Lsub Go

Francisco J. Ballesteros
TR LSUB-15-3 (rev 1)

ABSTRACT

For the Clive OS being developed at Lsub, we have modified the Go compiler in several important aspects. This document describes the changes made to the language, the compiler, and its run-time.

1. Introduction

Clive is written using the Go programming language [1]. Clive system services are organized by connecting them through a pipe-like abstraction. Like it has been done in UNIX for decades. The aim is to let applications leverage the CSP programming style while, at the same time, make them work across the network.

The problem with standard Go (or CSP-like) channels is that:

1. They do not behave well upon errors, regarding termination of pipelines.
2. They do not convey error messages when errors happen.

Therefore, we modified the channel abstraction as provided by Go to make it a better replacement for traditional pipes. When using channels in Clive's Go, each end of the pipe may close it and the channel implementation takes care of propagating the error indication to the other end. Furthermore, an error string can be supplied when closing a channel and the other end may inquire about the cause of the error. This becomes utterly important when channels cross the network because errors do happen.

For example, consider the pipeline

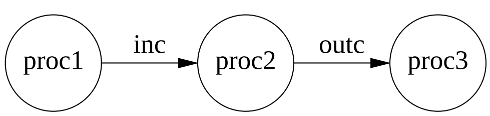

Figure 1: *Example pipeline of processes in clive*

In Clive, `proc2` can execute this code to receive data from an input channel, modify it, and send the result to an output channel:

```
var inc, outc chan[]byte
    ...
for data := range inc {
    ndata := modify(data)
    if ok := outc <-ndata; !ok {
        close(inc, cerror(outc))
        break
    }
}
close(outc, cerror(inc))
```

Should the first process, `proc1`, terminate normally (or abnormally), it calls `close` on the `inc` channel shown in the code excerpt. At this point, the code shown for `proc2` executes `close(outc, cerror(inc))`, which does two things:

1. retrieves the cause for the close of the input channel, by calling `cerror(inc)`.
2. closes the output channel providing exactly that error indication, by calling `close` with a second argument that provides the error.

Therefore, the error at a point of the pipe can be nicely propagated forward. In its interesting to to reconsider the implications of this for examples like that shown for removing files, and for similar system tools.

The most interesting case is when the third process, `proc3`, decides to cease consuming data. For example, because of an error or because it did find what it wanted. In this case, it calls `close` on the `outc` channel shown in the code.

The middle process is not able to send more data from that point in time. Instead of panicing, as the standard Go implementation would do, the send operation now returns `false`, thus `ok` becomes `false` when `proc2` tries to send more data. The loop can be broken cleanly, closing also the input channel to signal to the first process that there is no point in producing further data.

Furthermore, all involved processes can retrieve the actual error indicating the source of the problems (which is not just *"channel was closed"* and can be of more help).

As an aside, the last call to `close` becomes now a no-operation, because the output channel was already closed, and we don't need to add unnecessary code to prevent the call because in Clive this does not panic, unlike in standard Go.

The important point is that termination of the data stream is easy to handle for the program without resorting to exceptions (or panics), and we know which one is the error, so we can take whatever measures are convenient in each case.

There is a second change required by Clive: application contexts. We had to modify the runtime to include the concept of an application id that is inherited when new processes (goroutines) are created. Also, we had to access the current process (goroutine) id.

These were the two required changes. But, once we had to maintain our own Go compiler, we introduced other changes as well, as a convenience.

The following sections describe the changes made, as a reference for further ports. In all the changes we tried to be conservative and preserve as much as possible the existing structure, to make it easy to upgrade to future versions of the compiler.

Also, just in case we made a mistake regarding assumptions made by the compiler, adding more checks was preferred. The changes look worse but are safer.

2. Close

The `close` operation accepts now an optional second argument with the error status, and does not panic if the channel is already closed or is `nil`. Sending or receiving from a closed channel does not block and does not do anything. A new function `cerror` returns such error status, if any, for a given channel.

These calls are now equivalent:

```
close(c)
close(c, nil)
close(c, "")
```

2.1. Changes in the runtime package

The type `hchan` is changed to include an error string embedded in the structure, to preserve the invariant that there are no pointers to collect. This will change in the future, and we will keep an `error` instead garbage collected as everybode else.

- `runtime/chan.go:/^type.hchan`

```
type hchan struct {
    qcount   uint           // total data in the queue
    dataqsiz uint           // size of the circular queue
    buf      unsafe.Pointer // points to an array of dataqsiz elements
    elemsize uint16
    errlen   uint16
    closed   uint32
    elemtype *_type // element type
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // list of recv waiters
    sendq    waitq  // list of send waiters
    err      [maxerr]byte
    lock     mutex
}
```

  The new fields are `errlen` and `err`.

  The standard `closechan` is now a call to `closechan2` with `nil` as the second argument.

- `runtime/chan.go:/^func.closechan`

```
func closechan(c *hchan) {
    closechan2(c, nil)
}
```

  A new `chanerrstr` function returns the error string for the types accepted as a second argument to `close`:

- `runtime/chan.go:/^func.chanerrstr`

```
func chanerrstr(e interface{}) string {
    if e == nil {
        return ""
    }
    switch v := e.(type) {
    case nil:
        return ""
    case stringer:
        return v.String()
    case error:
        return v.Error()
    case string:
        return v
    default:
        panic("close errors must be a string or an error")
    }
}
```

The old `closechan` is now `closechan2`:

- `runtime/chan.go:/^func.closechan2`

```
func closechan2(c *hchan, e interface{}) {
    if c == nil {
        return
    }

    estr := chanerrstr(e)
    lock(&c.lock)
    if c.closed != 0 {
        unlock(&c.lock)
        return
    }
    ...
    c.errlen = uint16(0)
    if estr != "" {
        n := (*stringStruct)(unsafe.Pointer(&estr)).len
        if n > maxerr {
            n = maxerr
        }
        c.errlen = uint16(n)
        c.err[c.errlen] = 0
        p := (*stringStruct)(unsafe.Pointer(&estr)).str
        memmove(unsafe.Pointer(&c.err[0]), p, uintptr(c.errlen))
    }
    ...
}
```

The `chansend` function is changed not to panic when sending on a closed channel. It will be changed again later to return a boolean indicating if the send could proceed or not. For now, it returns `true` indicating the send is complete (and discarded).

- `runtime/chan.go:/^func.chansend`

```
func chansend(...) bool {
    ...
    if c.closed != 0 {
        unlock(&c.lock)
        return true
    }
    // and the same in a few other places that did panic.
    ...
}
```

In `selectgoImpl` we have to change the case for `sclose` so it does not panic. Instead, selects proceeds without actually doing anything.

- `runtime/select.go:/^sclose`

```
func selectgoImpl(...) (uintptr, uint16) {
    ...
sclose:
    selunlock(sel)
    goto retc
    ...
}
```

A new type and a couple of functions permits the user to call `cerror()` and retrieve the error for a channel (or nil), and to learn if the channel is closed and drained.

