Μεταφραστές 2020

Προγραμματιστική εργασία: Minimal++

Τζώρτζης Ευάγγελος

ΑΜ: 3088

Τελική Αναφορά Ορθής Λειτουργίας

Η λειτουργία του μεταφραστή για τη γλώσσα Minimal++ σε τελικό επίπεδο ελέγχθηκε χρησιμοποιώντας τα παρακάτω προγράμματα:

1. programplusc.min
2. program1workingfinal.min
3. program2working.min
4. decimalToBinary.min
5. fibonacci.min

Επιπλέον,υλοποίησα τα προγράμματα από τα παραδείγματά σας στον τελικό κώδικα, ώστε να συγκρίνω τα αποτελέσματα κώδικα assembly του mips με τα δικά μου:

6. finalex1.min
7. finalex2.min

Τα προγράμματα τρέχουν με την εντολή: **python3 compilerMainV4.py <somefile.min>**

Όταν τρέξει ο κώδικας και δεν υπάρχουν λάθη δημιουργούνται 4 αρχεία:

- Αρχείο <filename>.int, όπου περιέχεται ο ενδιάμεσος κώδικας.
- Αρχείο <filename>.c,όπου εφόσον δεν υπάρχουν συναρτήσεις δημιουργείται ένας ισοδύναμος κώδικας σε c.
- Αρχείο <filename>_symbol_table_history.txt, όπου περιέχεται ένα ιστορικό του πίνακα συμβολών που δημιουργήθηκε κατά τη μετάφραση του προγράμματος.
- Αρχείο <filename>.asm, όπου βρίσκεται ο τελικός κώδικας σε assembly και τρέχει από τον mars mips.

Παρακάτω αναλύονται τόσο οι διάφορες φάσεις, όσο και ο κώδικας των προγραμμάτων αυτών.

Πρόγραμμα 1:

programplusc.min

Το συγκεκριμένο πρόγραμμα έχει φτιαχτεί χωρίς να έχει κάποια συνάρτηση, ώστε να μπορεί να μετατραπεί σε κώδικα c για έλεγχο καλής λειτουργίας των υποστηριζόμενων statements και των λογικών συνθήκων της γλώσσας. Συγκεκριμένα, το πρόγραμμα ελέγχει την ορθότητα των statement:**if, input, while, print, forcase, :=(<assignment>),**καθώς και τις συνθήκες: **>,<,=,<=,and,or.** Ακόμα, ελέγχονται σε αρκετό βαθμό οι εκφράσεις με διάφορες πράξεις και εντολές **declare**.

Αρχικά, ζητούνται δυο input από τον χρήστη:

```
input(var1);
input(var2);
```

Αυτά γράφονται στον ενδιάμεσο κώδικα ως:

```
101: inp,var1,_,_
102: inp,var2,_,_
```

Στο αρχείο c γράφονται ως:

```
L_2: printf("var1 input: ");
scanf("%d",&var1);     //101: inp,var1,_,_
L_3: printf("var2 input: ");
scanf("%d",&var2);     //102: inp,var2,_,_
```

Όπου γράφονται και κάποια επεξηγηματικά prints.

Ο πίνακας συμβόλων είναι:

Final/main symbol table:

[[['var1', 12], ['var2', 16], ['var3', 20], ['var4', 24], ['T_1', 28], ['T_2', 32], ['T_3', 36], ['T_4', 40], ['T_5', 44], ['T_6', 48], ['T_7', 52], ['T_8', 56], ['T_9', 60], ['T_10', 64]]]

Final offset of program: 68

Ο τελικός κώδικας γίνεται:

```
L101:
li $v0,5
syscall
move $t1,$v0
sw $t1,-12($s0)
L102:
li $v0,5
syscall
move $t1,$v0
sw $t1,-16($s0)
```

Ο τελικός κώδικας αντιστοιχεί όπως πρέπει τις μεταβλητές με τα offset τους και δουλεύει.

Στη συνέχεια ελέγχεται ο βρόγχος while με συνθήκη var1<var2.

Μέσα στο βρόγχο ελέγχονται και ένα if και else, διάφορα print και ένα assignment.

```
if (var1 < 2 and var2 > 4) thenprint(-32767) elseprint(32767);
var3:=var1+var2;
print(var3);
var1 := 1 + var2
```

Ο ενδιάμεσος κώδικας γράφεται ως εξής:

103: <,var1,var2,105
104: jump,_,_,119
105: <,var1,2,107
106: jump,_,_,112
107: >,var2,4,109
108: jump,_,_,112
109: -,0,32767,T_1
110: out,T_1,_,_
111: jump,_,_,113
112: out,32767,_,_
113: +,var1,var2,T_2
114: :=,T_2,_,var3
115: out,var3,_,_
116: +,1,var2,T_3
117: :=,T_3,_,var1
118: jump,_,_,103

Σε c γράφεται:

```
L_4: if(var1<var2) gotoL_6;     //103: <,var1,var2,105
L_5: gotoL_20;     //104: jump,_,_,119
L_6: if(var1<2) gotoL_8;     //105: <,var1,2,107
L_7: gotoL_13;     //106: jump,_,_,112
L_8: if(var2>4) gotoL_10;     //107: >,var2,4,109
L_9: gotoL_13;     //108: jump,_,_,112
L_10: T_1=0-32767;     //109: -,0,32767,T_1
L_11: printf("output T_1: %d\n",T_1);     //110: out,T_1,_,_
L_12: gotoL_14;     //111: jump,_,_,113
L_13: printf("output 32767: %d\n",32767);     //112: out,32767,_,_
L_14: T_2=var1+var2;     //113: +,var1,var2,T_2
L_15: var3=T_2;     //114: :=,T_2,_,var3
L_16: printf("output var3: %d\n",var3);     //115: out,var3,_,_
L_17: T_3=1+var2;     //116: +,1,var2,T_3
L_18: var1=T_3;     //117: :=,T_3,_,var1
L_19: gotoL_4;     //118: jump,_,_,103
```

