function call... probably
```

Tricky construct #2: Calls, conversions and built-ins

Type conversions in Go look like function calls:

```
int(3.2)                // type conversion
fib(24)                 // function call... probably
```

- It depends: is `fib` is a type?
- How do we generate the proper AST node?
- We need to keep track of identifiers in scope, i.e. a symbol table
- More complex parsing:
  `call ::= expr_or_type '(' expr* ')'`

  e.g. `[]*int(z)`

Tricky construct #2: Calls, conversions and built-ins

Built-ins also look like function calls:

```
xs := make([]int, 3)   // [0, 0, 0]
xs = append(xs, 1)     // [0, 0, 0, 1]
len(xs)                // 4
```

What's different?

Tricky construct #2: Calls, conversions and built-ins

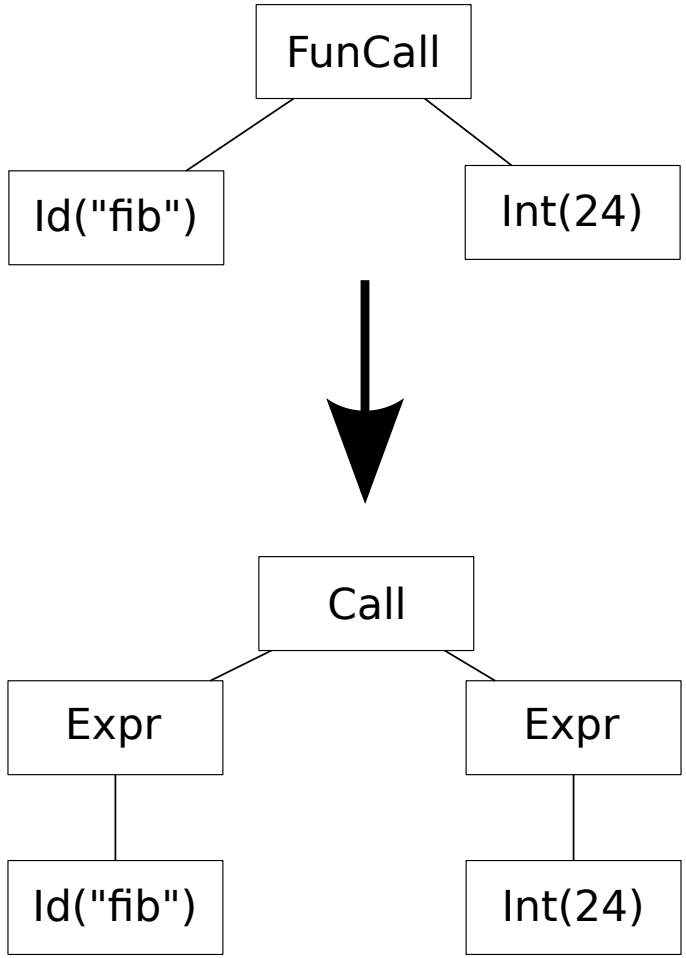
Built-ins also look like function calls:

