**Prototipo di preprocessore per usare tipi algebrici e pattern matching in go**

Il codice deriva dal prototipo originale che avero scritto per python , scritto in python usando ply come parser (<https://www.dabeaz.com/ply/>) ; il preprocessore lavora testualmente , non analizza il sorgente go.

Dato un esempio di input, prog0.go (Le istruzioni aggiuntive sono in corsivo):

package main

import (

"fmt"

)

*|data exp = Num int | Sum exp exp*

func main() {

if 1 == 1 {

var d1 exp = \_Sum(\_Sum(\_Num(3), \_Num(5)), \_Sum(\_Num(8), \_Num(9)))

*|match d1*

*|case Sum(Sum(el,Num(y)),Sum(ex,k))*

fmt.Println(el)

fmt.Println(y)

fmt.Println(ex)

fmt.Println(k)

*|endcase*

*|case Sum(el,Num(y))*

fmt.Println("el=",el)

fmt.Println("y =",y)

*|endcase*

*|endmatch*

} else {

fmt.Println("mah")

}

}

I preprocessore genera questo testo:

package main

import (

"fmt"

)

type exp interface {

isa\_exp()

}

type Num struct {

a0 int

}

func \_Num(a0 int) Num {

o := Num{}

o.a0 = a0

return o

}

func (o Num) isa\_exp() {}

func (o Num) String() string {

s := fmt.Sprintf("Num(%v)", o.a0)

return s

}

type Sum struct {

a0 exp

a1 exp

}

func \_Sum(a0 exp, a1 exp) Sum {

o := Sum{}

o.a0 = a0

o.a1 = a1

return o

}

func (o Sum) isa\_exp() {}

func (o Sum) String() string {

s := fmt.Sprintf("Sum(%v,%v)", o.a0, o.a1)

return s

}

func main() {

if 1 == 1 {

var d1 exp = \_Sum(\_Sum(\_Num(3), \_Num(5)), \_Sum(\_Num(8), \_Num(9)))

// if isinstance(d1, Sum) {

\_d1, ok0 := d1.(Sum)

if ok0 {

t0 := \_d1.a0

t1 := \_d1.a1

// if isinstance(t0, Sum) {

\_t0, ok1 := t0.(Sum)

if ok1 {

el := \_t0.a0

t2 := \_t0.a1

// if isinstance(t2, Num) {

\_t2, ok2 := t2.(Num)

if ok2 {

y := \_t2.a0

// if isinstance(t1, Sum) {

\_t1, ok3 := t1.(Sum)

if ok3 {

ex := \_t1.a0

k := \_t1.a1

fmt.Println(el)

fmt.Println(y)

fmt.Println(ex)

fmt.Println(k)

}

}

}

}

// if isinstance(d1, Sum) {

\_d1, ok4 := d1.(Sum)

if ok4 {

el := \_d1.a0

t3 := \_d1.a1

// if isinstance(t3, Num) {

\_t3, ok5 := t3.(Num)

if ok5 {

y := \_t3.a0

fmt.Println("el=", el)

fmt.Println("y =", y)

}

}

} else {

fmt.Println("mah")

}

}

L’istruzione *|data exp = Num int | Sum exp exp* viene analizzata per generare i tipi necesari:

il nome exp è usato per definire una interfaccia:

*type exp interface {*

*isa\_exp()*

*}*

Che richiede una funzione isa\_<nome-tipo> che verrà generata per ogni variante (Num e Sum) al solo scopo di poter verificare che un valore è veramente un “exp” ; per