Σε assembly γράφεται:

```
L103:
lw $t1,-12($s0)
lw $t2,-16($s0)
blt $t1,$t2,L105
L104:
b L119
L105:
lw $t1,-12($s0)
li $t2,2
blt $t1,$t2,L107
L106:
b L112
L107:
lw $t1,-16($s0)
li $t2,4
bgt $t1,$t2,L109
L108:
b L112
L109:
li $t1,0
li $t2,32767
sub $t1,$t1,$t2
sw $t1,-28($s0)
L110:
lw $t1,-28($s0)
li $v0,1
move $a0,$t1
syscall
L111:
b L113
```

```
(συνέχεια)
L112:
li $t1,32767
li $v0,1
move $a0,$t1
syscall
L113:
lw $t1,-12($s0)
lw $t2,-16($s0)
add $t1,$t1,$t2
sw $t1,-32($s0)
L114:
lw $t1,-32($s0)
sw $t1,-20($s0)
L115:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
syscall
L116:
li $t1,1
lw $t2,-16($s0)
add $t1,$t1,$t2
sw $t1,-36($s0)
L117:
lw $t1,-36($s0)
sw $t1,-12($s0)
L118:
b L103
```

Ύστερα υπαρχει μια εντολή print που τυπώνει και στα δυο το var3.
Μετά γινεται έλεγχος στο forcase:

```
forcase
    when (var3 > 1): {print (var3); var3:= var3-1}
    when (var3 = 1): {print (var3+var1+var2); var3:= var3-1}
    default: {var3:= var3-10;print(var3)};
```

Το forcase θα εκτελέσει το πρώτο when, θα το τυπώσει και θα το μειώσει κατά 1. Μετά θα παει στην αρχή και θα αυτό θα συνεχιστεί μέχρι το var3 να είναι 1. Τότε το δεύτερο when θα εκτελεστεί. Το var3 θα μειωθεί και στην επόμενη επανάληψη θα εκτελεστεί το default.

Σε int:

120: >,var3,1,122
121: jump,_,_,126
122: out,var3,_,_
123: -,var3,1,T_4
124: :=,T_4,_,var3
125: jump,_,_,120
126: =,var3,1,128
127: jump,_,_,134
128: +,var3,var1,T_5
129: +,T_5,var2,T_6
130: out,T_6,_,_
131: -,var3,1,T_7
132: :=,T_7,_,var3
133: jump,_,_,120
134: -,var3,10,T_8
135: :=,T_8,_,var3
136: out,var3,_,_

Σε c:

```
L_21: if (var3>1) goto L_23;     //120: >,var3,1,122
L_22: goto L_27;     //121: jump,_,_,126
L_23: printf("output var3: %d\n",var3);     //122: out,var3,_,_
L_24: T_4=var3-1;     //123: -,var3,1,T_4
L_25: var3=T_4;     //124: :=,T_4,_,var3
L_26: goto L_21;     //125: jump,_,_,120
L_27: if (var3==1) goto L_29;     //126: =,var3,1,128
L_28: goto L_35;     //127: jump,_,_,134
L_29: T_5=var3+var1;     //128: +,var3,var1,T_5
L_30: T_6=T_5+var2;     //129: +,T_5,var2,T_6
L_31: printf("output T_6: %d\n",T_6);     //130: out,T_6,_,_
L_32: T_7=var3-1;     //131: -,var3,1,T_7
L_33: var3=T_7;     //132: :=,T_7,_,var3
L_34: goto L_21;     //133: jump,_,_,120
L_35: T_8=var3-10;     //134: -,var3,10,T_8
L_36: var3=T_8;     //135: :=,T_8,_,var3
L_37: printf("output var3: %d\n",var3);     //136: out,var3,_,_
```

Σε assembly:

L120:
lw $t1,-20($s0)
li $t2,1
bgt $t1,$t2,L122
L121:
b L126
L122:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
syscall
L123:
lw $t1,-20($s0)
li $t2,1
sub $t1,$t1,$t2
sw $t1,-40($s0)
L124:
lw $t1,-40($s0)

```
sw $t1,-20($s0)
L125:
b L120
L126:
lw $t1,-20($s0)
li $t2,1
beq $t1,$t2,L128
L127:
b L134
L128:
lw $t1,-20($s0)
lw $t2,-12($s0)
add $t1,$t1,$t2
sw $t1,-44($s0)
L129:
lw $t1,-44($s0)
lw $t2,-16($s0)
add $t1,$t1,$t2
sw $t1,-48($s0)
L130:
lw $t1,-48($s0)
li $v0,1
move $a0,$t1
syscall
L131:
lw $t1,-20($s0)
li $t2,1
sub $t1,$t1,$t2
sw $t1,-52($s0)
L132:
lw $t1,-52($s0)
sw $t1,-20($s0)
L133:
b L120
L134:
lw $t1,-20($s0)
li $t2,10
sub $t1,$t1,$t2
sw $t1,-56($s0)
L135:
lw $t1,-56($s0)
sw $t1,-20($s0)
L136:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
syscall
```

Ύστερα δίνεται ως είσοδος η μεταβλητή var4 και γίνεται έλεγχος του statement IF χρητιμοποιώντας και το ELSE.

```
input (var4);
if (var4 <= var3 or var4>6) then {print(-var4);var4 :=-var4}
else print(var4)
```