- `runtime/chan.go:/^type.chanError`

```
type chanError string
func (e chanError) Error() string {
    return string(e)
}
```

- `runtime/chan.go:/^func.cerror`

```
func cerror(c *hchan) error {
    if c == nil {
        return nil
    }
    lock(&c.lock)
    if c.closed == 0 || c.errlen == 0 || c.err[0] == 0 {
        unlock(&c.lock)
        return nil
    }
    msg := gostringn(&c.err[0], int(c.errlen))
    unlock(&c.lock)
    return chanError(msg)
}
```

- `runtime/chan.go:/^func.cclosed`

```
func cclosed(c *hchan) bool {
    if c == nil {
        return true
    }
    lock(&c.lock)
    closed := c.closed != 0 && (c.dataqsiz == 0 ||
        c.qcount <= 0)
    unlock(&c.lock)
    return closed
}
```

2.2. Changes in the compiler

The compiler must add `cerror` and `cclosed` as new builtins, and must decide which one of `closechan` and `closechan2` should be called.

We define new constants for nodes that are calls to `cerror` or `cclosed`.

- `cmd/compile/internal/gc/syntax.go:/OCCLOSED`

```
// Node ops.
const (
    OXXX = iota
    ...
    OCCLOSED         // cclosed
    OCERROR          // cerror
    OCLOSE           // close
    ...
)
```

We give names for the new constants when printed:

- `cmd/compile/internal/gc/fmt.go:0+/goopnames/+/OCCLOSED/`

```
var goopnames = []string{
    ...
    OCCLOSED:  "cclosed",
    OCERROR:   "cerror",
    OCLOSE:    "close",
    ...
}
```

Precedence must be given to `cclosed` and `cerror`:

- `cmd/compile/internal/gc/fmt.go:0+/opprec/+/OCCLOSED/`

```
var opprec = []int{
    ...
    OCCLOSED:      8,
    OCERROR:       8,
    OCLOSE:        8,
    ...
}
```

Also, `exprfmt` has to check out if `close` has one or two arguments and must add cases for `cclosed` and `cerror`.

- `cmd/compile/internal/gc/fmt.go:/^func.exprfmt/+/OCLOSE/`

```
func exprfmt(n *Node, prec int) string {
    ...
    case OCLOSE:
        // nemo: close with 2nd arg
        if n.Left != nil && n.Right != nil {
            return fmt.Sprintf("%v(%v, %v)",
            Oconv(int(n.Op), obj.FmtSharp),
                n.Left, n.Right)
        }
        fallthrough
    case OREAL,
        OIMAG,
        ...
        OCERROR, OCCLOSED,
    ...
}
```

The predefined `syms` at `lex.go` must add `cerror` and `cclosed`.

- `cmd/compile/internal/gc/lex.go:/cclosed`

```
var syms = []struct {...} {
    ...
    {"cclosed", LNAME, Txxx, OCCLOSED},
    {"cerror", LNAME, Txxx, OCERROR},
    {"close", LNAME, Txxx, OCLOSE},
    ...
}
```

The `opnames` array is auto-generated and we don't have to add entries, but these are them.

- `cmd/compile/internal/gc/opnames.go:/CCLOSED`

```
var opnames = []string{
    ...
    OCCLOSED:         "CCLOSED",
    OCERROR:          "CERROR",
    OCLOSE:           "CLOSE",
    ...
}
```

In `order.go` we must add `cclosed` and `cerror` to `orderstmt`.

- `cmd/compile/internal/gc/order.go:/CCLOSED`

```
case OAS2,
    OCLOSE,
    OCCLOSED,
    OCERROR,
    ...
```

In `racewalk.go` must do the same for `racewalknode`.

- `cmd/compile/internal/gc/racewalk.go:/CCLOSED`

```
// should not appear in AST by now
case OSEND,
    ORECV,
    OCCLOSED,
    OCERROR,
    OCLOSE,
```

In `typecheck1`, `OCLOSE` must accept an optional second argument and don’t fail for send-only channels:

- `cmd/compile/internal/gc/typecheck.go:/ˆfunc.typecheck1/+/OCLOSE/`

```
case OCLOSE:
    // nemo: accept opt. second arg and don’t fail on close for
    // send only channels.
    args := n.List
    if args == nil {
        Yyerror("missing argument for close()")
        n.Type = nil
        return
    }
    if args.Next != nil && args.Next.Next != nil {
        Yyerror("too many arguments for close()")
        n.Type = nil
        return
    }

// nemo: this probably isn’tneeded. n should be ok already.
n.Left = args.N
if args.Next != nil {
    n.Right = args.Next.N
} else {
    n.Right = nil
}
n.List = nil

typecheck(&n.Left, Erv)
defaultlit(&n.Left, nil)
l := n.Left
t := l.Type
if t == nil {
    n.Type = nil
    return
}
if t.Etype != TCHAN {
    Yyerror("invalid operation: %v (non-chan type %v)", n, t)
    n.Type = nil
    return
}
```

```
if n.Right != nil {
    typecheck(&n.Right, Erv)
    defaultlit(&n.Right, nil)
    t = n.Right.Type
    if t == nil {
        n.Type = nil
        return
    }
    // TODO: check that the type is string or an error type.
}
ok |= Etop
break OpSwitch
```

Also in `typecheck1`, `cclosed` and `cerror` must be processed.

- `cmd/compile/internal/gc/typecheck.go:/^func.typecheck1/+/OCCLOSED/`

```
case OCCLOSED, OCERROR:
    // nemo: new builtins
    ok |= Erv
    args := n.List
    if args == nil {
        Yyerror("missing argument for %v", n)
        n.Type = nil
        return
    }
    if args.Next != nil {
        Yyerror("too many arguments for %v", n)
        n.Type = nil
        return
    }

n.Left = args.N
n.List = nil
typecheck(&n.Left, Erv)
defaultlit(&n.Left, nil)
l := n.Left
t := l.Type
if t == nil {
    n.Type = nil
    return
}
if t.Etype != TCHAN {
    Yyerror("invalid operation: %v (non-chan type %v)", n, t)
    n.Type = nil
    return
}
if n.Op == OCCLOSED {
    n.Type = Types[TBOOL]
} else {
    n.Type = errortype
}
break OpSwitch
```

In `checkdefergo` we must prevent discarding the result of `cclosed` and `cerror`.

- `cmd/compile/internal/gc/typecheck.go:/^func.checkdefergo/+/OCCLOSED/`

```
case OAPPEND,
    OCAP,
    OCCLOSED,
    OCERROR,
```

In `walkstmt` we must check walk the two new builtins.

- `cmd/compile/internal/gc/walk.go:/^func.walkstmt/+/OCCLOSED/`

```
case OAS,
    OCCLOSED,
    OCERROR,
```

In `walkexpr`, we must check if we have one or two arguments for `close` and then call one of `closechan` and `closechan2`.

- `cmd/compile/internal/gc/walk.go:/^func.walkexpr/+/OCLOSE/`

```
case OCLOSE:
    if n.Right == nil {
        fn := syslook("closechan", 1)
        substArgTypes(fn, n.Left.Type)
        n = mkcall1(fn, nil, init, n.Left)
    } else {
        fn := syslook("closechan2", 1)
        substArgTypes(fn, n.Left.Type)
        n = mkcall1(fn, nil, init, n.Left, n.Right)
    }
    goto ret
```

In `walkexpr`, we must add calls for the two new builtins:

- `cmd/compile/internal/gc/walk.go:/^func.walkexpr/+/OCCLOSED/`

```
case OCCLOSED:
    fn := syslook("cclosed", 1)
    substArgTypes(fn, n.Left.Type)
    n = mkcall1(fn, Types[TBOOL], init, n.Left)
    goto ret

case OCERROR:
    fn := syslook("cerror", 1)
    substArgTypes(fn, n.Left.Type)
    n = mkcall1(fn, errortype, init, n.Left)
    goto ret
```

The file `builtin.go` is generated, but anway these are the new runtime functions called:

- `cmd/compile/internal/gc/builtin.go:/closechan`

```
"func @\"\".closechan (@\"\".hchan.1 any)\n" +
"func @\"\".closechan2 (@\"\".hchan.1 any, @\"\".err.2 interface {})\n" +
"func @\"\".cerror (@\"\".hchan.2 any) (? error)\n" +
"func @\"\".cclosed (@\"\".hchan.2 any) (? bool)\n" +
```

A new file `lsub_test.go` tests for the changes in close.

