

УДК *Lorem ipsum dolor sit amet*

Операционная семантика выражений в языке Go на языке ABML

Харьков А.А. (Студент ММФ НГУ)

Ануреев И.С. (Институт систем информатики СО РАН)

Lorem ipsum dolor sit amet

1. Введение

Lorem ipsum dolor sit amet

2. Модель объектов языка Go

Прежде чем разбираться с моделированием действий различных программ или их составляющих, необходимо предварительно создать модель для элементов, составляющих язык Go, которые явно присутствуют в коде программы: переменные, функции, операторы, декларации и т.д. Далее, используя язык ABML, будет описано создание моделей соответствующих объектов языка Go.

Классификация и порядок изложения опираются на официальную спецификацию языка Go. Большая часть названий объектов, моделирующихся в этом разделе, будет совпадать с их названиями в официальной спецификации для удобства восприятия и облегчения поиска дополнительной информации по каждому из них. Однако некоторые сущности были признаны излишними с точки зрения моделирования их функциональности, а некоторые имена недостаточно отражающими суть описываемого объекта. Это вызвало некоторую разницу в именовании нескольких объектов в построенной модели.

2.1. Лексические элементы

Ниже представлено создание моделей для таких объектов, как комментарий "`comment`" - класс объектов, включающий в себя одиночные комментарии "`line comment`" и многострочные комментарии "`general comment`". А так же идентификатор "`identifier`", моделирующий имя объекта программы, например переменной, функции и т.д.

```

1 (typedef "comment" (uniont "line comment" "general comment"))
2 (mot "line comment" :at "text" string) ; // single-line
3 (mot "general comment" :at "text" string) ; /* multi-line */
4 (typedef "identifier" (uniont string))

```

2.2. Константы

Данный подраздел описывает модели объектов, представляющих постоянные значения, допускающие взаимодействие с программой: логические константы и числа, в том числе комплексные.

```

1 (typedef "constant" (uniont "bool constant" "numeric constant"
2   string))
3 (typedef "bool constant" (enumt "true" "false"))
4 (typedef "numeric constant" (uniont int real "complex constant"))
5 (mot "complex constant" :at "re" real :at "im" real)

```

2.3. Типы данных

В данном подразделе моделируются типы данных, большая часть которых должна быть интуитивно понятна из названий. Типы можно разделить на базовые - представляющие некоторые числа, символы или строки, и составные типы данных - состоящие зачастую совокупность базовых или моделирующих более сложные концепции хранения данных.

Сперва рассмотрим базовые типы данных.

```

1 (typedef "type" (uniont "base type" "composite type"))
2 (typedef "base type" (uniont "boolean type" "numeric type"
3   "string type" "rune type"))
4 (typedef "boolean type" (enumt "bool"))
5 (typedef "numeric type" (uniont "real-valued type" "complex
6   type" "byte type"))
7 (typedef "real-valued type" (enumt "int type" "float type"))
8 (typedef "int type" (uniont "signed int type" "unsigned int
9   type"))

```

```

7 (typedef "signed int type" (enumt "int" "int8" "int16" "int32"
     "int64"))
8 (typedef "unsigned int type" (enumt "uint" "uint8" "uint16" "
      uint32" "uint64" "uintptr"))
9 (typedef "float type" (enumt "float32" "float64"))
10 (typedef "complex type" (enumt "complex32" "complex64"))
11 (typedef "byte type" (enumt "byte"))
12 (typedef "rune type" (enumt "rune")) ; 'a' | '\t' | '\377',
13 (typedef "string type" (enumt "string"))

```

Теперь несколько слов про каждый из типов выше. **"boolean type"** - моделирует логический тип данных, **"real-valued type"** - вещественные числа, **"complex type"** - комплексные числа, **"byte type"** - двоичное представление чисел, **"rune type"** состоит из символов: буква, цифра, специальный символ и т.д., **"string type"** включает в себя любые строки.