Σε int:

```
137: inp,var4,_,_
138: <=,var4,var3,142
139: jump,_,_,140
140: >,var4,6,142
141: jump,_,_,147
142: -,0,var4,T_9
143: out,T_9,_,_
144: -,0,var4,T_10
145: :=,T_10,_,var4
146: jump,_,_,148
147: out,var4,_,_
148: halt,_,_,_
149: end_block,c1compatible,_,_
```

Σε c:

```
L_38: printf("var4 input: ");
scanf("%d",&var4);     //137: inp,var4,_,_
L_39: if (var4<=var3) goto L_43;     //138: <=,var4,var3,142
L_40: goto L_41;     //139: jump,_,_,140
L_41: if (var4>6) goto L_43;     //140: >,var4,6,142
L_42: goto L_48;     //141: jump,_,_,147
L_43: T_9=0-var4;     //142: -,0,var4,T_9
L_44: printf("output T_9: %d\n",T_9);     //143: out,T_9,_,_
L_45: T_10=0-var4;     //144: -,0,var4,T_10
L_46: var4=T_10;     //145: :=,T_10,_,var4
L_47: goto L_49;     //146: jump,_,_,148
L_48: printf("output var4: %d\n",var4);     //147: out,var4,_,_
L_49: {}
```

Σε assembly:

```
L137:
li $v0,5
syscall
move $t1,$v0
sw $t1,-24($s0)
L138:
lw $t1,-24($s0)
lw $t2,-20($s0)
ble $t1,$t2,L142
L139:
b L140
L140:
lw $t1,-24($s0)
li $t2,6
bgt $t1,$t2,L142
L141:
b L147
L142:
li $t1,0
lw $t2,-24($s0)
sub $t1,$t1,$t2
sw $t1,-60($s0)
L143:
lw $t1,-60($s0)
li $v0,1
move $a0,$t1
syscall
L144:
li $t1,0
lw $t2,-24($s0)
sub $t1,$t1,$t2
sw $t1,-64($s0)
L145:
lw $t1,-64($s0)
sw $t1,-24($s0)
L146:
b L148
L147:
lw $t1,-24($s0)
li $v0,1
move $a0,$t1
syscall
L148:
li $v0,10
syscall
```

Ο κώδικας τόσο σε c όσο και στον mips επιστρέφει τα ίδια σωστά και έγκυρα αποτελέσματα όπως φαίνεται και παρακάτω.

Ένα παράδειγμα:

Είσοδοι: 1, 4 και -5

Σε c τυπώνεται:

var1 input: 1
var2 input: 4
output 32767: 32767
output var3: 5
output var3: 5
output var3: 5
output var3: 4
output var3: 3
output var3: 2
output T_6: 10
output var3: -10
var4 input: -5
output var4: -5

Σε assembly τυπώνεται το ίδιο:

1

4

3276755543210-10-5

-5

Το προγραμμα 1 σε c:

```c
#include <stdio.h>
int main()
{
 int var1,var2,var3,var4,T_1,T_2,T_3,T_4,T_5,T_6,T_7,T_8,T_9,T_10;
 L_1:
 L_2: printf("var1 input: ");
 scanf("%d",&var1);    //101: inp,var1,_,_
 L_3: printf("var2 input: ");
 scanf("%d",&var2);    //102: inp,var2,_,_
 L_4: if (var1<var2) goto L_6;    //103: <,var1,var2,105
 L_5: goto L_20;    //104: jump,_,_,119
 L_6: if (var1<2) goto L_8;    //105: <,var1,2,107
 L_7: goto L_13;    //106: jump,_,_,112
 L_8: if (var2>4) goto L_10;    //107: >,var2,4,109
 L_9: goto L_13;    //108: jump,_,_,112
 L_10: T_1=0-32767;    //109: -,0,32767,T_1
 L_11: printf("output T_1: %d\n",T_1);    //110: out,T_1,_,_
 L_12: goto L_14;    //111: jump,_,_,113
 L_13: printf("output 32767: %d\n",32767);    //112: out,32767,_,_
 L_14: T_2=var1+var2;    //113: +,var1,var2,T_2
 L_15: var3=T_2;    //114: :=,T_2,_,var3
 L_16: printf("output var3: %d\n",var3);    //115: out,var3,_,_
 L_17: T_3=1+var2;    //116: +,1,var2,T_3
 L_18: var1=T_3;    //117: :=,T_3,_,var1
 L_19: goto L_4;    //118: jump,_,_,103
 L_20: printf("output var3: %d\n",var3);    //119: out,var3,_,_
 L_21: if (var3>1) goto L_23;    //120: >,var3,1,122
 L_22: goto L_27;    //121: jump,_,_,126
 L_23: printf("output var3: %d\n",var3);    //122: out,var3,_,_
 L_24: T_4=var3-1;    //123: -,var3,1,T_4
 L_25: var3=T_4;    //124: :=,T_4,_,var3
 L_26: goto L_21;    //125: jump,_,_,120
 L_27: if (var3==1) goto L_29;    //126: =,var3,1,128
 L_28: goto L_35;    //127: jump,_,_,134
 L_29: T_5=var3+var1;    //128: +,var3,var1,T_5
 L_30: T_6=T_5+var2;    //129: +,T_5,var2,T_6
 L_31: printf("output T_6: %d\n",T_6);    //130: out,T_6,_,_
 L_32: T_7=var3-1;    //131: -,var3,1,T_7
 L_33: var3=T_7;    //132: :=,T_7,_,var3
 L_34: goto L_21;    //133: jump,_,_,120
 L_35: T_8=var3-10;    //134: -,var3,10,T_8
 L_36: var3=T_8;    //135: :=,T_8,_,var3
 L_37: printf("output var3: %d\n",var3);    //136: out,var3,_,_
 L_38: printf("var4 input: ");
 scanf("%d",&var4);    //137: inp,var4,_,_
 L_39: if (var4<=var3) goto L_43;    //138: <=,var4,var3,142
 L_40: goto L_41;    //139: jump,_,_,140
 L_41: if (var4>6) goto L_43;    //140: >,var4,6,142
 L_42: goto L_48;    //141: jump,_,_,147
 L_43: T_9=0-var4;    //142: -,0,var4,T_9
 L_44: printf("output T_9: %d\n",T_9);    //143: out,T_9,_,_
 L_45: T_10=0-var4;    //144: -,0,var4,T_10
 L_46: var4=T_10;    //145: :=,T_10,_,var4
 L_47: goto L_49;    //146: jump,_,_,148
 L_48: printf("output var4: %d\n",var4);    //147: out,var4,_,_
 L_49: {}
}
```