Ogni variante, specificata tramite un nome e una lista di tipi determina la creazione di un tipo struct che implementa l’interfaccia Stringer e come detto il tipo “base”, cioe’ exp in questo esempio; per esempio per Sum exp exp viene generato questo codice:

type Sum struct {

a0 exp

a1 exp

}

func \_Sum(a0 exp, a1 exp) Sum {

o := Sum{}

o.a0 = a0

o.a1 = a1

return o

}

func (o Sum) isa\_exp() {} // cosi’ un Sum e’ anche un exp

func (o Sum) String() string {

s := fmt.Sprintf("Sum(%v,%v)", o.a0, o.a1)

return s

}

Come si vede, dalla dichiarazione Sum exp exp oltre al nome della struct vengono rilevati i tipi dei campi della struct, che in questo caso sono entrambi di tipo exp; nomi dei campi hanno nome a<n> dove

<n> è un progressivo, da 0. La prima funzione definita ha come detto la funzione di “tag”; la funzione String() string associata al tipo Sum ritorna una stringa nel formato *<nome-tipo>( s0,s1….)* dove *si* è il risultati di String() su ogni campo i-esimp, grazie ai segnaposto %v; per il tipo Num int, il codice e’:

func \_Num(a0 int) Num {

o := Num{}

o.a0 = a0

return o

}

func (o Num) isa\_exp() {}

func (o Num) String() string {

s := fmt.Sprintf("Num(%v)", o.a0)

return s

}

Dall’istruzione *|case Sum(Sum(el,Num(y)),Sum(ex,k))* , che richiede il matching con la variabile *d1* indicata in *|match d1* (nota: per semplicità la match accetta solo un nome e non una espressione go) viene generato il seguente codice:

\_d1, ok0 := d1.(Sum)

if ok0 {

t0 := \_d1.a0

t1 := \_d1.a1

\_t0, ok1 := t0.(Sum)

if ok1 {

el := \_t0.a0

t2 := \_t0.a1

\_t2, ok2 := t2.(Num)

if ok2 {

y := \_t2.a0

\_t1, ok3 := t1.(Sum)

if ok3 {

ex := \_t1.a0

k := \_t1.a1

fmt.Println(el)

fmt.Println(y)

fmt.Println(ex)

fmt.Println(k)

}

}

}

}

Il generatore a seconda dei casi usa i nomi speficati come variabili nel pattern, come per ex, k nella sotto espressione *Sum(ex,k) ,* oppure “inventa” dei nomi temporanei (quando il figlio di un costruttore è una stringa; il codice generato usa le asserzioni di tipo di go per verificare se una (sotto) radice è del tipo indicato nel corrispondente elemento del pattern.

Segue Il codice del prototipo ; per ulteriori dettagli, vedi il documento sulla versione per python.

Il “main”, go\_test1.py:

import sys

import re

import go\_gen\_case

import go\_gen\_class

import go\_pdata

import pcase

import pmatch

#import ply.yacc as yacc

#from lexer\_data import get\_lexer

'''

import sys

ss = [ '|data exp = Num n | Sum e1 e2' ]

for s in ss:

if s.startswith('|data'):

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#dataparser = yacc.yacc(tabmodule='dataparsetab')

result = pdata.dataparser.parse(s[1:],lexer=pdata.lexer) #, debug=True) #, debug=True)

for d in result:

gen\_class.gen\_class(d)

'''

ca = re.compile(r' \*(\|case )')

cm = re.compile(r' \*(\|match )')

ce = re.compile(r' \*(\|endmatch)')

#test\_parser()

inp = open(sys.argv[1])

ind = 0

primo\_case = True

curr\_match = None

interfaceName = ''

for r in inp:

s = r.rstrip()

if s.startswith('|data'):

result = go\_pdata.dataparser.parse(s[1:],lexer=go\_pdata.lexer)

interfaceName = result[0]

go\_gen\_class.gen\_interface(interfaceName)

for d in result[1]:

member\_names = go\_gen\_class.gen\_class(d, interfaceName)

go\_gen\_case.diz[d.getName()] = d

go\_gen\_case.diz\_struct\_members\_by\_struct\_name[d.getName()] = member\_names

elif ca.match(s):#s.startswith('|case'):

ic = ca.match(s).start(1)

result = pcase.parser.parse(s[ic+1:], lexer=pcase.lexer)

