


HY340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

```
VAR i:Integer;  
FUNCTION(Symbol) replicate  
  x = (function(x,y){return x*y;});  
  class DelFunctor: public std::unary_function<
```

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

1




HY340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

Διάλεξη 11η
ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ – III (τελευταία)

HY340 Α. Σαββίδης Slide 2 / 26

2




Περιεχόμενα

- **Εκφράσεις**
- Εντολή διακλάδωσης
- Ανακυκλώσεις
- Ειδικές εντολές

HY340 Α. Σαββίδης Slide 3 / 26

3



Εκφράσεις (1/4)

- Αριθμητικών τελεστών
- Συσχετιστικών τελεστών
- Λογικών τελεστών

HY340 Α. Σαββίδης Slide 4 / 26

4



```
arithop → + { $arithop = add; }
arithop → - { $arithop = sub; }
arithop → * { $arithop = mul; }
arithop → / { $arithop = div; }
arithop → % { $arithop = mod; }
```

Εδώ επίσης δεν δείχνουμε και την εφαρμογή της τακτικής επαναχρησιμοποίησης κρυφών μεταβλητών, ωστόσο μπορεί να εφαρμοστεί κανονικά.

```

expr → expr1 arithop expr2
{
    $expr = newexpr(arithexpr_e);
    $expr->sym = newtemp();
    emit($arithop, $expr1, $expr2, $expr);
}

```

$a = (x+y) - (z*w)$	1: ADD x y t1
	2: MUL z w t2
	3: SUB t1 t2 t3
	4: ASSIGN t3 a

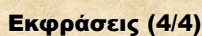
Η παραπάνω είναι η πολύ απλή περίπτωση υλοποίησης. Υπάρχουν και οι εξής περιπτώσεις που πρέπει να ελεγχθούν:

- ❑ Εάν κάποιο expression είναι ήδη γνωστού τύπου που δεν επιτρέπεται να συμμετέχει σε αριθμητική έκφραση, όπως: *programfunc_e*, *libraryfunc_e*, *boolexpr_e*, *newtable_e*, *constbool_e*, *conststring_e* και *nil_e*.
- ❑ Εάν έχουμε δύο expressions τύπου *costnum_e*, τότε υπολογίζουμε το αποτέλεσμα και το *δερφρ* γίνεται και αυτό τύπου *costnum_e*.

HY340

A. Σαββίδης

Slide 5 / 26



```
boolop → && { $boolop = and;
boolop → || { $boolop = or; }
```

```
boolop → && { $boolop = and;
boolop → || { $boolop = or; }
```

a = x y && z;	1: AND y z t1
	2: OR x t1 t2
	3: SETCCM t2

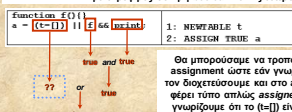
```

expr → expr1 boolop expr2
{
    $expr = newexpr(boolexpr_e);
    $expr->sym = newtemp();
    emit($boolop, $expr1, $expr2, $expr);
}

```

Η μέθοδος μερικής αποτίμησης είναι ιδιαίτερα πονηρή, εσείς ακολουθείτε πιστά τη μέθοδο (φροντιστήριο)

Και πάλι ισχύουν όσα και στις προηγούμενες περιπτώσεις εκφράσεων, μόνο που όταν έχουμε σταθερές τιμές ή συναρτήσεις ή δυναμική κατασκευή πίνακα, γνωρίζουμε ότι είναι μετατρέψιμες σε τιμή boolean και τις θεωρούμε constants.

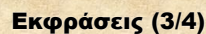


Θα μπορούσαμε να τροποποιήσουμε την υλοποίηση του assignment ώστε εάν γνωρίζουμε τον τύπο του r-value να τον διοχετεύσουμε και στο `assign expression` (το οποίο τώρα φέρει τύπο απλώς `assignexpr_e`). Έτσι θα μπορούσαμε να γνωρίζουμε ότι το `(t[=])` είναι δυναμικός πίνακας που είναι `true`.