Πρόγραμμα 2:

program1workingfinal.min

Σε αυτό το πρόγραμμα ελέγχονται διάφορες κατηγορίες συναρτήσεων(function, procedure), καθώς και κλήσεις συναρτήσεων από διάφορα σημεία στον κώδικα, όπως συνάρτηση καλείται από την main, συνάρτηση καλείται από άλλη συνάρτηση, συνάρτηση καλεί άλλο υποπρόγραμμα μέσα στην ίδια και συνάρτηση καλεί τον εαυτό της(αναδρομή).Επιπλέον, ελέγχεται το σωστό πέρασμα παραμέτρων(in, inout) και η επιστροφή (return) των συναρτήσεων.

Οι συναρτήσεις/διαδικασίες που υπάρχουν είναι οι: **foo, pro1, f2, encap**. Από αυτές οι foo,f2 και encap είναι function και η pro1 είναι procedure. Ακόμα, η encap είναι εμφωλευμένη και η δήλωσή της γίνεται μέσα στην f2.

Αφού δεν μπορεί να δημιουργηθεί κώδικας .c στο αρχείο .c τυπώνεται:

THE PROGRAM COULD NOT BE CONVERTED TO C FILE PROPERLY BECAUSE OF FUNCTION/PROCEDURE CALLS.

Δήλωση υποπρογραμμάτων:

Πρώτα γίνεται η δήλωση της foo με τυπική παράμετρο x:

```
function foo(in x){
    {x := x+1;
    if (x>10) then print(-x+100) else print(x-100);
    return x+10
    }
}
```

Στο .int αρχείο τυπώνεται :

```
100: begin_block,foo,_,_
101: +,x,1,T_1
102: :=,T_1,_,x
103: >,x,10,105
104: jump,_,_,109
105: -,0,x,T_2
106: +,T_2,100,T_3
107: out,T_3,_,_
108: jump,_,_,111
109: -,x,100,T_4
110: out,T_4,_,_
111: +,x,10,T_5
112: retv,T_5,_,_
113: end_block,foo,_,_
```

Σε κώδικα assembly τυπώνεται:

```
L100:
sw $ra,($sp)
L101:
lw $t1,-12($sp)
li $t2,1
add $t1,$t1,$t2
sw $t1,-16($sp)
L102:
lw $t1,-16($sp)
sw $t1,-12($sp)
L103:
lw $t1,-12($sp)
li $t2,10
bgt $t1,$t2,L105
L104:
b L109
L105:
li $t1,0
lw $t2,-12($sp)
sub $t1,$t1,$t2
sw $t1,-20($sp)
L106:
lw $t1,-20($sp)
li $t2,100
add $t1,$t1,$t2
sw $t1,-24($sp)
L107:
lw $t1,-24($sp)
```

```
(συνέχεια)
li $v0,1
move $a0,$t1
syscall
L108:
b L111
L109:
lw $t1,-12($sp)
li $t2,100
sub $t1,$t1,$t2
sw $t1,-28($sp)
L110:
lw $t1,-28($sp)
li $v0,1
move $a0,$t1
syscall
L111:
lw $t1,-12($sp)
li $t2,10
add $t1,$t1,$t2
sw $t1,-32($sp)
L112:
lw $t1,-32($sp)
lw $t0,-8($sp)
sw $t1,($t0)
L113:
lw $ra,($sp)
jr $ra
```

Στη συνέχεια είναι η διαδικασία pro1:

```
procedure pro1(){
    {
        var3:= var2;
        print(var3);
        print(var2)
    }
}
```

Στο .int αρχείο τυπώνεται :

```
114: begin_block,pro1,_,_
115: :=,var2,_,var3
116: out,var3,_,_
117: out,var2,_,_
118: end_block,pro1,_,_
```

Σε κώδικα assembly τυπώνεται:

```
L114:
sw $ra,($sp)
L115:
lw $t1,-16($s0)
sw $t1,-20($s0)
L116:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
```

```
(συνέχεια)
syscall
L117:
lw $t1,-16($s0)
li $v0,1
move $a0,$t1
syscall
L118:
lw $ra,($sp)
jr $ra
```

Μετά βρίσκεται η f2, αλλά αφού μέσα σε αυτή βρίσκεται η encap πρώτα θα ελεγθεί η encap και μετά η f2.Ακόμα η f2 καλεί και την αδελφή συνάρτησή της foo:

```
function f2(in a,inout b){
    declare y;
    function encap(){      //encapsulated function
        {
        var1:=10;
        return var1
        }
    }
    {
    y:=5*a-20;
    if (y>0) then print(2*b) else return 5*b;
    y:=encap();
    y :=y+b+foo(in b);
    return y
    }
}
```

Συνάρτηση encap:

Στο .int αρχείο τυπώνεται :

```
119: begin_block,encap,_,_
120: :=,10,_,var1
121: retv,var1,_,_
122: end_block,encap,_,_
```