3. Send

The send operation on a closed chan was changed to proceed, doing nothing in that case. It must be changed to report if the send could be done or not, as in:

```
if ok := c <- v; !ok {
    ...
}
```

3.1. Changes in the runtime package

A new function `chansend2`, replaces `chansend1` as the entry point for sends. It returns a `bool` reporting if the send was done or not (i.e., if the channel was open or closed).

- `runtime/chan.go:/^func.chansend2`

```
func chansend2(t *chantype, c *hchan, elem unsafe.Pointer) bool {
    if t == nil {
        return false // prevent this from inlining
    }
    _, did := chansend(t, c, elem, true,
        getcallerpc(unsafe.Pointer(&t)))
    return did
}
```

The old `chansend` is changed to return two booleans instead of one: could we send without blocking?, and did the send happen? (i.e., was the channel not closed).

When it did return `false`, it now:

- `runtime/chan.go:/^func.chansend\(`

```
func chansend(...) (bool, bool) {
    ...
    if !block {
        return false, false
    }
    ...
}
```

When it did return `true` because it could send, it now does

```
return true, true
```

Also, when the channel is found closed:

```
if c.closed != 0 {
    unlock(&c.lock)
    return true, false
}
```

Note that this did panic before we changed anything.

This part of the code is also changed:

```
gp.waiting = nil
done := true
if gp.param == nil {
    if c.closed == 0 {
        throw("chansend: spurious wakeup")
    }
    // nemo: don't panic("send on closed channel")
    done = false
}
gp.param = nil
if mysg.releasetime > 0 {
    blockevent(int64(mysg.releasetime)-t0, 2)
}
releaseSudog(mysg)
return true, done
```

Because of this change, `selectnbsend` has to be changed to use one of the two returned values.

- `runtime/chan.go:/^func.selectnbsend`

```
func selectnbsend(...) (selected bool) {
    can, _ := chansend(...)
    return can
}
```

The same happens to `reflect_chansend`.

- `runtime/chan.go:/^func.reflect_chansend`

```
func reflect_chansend(...) (selected bool) {
    can, _ := chansend(...)
    return can
}
```

3.2. Changes in the compiler

The syntax must now accept using `c<-x` as a value. In the grammar we must note that

- `cmd/compile/internal/gc/go.y:0+/^expr/+/LCOMM`

```
expr LCOMM expr
{
    $$ = Nod(OSEND, $1, $3);
}
```

is now a valid expression once again. This does not change the code, but there was a comment indicating that this was here just to report syntax errors.

The file `builtin.go` is generated, but anway this function is added:

- cmd/compile/internal/gc/builtin.go:/closechan

```
"func @\"\".chansend2 (@\"\".chanType.2 *byte, @\"\".hchan.3 chan<- any, @\"\".elem.4 *any) (@\
```

The function hascallchan is used to see if something has a call to a channel, and must now consider OSEND as part of expressions:

- cmd/compile/internal/gc/const.go:/^func.hascallchan/+/OSEND

```
func hascallchan(n *Node) bool {
    ...
    switch n.Op {
    case OAPPEND,
        ...
        OSEND:
        return true
    }
    ...
}
```

It is ok to use send in assignments in a select. We introduce a new OSELSEND node type that will later be used like OSELRECV nodes. First we define the new node type.

- cmd/compile/internal/gc/syntax.go:/OSELSEND

```
// Node ops.
const (
    OXXX = iota
    OSELRECV        // case x = <-c:
    OSELRECV2       // case x, ok = <-c:
    OSELSEND        // case ok = c <- x:
    ...
)
```

This is generated, but anyway...

- cmd/compile/internal/gc/opnames.go:/opnames/

```
var opnames = []string{
    ...
    OSELRECV:          "SELRECV",
    OSELRECV2:         "SELRECV2",
    OSELSEND:          "SELSEND",
    ...
}
```

In order, a send can now happen within a expression.

- cmd/compile/internal/gc/order.go:/^func.orderexpr/

```
func orderexpr(np **Node, order *Order, lhs *Node) {
    ...
    case OSEND:
        t := marktemp(order);
        orderexpr(&n.Left, order, nil)
        orderexpr(&n.Right, order, nil)
        orderaddrtemp(&n.Right, order)
        cleantemp(t, order)
}
```

In `select`, we must prepare to accept assignments using sends.

- `cmd/compile/internal/gc/select.go:/ˆfunc.typecheckselect/+/OAS/`

```
func typecheckselect(sel *Node) {
    ...
    case OAS:
        switch n.Right.Op {
        case ORECV:
            n.Op = OSELRECV
        case OSEND:
            // n.Op = OSELSEND
            Yyerror("BUG: TODO")
        default:
            Yyerror("must have chan op on rhs")
        }
}
```

- `cmd/compile/internal/gc/select.go:/ˆfunc.walkselect/+/OSEND/`

```
func walkselect(sel *Node) {
    ...
    // optimization: one-case select: single op.
    ...
    case OSEND:
        ch = n.Left
    case OSELSEND:
        Fatal("walkselect OSELSEND not implemented")
    ...
    // convert case value arguments to addresses.
    case OSELSEND:
        Fatal("walkselect OSELSEND not implemented")
    ...
    // optimization: two-case select but one is default
    case OSELSEND:
        Fatal("walkselect OSELSEND not implemented")
    ...
    // register cases
    case OSELSEND:
        Fatal("walkselect OSELSEND not implemented")
}
```

In `typecheck`, `callrecv` must be updated so it does not indicate if a node is just a call or receive, but also a send.

- `cmd/compile/internal/gc/typecheck.go:/^func.callrecv`

```
func callrecv(n *Node) bool {
    ...
    case OCALL,
        OSEND,
    ...
}
```

The main change is making `typecheck1` accept `OSEND` as `Erv`.

- `cmd/compile/internal/gc/typecheck.go:/^func.typecheck1/+/OSEND/`

```
func typecheck1(np **Node, top int) {
    ...
    case OSEND:
        ok |= Etop|Erv
        ...
        // TODO: more aggressive
        // n.Etype = 0
        n.Type = Types[TBOOL]
        break OpSwitch
}
```

Also, in `walk`, calling `chansend2` so it can return its value.

- `cmd/compile/internal/gc/walk.go:/^func.walkexpr/+/OSEND/`

```
func walkexpr(...) {
    ...
    case OSEND:
        n1 := n.Right
        n1 = assignconv(n1, n.Left.Type.Type, "chan send")
        walkexpr(&n1, init)
        n1 = Nod(OADDR, n1, nil)
        n = mkcall1(chanfn("chansend2", 2, n.Left.Type),
            Types[TBOOL], init,
            typename(n.Left.Type), n.Left, n1)
        n.Type = Types[TBOOL]
        goto ret
}
```

4. Send in selects

This change permits using

```
select {
case ok := c <- v:
    ...
}
```

4.1. Changes in the runtime

Two new functions accept a pointer to the returned value in sends, one blocks and one doesn’t.

