Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Μάθημα: Μεταφραστές
Κωδικός Μαθήματος: myy802

Αναφορά Προγραμματιστικής Άσκησης Για Το Έτος 2020-2021

Ομάδα: Γεωργάκης Βασίλειος Α.Μ.: 3197

Ζήσης Προκόπιος Ταλαμάγκας Α.Μ.: 3340

Σύντομη Περιγραφή της Άσκησης

Στο μάθημα των Μεταφραστών ζητήθηκε η κατασκευή ενός compiler για την γλώσσα cimple. Ο μεταγλωττιστής θα πρέπει να παράγει ως τελική γλώσσα, τη γλώσσα assembly του επεξεργαστή MIPS. Στην συνέχεια γίνεται η ανάλυση των 3 φάσεων από τις οποίες αποτελείται η υλοποίηση της εργασίας, και πως συνδέονται ώστε να επιτευχθεί η άσκηση. Η Πρώτη φάση αποτελείται από τον λεκτικό και συντακτικό αναλυτή. Η Δεύτερη

φάση από τον ενδιάμεσο κώδικα και τέλος η Τρίτη φάση από τον πίνακα συμβόλων και τον τελικό κώδικα.

Περιεχόμενα Υλοποίησης Άσκησης

Λεκτικός Αναλυτής……σελίδα 2.

Συντακτικός Αναλυτής……σελίδα 3.

Ενδιάμεσος Κώδικας……σελίδα 17.

Πίνακας Συμβόλων……σελίδα 19.

Τελικός Κώδικας……σελίδα 19.

Λεκτικός Αναλυτής

Ο λεκτικός αναλυτής καλείται ως συνάρτηση από τον συντακτικό αναλυτή έχοντας ως σκοπό να διαβάσει το πρόγραμμα που του δίνεται από τον χρήστη γράμμα-γράμμα και να επιστρέφει κάθε φορά την επόμενη λεκτική μονάδα. Ο λεκτικός αναλυτής λειτουργεί σαν ένα αυτόματο καταστάσεων που ξεκινώντας από μία αρχική κατάσταση, περνώντας από ενδιάμεσες καταστάσεις, να φτάσει σε μία τελική κατάσταση. Οι καταστάσεις που αποτελούν το αυτόματο υλοποιούνται στις γραμμές 20-28 του κώδικα και φαίνονται στην εικόνα 1.1.

```
20    starting_state = 0
21    identifier_state = 100
22    symbols_state = 200
23    number_state = 300
24    comment_state = 400
25    asgn_state = 500
26    smaller_state = 600
27    larger_state = 700
28    end_state = 1100
```

Εικόνα 1.1

Έχοντας κάνει αυτή την δουλειά ο λεκτικός αναλυτής επιστρέφει στον συντακτικό αναλυτή την λεκτική μονάδα και έναν ακέραιο που την χαρακτηρίζει.

- Το αυτόματο καταστάσεων αναγνωρίζει το αλφάβητο της Cimple το οποίο περιέχει τα εξής:
- Τα μικρά και κεφαλαία γράμματα της λατινικής αλφαβήτου (A , … , Z και a , … , z)
- Τα αριθμητικά ψηφία (0 , … , 9)
- Τα σύμβολα των αριθμητικών πράξεων (+ , - , * , /)
- Τους τελεστές συσχέτισης (< , > , = , <= , >= , <>)
- Το σύμβολο ανάθεσης (:=)
- Τους διαχωριστές (; , «,» , :)
- Τα σύμβολα ομαδοποίησης ([,] , (,) , { , })
- Τον τερματισμό του προγράμματος (.)
- Τον διαχωρισμό των σχολίων (#)

Οι δεσμευμένες λέξεις είναι:

program	declare				
If	else	while			
switchcase	forcase	incase	case	default	
not	and	or			
function	procedure	call	return	in	inout
input	print				

Η υλοποίηση του λεκτικού αναλυτή βρίσκεται στις γραμμές 16 – 155 του κώδικα.

Συντακτικός Αναλυτής

Η λειτουργία του συντακτικού αναλυτή είναι αρχικά να γίνει έλεγχος για να διαπιστωθεί αν το πηγαίο πρόγραμμα ανήκει ή όχι στη γλώσσα Cimple και στη συνέχεια να

δημιουργήσει το κατάλληλο περιβάλλον μέσα στο οποίο θα κληθούν οι σημαντικές ρουτίνες. Οι ρουτίνες του κώδικα είναι οι παρακάτω.