Далее перейдём к рассмотрению составных типов данных.

```

1 (typedef "composite type" (uniont "array type" "slice type"
        "struct type" "pointer type" "function type" "interface type"
        "map type" "channel type"))

```

Начнём с простейших из них.

```

1 (mot "array type" :at "len" nat :at "element type" "type")
2 (mot "slice type" :at "element type" "type")
3 (mot "struct type" :at "fields" (listt "field decl"))
4 (typedef "field decl" (uniont "named field" "embedded field"))
5 (mot "named field" :at "name" "identifier" :at "type" "type")
6 (mot "embedded field" :at "type" "type")
7 (mot "pointer type" :at "type" "type")

```

Тип список **"array type"** - структура данных, которую можно описать как упорядоченный набор элементов одного типа.

Тип срез **"slice type"** - ссылка на последовательную часть исходного массива.

Тип структура **"struct type"** позволяет создавать новые объекты, состоит из множе-

ства полей. Поля структуры могут быть именованными "named field" или встроенными "embedded field".

Именованные поля представляют собой пару имя-значение, устроенные аналогично переменным. Встроенные поля обычно являются другими структурами, таким образом позволяя встраивать все поля вложенной структуры в текущую.

Тип указатель "pointer type" - ссылка на исходный объект.

Далее речь пойдёт о функциях.

```

1 (mot "function type" :at "signature" "signature")
2 (mot "signature" :at "parameters" (listt "parameter decl") :at
   "result" "function result")
3 (mot "parameter decl" :at "names" (listt "identifier") :at
   "type" (uniont "type" "variadic type"))
4 (mot "variadic type" :at "type" "type") ; func f(numbers
   ...int) -> f(1, 2, 3, 4)
5 (typedef "function result" (uniont (listt "parameter decl")
   "type"))

```

Тип функция "function type" характеризует своей сигнатурой "signature", которая в свою очередь является совокупностью параметров "parameter decl" и типом результата функции "function result".

В следующем блоке определяется интерфейс и связанные с ним объекты.

```

1 (mot "interface type" :at "elements" (listt "interface elem"))
2 (typedef "interface elem" (uniont "method elem" "type elem"))
3 (mot "method elem" :at "name" "identifier" :at "signature"
   "signature")
4 (typedef "type elem" (uniont (listt "type term")))
5 (typedef "type term" (uniont "type" "underlying type"))
6 ; type A interface {
7 ;   f(n int) (q bool) // method elem: "name" = 'f',
8 ; } // "signature" = '(n int) (q bool)'

```

Тип интерфейс "interface type" состоит из элементов-методов и элементов-типов.

Тип элементы-методы "method elem" представляют собой пару из имени функции и её

сигнатуры.

Тип элементы-типы "type elem" обычно являются описанными ранее интерфейсами, что позволяет включить в элементы текущего интерфейса все элементы описанного ранее.

Осталось определить ещё несколько типов данных.

```

1 (mot "underlying type" :at "type" "type") ; ~Type
2 (mot "map type" :at "key type" "type" :at "element type" "type")
3 (mot "channel type" :at "direction" "direction" :at "type"
      "type")
4 (typedef "direction" (enumt "bidirectional" "send" "receive"))

```

Объемлющий тип "underlying type" определяется следующим образом: если это логический, числовой или сточный тип, то объемлющим для него является он сам; иначе, за некоторым исключением, объемлющим для него является тип, на который он ссылается в своём объявлении.

Тип словарь "map type" является реализацией хэш-таблицы.

Тип канал "channel type" используется преимущественно в многопоточных процессах. Каналы можно представить как переменную с дополнительной особенностью - при считывании значения "пустой" ячейки происходит блокирование процесса до тех пор, пока другой процесс не положит в данную ячейку какое-либо значение. Причём каждый канал имеет своё направление "direction" - некоторые каналы предназначены только для записи или только для получения данных из него, или всё вместе.

2.4. Блоки

Этот подраздел посвящён только одному типу блоков "block", в том числе по причине обособленности с точки зрения описания функциональности.

```

1 (mot "block"
2   :at "statements" (listt "statement")
3   ; semantic attributes
4   :at "variables" (listt "identifier")
5   :at "variable location" (cot :amap "identifier"
6     "location")
7   :at "labels" (listt "label")

```

```

7      :at "label position" (cot :амар "label" nat)
8 )

```

Атрибут "**statements**" представляет собой список команд из блока, переведённых в модельное представление.