- `runtime/chan.go:/^func.chanselsend/`

```
func chanselsend(t *chantype, c *hchan, elem unsafe.Pointer, okp *bool) bool {
    if t == nil {
        return false    // prevent this from inlining
    }
    ok, did := chansend(t, c, elem, true, getcallerpc(unsafe.Pointer(&t)))
    if okp != nil {
        *okp = did
    }
    return ok
}


func channbselsend(t *chantype, c *hchan, elem unsafe.Pointer, okp *bool) bool {
    if t == nil {
        return false    // prevent this from inlining
    }
    ok, did := chansend(t, c, elem, false, getcallerpc(unsafe.Pointer(&t)))
    if okp != nil {
        *okp = did
    }
    return ok
}
```

4.2. Changes in the compiler

In `typecheckselect`, we will convert cases like`ok=c<-v` to `OSELSEND` nodes, like done for receives.

- `cmd/compile/internal/gc/select.go:/^func.typecheckselect/+/OAS/`

```
case OAS:
    ...
    switch n.Right.Op {
    case ORECV:
        n.Op = OSELRECV
    case OSEND:
        n.Op = OSELSEND
    default:
        Yyerror("select assignment must have receive on rhs")
    }
```

In `orderstmt`, we must add a case for `OSELSEND` within `OSELECT`.

- `cmd/compile/internal/gc/order.go:/^func.orderstmt/+/OSE-LECT/+/OSELSEND/`

```
case OSELSEND:
    if r.Colas {
        t = r.Ninit
        if t != nil && t.N.Op == ODCL && t.N.Left == r.Left {
            t = t.Next
        }
        if t != nil && t.N.Op == ODCL && t.N.Left == r.Ntest {
            t = t.Next
        }
        if t == nil {
            r.Ninit = nil
        }
    }
    if r.Ninit != nil {
        Yyerror("ninit on select send")
        dumplist("ninit", r.Ninit)
    }

// case ok = c <- x
// r->left is ok, r->right is SEND, r->right->left is c, r->right->right is x
// r->left == N means 'case c<-x'.
// c is always evaluated; ok is only evaluated when assigned.
orderexpr(&r.Right.Left, order, nil)
if r.Right.Left.Op != ONAME {
    r.Right.Left = ordercopyexpr(r.Right.Left, r.Right.Left.Type, order, 0)
}

if r.Left != nil && isblank(r.Left) {
    r.Left = nil
}
if r.Left != nil {
    tmp1 = r.Left
    if r.Colas {
        tmp2 = Nod(ODCL, tmp1, nil)
        typecheck(&tmp2, Etop)
        l.N.Ninit = list(l.N.Ninit, tmp2)
    }
    r.Left = ordertemp(tmp1.Type, order, false)
    tmp2 = Nod(OAS, tmp1, r.Left)
    typecheck(&tmp2, Etop)
    l.N.Ninit = list(l.N.Ninit, tmp2)
}
orderblock(&l.N.Ninit)
```

We keep the old `OSEND` case within selects to leave the previous setup undisturbed, in case we introduce any bugs.

In `walkselect`, we must handle the new case. First in the one-case select.

- `cmd/compile/internal/gc/select.go:/^func.walkselect/+/OSELSEND/`

```
// optimization: one-case select: single op.
...
case OSELSEND:
    ch = n.Right.Left
    if n.Op == OSELSEND || n.Ntest == nil {
        if n.Left == nil {
            n = n.Right
        } else {
            n.Op = OAS
        }
        break
    }
    Fatal("walkselect OSELSEND with OAS2")
```

Then while converting case arguments to addresses.

```
// convert case value arguments to addresses.
...
case OSELSEND:
    n.Left = Nod(OADDR, n.Left, nil)
    typecheck(&n.Left, Erv)
    n.Right.Right = Nod(OADDR, n.Right.Right, nil)
    typecheck(&n.Right.Right, Erv)
```

Next, in the two-case select with default optimization.

```
// optimization: two-case select but one is default
...
case OSELSEND:
    r = Nod(OIF, nil, nil)
    r.Ninit = cas.Ninit
    ch := n.Right.Left
    r.Ntest = mkcall1(chanfn("channbselsend", 2, ch.Type),
        Types[TBOOL], &r.Ninit, typename(ch.Type),
        ch, n.Right.Right, n.Left)
```

Finally, in the plain select cases.

```
// register cases
...
case OSELSEND:
    r.Ntest = mkcall1(chanfn("chanselsend", 2, n.Right.Left.Type),
        Types[TBOOL], &r.Ninit, var_,
        n.Right.Left, n.Right.Right, n.Left)
```

The file `builtin.go` is generated, but anway this is added:

- `cmd/compile/internal/gc/builtin.go:/channbselsend`

```
"func @\"\".channbselsend (@\"\".chanType.2 *byte, @\"\".hchan.3 chan<- any, @\"\".elem.4 *any,
"func @\"\".chanselsend (@\"\".chanType.2 *byte, @\"\".hchan.3 chan<- any, @\"\".elem.4 *any, @
```

5. App ids

This change provides each process (goroutine) with a new application id, inherited when new processes are created.

First, a new `gappid` is added to `g`.

- `runtime/runtime2.go:/^.readyg /`

```
type g struct {
    ...
    readyg      *g
    gappid      int64
    ...
}
```

It is initialized to the `goid` for top-level processes.

- `runtime/proc1.go:/^func.newextram`

```
func newextram() {
    ...
    gp.goid = int64(xadd64(&sched.goidgen, 1))
    gp.gappid = gp.goid
    ...
}
```

- `runtime/proc.go:/^func.main`

```
func main() {
    g := getg()
    g.gappid = g.goid
    ...
}
```

And it is inherited. We pass the application id as an argumetn because `systemstack` is likely to run on `g0` and not on the caller process context.

- `runtime/proc1.go:/^/func.newproc\(`

```
func newproc(...) {
    argp := add(unsafe.Pointer(&fn), ptrSize)
    pc := getcallerpc(unsafe.Pointer(&siz))
    appid := int64(0)
    if _g_ := getg(); _g_ != nil {
        appid = _g_.gappid
    }
    systemstack(func() {
        newproc1(fn, (*uint8)(argp), siz, 0, appid, pc)
    })
}
```

```
func newproc1(..., appid int64,...) {
    ...
    newg.goid = int64(_p_.goidcache)
    newg.gappid = appid
    ...
```

The interface for the user is like follows.

- `runtime/proc.go:/^/func.AppId`

```
// Return the application id for the current process (goroutine).
func AppId() int64 {
    g := getg()
    return g.gappid
}

// Return the process id (goroutine id)
func GoId() int64 {
    g := getg()
    return g.goid
}

// Make the current process the leader of a new application, with its own id
// set to that of the process id.
func NewApp() {
    g := getg()
    g.gappid = g.goid
}
```

6. Looping select construct

This change was not strictly required, but, because we had to change the compiler as shown before, it was made for the programmer's convenience.