- program : program ID block

```
161 def program():
162     global token[]
163     token = lex()
164     if token[0] == 'program':
165         token = lex()
166         if ID(token):
167             token[0] = lex()
168             block(ID)
169             genquad('halt','','','')
170             genquad('end_block',name,'','')
171         else:
172             error("Expected an ID")
173 
174     else:
175         error("Expected 'program'")
176     if token[0] == '.':
177         print("The lexical and synxtax analysis were correct")
178     return
```

- block : declarations subprograms statements

```
180 def block(name):
181     global token[]
182     declarations()
183     subprograms()
184     genquad('begin_block',name,'','')
185     statements()
186     return
```

- declarations : (declare varlist ;)*

```
188 def declarations():
189     global token[]
190     while token[0] == 'declare':
191         varlist()
192         if token[0] == ';':
193             token = lex()
194         else:
195             error("Expected ';'")
196     return
```

- varlist : ID (, ID)*
  | ε

```
198 def varlist():
199     global token[]
200     if ID(token):
201         token = lex()
202         while token[0] == ',':
203             token = lex()
204             if ID(token):
205                 token = lex()
206             else:
207                 error("Expected ID")
208     return
```

- subprograms : function ID (formalparlist) block
  | procedure ID (formalparlist) block

```
210 def subprograms():
211     global token[]
212     while token[0] == 'function' or token[0] == 'procedure':
213         token = lex()
214         subprogram()
215         genquad('end_block',name,'','')
216     return
```

- subprogram : function ID (formalparlist) block
  | procedure ID (formalparlist) block

```
218 def subprogram():
219     global token[]
220     if token[0] == 'function':
221         if ID(token):
222             if token[0] == '(':
223                 token = lex()
224                 formalparlist()
225                 if token[0] == ')':
226                     token = lex()
227                     block()
228                 else:
229                     error("Expected ')'")
230             else:
231                 error("Expected '('")
232     if token[0] == 'procedure':
233         if ID(token):
234             if token[0] == '(':
235                 token = lex()
236                 formalparlist()
237                 if token[0] == ')':
238                     token = lex()
239                     block()
240                 else:
241                     error("Expected ')'")
242             else:
243                 error("Expected '('")
244     return
```

- formalparlist : formalparitem (, formalparitem)*
  | ε

```
246 def formalparlist():
247     global token[]
248     formalparitem()
249     while token[0] == ',':
250         token = lex()
251         if token[0]== 'in' or token[0]=='inout':
252             formalparitem()
253         else:
254             error("Expected 'in' or 'inout'")
255     return
```

- formalparitem : in ID
  | inout ID

```
257 def formalparitem():
258     global token[]
259     if token[0] == 'in':
260         token = lex()
261         if ID(token):
262             token = lex()
263             genquad(par,a,CV,'')
264             genquad(par,b,REF,'')
265             genquad(call,assign_v,'','')
266             return
267         else:
268             error("Expected id")
269     elif token[0] == 'inout':
270         token = lex()
271         if ID(token):
272             token=lex()
273             genquad(par,a,CV,'')
274             genquad(par,b,REF,'')
275             genquad(call,assign_v,'','')
276             return
277         else:
278             error("Expected id")
279     else:
280         error("Expected 'in' or 'inout'")
281         genquad(par,a,CV,'')
282         genquad(par,b,REF,'')
283         w = newtemp()
284         genquad(par,w,RET,'')
285         genquad(call,assign_v,'','')
286     return
```

- statements : statement ;
  | { statement (; statement)* }

```
288 def statements():
289     global token[]
290     statement()
291     if token[0] == ';':
292         token= lex()
293         return
294     elif token[0] == '{':
295         statement()
296         token=lex()
297         while token[0] == ';':
298             statement()
299             token = lex()
300         if token[0]=='}'
301             token = lex()
302             return
303         else:
304             error("Expected '}'")
305 
306     else:
307         statement()
308         if token[0] == ';':
309             token=lex()
310         else:
311             error("Expected ';'")
312     return
```