HY340

A. Σαββιῶνος

Slide 7 / 26



```
relop → >      { $relop = if_greater; }
relop → >=     { $relop = if_greatereq; }
```

$rel_{op} \rightarrow >$	{ \$rel_{op} = if_greater; }
$rel_{op} \rightarrow >=$	{ \$rel_{op} = if_greater_eq; }
$rel_{op} \rightarrow <$	{ \$rel_{op} = if_less; }
$rel_{op} \rightarrow <=$	{ \$rel_{op} = if_less_eq; }
$rel_{op} \rightarrow ==$	{ \$rel_{op} = if_eq; }
$rel_{op} \rightarrow !=$	{ \$rel_{op} = if_not_eq; }

a = x > y;	1: IF GREATEREQ x y 4 2: ASSIGN t1 FALSE 3: JUMP 5 4: ASSIGN t1 TRUE 5: ASSIGN t1 a
------------	---

b = f()>g();	1: CALL f
	2: GETRETVAL _t1
	3: CALL g
	4: GETRETVAL _t2
	5: IF GREATEREQ _t1 _t2 B
	6: ASSIGN t3 FALSE
	7: JUMP 9
	8: ASSIGN t3 TRUE
	9: ASSIGN _t3 b

```

expr → expr1 relop expr2
{
    $expr = newexpr(boolexpr_e);
    $expr->sym = newtemp();

    emit($relop, $expr1, $expr2, nextquad()+3);
    emit(assign, newexpr_constbool(0), $expr);
    emit(jump, nextquad()+2);
    emit(assign, newexpr_constbool(1), $expr);
}

```

Και σε αυτή την περίπτωση παραλείπονται οι στατικοί έλεγχοι συμφωνίας τύπων αλλά και υπολογισμού τιμής της λογικής έκφρασης από σταθερά ορίσματα (στην περίπτωση αυτή προφανώς δεν παράγεται κώδικας αφού ο υπολογισμός γίνεται από τον compiler).

HY340

A. Σαββίδης

Slide 6 / 26



- Εκφράσεις
- **Εντολή διακλάδωσης**
- Ανακυκλώσεις
- Ειδικές εντολές

8

Εντολή διακλάδωσης (1/4)

if (expr) stmt (1/2)

Ενδιάμεσος κώδικας που παράγεται λόγω του *expr*

```
IF EQ $expr TRUE
JUMP
```

Ενδιάμεσος κώδικας που παράγεται λόγω του *stmt*

```
IF (x >= y) {
  1: ADD x y t1
  2: SUB z w t2
  3: IF GREATEREQ t1 t2 6
  4: ASSIGN FALSE t3
  5: JUMP 7
  6: ASSIGN TRUE t3
  7: IF EQ TRUE t3 9
  8: JUMP 14
  9: PARAM x
  10: PARAM y
  11: PARAM z
  12: CALL f
  13: GETRETVAR t4
  14:
}
```

Ενδολεκτικό μπορούμε να έχουμε IF, NE με jump στο πρώτο quad μετά το stmt που σημαίνει ότι δεν απαιτείται το JUMP.

Πρέπει με κάποιο τρόπο να μπορούσαμε να προκαλέσουμε την παραγωγή του ενδιάμεσου κώδικα των δύο αυτών εντολών ανάμεσα στο *expr* και στο *stmt*.

Επίσης, ο μόνος τρόπος να γνωρίζουμε ακριβώς την θέση των τελικών quads και για τις δύο εντολές είναι αφού έχει παραχθεί ο κώδικας για το *stmt*.

Το πρώτο quad του *stmt*

Το πρώτο quad ακριβώς μετά το *stmt*

HY340 A. Σαββίδης Slide 9 / 26

9

Εντολή διακλάδωσης (2/4)

if (expr) stmt (2/2)

ifprefix → if (expr)

```
{
  emit(
    if_eq, $expr,
    newexpr_constbool(1),
    nextquad() + 2
  );
  $ifprefix = nextquad();
  emit(jump, NULL, NULL, 0);
}

if → ifprefix stmt
{
  patchlabel($ifprefix, nextquad());
}
```

void patchlabel(unsigned quadNo, unsigned label) {
 assert(quadNo < curQuad && lquad[quadNo].label);
 }
 quads[quadNo].label = label;
}
 expr* newexpr_constbool(unsigned int b) {
 expr* e = newexpr(constbool_e);
 e->boolConst = !!b;
 return e;
}
 unsigned nextquad(void) { return curQuad; }

Βλέπουμε ότι λύνουμε εύκολα το πρόβλημα παραγωγής κώδικα ανάμεσα στο *expr* και *stmt* με την εισαγωγή ενός επιπλέον κανόνα ο οποίος παράγει τις δύο αναγκαίες εντολές if_eq και jump.

Ουσιαστικά ο σημασιολογικός κανόνας του *ifprefix* καλείται μετά την παραγωγή του κώδικα για το *expr* αλλά πάντα πριν την παραγωγή του κώδικα για το *stmt* (καθώς το *ifprefix* γίνεται reduced πριν καν γίνει parsed οπότε παράγεται από το *stmt*).