The change introduces a new `doselect` construct that is a looping select (similar to CSP's *do* control structure). Within the construct, a `break` breaks the entire loop and a `continue` continues looping. This is an example:

```
doselect {
case <-a:
    ...
case <-b:
    if foo {
        break
    }
case <-c: {
    if bar {
        continue
    }
    ...
}
```

The meaning is:

```
Loop:
for {
    select {
    case <-a:
        ...
    case <-b:
        if foo {
            break Loop
        }
    case <-c: {
        if bar {
            continue Loop
        }
        ...
    }
    }
}
```

First, we add a new token for `doselect`.

- `cmd/compile/internal/gc/go.y:/LDOSELECT/`

```
...
%token    <sym>    LTYPE LVAR
%token    <sym>    LDOSELECT
...
```

Then we add it to the lexer.

- `cmd/compile/internal/gc/lex.go:/func._yylex/+/LDOSELECT/`

```
...
case LFOR, LIF, LSWITCH, LSELECT, LDOSELECT:
    loophack = 1 // see comment about loophack above
...
```

- `cmd/compile/internal/gc/lex.go:/^var.syms/+/LDOSELECT/`

```
var syms = ... {
    ...
    {"default", LDEFAULT, Txxx, OXXX},
    {"doselect", LDOSELECT, Txxx, OXXX},
    {"else", LELSE, Txxx, OXXX},
    ...
}
```

- `cmd/compile/internal/gc/lex.go:/^var.lexn/+/LDOSELECT/`

```
var lexn = ... {
    ...
    {LDEFER, "DEFER"},
    {LDOSELECT, "DOSELECT"},
    {LELSE, "ELSE"},
    ...
}
```

- `cmd/compile/internal/gc/lex.go:/^var.yytfix/+/LDOSELECT/`

```
var yytfix = ... {
    ...
    {LDEFER, "DEFER"},
    {LDOSELECT, "DOSELECT"},
    {LELSE, "ELSE"},
    ...
}
```

The grammar is changed to include the construct. A `doselect` is built as a `for` with a `select` in it, but the node for `select` uses `ODOSELECT` instead of `OSELECT`, to let us handle breaks.

- `cmd/compile/internal/gc/go.y:/select_stmtd/`

```
%type    <node>    doselect_stmt  doselect_hdr
```

- `cmd/compile/internal/gc/go.y:/^non_dcl_stmt/`

```
non_dcl_stmt:
    ...
|    select_stmt
|    doselect_stmt
    ...
```

- `cmd/compile/internal/gc/go.y:/^doselect_stmt/`

```
doselect_stmt:
    LDOSELECT
    {
        // for
        markdcl();
    }
    doselect_hdr
    {
        // select
        typesw = Nod(OXXX, typesw, nil);
    }
    LBODY caseblock_list '}'
    {
        // select
        nd := Nod(ODOSELECT, nil, nil);
        nd.Lineno = typesw.Lineno;
        nd.List = $6;
        typesw = typesw.Left;

        // for
        $$ = $3;
        $$.Nbody = list1(nd)
        popdcl();
    }
```

The header works like in a `for` construct, so we can do things like limit the number of loops, etc.

- `cmd/compile/internal/gc/go.y:/ˆdoselect_hdr/`

```
doselect_hdr:
    osimple_stmt ';' osimple_stmt ';' osimple_stmt
    {
        // init ; test ; incr
        if $5 != nil && $5.Colas {
            Yyerror("cannot declare in the doselect-increment");
        }
        $$ = Nod(OFOR, nil, nil);
        if $1 != nil {
            $$.Ninit = list1($1);
        }
        $$.Ntest = $3;
        $$.Nincr = $5;
    }
|    osimple_stmt
    {
        // normal test
        $$ = Nod(OFOR, nil, nil);
        $$.Ntest = $1;
    }
```

A new node `ODOSELECT` is added mainly to handle `break` and `continue` as expected in the new construct.

- `cmd/compile/internal/gc/syntax.go:/OSELECT/`

```
// Node ops.
const (
    OXXX = iota
    ...
    OSELECT   // select
    ODOSELECT // doselect
    ...
)
```

- `cmd/compile/internal/gc/fmt.go:/ˆvar.goopnames/`

```
var goopnames = []string{
    ...
    OSELECT:   "select",
    ODOSELECT: "doselect",
    ...
}
```

- `cmd/compile/internal/gc/fmt.go:/ˆfunc.stmtfmt/+/OSELECT/`

```
func stmtfmt(n *Node) string {
    ...
    case OSELECT, ODOSELECT, OSWITCH:
    ...
}
```

- `cmd/compile/internal/gc/fmt.go:/^var.opprec/`

```
var opprec = []int{
    ...
    OSELECT:     -1,
    ODOSELECT:   -1,
    ...
}
```

This one is generated, but anyway...

- `cmd/compile/internal/gc/opnames.go`

```
...
OSELECT:          "SELECT",
ODOSELECT:        "DOSELECT",
...
```

Now we have to honor the new node. In general, a `ODOSELECT` is to be handled as a `OSELECT` node, because it is already within a `OFOR` node.

- `cmd/compile/internal/gc/inl.go:/^func.ishairy/+/OSELECT/`

```
func ishairy(n *Node, budget *int) bool {
    ...
    case OCLOSURE,
        OCALLPART,
        ORANGE,
        OFOR,
        OSELECT,
        ODOSELECT,
    ...
}
```

- `cmd/compile/internal/gc/order.go:/^func.orderstmt\(/+/OSELECT/`

```
func orderstmt(n *Node, order *Order) {
    ...
    case OSELECT, ODOSELECT:
    ...
}
```

- `cmd/compile/internal/gc/racewalk.go:/^func.racewalknode\(/+/OSE-LECT/`

```
func racewalknode(np **Node, init **NodeList, wr int, skip int) {
    ...
    // just do generic traversal
    case OFOR,
        ...
        OSELECT,
        ODOSELECT,
    ...
}
```

- cmd/compile/internal/gc/typecheck.go:/^func.typecheck1\(/+/OSELECT/

```
func typecheck1(np **Node, top int) {
    ...
    case OSELECT, ODOSELECT:
        ok |= Etop
        typecheckselect(n)
        break OpSwitch
    ...
}
```

- cmd/compile/internal/gc/typecheck.go:/^func.markbreak\(/+/OSELECT/

```
func markbreak(n *Node, implicit *Node) {
    ...
    case OFOR,
        OSWITCH,
        OTYPESW,
        OSELECT,
        ODOSELECT,
        ORANGE:
        implicit = n
        fallthrough
    ...
}
```

- cmd/compile/internal/gc/typecheck.go:/^func.markbreaklist\(/+/OSE-LECT/

```
func markbreaklist(...) {
    ...
    case OFOR,
        OSWITCH,
        OTYPESW,
        OSELECT,
        ODOSELECT,
        ORANGE:
    ...
}
```

- cmd/compile/internal/gc/typecheck.go:/^func.isterminating\(/+/OSE-LECT/

```
func isterminating(...) {
    ...
    case OSWITCH, OTYPESW, OSELECT, ODOSELECT:
        if n.Hasbreak {
            return false
        }
    ...
    if n.Op != OSELECT && n.Op != ODOSELECT && def == 0 {
        return false
    }
}
```

- `cmd/compile/internal/gc/walk.go:/^func.walkstmt\(/+/OSELECT/`

```
func walkstmt(np **Node) {
    ...
    case OSELECT, ODOSELECT:
        walkselect(n)
    ...
}
```

- `cmd/compile/internal/gc/gen.go:/^func.gen\(/+/OSELECT/`

```
func gen(n *Node) {
    ...
    if n.Defn != nil {
        switch n.Defn.Op {
        // so stmtlabel can find the label
        case OFOR, OSWITCH, OSELECT, ODOSELECT:
            n.Defn.Sym = lab.Sym
        }
    }
    ...
}
```

And this is the main change for a `ODOSELECT`. It works like a `select` but does not redefine the user break PC, so that breaks and continues always refer to the enclosing, implicit, `for` loop.