go\_gen\_case.pila = []

ind = go\_gen\_case.gen(result, curr\_match , primo\_case, interfaceName, ic ) - 1

primo\_case = False

elif cm.match(s):

ic = cm.match(s).start(1)

curr\_match = pmatch.matchparser.parse(s[ic+1:], lexer=pmatch.lexer)

elif ce.match(s):

ind = 0

else:

if ind > 0:

if '|endcase' == s.strip():

go\_gen\_case.pila.reverse()

for p in go\_gen\_case.pila:

print(' '\* p[0],p[1],sep='')

else:

print(' ' \* ind,s.strip())

else:

print(s)

**go\_gen\_case.py:**

# rappresenta un caso di match:

'''

class Case:

def \_\_init\_\_(self, nomeCostruttore, figli):

self.nomeCostruttore = nomeCostruttore

self.figli = figli

def \_\_str\_\_(self):

return self.nomeCostruttore + "(" + ','.join([str(c) for c in self.figli]) + ")"

class Data:

def \_\_init\_\_(self, costrname, cnames):

self.costrname = costrname

self.cnames = cnames # nella versione go, questi sono tipi

# e i nomi sono generati in sequenza

def getName(self):

return self.costrname

def \_\_str\_\_(self):

return 'data(%s)' % ','.join([str(c) for c in self.cnames])

'''

# XXXXXXXXXX PER TEST !!!!!

'''

d1 = Data('Num', ['n'])

d2 = Data('Sum', ['e1', 'e2'])

diz = { d1.getName() : d1, d2.getName() : d2 }

'''

diz = {}

diz\_struct\_members\_by\_struct\_name = {} # contiene liste di coppie [n,t] per ogni nome di struct, con nome membro e tipo

temp=0

# temporanei per il patter matching

def newtemp():

global temp

r = 't%d' % temp

temp += 1

return r

# genera il codice per il pattern matching di un "case"

pila = []

curr\_ok = 0

def gen(c, match\_id, primocase, interfaceName, ind=0):

global pila, curr\_ok

cname = c.nomeCostruttore

ic = 0

cldef = diz[cname] # trova la definizione del costruttore di valore del dato

# algebrico

ifs = 'if'

if\_ind = ind - 1

print("%s// %s isinstance(%s, %s) {" % (' ' \* ind, ifs, match\_id, cname))

c\_match\_id = '\_' + match\_id

print('%s%s, ok%s := %s.(%s)' % (' ' \* ind,c\_match\_id, curr\_ok, match\_id, cname))

print("%sif ok%s {" % (' ' \* ind, curr\_ok))

pila.append((ind,'}'))

curr\_ok += 1

temps = []

for ch in c.figli:

name\_m = diz\_struct\_members\_by\_struct\_name[cname][ic][0] # [1] e' il tipo del

# membro

if isinstance(ch,str):

print("%s%s := %s.%s" % (' ' \* (ind+3), ch, c\_match\_id, name\_m))

else:

t = newtemp()

temps.append(t)

print("%s%s := %s.%s" % (' ' \* (ind+3), t, c\_match\_id, name\_m))

ic += 1

nt = 0

ind += 3

sind = ind

for ch in c.figli:

if not isinstance(ch,str):

#print('ch=>',ch)

sind = gen(ch, temps[nt], True, ind)

nt += 1

#ind += 6

ind = sind

return sind # ritorna la massima indentazione

**go\_gen\_class.py:**

# genera la classe da una definizione Data creata

# dal parser del (futuro) preprocessore:

def gen\_interface(interfaceName):

print('type %s interface {' % interfaceName)

print(' isa\_%s()' % interfaceName)

print('}')

print('\n')

def gen\_class(dt, interfaceName):

ints = [i for i in range(len(dt.costrname))]

dt\_cnames = [['a%d' % v[0], '%s' % v[1]] for v in zip(ints, dt.cnames)]

print('type %s struct {' % dt.costrname)

for e in dt\_cnames:

print( ' %s %s' % (e[0],e[1]))

print('}')

