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')
```