The idea is that implicit `breaks` inserted by the compiler will not be `OBREAK`, but `OCBREAK`. The new `OCBREAK` is a compiler-inserted break and `gen.go` can skip those breaks when jumping on `break` and `continue` within `doselect` structures.

- `cmd/compile/internal/gc/syntax.go:/OBREAK`

```
// Node ops.
const (
    OXXX = iota
    ...
    OBREAK    // break
    OCBREAK    // break generated by the compiler
```

- `cmd/compile/internal/gc/opnames.go:/OCBREAK`

```
...
OBREAK:           "BREAK",
OCBREAK:          "CBREAK",
...
```

- cmd/compile/internal/gc/fmt.go:/^var.goopnames/

```
var goopnames = []string{
    ...
    OBREAK:    "break",
    OCBREAK:    "break",
    ...
}
```

- cmd/compile/internal/gc/fmt.go:/^func.stmtfmt/

```
func stmtfmt(n *Node) string {
    ...
    case OBREAK, OCBREAK,
        OCONTINUE,
        OGOTO,
        OFALL,
        OXFALL:
    ...
}
```

- cmd/compile/internal/gc/fmt.go:/^var.opprec/

```
var opprec = []int{
    ...
    OBREAK:      -1,
    OCBREAK:      -1,
    ...
}
```

In `select` we insert `OCBREAK` nodes instead of `OBREAK`, which are now left for the user breaks.

- cmd/compile/internal/gc/select.go:/^func.racewalknode/

```
func walkselect(sel *Node) {
    ...
    r.Nbody = concat(r.Nbody, cas.Nbody)
    r.Nbody = list(r.Nbody, Nod(OCBREAK, nil, nil))
    init = list(init, r)
    ...
}
```

The same must be done in `swt` for switches.

- cmd/compile/internal/gc/swt.go:/^func.casebody/

```
func casebody(sw *Node, typeswvar *Node) {
    ...
    var cas *NodeList  // cases
    var stat *NodeList // statements
    var def *Node      // defaults
    br := Nod(OCBREAK, nil, nil)
    ...
}
```

- `cmd/compile/internal/gc/swt.go:/^func.*exprswitch.*walk/`

```
func (s *exprSwitch) walk(sw *Node) {
    ...
    if len(cc) > 0 && cc[0].typ == caseKindDefault {
        def = cc[0].node.Right
        cc = cc[1:]
    } else {
        def = Nod(OCBREAK, nil, nil)
    }
    ...
}
```

- `cmd/compile/internal/gc/swt.go:/^func.*typeSwitch.*walk/`

```
func (s *typeSwitch) walk(sw *Node) {
    ...
    if len(cc) > 0 && cc[0].typ == caseKindDefault {
        def = cc[0].node.Right
        cc = cc[1:]
    } else {
        def = Nod(OCBREAK, nil, nil)
    }
    ...
}
```

And almost all processing is shared with the user `OBREAK` node.

- `cmd/compile/internal/gc/order.go:/^func.orderstmt/`

```
func orderstmt(n *Node, order *Order) {
    ...
    case OBREAK, OCBREAK,
        OCONTINUE,
        ODCL,
        ODCLCONST,
    ...
}
```

- `cmd/compile/internal/gc/racewalk.go:/^func.racewalknode/`

```
func racewalknode(...) {
    ...
    case OFOR,
        OBREAK,
        OCBREAK,
        OCONTINUE,
    ...
}
```

- `cmd/compile/internal/gc/typecheck.go:/^func.typecheck1/`

```
func typecheck1(np **Node, top int) {
    ...
    case OBREAK,
        OCBREAK,
        OCONTINUE,
    ...
}
```

- `cmd/compile/internal/gc/typecheck.go:/^func.markbreak/`

```
func markbreak(n *Node, implicit *Node) {
    ...
    switch n.Op {
    case OBREAK, OCBREAK:
    ...
}
```

- `cmd/compile/internal/gc/walk.go:/^func.markbreak/`

```
func func walkstmt(np **Node) {
    ...
    case OBREAK,
        OCBREAK,
        ODCL,
    ...
}
```

Here is where things start to change. A new `ubreakpc` records the PC for user (not compiler) breaks.

- `cmd/compile/internal/gc/go.go:/^var.breakpc/`

```
var breakpc, ubreakpc *obj.Prog
```

- `cmd/compile/internal/gc/pgen.go:/^func.compile/+/breakpc/`

```
func compile(fn *Node) {
    ...
    continpc = nil
    breakpc = nil
    ubreakpc = nil
    ...
}
```

The code in `gen` is changed now so that `ubreakpc` is recorded for user breaks but not for compiler-inserted breaks.

The processing for `OBREAK` and `OCBREAK` differs in the `breakpc` used (which is be `ubreakpc` for user breaks).

Processing for `ODOSELECT` is like that for `OSELECT` but does not redefine the user break, so that breaks and continues refer to the enclosing for loop inserted by the compiler.

- `cmd/compile/internal/gc/gen.go:/^func.gen/+/^.case.OBREAK/`

```
case OBREAK, OCBREAK:
    ...
    if breakpc == nil || ubreakpc == nil {
        Yyerror("break is not in a loop")
        break
    }
    if n.Op == OBREAK {
        gjmp(ubreakpc)
    } else {
        gjmp(breakpc)
    }
```

- `cmd/compile/internal/gc/gen.go:/^func.gen/+/^.case.OFOR/`

```
case OFOR:
    sbreak, subreak := breakpc, ubreakpc
    p1 := gjmp(nil)     //        goto test
    breakpc = gjmp(nil) // break:    goto done
    ubreakpc = breakpc
    ...
    Patch(breakpc, Pc) // done:
    Patch(ubreakpc, Pc) // done:
    continpc = scontin
    breakpc, ubreakpc = sbreak, subreak
    if lab != nil {
        lab.Breakpc = nil
        lab.Continpc = nil
    }
```

- `cmd/compile/internal/gc/gen.go:/^func.gen/+/^.case.OSWITCH/`

```
case OSWITCH:
    sbreak, subreak := breakpc, ubreakpc
    p1 := gjmp(nil)     //        goto test
    breakpc = gjmp(nil) // break:    goto done
    ubreakpc = breakpc
    // define break label
    lab := stmtlabel(n)
    if lab != nil {
        lab.Breakpc = breakpc
    }

    Patch(p1, Pc)      // test:
    Genlist(n.Nbody)   //        switch(test) body
    Patch(breakpc, Pc) // done:
    Patch(ubreakpc, Pc) // done:
    breakpc, ubreakpc = sbreak, subreak
    if lab != nil {
        lab.Breakpc = nil
    }
```

- `cmd/compile/internal/gc/gen.go:/^func.gen/+/^.case.OSELECT/`

```
case OSELECT, ODOSELECT:
    sbreak, subreak := breakpc, ubreakpc
    p1 := gjmp(nil)     //        goto test
    breakpc = gjmp(nil) // break:    goto done
    if n.Op == OSELECT {
        ubreakpc = breakpc
    }
    // define break label
    lab := stmtlabel(n)
    if lab != nil {
        lab.Breakpc = breakpc
    }

    Patch(p1, Pc)      // test:
    Genlist(n.Nbody)   //        select() body
    Patch(breakpc, Pc) // done:
    breakpc = sbreak
    if n.Op == OSELECT {
        Patch(ubreakpc, Pc) // done:
        ubreakpc = subreak
    }
    if lab != nil {
        lab.Breakpc = nil
    }
```

7. Implicit structure and interface declarations

This is yet another convenience change, added because we already had to change the compiler.