Σε κώδικα assembly τυπώνεται:

```
L119:
sw $ra,($sp)
L120:
li $t1,10
sw $t1,-12($s0)
L121:
```

```
(συνέχεια)
lw $t1,-12($s0)
lw $t0,-8($sp)
sw $t1,($t0)
L122:
lw $ra,($sp)
jr $ra
```

Συνάρτηση f2:

Στο .int αρχείο τυπώνεται :

```
123: begin_block,f2,_,_
124: *,5,a,T_6
125: -,T_6,20,T_7
126: :=,T_7,_,y
127: >,y,0,129
128: jump,_,_,132
129: *,2,b,T_8
130: out,T_8,_,_
131: jump,_,_,134
132: *,5,b,T_9
133: retv,T_9,_,_
```

```
(συνέχεια)
134: par,T_10,RET,_
135: call,encap,_,_
136: :=,T_10,_,y
137: +,y,b,T_11
138: par,b,CV,_
139: par,T_12,RET,_
140: call,foo,_,_
141: +,T_11,T_12,T_13
142: :=,T_13,_,y
143: retv,y,_,_
144: end_block,f2,_,_
```

Σε κώδικα assembly τυπώνεται:

```
L123:
sw $ra,($sp)
L124:
li $t1,5
lw $t2,-12($sp)
mul $t1,$t1,$t2
sw $t1,-24($sp)
L125:
lw $t1,-24($sp)
li $t2,20
sub $t1,$t1,$t2
sw $t1,-28($sp)
L126:
lw $t1,-28($sp)
sw $t1,-20($sp)
L127:
lw $t1,-20($sp)
li $t2,0
bgt $t1,$t2,L129
L128:
b L132
L129:
li $t1,2
lw $t0,-16($sp)
lw $t2,($t0)
mul $t1,$t1,$t2
sw $t1,-32($sp)
L130:
lw $t1,-32($sp)
li $v0,1
```

```
(συνέχεια)
move $a0,$t1
syscall
L131:
b L134
L132:
li $t1,5
lw $t0,-16($sp)
lw $t2,($t0)
mul $t1,$t1,$t2
sw $t1,-36($sp)
L133:
lw $t1,-36($sp)
lw $t0,-8($sp)
sw $t1,($t0)
L134:
addi $fp,$sp,12
addi $t0,$sp,-40
sw $t0,-8($fp)
L135:
sw $sp,-4($fp)
addi $sp,$sp,12
jal L119
addi $sp,$sp,-12
L136:
lw $t1,-40($sp)
sw $t1,-20($sp)
L137:
lw $t1,-20($sp)
lw $t0,-16($sp)
lw $t2,($t0)
```

```
(συνέχεια)
add $t1,$t1,$t2
sw $t1,-44($sp)
L138:
addi $fp,$sp,36
lw $t0,-16($sp)
lw $t0,($t0)
sw $t0,-12($fp)
L139:
addi $t0,$sp,-48
sw $t0,-8($fp)
L140:
sw $sp,-4($fp)
addi $sp,$sp,36
jal L100
addi $sp,$sp,-36
L141:
lw $t1,-44($sp)
lw $t2,-48($sp)
add $t1,$t1,$t2
sw $t1,-52($sp)
L142:
lw $t1,-52($sp)
sw $t1,-20($sp)
L143:
lw $t1,-20($sp)
lw $t0,-8($sp)
sw $t1,($t0)
L144:
lw $ra,($sp)
jr $ra
```

Στο κυρίως πρόγραμμα:

Αρχικά ζητείται μια είσοδος από τον χρήστη:

```
input(var1); //user input
```

Μετά καλείται η foo και γίνεται assign στη var2 μέσω της εντολής:

```
var2:=foo(in var1);
```

με πραγματική παράμετρο την var1 και πέρασμα με τιμή.

Ύστερα καλείται η διαδικασία pro1 χωρίς παραμέτρους :

```
call pro1();
```

Μετά καλείται η f2 με δυο παραμετρους var2 με τιμή και var3 με αναφορά και γίνεται assign στη var1:

```
var1:= f2(in var2,inout var3);
```

Τέλος γίνεται ένα print της var1:

```
print(var1)
```

Στην αρχή του προγράμματος σε assembly υπάρχει μια εντολή **j L145** ώστε να πάει το πρόγραμμα στην αρχή της εκτέλεσης του κύριου μέρους του.

Το κυρίως πρόγραμμα σε int:

145: begin_block,testpWorking,_,_
146: inp,var1,_,_
147: par,var1,CV,_
148: par,T_14,RET,_
149: call,foo,_,_
150: :=,T_14,_,var2
151: call,pro1,_,_
152: par,var2,CV,_
153: par,var3,REF,_
154: par,T_15,RET,_
155: call,f2,_,_
156: :=,T_15,_,var1
157: out,var1,_,_
158: halt,_,_,_
159: end_block,testpWorking,_,_

Το κυρίως πρόγραμμα σε mips assembly:

```
L145:
addi $sp,$sp,32
move $s0,$sp
L146:
li $v0,5
syscall
move $t1,$v0
sw $t1,-12($s0)
L147:
addi $fp,$sp,36
lw $t0,-12($s0)
sw $t0,-12($fp)
L148:
addi $t0,$sp,-24
sw $t0,-8($fp)
L149:
sw $sp,-4($fp)
addi $sp,$sp,36
jal L100
addi $sp,$sp,-36
L150:
lw $t1,-24($s0)
sw $t1,-16($s0)
L151:
addi $fp,$sp,12
sw $sp,-4($fp)
addi $sp,$sp,12
jal L114
addi $sp,$sp,-12
L152:
addi $fp,$sp,56
lw $t0,-16($s0)
sw $t0,-12($fp)
L153:
addi $t0,$sp,-20
sw $t0,-16($fp)
L154:
addi $t0,$sp,-28
sw $t0,-8($fp)
L155:
sw $sp,-4($fp)
addi $sp,$sp,56
jal L123
addi $sp,$sp,-56
L156:
lw $t1,-28($s0)
sw $t1,-12($s0)
L157:
lw $t1,-12($s0)
li $v0,1
move $a0,$t1
syscall
L158:
li $v0,10
syscall
```