Следующие атрибуты не являются необходимыми для представления типа "**block**", однако упрощают реализацию действий некоторых команд, включающих в себя взаимодействие с блоками. Поэтому допустимо называть следующие атрибуты семантическими.

Атрибут "**variables**" включает в себя все переменные, значения которых меняются в ходе исполнения блока.

Каждый элемент "**variable location**" является ячейкой памяти до начала исполнения блока для переменной из списка выше. Если переменная с соответствующим названием раньше не существовала, то элементу присваивается значение **nil**.

Атрибут "**labels**" содержит в себе все метки, встречающиеся в теле блока, каждой из которых соответствует позиция в блоке, хранящаяся в атрибуте "**label position**".

2.5. Декларации

Настало время определиться с декларированием различных объектов. Начнём с обобщения до понятия декларации объекта "**declaration**".

```

1 (typedef "declaration" (unionint "const decl" "type decl" "var
  decl"))

```

Дальше будет описано декларирование постоянных величин (констант) "**const decl**".

```

1 (mot "const decl" :at "specifiers" (listt "const spec"))
2 (mot "const spec" :at "names" (listt "identifier") :at "type"
  "type" :at "initializers" (listt "expression"))

```

Далее опишем декларирование типов "**type decl**"

```

1 (typedef "type decl" (unionint (listt (unionint "alias decl" "type
  def"))))
2 (mot "alias decl" :at "name" "identifier" :at "type parameters"
  "type parameters" :at "type" "type")
3 ; type (

```

```

4 ;           nodeList = []*Node
5 ;           Polar     = polar
6 ;
7 (mot "type def" :at "name" "identifier" :at "type parameters"
8   "type parameters" :at "type" "type")
9 ; type (
10 ;       Point struct{ x, y float64 }
11 ;       polar Point
12 ;
13 (mot "type parameters" :at "param declarations" (listt "type"
14   "param decl"))
15 (mot "type param decl" :at "names" (listt "identifier") :at
16   "type constraint" "type constraint")
17 (mot "type constraint" :at "constraints" "type elem") ; [T1
18   comparable, T2 any]

```

Теперь опишем декларацию переменных, разделяя её на 2 типа: полная декларация "var decl" с явным указанием типа переменной, и короткая декларация "short var decl", тип которой устанавливается по соответствующему начальному значению при декларации.

```

1 (mot "var decl" :at "specifiers" (listt "var spec"))
2 (mot "var spec" :at "names" (listt "identifier") :at "type"
3   "type" :at "expressions" (listt "expression"))
4 ; var x, y float32 = -1, -2
5 (mot "short var decl" :at "names" (listt "identifier") :at
6   "expressions" (listt "expression"))
7 ; f := func() int { return 7 }

```

Наконец, перейдём к декларированию функциональных объектов: функций "function decl" и методов "method decl". Причём, можно отметить, что функционально вторые ничем не отличаются от первых.

```

1 (mot "function decl" :at "name" "identifier" :at "type"
2   "parameters" "type parameters" :at "signature" "signature" :at

```

```

"body" "block")
2 ; func min[T ~int|~float64](x, y T) T {
3 ;     if x < y {
4 ;         return x
5 ;     }
6 ;     return y
7 ;
8 (mot "method decl" :at "reciever" (listt "parameter decl") :at
9   "name" "identifier" :at "signature" "signature" :at "body"
10  "block")
11 ; func (p *Point) Scale(factor float64) {
12 ;     p.x *= factor
13 ;     p.y *= factor
14 ;

```

2.6. Выражения

Данный раздел посвящён моделированию выражений - некоторых команд, что после исполнения возвращают некоторое значение. В соответствии с официальной спецификацией языка начнём с описания операндов.

```

1 (typedef "operand" (uniont "literal" "operand[T]"
2   "(expression)"))
3 (typedef "literal" (uniont "constant" "composite literal"
4   "function literal"))
5 (mot "composite literal" :at "type" "literal type" :at "value"
6   (listt "keyed element")); It construct new values for
7   structs, arrays, slices, and maps each time they are evaluated
8 (mot "keyed element" :at "key" "key" :at "element" "element")
9 (typedef "literal type" (uniont "struct type" "array type"
10  "slice type" "map type"))
11 (typedef "key" (uniont "field name" "expression" (listt "keyed
12  element")))