- statement : assignStat
  | ifStat
  | whileStat
  | switchcaseStat
  | forcaseStat
  | incaseStat
  | callStat
  | returnStat
  | inputStat
  | printStat
  | ε

```
314 def statement():
315     global token[]
316     if token[0] == ':':
317         assignStat()
318     elif token[0] == 'if':
319         ifStat()
320     elif token[0] == 'while':
321         whileStat()
322     elif token[0] == 'switch':
323         switchStat()
324     elif token[0] == 'forcase':
325         forcaseStat()
326     elif token[0] == 'incase':
327         incaseStat()
328     elif token[0] == 'call':
329         callStat()
330     elif token[0] == 'return':
331         returnStat()
332     elif token[0] == 'input':
333         inputStat()
334     elif token[0] == 'print':
335         printStat()
336     return
```

- assignStat : ID := expression

```
338 def assignStat():
339     global token[]
340     token = lex()
341     if ID(token):
342         if token[0] == ':=':
343             token = lex()
344             A = expression()
345             genquad(':=',A,'',id)
346         else:
347             error("Expected ':='")
348     return A
```

- ifStat : if (condition) statements elsepart

```
350 def ifStat():
351     global token[]
352     token = lex()
353     if token[0] == 'if':
354         token = lex()
355         if token[0] == '(':
356             token=lex()
357             B = condition()
358             if token[0] == ')':
359                 token = lex()
360                 backpatch(B[0],nextquad())
361                 statements()
362                 ifList = makelist(nextquad())
363                 genquad('jump','','','')
364                 backpatch(B[1],nextquad())
365                 elsepart()
366                 backpatch(ifList,nextquad())
367             else:
368                 error("Expected ')'")
369         else:
370             error("Expected '('")
371     return
```

- elsepart : else statements
  | ε

```
373 def elsepart():
374     global token[]
375     if token[0]=='else':
376         token = lex()
377         statements()
378     return
```

- whileStat : while (condition) statements

```
380 def whileStat():
381     global token[]
382     token = lex()
383     if token[0] == 'while':
384         token = lex()
385         Bquad = nextquad()
386         if token[0] == '(':
387             token=lex()
388             B = condition()
389             if token[0] == ')':
390                 token = lex()
391                 backpatch(B[0],nextquad())
392                 S = statements()
393                 genquad('jump','','',Bquad)
394                 backpatch(B[1],nextquad())
395             else:
396                 error("Expected ')'")
397         else:
398             error("Expected '('")
399     return
```

- switchcaseStat : switchcase
  (case (condition) statements)*
  default statements

```
401 def switchcase():
402     global token[]
403     token=lex()
404     if token[0] == 'switchcase':
405         while token=='case':
406             token=lex()
407             if token=='(':
408                 token=lex()
409                 condition()
410                 if token==')':
411                     statements()
412                 else:
413                     error("Expected ')'")
414             else:
415                 error("Expcted '('")
416         if token=='default':
417             token=lex()
418             statements()
419     return
```

- forcaseStat : forcase

  (case (condition) statements)*

  default statements

```
421 def forcaseStat():
422     global token[]
423     token=lex()
424     if token[0] == 'forcase':
425         plQuad = nextquad()
426         while token=='case':
427             token=lex()
428             if token=='(':
429                 token=lex()
430                 condition()
431                 backpatch(cond.true.nextquad())
432                 if token==')':
433                     statements()
434                     genquad('jump','','',plQuad)
435                     backpatch(cond.false,nextquad())
436                 else:
437                     error("Expected ')'")
438             else:
439                 error("Expcted '('")
440         if token=='default':
441             token=lex()
442             statements()
443     return
```

- incaseStat : incase

  (case (condition) statements)*

```
445 def incaseStat():
446     global token[]
447     token=lex()
448     if token[0] == 'incase':
449         while token=='case':
450             token=lex()
451             w = newtemp()
452             plQuad = nextquad()
453             genquad(':=',1,'',w)
454             if token=='(':
455                 token=lex()
456                 condition()
457                 backpatch(cond.true.nextquad())
458                 genquad(':=',0,'',w)
459                 if token==')':
460                     statements()
461                     backpatch(cond.false.nextquad())
462                 else:
463                     error("Expected ')'")
464             else:
465                 error("Expcted '('")
466     genquad('=',w,0,plQuad)
467     return
```

- returnStat : return (expression)

```
469 def returnStat():
470     global token[]
471     if token[0]=='return':
472         token=lex()
473         if token[0]=='(':
474             token=lex()
475             A = expression()
476             genquad('retv',A,'','')
477             if token==')':
478                 token = lex()
479             else:
480                 error("Expected ')'")
481         else:
482             error("Expcted '('")
483     return A
```