# costruttore:

print('func \_%s(' % dt.costrname, end='')

if len(dt.cnames) == 0:

print('):')

print(' pass')

else:

print(','.join([' '.join(e) for e in dt\_cnames]),') %s {' % dt.costrname)

print(' o := %s{}' % dt.costrname)

for c in dt\_cnames:

print(' o.%s = %s' % (c[0],c[0]))

print(' return o')

print('}')

print('func (o %s) isa\_%s() {}' % (dt.costrname, interfaceName))

# String() string

# 'Sum(%s,%s)' % (str(self.e1), str(self.e2))

print('func', '(o %s) String() string {' % dt.costrname)

#fmt = ','.join(['%s' for c in dt.cnames])

nplaces = ','.join(['%v' for x in range(len(dt.cnames))])

s = 's := fmt.Sprintf("%s(%s)",%s)' % (dt.costrname, nplaces, ','.join(['o.%s' % c[0] for c in dt\_cnames]))

print(' %s' % s)

print(' return s') # % s)

print('}')

return dt\_cnames

**go\_pdata.py:**

#calcparser.py

# Yacc example

import ply.yacc as yacc

reserved = { 'data' : 'DATA' }

tokens = [ 'OR','ID', 'EQ' ] + list(reserved.values())

# Tokens

#t\_LP = r'\('

#t\_RP = r'\)'

#t\_CASE = 'case'

# t\_COMMA = r','

t\_EQ = r'='

t\_OR = r'\|'

def t\_ID(t):

r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

t.type = reserved.get(t.value,'ID') # Check for reserved words

return t

# Ignored characters

t\_ignore = " \t"

def t\_newline(t):

r'\n+'

t.lexer.lineno += t.value.count("\n")

def t\_error(t):

print("Illegal character '%s'" % t.value[0])

t.lexer.skip(1)

# Build the lexer

import ply.lex as lex

lexer = lex.lex()

# ============ classi di supporto ======================

# rappresenta un caso di match:

class Data:

def \_\_init\_\_(self, costrname, cnames):

self.costrname = costrname

self.cnames = cnames

def getName(self):

return self.costrname

def \_\_str\_\_(self):

return 'data(%s, %s)' % (self.costrname, ','.join([str(c) for c in self.cnames]))

def p\_data(p):

'data : DATA ID EQ vlist'

#p[0] = p[4]

p[0] = (p[2],p[4])

def p\_vlist(p):

'''vlist : vlist OR v

| v'''

if len(p) == 2:

p[0] = [p[1]]

else:

p[1].append(p[3])

p[0] = p[1]

def p\_v(p):

'v : ID vs'

p[0] = Data(p[1], p[2])

def p\_vs(p):

'''vs : vs ID

| empty'''

if len(p) == 2:

p[0] = []

else:

p[1].append(p[2])

p[0] = p[1]

def p\_empty(p):

'empty :'

pass

# Error rule for syntax errors

def p\_error(p):

print("Syntax error in input!")

dataparser = yacc.yacc(tabmodule='dataparsetab')

**pmatch.py:**

import ply.yacc as yacc

reserved = { 'match' : 'MATCH' }

tokens = [ 'ID' ] + list(reserved.values())

# Tokens

def t\_ID(t):

r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

t.type = reserved.get(t.value,'ID') # Check for reserved words

return t

# Ignored characters

t\_ignore = " \t"

def t\_newline(t):

r'\n+'

t.lexer.lineno += t.value.count("\n")

def t\_error(t):

print("Illegal character '%s'" % t.value[0])

t.lexer.skip(1)

# Build the lexer

import ply.lex as lex

lexer = lex.lex()

def p\_match(p):

'match : MATCH ID'

p[0] = p[2]

# Error rule for syntax errors

def p\_error(p):

print("Syntax error in input!")

matchparser = yacc.yacc(tabmodule='matchparsetab')