In most cases types are `struct` types. It can be easy for the compiler in certain cases to assume that a type declaration where the `struct` keyword is missing is a `struct` type declaration. We assume that a structure is declared if we see something like

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

while a type is declared (i.e., in the `typedcl` node of the grammar).

In the same way, because `interface{}` is a very popular type for channels in Clive, the `interface` keyword can be removed when declaring the type for a channel. These two are equivalent:

```
chan {}
chan interface{}
```

The changes in the grammar are as shown here.

- `cmd/compile/internal/gc/go.y`

```
%type    <node>    implstructtype implinterfacetype
...

typedcl:
    typedclname ntype
    {
        $$ = typedcl1($1, $2, true);
    }
|
    typedclname implstructtype
    {
        $$ = typedcl1($1, $2, true);
    }
...

implstructtype:
    lbrace structdcl_list osemi '}'
    {
        $$ = Nod(OTSTRUCT, nil, nil);
        $$.List = $2;
        fixlbrace($1);
    }
|    lbrace '}'
    {
        $$ = Nod(OTSTRUCT, nil, nil);
        fixlbrace($1);
    }
...

implinterfacetype:
    lbrace '}'
    {
        $$ = Nod(OTINTER, nil, nil);
        fixlbrace($1);
    }
...
```

```
othertype:
    ...
|    LCHAN non_recvchantype
    {
        $$ = Nod(OTCHAN, $2, nil);
        $$.Etype = Cboth;
    }
|    LCHAN LCOMM ntype
    {
        $$ = Nod(OTCHAN, $3, nil);
        $$.Etype = Csend;
    }
|    LCHAN implinterfacetype
    {
        $$ = Nod(OTCHAN, $2, nil);
        $$.Etype = Cboth;
    }
|    LCHAN LCOMM implinterfacetype
    {
        $$ = Nod(OTCHAN, $3, nil);
        $$.Etype = Csend;
    }
...

recvchantype:
    LCOMM LCHAN ntype
    {
        $$ = Nod(OTCHAN, $3, nil);
        $$.Etype = Crecv;
    }
|
    LCOMM LCHAN implinterfacetype
    {
        $$ = Nod(OTCHAN, $3, nil);
        $$.Etype = Crecv;
    }
```

8. Go package and Go tools

Previous changes should suffice, given that the compiler is now written in Go. However, there is a `go` package that contains yet another parser for the language, and it has to be changed as well. Most Go tools (commands) use it, and we must update it.

8.1. Channel sends

We must add `<-` in the predecende table. To preserve the levels, hardwired into `gofmt`, we set for the send operation the lowest one.

- `/usr/local/go/src/go/token/token.go:/^.LowestPrec`

```
const (
    LowestPrec  = 0 // non-operators
    UnaryPrec   = 6
    HighestPrec = 7
)

func (op Token) Precedence() int {
    switch op {
    case ARROW, LOR:
        return 1
    case LAND:
        return 2
    case EQL, NEQ, LSS, LEQ, GTR, GEQ:
        return 3
    case ADD, SUB, OR, XOR:
        return 4
    case MUL, QUO, REM, SHL, SHR, AND, AND_NOT:
        return 5
    }
    return LowestPrec
}
```

8.2. Looping selects

The main change id adding `DOSELECT` as a new token.

- `/usr/local/go/src/go/token/token.go`

```
// The list of tokens.
const (
    ...
    DEFAULT
    DEFER
    DOSELECT
    ELSE
    FALLTHROUGH
    FOR
    ...
)

var tokens = [...]string{
    ...
    DEFAULT:     "default",
    DEFER:       "defer",
    DOSELECT:    "doselect",
    ELSE:        "else",
    FALLTHROUGH: "fallthrough",
    FOR:         "for",
    ...
}
```

The AST must include a `DoSelectStmt`.

- `/usr/local/go/src/go/ast/ast.go:/^.DoSelectStmt`

```
// A DoSelectStmt node represents a doselect statement.
DoSelectStmt struct {
    DoSelect token.Pos  // position of "doselect" keyword
    Init Stmt           // initialization statement; or nil
    Cond Expr           // condition; or nil
    Post Stmt           // post iteration statement; or nil
    Body   *BlockStmt   // CommClauses only
}
```

And its methods...

- `/usr/local/go/src/go/ast/ast.go`

```
...
func (s *SelectStmt) Pos() token.Pos     { return s.Select }
func (s *DoSelectStmt) Pos() token.Pos   { return s.DoSelect }
...
func (s *SelectStmt) End() token.Pos { return s.Body.End() }
func (s *DoSelectStmt) End() token.Pos { return s.Body.End() }
...
func (*SelectStmt) stmtNode()     {}
func (*DoSelectStmt) stmtNode()   {}
```

Plus a `walk` for it.

- `/usr/local/go/src/go/ast/walk.go`

```
func Walk(v Visitor, node Node) {
    ...
    case *DoSelectStmt:
        if n.Init != nil {
            Walk(v, n.Init)
        }
        if n.Cond != nil {
            Walk(v, n.Cond)
        }
        if n.Post != nil {
            Walk(v, n.Post)
        }
        Walk(v, n.Body)
    case *ForStmt:
    ...
}
```

Then the parser. There is a new statement to synchronize on errors.

- `/usr/local/go/src/go/parser/parser.go:/^func.syncStmt\(`

```
func syncStmt(p *parser) {
    for {
        switch p.tok {
        case token.BREAK, ...
            token.DOSELECT, ...
            token.VAR:
        ...
        case token.EOF:
            return
        }
        p.next()
    }
}
```

And there is a new statement.

• `/usr/local/go/src/go/parser/parser.go:/ˆfunc.parseStmt\(`

```
func (p *parser) parseStmt() (s ast.Stmt) {
    ...
    case token.SELECT:
        s = p.parseSelectStmt()
    case token.DOSELECT:
        s = p.parseDoSelectStmt()
    ...
}
```

The parsing is taken from the parsing of a `for` header and a `select` body.