- callStat : call ID (actualparlist)

```
485 def callStat():
486     global token[]
487     if token[0]=='call':
488         token=lex()
489         if ID(token):
490             if token[0]=='(':
491                 token=lex()
492                 actualparlist()
493                 if token==')':
494                     token = lex()
495                 else:
496                     error("Expected ')'")
497             else:
498                 error("Expcted '('")
499     return
```

- printStat : print (expression)

```
501 def printStat():
502     global token[]
503     if token[0]=='print':
504         token=lex()
505         if token[0]=='(':
506             token=lex()
507             E = expression()
508             if token==')':
509                 token = lex()
510                 genquad('out',E,'','')
511             else:
512                 error("Expected ')'")
513         else:
514             error("Expcted '('")
515     return
```

- inputStat : input (ID)

```
517 def inputStat():
518     global token[]
519     if token[0]=='return':
520         token=lex()
521         if token[0]=='(':
522             token=lex()
523             if ID(token):
524                 input_id = token
525                 token = lex()
526                 if token==')':
527                     token = lex()
528                     genquad('inp', input_id, '','')
529                 else:
530                     error("Expected ')'")
531         else:
532             error("Expcted '('")
533     return
```

- actualparlist : actualparitem (, actualparitem)*

  | ε

```
535 def actualparlist():
536     global token[]
537         actualparitem()
538         while token[0] == ',':
539             token = lex()
540             if token[0]== 'in' or token[0]=='inout':
541                 actualparitem()
542             else:
543                 error("Expected 'in' or 'inout'")
544         return
```

- actualparitem : in expression

  | inout ID

```
546 def actualparitem():
547     global token[]
548     if token[0] == 'in':
549         token = lex()
550         expression()
551     elif token[0] == 'inout':
552         token = lex()
553         if ID(token):
554             token=lex()
555         else:
556             error("Expected id")
557     else:
558         error("Expected 'in' or 'inout'")
559     return
```

- condition : boolterm (or boolterm)*

```
561 def condition():
562     global token
563     BTrue = []
564     BFalse = []
565     B1 = boolterm()
566     BTrue = B1[0]
567     BFalse = B1[1]
568     while token[0]=='or':
569         token=lex()
570         backpatch(BFalse, nextquad())
571         B2 = boolterm()
572         BTrue = merge(BTrue, B2[0])
573         BFalse = B2[1]
574     return [BTrue,BFalse]
```

- boolterm : boolfactor (and boolfactor)*

```
576 def boolterm():
577     global token
578     BTrue2 = []
579     BFalse2 = []
580     Q1 = boolfactor()
581     BTrue2 = Q1[0]
582     BFalse2 = Q2[1]
583     while token[0]=='and':
584         token=lex()
585         backpatch(BTrue2, nextquad())
586         Q2 = boolfactor()
587         BFalse2 = merge(BFalse2, Q2[1])
588         BTrue2 = Q2[0]
589     return [BTrue2,BFalse2]
```

- boolfactor : not [condition]

  | [condition]

  | expression REL_OP expression

```
591  def boolfactor():
592      global token[]
593      RTrue = []
594      RFalse = []
595      if token[0] == 'not':
596          token=lex()
597          if token[0] == '[':
598              token = lex()
599              B = condition()
600              RTrue = B[0]
601              RFalse = B[1]
602              if token[0] == ']':
603                  token = lex()
604              else:
605                  error("Expected ']'")
606          else:
607              error("Expected '['")
608      elif token[0] == '[':
609          token = lex()
610          B = condition()
611          RTrue = B[0]
612          RFalse = B[1]
613          if token[0] == ']':
614              token = lex()
615          else:
616              error("Expected ']'")
617      else:
618          E1 = expression()
619          relop = REL_OP()
620          E2 = expression()
621          RTrue = makelist(nextquad())
622          genquad(relop,E1,E2,'')
623          RFalse = makelist(nextquad())
624          genquad('jump','','','')
625      return [RTrue,RFalse]
```

- expression : optionalSign term (ADD_OP term)*

```
627  def expression():
628      global token[]
629      optimalSign()
630      A = term()
631      while token[0] == '+' or token[0]=='-':
632          C = ADD_OP()
633          B = term()
634          w = newtemp()
635          genquad(C,A,B,w)
636          A = w
637      return A
```