```
xs := make([]int, 3)    // [0, 0, 0]
xs = append(xs, 1)      // [0, 0, 0, 1]
len(xs)                 // 4
```

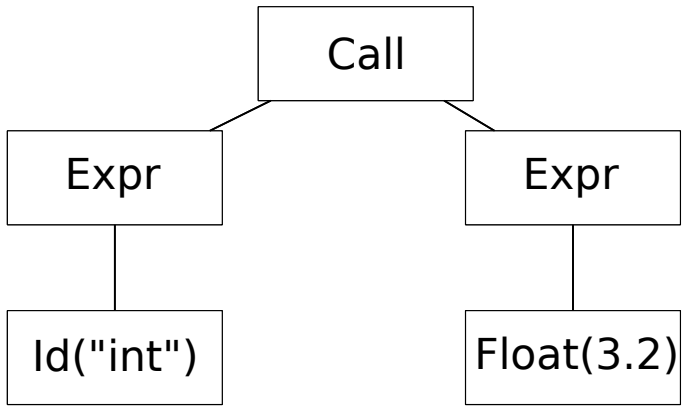
What's different?

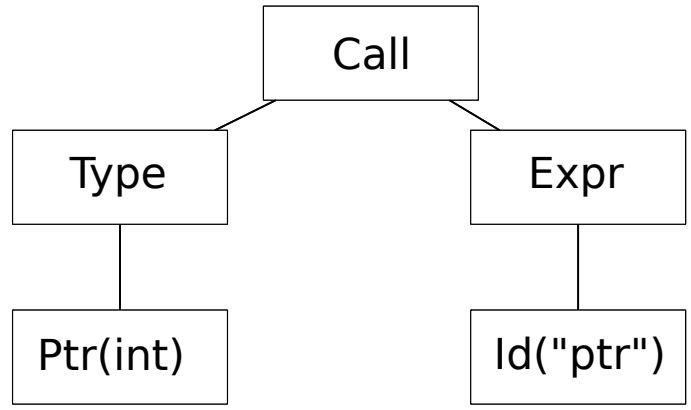
- The first parameter of a built-in can be a type
- `call ::= expr_or_type '(' expr_or_type* ')'`
- Very difficult to get right: `expr` and `type` conflict (i.e. identifier)
  - Factor the type non-terminals (*expr-term-factor*)
- AST "pollution"

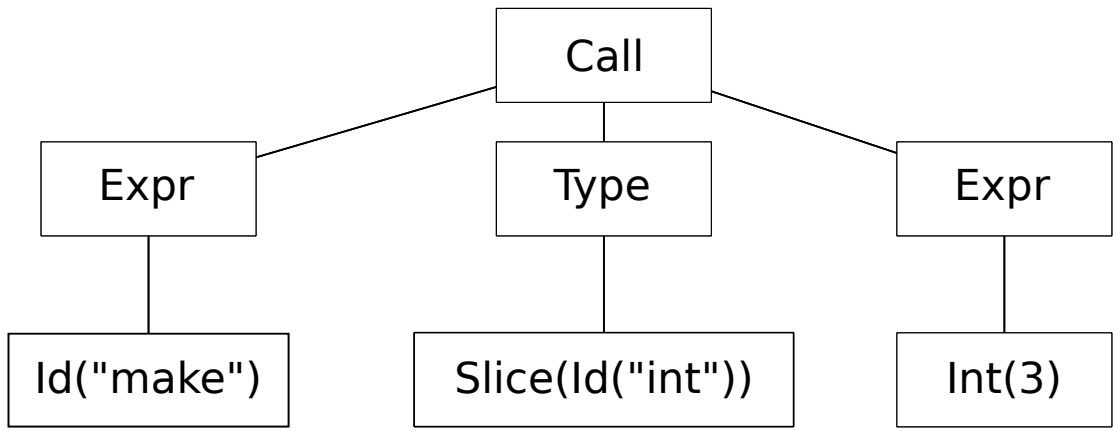
Tricky construct #2: Calls, conversions and built-ins



Tricky construct #2: Calls, conversions and built-ins







Pause philosophique

What does it mean to parse a language?

Pause philosophique

What does it mean to parse a language?

- For theorists: does a sequence of symbol belong to a language?
- For compiler writers: can I generate a semantically-precise AST from this sequence of symbols?

Tricky construct #3: chan directionality

- `chan int`: channel of ints
- `chan<- int`: send-only channel of ints
- `<-chan int`: receive-only channel of ints

What is `chan <- chan int`?

Tricky construct #3: chan directionality

- `chan int`: channel of ints
- `chan<- int`: send-only channel of ints
- `<-chan int`: receive-only channel of ints

What is `chan <- chan int`? `chan<- (chan int)`

Tricky construct #3: chan directionality

How do we implement this?

- Apply *expr-term-factor* factorization to types
- No *PEDMAS* for types
- Remember `expr_or_type`? Fun times :/
- Complicates parser

Semantics

Semantics - interfaces

A type **implicitly** implements an interface if it has the right methods.

```
type Point struct {
  x, y int
}

type Summable interface {
  Sum() int
}

func (p Point) Sum() int {
  return p.x + p.y
}

func Test(s Summable) {
  fmt.Println(s.Sum())
}

func main() {
  p := Point{ 3, 4 }
  Test(p)
}
```

Semantics - constants

Go does not perform automatic type conversions:

```
var x int = 15        // OK
var y float64 = x     // Error
```

Constants are “untyped” however:

```
var x int = 15        // OK
var y float64 = 15    // OK
var z int = 3.14      // Error
```

Constants are high-precision:

```
Pi = 3.14159265358979323846264338327950288419716939937510
HalfPi = Pi / 2  // Also high-precision
```

Status

Status - Lexer

- As complete as I want it at the moment
- Don't intend to add unicode support

Status - Parser

- Lacks support for many constructs (e.g. type switches, implicitly initialized consts, chan directionality, etc.)
- Some cheating to make productions easier (e.g. disallow parenthesized types in `expr_or_type`)

Status - AST

- In the process of simplifying the AST (e.g. generalized call node)
- Create a new, semantically richer AST that will be a result of type checking: type and scope information will be used to create appropriate nodes

Status - Semantic analysis

- On-going, currently doing basic types
- Scrapped some phases that turned out to be unnecessary
- Cheating by simplifying the rules of constants (e.g. `var x float64 = 3` is a type error)
- Haven't started thinking about interface types yet; maybe leave them out

Status - Code generation

- Not started at the moment
- Probably going to target JS or C
- JS: easy support for closures and multi-value returns

Misc.

- How to support a mini stdlib?
- GC? Allocate and forget, that's my motto!

Language subset

Language subset

- No concurrency support (channels, goroutines, `select`)
- Simplify some of the syntax (unnamed parameters)
- Eliminate "exotic" features (complex numbers, `iota`)
- Simplify constants
- Eliminate methods and interfaces
- No GC

Questions?