• `/usr/local/go/src/go/parser/parser.go:/ˆfunc.parseStmt\(`

```
func (p *parser) parseDoSelectStmt() *ast.DoSelectStmt {
    if p.trace {
        defer un(trace(p, "DoSelectStmt"))
    }
    pos := p.expect(token.DOSELECT)
    p.openScope()
    defer p.closeScope()

    var s1, s2, s3 ast.Stmt
    if p.tok != token.LBRACE {
        prevLev := p.exprLev
        p.exprLev = -1
        if p.tok != token.SEMICOLON {
            isRange := false
            if p.tok == token.RANGE {
                isRange = true
            } else {
                s2, isRange = p.parseSimpleStmt(basic)
            }
            if isRange {
                p.error(pos, "unexpected range")
                // but ignore it for now
            }
        }
        if p.tok == token.SEMICOLON {
            p.next()
            s1 = s2
            s2 = nil
            if p.tok != token.SEMICOLON {
                s2, _ = p.parseSimpleStmt(basic)
            }
            p.expectSemi()
            if p.tok != token.LBRACE {
                s3, _ = p.parseSimpleStmt(basic)
            }
        }
        p.exprLev = prevLev
    }

    lbrace := p.expect(token.LBRACE)
    var list []ast.Stmt
    for p.tok == token.CASE || p.tok == token.DEFAULT {
        list = append(list, p.parseCommClause())
    }
    rbrace := p.expect(token.RBRACE)
    p.expectSemi()
    body := &ast.BlockStmt{Lbrace: lbrace, List: list, Rbrace: rbrace}

    return &ast.DoSelectStmt {
        DoSelect: pos,
        Init: s1,
        Cond: p.makeExpr(s2, "boolean expression"),
        Post: s3,
        Body: body,
```

```
    }
}
```

Now we can print it.

- `/usr/local/go/src/go/printer/nodes.go:/^func.*printer.*stmt\(/`

```
func (p *printer) stmt(stmt ast.Stmt, nextIsRBrace bool) {
    ...
    case *ast.DoSelectStmt:
        p.print(token.DOSELECT, blank)
        p.controlClause(true, s.Init, s.Cond, s.Post)
        body := s.Body
        if len(body.List) == 0 && !p.commentBefore(p.posFor(body.Rbrace)) {
            // print empty select statement w/o comments on one line
            p.print(body.Lbrace, token.LBRACE, body.Rbrace, token.RBRACE)
        } else {
            p.block(body, 0)
        }
    ...
}
```

8.3. Implicit keywords

We are going to flag `StructType` for implicit `struct` and `interface` declarations.

- `/usr/local/go/src/go/ast/ast.go:/^.StructType`

```
// A StructType node represents a struct type.
StructType struct {
    Struct     token.Pos  // position of "struct" keyword
    Fields     *FieldList // list of field declarations
    Incomplete bool
    Implicit bool
}
```

- `/usr/local/go/src/go/ast/ast.go:/^.InterfaceType`

```
// An InterfaceType node represents an interface type.
InterfaceType struct {
    Interface  token.Pos  // position of "interface" keyword
    Methods    *FieldList // list of methods
    Incomplete bool
    Implicit bool
}
```

Globals in the parser records if we can accept implicit keywords.

- `/usr/local/go/src/go/parser/parser.go:/^type.parser`

```
type parser struct {
    ...
    implStructOk, implInterOk bool
}
```

In a global type declaration, we accept `struct` to be implicit. This is not exactly what the Go compiler does, but it is close enough.

- `/usr/local/go/src/go/parser/parser.go:/^func.*parser.*parseDecl\(`

```
func (p *parser) parseDecl(sync func(*parser)) ast.Decl {
    if p.trace {
        defer un(trace(p, "Declaration"))
    }
    p.implStructOk = false
    defer func() {p.implStructOk = false}()
    var f parseSpecFunction
    switch p.tok {
    ...
    case token.TYPE:
        p.implStructOk = true
        f = p.parseTypeSpec
    ...
    }
    return p.parseGenDecl(p.tok, f)
}
```

- `/usr/local/go/src/go/parser/parser.go:/^func.*parser.*parseGenDecl\(`

```
func (p *parser) parseGenDecl(...) *ast.GenDecl {
    ...
    old := p.implStructOk
    for ... {
        p.implStructOk = old
        list = append(...)
    }
    ...
}
```

Later, `parseStructType` can honor the flag.

- `/usr/local/go/src/go/parser/parser.go:/^func.*parser.*parseStructType\(`

```
func (p *parser) parseStructType() *ast.StructType {
    if p.trace {
        defer un(trace(p, "StructType"))
    }
    var pos, lbrace token.Pos
    implicit := p.implStructOk
    if implicit  && p.tok == token.LBRACE {
        pos = p.expect(token.LBRACE)
        lbrace = pos
    } else {
        pos = p.expect(token.STRUCT)
        lbrace = p.expect(token.LBRACE)
    }
    old := p.implStructOk
    p.implStructOk = false
    defer func() {p.implStructOk = old}()

    scope := ast.NewScope(nil) // struct scope
    ...
    return &ast.StructType{
        Struct: pos,
        Fields: &ast.FieldList{
            Opening: lbrace,
            List:    list,
            Closing: rbrace,
        },
        Implicit: implicit,
    }
}
```

The flag is saved, cleared, and restored to prevent implicit `struct` declarations anywhere but at the top-level.

To accept implicit `interface` declarations, we set the flag while declaring a channel type.

- `/usr/local/go/src/go/parser/parser.go:/^func.*parser.*parseChanType\(`

```
func (p *parser) parseChanType() *ast.ChanType {
    ...
    p.implInterOk = true
    value := p.parseType()
    p.implInterOk = false
    ...
}
```

And `parseInterfaceType` takes care of the flag.

- `/usr/local/go/src/go/parser/parser.go:/^func.*parser.*parseInterfaceType\(`

```
func (p *parser) parseInterfaceType() *ast.InterfaceType {
    if p.trace {
        defer un(trace(p, "InterfaceType"))
    }
    var pos, lbrace token.Pos
    implicit := p.implInterOk
    if implicit  && p.tok == token.LBRACE {
        pos = p.expect(token.LBRACE)
        lbrace = pos
    } else {
        pos = p.expect(token.INTERFACE)
        lbrace = p.expect(token.LBRACE)
    }
    p.implInterOk = false
    scope := ast.NewScope(nil) // interface scope
    var list []*ast.Field
    for p.tok == token.IDENT {
        list = append(list, p.parseMethodSpec(scope))
    }
    if implicit && len(list) > 0 {
        p.error(pos, "ok only for empty interfaces")
    }
    rbrace := p.expect(token.RBRACE)
    return &ast.InterfaceType{
        Interface: pos,
        Methods: &ast.FieldList{
            Opening: lbrace,
            List:    list,
            Closing: rbrace,
        },
        Implicit: implicit,
    }
}
```

This time we clear the flag right after using it, because the implicit interface declaration works only right after the `chan` keyword (but for send/receive only indications).

In the printer, we define

- `/usr/local/go/src/go/printer/printer.go:/^type.Config`

```
type Config struct {
    Mode     Mode // default: 0
    Tabwidth int  // default: 8
    Indent   int  // default: 0 (all code is indented at least by this much)
    DontPrintImplicits bool
}
```

The flag `DontPrintImplicits` may be set by the code using this package to instruct nodes not to print the implicit keywords. By default, they are printed.

The `gofmt` command is given a flag to set it.

- `/usr/local/go/src/cmd/gofmt/gofmt.go`

```
var noImpls = flag.Bool("S", false,
    "omit struct keyword in top-level type declarations")
```

And to process file...

- `/usr/local/go/src/cmd/gofmt/gofmt.go:/^func.processFile`

```
func processFile(...) error {
    cfg := printer.Config{..., DontPrintImplicits: noImpls}
    res, err := format.Format(..., cfg)
}
```

References

1. The Go Programming Language. The Go Authors. http://golang.org.