- term : factor (MUL_OP factor)*

```
639 def term():
640     global token[]
641     A = factor()
642     while token[0] == '*' or token[0]=='/':
643         C = MUL_OP()
644         B = factor()
645         w = newtemp()
646         genquad(C,A,B,w)
647         A = w
648     return A
```

- factor : INTEGER

  | (expression)

  | ID idtail

```
650 def factor():
651     global token[]
652     if token[0] == '(':
653         token = lex()
654         A = expression()
655         if token[0] == ')':
656             token = lex()
657         else:
658             error("Expected ')'")
659         F = A
660     elif ID(token):
661         token = lex()
662         B = idtail()
663         F = B
664     else:
665         INTEGER()
666     return F
```

- idtail : (actualparlist)

  | ε

```
668 def idtail():
669     global token[]
670     if token[0] == '(':
671         token=lex()
672         actualparlist()
673         if token[0] == ')':
674             token = lex()
675         else:
676             error("Expected ')'")
677     return
```

- optionalSign : ADD_OP

  | ε

```
679 def optimalSign():
680     global token
681     if token == '+' or '-':
682         ADD_OP()
683     return
```

- REL_OP : = | <= | >= | > | < | <>

```
685 def REL_OP():
686     global token
687     if token == '=' or token == '<=' or token == '>=' or token == '>' or token == '<' or token == '<>':
688         token = lex()
689     else:
690         error("Expected realtional operator")
691     return
```

- ADD_OP : + | -

```
693 def ADD_OP():
694     global token
695     if token =='+' or token == '-':
696         add_opp = token
697         token = lex()
698     else :
699         error("Expected '+' or '-'")
700     return add_opp
```

- MUL_OP : * | /

```
702 def MUL_OP():
703     global token
704     if token == '*' or token == '/':
705         mul_opp
706         token = lex()
707     else:
708         error("Expected '*' or '/'")
709     return mul_opp
```

- INTEGER : [0-9]+

```
711 def INTEGER():
712     global token
713     if token.isdigit():
714         return True
715     return False
```

- ID : [a-zA-Z] [a-zA-Z0-9]*

```
717 def ID(unit): #[a-zA-Z][a-zA-Z0-9]* The first char must be a char
718     global token
719     if token not in keywords:
720         if not token.isdigit():
721             if token.isalnum():
722                 return True
723     return False
```

Για κάθε ένα από τους παραπάνω κανόνες υπάρχει όπως φαίνεται και στις αντίστοιχες εικόνες ένα υποπρόγραμμα (συνάρτηση στην γλώσσα python) που υλοποιεί τον κανόνα και καλείται ανάλογα βρέθηκε η δεσμευμένη λέξη ή κάποιο σύμβολο. Κατά την διάρκεια του προγράμματος όσο δεν συναντάμε κάποιο τερματικό σύμβολο καλούμε το αντίστοιχο υποπρόγραμμα. Στην περίπτωση που βρούμε ένα τερματικό σύμβολο τότε :

- Αν ο λεκτικός αναλυτής επιστρέψει λεκτική μονάδα που αντιστοιχεί στο τερματικό αυτό σύμβολο τότε γνωρίζουμε πως έχουμε αναγνωρίσει επιτυχώς την λεκτική μονάδα
- Σε αντίθετη περίπτωση έχουμε κάνει κάποιο λάθος και καλείται ο διαχειριστής σφαλμάτων που έχουμε υλοποιήσει

Όταν αναγνωριστεί και η τελευταία λέξη του δοθέντος προγράμματος , τότε έχει τελειώσει με επιτυχία η συντακτική ανάλυση.

Ενδιάμεσος Κώδικας

Ο ενδιάμεσος κώδικας είναι ένα σύνολο από τετράδες. Οι τετράδες αποτελούνται από έναν τελεστή και τρία τελούμενα. Κάθε τετράδα έχει μπροστά της έναν μοναδικό αριθμό που τη χαρακτηρίζει. Οι τετράδες αυτές είναι της μορφής op, x, y, z όπου:

- Το op είναι ο τελεστής
- Τα τελούμενα x, y μπορεί να είναι:
  1. Ονόματα μεταβλητών
  2. Αριθμητικές σταθερές
- Το τελούμενο z μπορεί να είναι:
  - Όνομα μεταβλητής
- Ο τελεστής op εφαρμόζεται στα τελούμενα x, y και το αποτέλεσμα τοποθετείται στο τελούμενο z.