Το ιστορικό του πίνακα συμβόλων του προγράμματος:
Symbol table history:

(function foo)
Current nesting level: 1
Current symbol table before scope delete:
Scope #1: [['x', 1, 12], ['T_1', 16], ['T_2', 20], ['T_3', 24], ['T_4', 28], ['T_5', 32]]

Scope #0: [['var1', 12], ['var2', 16], ['var3', 20], ['foo', 1, 101, [1], -1]]

(procedure pro1)
Current nesting level: 1
Current symbol table before scope delete:
Scope #1: []

Scope #0: [['var1', 12], ['var2', 16], ['var3', 20], ['foo', 1, 101, [1], 36], ['pro1', 2, 115, [], -1]]

(function encap μέσα στην f2)
Current nesting level: 2
Current symbol table before scope delete:
Scope #2: []

Scope #1: [['a', 1, 12], ['b', 2, 16], ['y', 20], ['encap', 1, 120, [], -1]]

Scope #0: [['var1', 12], ['var2', 16], ['var3', 20], ['foo', 1, 101, [1], 36], ['pro1', 2, 115, [], 12], ['f2', 1, -1, [1, 2], -1]]

(function f2)
Current nesting level: 1
Current symbol table before scope delete:
Scope #1: [['a', 1, 12], ['b', 2, 16], ['y', 20], ['encap', 1, 120, [], 12], ['T_6', 24], ['T_7', 28], ['T_8', 32], ['T_9', 36], ['T_10', 40], ['T_11', 44], ['T_12', 48], ['T_13', 52]]

Scope #0: [['var1', 12], ['var2', 16], ['var3', 20], ['foo', 1, 101, [1], 36], ['pro1', 2, 115, [], 12], ['f2', 1, 124, [1, 2], -1]]

(main)
Current nesting level: 0
Final/main symbol table: [[['var1', 12], ['var2', 16], ['var3', 20], ['foo', 1, 101, [1], 36], ['pro1', 2, 115, [], 12], ['f2', 1, 124, [1, 2], 56], ['T_14', 24], ['T_15', 28]]]

Final offset of program: 32

Παρατηρήσεις που αφορούν την ορθή λειτουργία του προγράμματος:

Οι κώδικες του ενδιάμεσου κώδικα και του τελικού κώδικα για τις διάφορες εντολές και τα υποπρογράμματα ανταποκρίνονται στις διαφάνειες του μαθήματος.

Μέσα από το ιστορικό του πίνακα συμβόλων φαίνεται το σωστό πέρασμα μεταβλητών και εκτέλεση συναρτήσεων/διαδικασιών με τα πεδία τους.

Κατά τη διάρκεια του προγράμματος υπάρχουν διάφορα print ώστε να ελεγθούν οι λειτουργίες των συναρτήσεων στη Minimal++.Παρακάτω περιγράφεται η χρήση αυτών των εντολών.

- Το πρώτο print εμφανίζεται στη συνάρτηση *foo*: **if (x>10) then print(-x+100) else print(x-100);** , η οποία καλείται από την δεύτερη εντολή του κύριου μέρους: **var2:=foo(in var1);**
- Τα επόμενα δύο print προκύπτουν στη διαδικασία *pro1*: **print(var3); print(var2)**.Ο ίδιος αριθμός θα πρέπει να τυπωθεί (var1 + 11), ο οποίος επιστρέφεται από την προηγούμενη κλήση της foo και μέσα στη pro1 υπάρχει: var3:= var2;
- Το επόμενο print προκύπτει στη συνάρτηση *f2*: **if (y>0) then print(2*b) else return 5*b;** ,όπου y:=5*a-20 και a είναι η πρώτη τυπική παράμετρος της f2 και b η δεύτερη τυπική παράμετρος της f2.
- Η εντολή print που καλείται ύστερα είναι μέσα στη foo,παρόμοια με το πρώτο print και καλείται μέσα από την f2.
- Το τελικό print βρίσκεται στο τέλος του κύριου μέρους του προγράμματος**: print(var1**), όπου **var1:= f2(in var2,inout var3)**, το οποίο πρέπει να είναι είτε: y, αφού από την f2 επιστρέφεται y :=y+b+foo(in b); return y , είτε 5*var3 από την εντολή στην f2: if (y>0) then print(2*b) else return 5*b; , όπου b η δεύτερη τυπική παράμετρος της f2.

Μερικά παραδείγματα εκτέλεσης του κώδικα σε mips:

Είσοδος: 1

Έξοδος: -98 12 12 248745

Είσοδος: 34

Έξοδος: 65 45 45 9054111

Είσοδος: 6

Έξοδος: -93 17 17 348255

Πρόγραμμα 3:

program2working.min

Σε αυτό το αρχείο ελέγχονται συμπληρωματικά οι υποστηριζόμενες δομές της γλώσσας, καθώς και συνδυασμός αυτών: **while** με **ανάθεση**(**:=**), **if** και **print** μέσα του, **input** , κάλεσμα διαδικασίας **call** , **forcase** με **ανάθεση**(**:=**), **while** και **print** μέσα του, αναδρομή διαδικασίας.