```

```

7  (typedef "element" (uniont "expression" (listt "keyed element")))
8  (mot "function literal" :at "signature" "signature" :at "body"
      "block") ; func(a, b int, z float64) bool { return a*b <
      int(z) }
9  (mot "operand[T]" :at "name" "identifier" :at "types" (listt
      "type")) ; substituting types into a template
10 (mot "(expression)" :at "expression" "expression")

```

Далее, будут описаны простейшие выражения "**primary expression**", которые впоследствии будут использованы для моделирования более сложных выражений.

```

1  (typedef "primary expression" (uniont "operand" "conversion"
      "method expr" "selector expression" "index expr" "slice expr"
      "type assertion" "function call"))
2  (mot "conversion" :at "type" "type" :at "expression"
      "expression") ; (*Point)(p) // p is converted to *Point
3  (mot "method expression" :at "receiver type" "type" :at "name"
      "identifier") ; T.Mv
4  (mot "selector expression" :at "expression" "primary expression"
      :at "selector" "identifier") ; x.id
5  (mot "index expr" :at "list" "primary expression" :at "index"
      "expression") ; x[e]
6  (mot "slice expr" :at "list" "primary expression" :at "slice"
      "slice")
7  (mot "slice" :at "low" "expression" :at "high" "expression" :at
      "max" "expression") ; x[e1 : e2] | x[e1 : e2 : e3]
8  (mot "type assertion" :at "expression" "primary expression" :at
      "type" "type") ; .(Type)
9  (mot "function call" :at "function" "primary expression" :at
      "arguments" (listt "argument")) ; math.Sin(2)
10 (mot "argument" :at "expressions" (listt "expression") :at
      "type" "type") ; func f(a, b int, c, d float64)

```

Теперь можно переходить к описанию полноценных выражений. Начнём, как обычно,

с общего понятия выражения "expression"

```
1 (typedef "expression" (uniont "unary expression" "binary
expression"))
```

Далее описано создание моделей для унарных выражений "unary expression" - простейших выражений и унарных операторов с аргументом.

```
1 (typedef "unary expression" (uniont "primary expression" "+1"
"-1" "^1" "!1" "*1" "&1" "<-"))
2 (mot "+1" :at 1 "unary expression") ; +x == x
3 (mot "-1" :at 1 "unary expression") ; -x == -1 * x
4 (mot "^1" :at 1 "unary expression") ; ^10(2) == 01(2)
5 (mot "!1" :at 1 "unary expression") ; !true == false
6 (mot "*1" :at 1 "unary expression")
7 (mot "&1" :at 1 "unary expression")
8 (mot "<-1" :at 1 "unary expression")
```

Теперь перейдём к описанию бинарных выражений - бинарные операторы с двумя аргументами.

```
1 (typedef "binary expression" (uniont "1||2" "1&&2" "rel
expression" "add expression" "mul expression"))
2 (mot "1||2" :at 1 "expression" :at 2 "expression")
3 (mot "1&&2" :at 1 "expression" :at 2 "expression")
```

Следующий подтип бинарных выражений можно связать с операцией умножения.

```
1 (typedef "mul expression" (uniont "1*2" "1/2" "1%2" "1<<2"
"1>>2" "1&2" "1&^2"))
2 (mot "1*2" :at 1 "expression" :at 2 "expression")
3 (mot "1/2" :at 1 "expression" :at 2 "expression")
4 (mot "1%2" :at 1 "expression" :at 2 "expression")
5 (mot "1<<2" :at 1 "expression" :at 2 "expression")
6 (mot "1>>2" :at 1 "expression" :at 2 "expression")
7 (mot "1&2" :at 1 "expression" :at 2 "expression")
```

```

8 | (mot "1&^2" :at 1 "expression" :at 2 "expression") ; 11001010(2)
   &^ 10101100(2) == 01000010(2)      // 1 <=> 1 &^ 0