Οι συναρτήσεις που υλοποιούνται στον ενδιάμεσο κώδικα είναι :

nextquad() : Επιστρέφει τον αριθμό της επόμενης τετράδας που πρόκειται να παραχθεί

genquad(op, x, y, z) : Δημιουργεί την επόμενη τετράδα (op, x, y, z)

newtemp() : Δημιουργεί και επιστρέφει μία νέα προσωρινή μεταβλητή. Οι προσωρινές μεταβλητές είναι της μορφής T_1, T_2, …

emptylist() : Δημιουργεί μία κενή λίστα ετικετών τετράδων

makelist(x) : Δημιουργεί μία λίστα ετικετών τετράδων που περιέχει μόνο το x

merge(list1, list2) : Δημιουργεί μία λίστα ετικετών τετράδων από τη συνένωση των λιστών list1,list2

backpatch(list, z) : Η λίστα list αποτελείται από δείκτες σε τετράδες των οποίων το τελευταίο τελούμενο δεν είναι συμπληρωμένο. Η backpatch επισκέπτεται μία μία τις τετράδες αυτές και τις συμπληρώνει με την ετικέτα z

```
736  global quad_identifier = 0
737  global temp_counter = 0
738
739  def nextquad():
740      global quad_identifier
741      quad_identifier += 1
742      return quad_identifier
743
744  def genquad(op,x,y,z):
745      global quad = [None]*4
746      global quad_final
747      quad.append(op)
748      quad.append(x)
749      quad.append(y)
750      quad.append(z)
751      quad_final.append(nextquad())
752      quad_final.append(quad)
753      return quad_final
754
755  def newtemp():
756      global temp_counter
757      temp_counter += 1
758      return 'T_' + str(temp_counter)
759
760  def emptylist():
761      emptyquad = [None]*4
762      return emptyquad
763
764  def makelist(x):
765      xquad = [x]
766      return quad
767
768  def merge(list1,list2):
769      new_list = list1.extend(list2)
770      return new_list
771
772  def backpatch(list,z):
773      global quad_final
774      for i in range(len(list)):
775          for j in range(len(quad_final)):
776              if(list[i]==quad_final[j][0] and quad_final[j][4]=='_'):
777                  quad_final[j][4] = z
778      return
```

Πίνακας Συμβόλων

Ο πίνακας συμβόλων είναι υπεύθυνος για την ανάγνωση των μεταβλητών των σταθερών και των προσωρινών μεταβλητών μιας συνάρτησης του προγράμματος που δίνεται από τον χρήστη. Περιλαμβάνει τα εξής:

- Κλάση Entity :
  - Συνάρτηση init
  - Κλάση Variable
  - Κλάση Function
  - Κλάση Constant
  - Κλάση Parameter
  - Κλάση TempVar
- Κλάση Scope :
  - Συνάρτηση init
- Κλάση Argument :
  - Συνάρτηση init
- Συνάρτηση new_scope (Προσθήκη νέου Scope)
- Συνάρτηση deletescope (Διαγραφή Scope)
- Συνάρτηση new_entity (Προσθήκη νέου Entity)
- Συνάρτηση new_argument (Προσθήκη νέου Argument)
- Συνάρτηση search_entity (Αναζήτηση Entity)

Τα παραπάνω βρίσκονται στις γραμμές κώδικα 784 – 861.

Τελικός Κώδικας

Ο τελικός κώδικας παίρνει κάθε εντολή ενδιάμεσου κώδικα και παράγει τις αντίστοιχες εντολές του τελικού κώδικα. Οι κύριες ενέργειες στην φάση αυτή είναι :

- Οι μεταβλητές απεικονίζονται στην μνήμη (στοίβα)
- Γίνεται το πέρασμα συναρτήσεων και η κλήση συναρτήσεων

Τέλος, δημιουργείται ο κώδικας για τον επεξεργαστή MIPS.

Δυστυχώς σε αυτό το σημείο δεν έχουμε καταφέρει την υπολοποίηση. Η προσπάθεια μας σταμάτησε στην παραγωγή του πίνακα συμβόλων οπότε ο compiler της γλώσσας Cimple δεν ολοκληρώθηκε.

Αυτή ήταν η αναφορά μας για την προγραμματιστική εργασία για το μάθημα των Μεταφραστών όπου έπρεπε να κατασκευάσουμε έναν compiler για την γλώσσα Cimple. Ευχαριστούμε!