Η διαδικασία repeat με είσοδο έναν αριθμό, τυπώνει τον αριθμό αυτό όσες φορές περιγράφονται από τον ίδιο τον αριθμό χρησιμοποιώντας αναδρομή.

```
procedure repeat(in seed){
    {
    print(seed);
    v2:=v2-1;
    if (v2 > 0) then
        call repeat(in seed)
    }
}
```

Στο κυρίως μέρος του προγράμματος το while τυπώνει τους αριθμούς 9-0 και ενδιάμεσα, αν είναι μεγαλύτερος του 5 τυπώνει 0, αλλιώς 1:

```
v1 := 10;
while (v1>0)
    {v1:=v1-1;
    print(v1);
    if (v1>5) then print(0) else print(1)
    };
```

Έξοδος: 908070605141312111101

Μετά υπάρχει ένα input και καλείται η διαδικασία repeat:

```
input(v1);
v2:=v1;
call repeat(in v1);
```

Είσοδος: 7
Έξοδος: 7777777

Ύστερα υπάρχει ένα input και μετά το forcase, όπου τυπώνει από τον αριθμό της εισόδου μέχρι τον αρνητικό αριθμό της και ξανά μέχρι τον αριθμό:

```
forcase
    when (v1 > 0): {print (v1);
                    v1:=v1-1}
    when (v1 >= -v2): {print (v1);
                       v1:=v1-1}
    default: {v1:=v1+1;
              while(v1<v2)
                {v1:=v1+1;
                print (v1)}
             }
```

Είσοδος: 4
Έξοδος: 43210-1-2-3-4-3-2-101234

Τα παρακάτω δύο προγράμματα αποτελούν παραδείγματα χρήσης της γλώσσας για προβλήματα τα οποία έχουν πραγματική λειτουργία, ενώ παράλληλα δοκιμάζεται περαιτέρω η ορθότητα της μετάφρασης σε mips assembly των δομών της Minimal++.

Πρόγραμμα 4:

decimalToBinary.min

Αυτό το πρόγραμμα μετατρέπει μια αριθμητική είσοδο μικρότερη ή ίση του 1023 σε δυαδικό αριθμό. Στη συνέχεια για μια ακόμα είσοδο < = 1023, από τον αριθμό της εισόδου μέχρι το 1 τυπώνει τη δυαδική και τη δεκαδική αναπαράσταση των αριθμών.

Μερικά παραδείγματα:

Είσοδος 1: 9

Έξοδος 1: 1001

Είσοδος 2: 9

Έξοδος 2: 9 1001 8 1000 7 111 6 110 5 101 4 100 3 11 2 10 1 1

Είσοδος 1: 1023

Έξοδος 1: 1111111111

Είσοδος 2: 13

Έξοδος 2: 13 1101 12 1100 11 1011 10 1010 9 1001 8 1000 7 111 6 110 5 101 4 100 3 11 2 10 1 1

Πρόγραμμα 5:

fibonacci.min

Το παρών πρόγραμμα υπολογίζει τους αριθμούς της ακολουθίας Fibonacci. Στην αρχή παίρνεται ως είσοδος από τον χρήστη το επιθυμητό πλήθος μετά την αρχικοποίηση της ακολουθίας (1,1).

Μερικά παραδείγματα:

Είσοδος : 6

Έξοδος : 1 1 2 3 5 8 13 21

Είσοδος : 18

Έξοδος : 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765

Είσοδος : 9

Έξοδος : 1 1 2 3 5 8 13 21 34 55 89

Τα παρακάτω προγράμματα είναι αντιγραμμένα από τις διαφάνειες με τα παραδείγματα τελικού κώδικα για σύγκριση αποτελεσμάτων.

Παράδειγμα*: finalex1.min*

(δικός μου κώδικας)

```
j L106
L100:
sw $ra,($sp)
L101:
lw $t1,-12($sp)
li $t2,1
add $t1,$t1,$t2
sw $t1,-20($sp)
L102:
lw $t1,-20($sp)
lw $t0,-16($sp)
sw $t1,($t0)
L103:
li $t1,4
sw $t1,-20($s0)
L104:
lw $t0,-16($sp)
lw $t1,($t0)
lw $t0,-8($sp)
sw $t1,($t0)
L105:
lw $ra,($sp)
jr $ra
```

(διαφάνειες)

```
L0: b Lmain  (κάνουν το ίδιο πράγμα)
L1:
sw $ra,-0($sp)
L2:
lw $t1,-12($sp)
li $t2,1
add $t1,$t1,$t2
sw $t1,-20($sp)
L3:
lw $t1,-20($sp)
lw $t0,-16($sp)
sw $t1,($t0)
L4:
li $t1,4
sw $t1,-20($s0)
L5:
lw $t0,-16($sp)
lw $t1,($t0)
lw $t0,-8($sp)
sw $t1,($t0)
L6:
lw $ra,-0($sp)
jr $ra
```

```
L106:
addi $sp,$sp,28
move $s0,$sp
L107:
li $t1,1
sw $t1,-12($s0)
L108:
addi $fp,$sp,24
lw $t0,-12($s0)
sw $t0,-12($fp)
L109:
addi $t0,$sp,-16
sw $t0,-16($fp)
L110:
addi $t0,$sp,-24
sw $t0,-8($fp)
L111:
sw $sp,-4($fp)
addi $sp,$sp,24
jal L100
addi $sp,$sp,-24
```

```
Lmain: L7:
addi $sp, $sp, 28
move $s0,$sp
L8:
li $t1,1
sw $t1,-12($sp)
L9:
addi $fp, $sp, 24
lw $t1,-12($sp)
sw $t1,-12($fp)
L10:
addi $t0,$sp,-16
sw $t0,-16($fp)
L11:
addi $t0,$sp,-24
sw $t0,-8($fp)
L12:
sw $sp,-4($fp)
addi $sp, $sp, 24
jal L1
addi $sp, $sp, -24
```

```
L112:
lw $t1,-24($s0)
sw $t1,-20($s0)
L113:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
syscall
L114:
lw $t1,-16($s0)
li $v0,1
move $a0,$t1
syscall
L115:
li $v0,10
syscall
```

```
L13:
lw $t1,-24($sp)
sw $t1,-20($sp)
L14:
lw $t1,-20($sp)
li $v0,1
move $a0, $t1
syscall
L15:
lw $t1,-16($sp)
li $v0,1
move $a0, $t1
syscall
L16:
(σημείωση: η τελευταίες εντολές χρειάζονται στο τέλος του προγράμματος)
L17:
```