```

Следующий подтип бинарных выражений можно связать с операцией сложения.

```

1 | (typedef "add expression" (uniont "1+2" "1-2" "1|2" "1^2"))
2 | (mot "1+2" :at 1 "expression" :at 2 "expression")
3 | (mot "1-2" :at 1 "expression" :at 2 "expression")
4 | (mot "1|2" :at 1 "expression" :at 2 "expression")
5 | (mot "1^2" :at 1 "expression" :at 2 "expression")

```

Следующий подтип бинарных выражений можно связать с операторами сравнения.

```

1 | (typedef "rel expression" (uniont "1=2" "1!=2" "unequality
   expression"))
2 | (mot "1==2" :at 1 "expression" :at 2 "expression")
3 | (mot "1!=2" :at 1 "expression" :at 2 "expression")
4 | (typedef "unequality expression" (uniont "1<2" "1<=2" "1>2"
   "1>2"))
5 | (mot "1<2" :at 1 "expression" :at 2 "expression")
6 | (mot "1<=2" :at 1 "expression" :at 2 "expression")
7 | (mot "1>2" :at 1 "expression" :at 2 "expression")
8 | (mot "1>=2" :at 1 "expression" :at 2 "expression")

```

Внимание! Дальнейший текст находится в процессе доработки!

2.7. Заявления

```

1 | (typedef "statement" (uniont "declaration" "label stmt" "simple
   stmt" "go stmt" "return stmt" "break stmt" "continue stmt"
   "goto stmt" "fallthrough stmt" "block" "if stmt" "switch
   stmt" "select stmt" "for stmt" "defer stmt"))

```

```

1 | (typedef "simple statement" (uniont "empty stmt" "expression
   stmt" "send stmt" "1++" "1--" "assignment stmt" "short var
   decl"))

```

```

2 (typedef "empty stmt") ; The empty statement does nothing
3 (mot "label stmt" :at "label" "label" :at "statement"
4   "statement")
5 (mot "label" :at "name" "identifier")
6 ; "label": "statement"
7 ; Error: log.Panic("error encountered")
8 (mot "expression stmt" :at "expression" "expression")
9 ; h(x+y)
10 ; f.Close()
11 (mot "send stmt" :at "channel" "expression" :at "message"
12   "expression")
13 (mot "1++ stmt" :at "1" "expression") ; x += 1
14 (mot "1-- stmt" :at "1" "expression") ; x -= 1

```

```

1 (typedef "assignment stmt" (unionint "1=2" "1+=2" "1-=2" "1|=2"
2   "1^=2" "1*=2" "1/=2" "1%=?2" "1<=?2" "1>=?2" "1&=?2" "1&^=?2"))
3 (mot "1=2" :at 1 (listt "expression") :at 2 (listt "expression"))
4 (mot "1+=2" :at 1 (listt "expression") :at 2 (listt
5   "expression"))
6 (mot "1-=2" :at 1 (listt "expression") :at 2 (listt
7   "expression"))
8 (mot "1|=2" :at 1 (listt "expression") :at 2 (listt
9   "expression"))
10 (mot "1^=2" :at 1 (listt "expression") :at 2 (listt
11   "expression"))
12 (mot "1*=2" :at 1 (listt "expression") :at 2 (listt
13   "expression"))
14 (mot "1/=2" :at 1 (listt "expression") :at 2 (listt
15   "expression"))
16 (mot "1%=?2" :at 1 (listt "expression") :at 2 (listt
17   "expression"))
18 (mot "1<=?2" :at 1 (listt "expression") :at 2 (listt
19   "expression"))