Οι κώδικες των προγραμμάτων ταιριάζουν επί το πλείστον, και εκεί που δεν ταιριάζουν δεν υπάρχει κάποια διαφορά στην εκτέλεση του προγράμματος, συγκεκριμένα στο κυρίως μέρος οι καταχωρητές $s0 και $sp αντιστιχούν στην ίδια θέση.

Παράδειγμα: *finalex2.min*

```
(δικός  μου κώδικας)
j L108
L100:
sw $ra,($sp)
L101:
lw $t1,-16($s0)
sw $t1,-20($s0)
L102:
lw $t0,-4($sp)
addi $t0,$t0,-12
lw $t1,($t0)
lw $t0,-4($sp)
addi $t0,$t0,-16
lw $t0,($t0)
sw $t1,($t0)
L103:
lw $ra,($sp)
jr $ra
```

```
(διαφάνειες)
L0:b Lmain
L1:
sw $ra,-0($sp)
L2:
lw $t1,-16($s0)
sw $t1,-20($s0)
L3:
lw $t0,-4($sp)
addi $t0,$t0,-12
lw $t1,($t0)
lw $t0,-4($sp)
addi $t0,$t0,-16
lw $t0,($t0)
sw $t1,($t0)
L4:
lw $ra,-0($sp)
jr $ra
```

(δικός μου κώδικας)

```
L104:
sw $ra,($sp)
L105:
li $t1,2
sw $t1,-16($s0)
L106:
addi $fp,$sp,12
sw $sp,-4($fp)
addi $sp,$sp,12
jal L100
addi $sp,$sp,-12
L107:
lw $ra,($sp)
jr $ra


L108:
addi $sp,$sp,24
move $s0,$sp
L109:
li $t1,3
sw $t1,-12($s0)
L110:
li $t1,4
sw $t1,-16($s0)
L111:
addi $fp,$sp,20
lw $t0,-12($s0)
sw $t0,-12($fp)
L112:
addi $t0,$sp,-16
sw $t0,-16($fp)
L113:
sw $sp,-4($fp)
addi $sp,$sp,20
jal L104
addi $sp,$sp,-20
L114:
lw $t1,-16($s0)
li $v0,1
move $a0,$t1
syscall
L115:
lw $t1,-20($s0)
li $v0,1
move $a0,$t1
syscall
L116:
li $v0,10
syscall
```

(διαφάνειες)

```
L5:
sw $ra,-0($sp)
L6:
li $t1,2
sw $t1,-16($s0)
L7:
addi $fp, $sp, 12
sw $sp, -4($fp)
addi $sp, $sp, 12
jal L1
addi $sp, $sp, -12
L8:
lw $ra,-0($sp)
jr $ra


Lmain: L9:
addi $sp, $sp, 24
move $s0,$sp
L10:
li $t1,3
sw $t1,-12($sp)
L11:
li $t1,4
sw $t1,-16($sp)
L12:
addi $fp, $sp, 20
lw $t1,-12($sp)
sw $t1,-12($fp)
L13:
addi $t0, $sp, -16
sw $t0, -16($fp)
L14:
sw $sp, -4($fp)
addi $sp, $sp, 20
jal L5
addi $sp, $sp, -20
L15:
lw $t1,-16($sp)
li $v0,1
move $a0, $t1
syscall
L16:
lw $t1,-20($sp)
li $v0,1
move $a0, $t1
syscall
L17:
```

(οι εντολές χρειάζονται στο τέλος)

```
L18:
```

Όπως και πριν, οι κώδικες των προγραμμάτων ταιριάζουν επί το πλείστον, και εκεί που δεν ταιριάζουν δεν υπάρχει κάποια διαφορά στην εκτέλεση του προγράμματος, συγκεκριμένα στο κυρίως μέρος οι καταχωρητές $s0 και $sp αντιστιχούν στην ίδια θέση.

Σε όλες τις φάσεις του κώδικα γίνονται έλεγχοι λαθών.
Συγκεκριμένα:

-Στο λεκτικό αναλυτή ελέγχονται χαρακτήρες που δεν ανήκουν στη γλώσσα.

-Στο συντακτικό αναλυτή ελέγχεται η σωστή σύνταξη των δεσμευμένων λέξεων, καθώς και η σωστή σειρά των στοιχείων που δομούν τη γλώσσα(πχ:σωστό άνοιγμα παρενθέσεων, αγκυλών, κλπ).

-Στις υπόλοιπες φάσεις (ενδιάμεσος κώδικας, πίνακας συμβόλων, παραγωγή τελικού κώδικα) γίνονται σε διάφορα σημεία έλεγχοι (όπως ονόματα μεταβλήτων, πεδίων συναρτήσεων, ορατότητας μεταβλητών/συναρτήσεων) που συμβάλλουν στο "πιάσιμο" λαθών ώστε να παραχθεί επιτυχημένα ο τελικός κώδικας mips assembly.