```

```
    "expression"))
11 (mot "1>>=2" :at 1 (listt "expression") :at 2 (listt
    "expression"))
12 (mot "1&=2" :at 1 (listt "expression") :at 2 (listt
    "expression"))
13 (mot "1&^=2" :at 1 (listt "expression") :at 2 (listt
    "expression"))
```

```
; ; if statement
1 (mot "if stmt" :at "init" "simple stmt" :at "condition"
      "expression" :at "then" "block" :at "else" (uniont "if
      statement" "block"))
2 ; if x := f(); x < y {
3 ;     return x
4 ; } else if x > z {
5 ;     return z
6 ; }
7 ;
8 ; ; switch statement
9 (typedef "switch stmt" (uniont "expr switch stmt" "type switch
      stmt"))
10 (mot "expr switch stmt" :at "init" "simple stmt" :at
      "controlling expression" "expression" :at "cases" (listt
      "expr case clause"))
11 (mot "expr case clause" :at "cases" (uniont (listt "expression")
      "default") :at "statements" (listt "statement"))
12 (mot "default")
13 ; switch tag {
14 ; case 0, 1, 2, 3: s1()
15 ; case 4, 5, 6, 7: s2()
16 ; default: s3()
17 ;
18 (mot "type switch stmt" :at "init" "simple stmt" :at "guard"
```

```

    "type switch guard" :at "cases" (listt "type case clause"))
19 (mot "type switch guard" :at "name" "identifier" :at "variable"
      "primary expression")
20 (mot "type case clause" :at "cases" (uniont (listt "type")
      "default") :at "statements" (listt "statement"))
21 ; switch t := x.(type) {                      // type checker
22 ; case nil:
23 ;     fmt.Println("x is nil")
24 ; default:
25 ;     fmt.Println("don't know the type")
26 ;;for statement
27 (typedef "for statement" (uniont "for stmt with condition
      clause" "for stmt with range clause"))
28 (mot "for stmt with condition clause" :at "init" "simple stmt"
      :at "condition" "expression" :at "post" "simple stmt" :at
      "body" "block")
29 (mot "for stmt with range clause" :at "names" (uniont (listt
      "expression") (listt "identifier")) :at "expression"
      "expression" :at "body" "block")
30 ; for x < y { a *= 2 }                      // the usual while loop
31 ; for i := 0; i < 10; i++ { f(i) }        // the usual for loop
32 ; a := [5]int{1, 2, 3, 4, 5}
33 ; for i, v := range a {
34 ;     fmt.Println(i, v)                      // i a[i]
35 ; }
36 (mot "go stmt" :at "body" "expression") ; It starts the
      execution of a function call as an independent concurrent
      thread of control
37 ; go Server()
38 ; go func(ch chan<- bool) { for { sleep(10); ch <- true } } (c)
39 (mot "select stmt" :at "statement" (listt "common clause"))
40 (mot "common clause" :at "case" "common case" :at "statements"

```

```
(listt "statement"))

41 (typedef "common case" (uniont "send stmt" "recive stmt"
        "default"))

42 (mot "recive stmt" :at "names" (uniont (listt "expression")
        (listt "identifier")) :at "expression" "expression")

43 ; func f1(c1 chan int) {
44 ;
45 ;     a := <-c1
46 ;
47 ;     c1 <- a
48 ;
49 ; }
50 ;
51 ; c1, c2 := make(chan int), make(chan int)
52 ; go f1(c1)
53 ; go f2(c2)
54 ;
55 ; c1 <- 3
56 ;
57 ; c2 <- 4
58 ;
59 ; select {
60 ;     case a := <-c1:
61 ;         fmt.Println(1, a)
62 ;     case b := <-c2:
63 ;         fmt.Println(2, b)
64 ;
65 ; }
66 ;
67 (mot "return" :at "expressions" (listt "expression"))

68 (mot "break" :at "label" "label")

69 ; OuterLoop:
70 ;
71     for {
72 ;
73         for {
74 ;
75             break OuterLoop
76 ;
77         }
78 ;
79     }
```

```
70 (mot "continue" :at "label" "label")
71 (mot "goto" :at "label" "label")
72 (typedef "fallthrough" (enumt "fallthrough"))
73 ; switch {                                // prints "true & false"
74 ;     case true:
75 ;         fmt.Println("true & ")
76 ;         fallthrough                      // It transfers control to the
77 ;             next case clause in a switch
77 ;     case false:
78 ;         fmt.Println("false")
79 ;
80 (mot "defer stmt" :at "expression" "expression") ; A "defer"
81 ; statement invokes a function whose execution is deferred to
82 ; the moment the surrounding function returns
81 ; for i := 0; i <= 3; i++ {      // prints 3 2 1 0
82 ;     defer fmt.Println(i)
83 ; }
```