HY340 A. Σαββίδης Slide 10 / 26

10

Εντολή διακλάδωσης (3/4)

if (expr) stmt else stmt (1/2)

Ενδιάμεσος κώδικας που παράγεται λόγω του *expr*

```
IF EQ $expr TRUE
JUMP
```

Ενδιάμεσος κώδικας που παράγεται λόγω του *stmt₁*

```
JUMP
```

Ενδιάμεσος κώδικας που παράγεται λόγω του *stmt₂*

```
IF (a || b) {
  1: OR a b t1
  2: IF EQ TRUE t1 4
  3: JUMP 6
  4: ASSIGN y x
  5: JUMP 7
  6: ASSIGN z y
  7:
}
```

Και πάλι έχουμε παρόμοιο πρόβλημα με πριν καθώς είναι αναγκαίο να εισάγουμε ένα JUMP ανάμεσα στον κώδικα των δύο statements.

Και πάλι εφαρμόζουμε την ίδια λύση με την εισαγωγή ενός γραμματικού κανόνα που γίνεται πάντα reduced μετά το 1ο *stmt* αλλά πάντα πριν το 2ο *stmt*.

Το πρώτο quad του *stmt₁*

Το πρώτο quad ακριβώς μετά το *stmt₂*

HY340 A. Σαββίδης Slide 11 / 26

11

Εντολή διακλάδωσης (4/4)

if (expr) stmt else stmt (2/2)

elseprefix → else

```
{
  $elseprefix = nextquad();
  emit(jump, NULL, NULL, 0);
}

if → ifprefix stmt, elseprefix stmt2
{
  patchlabel($ifprefix, $elseprefix + 1);
  patchlabel($elseprefix, nextquad());
}
```

Η τιμή του *elseprefix* είναι ο αριθμός του quad ακριβώς πριν το 1ο quad του *stmt₂*, και αντιστοιχεί σε ένα εμβόλιμο JUMP που ακολουθεί την εκτέλεση του *stmt₁*.

Η τιμή του *ifprefix* αντιστοιχεί στο JUMP στην περίπτωση που το if expression είναι false, άρα θα πρέπει να οδηγήσει στο 1ο quad του *stmt₂*. Όμως από τα προηγούμενα, ο αριθμός αυτού του quad είναι ακριβώς *\$elseprefix*+1, άρα αρκεί να κάνουμε patch το target label του quad *\$ifprefix* με την τιμή αυτή.

Τώρα, καθώς το quad στη θέση *\$elseprefix* αντιπροσωπεύει το JUMP όταν περαστεί η εκτέλεση του *stmt₁*, θα πρέπει να γίνει patched με τον αριθμό του 1ου quad αμέσως μετά το *stmt₂*. Όμως τη στιγμή που εκτελείται ο σημασιολογικός κανόνας, η τιμή αυτή είναι ακριβώς το *nextquad*.

HY340 A. Σαββίδης Slide 12 / 26

12

Περιεχόμενα

- Εκφράσεις
- Εντολή διακλάδωσης
- **Ανακυκλώσεις**
- Ειδικές εντολές

HY340 Α. Σαββίδης Slide 13 / 26

13

Ανακυκλώσεις (1/5)

while (expr) stmt (1/2)

Το πρώτο quad του *expr*

Ενδιάμεσος κώδικας που παράγεται λόγω του *expr*

IF EQ *Sexpr* TRUE

JUMP

Το πρώτο quad του *stmt*

Ενδιάμεσος κώδικας που παράγεται λόγω του *stmt*

break list

continue list

JUMP

Το πρώτο quad ακριβώς μετά το *stmt*

+2

```

1: SUB x y t1
2: SUB z w t2
3: IF GREATER t1 t2 6
4: ASSIGN FALSE t3
5: JUMP 7
6: ASSIGN TRUE t3
7: IF EQ TRUE t3 9
8: JUMP 16
9: IF EQ TRUE b 11
10: JUMP 13
11: JUMP 16
12: JUMP 15
13: SUB x z t4
14: ASSIGN t4 x
15: JUMP 1
16:

```

while (x > y) { if (b) break; x = x + z; }

• Εδώ οι ανάγκες είναι ίδιες με το απλό if ως προς την πρώτη φορά εκτέλεσης. Όμως επειδή πρόκειται για ανακύκλωση (δηλ. επαναλαμβανόμενο if) πρέπει στο τέλος να έχουμε πάντα ένα jump στην αρχή, στο 1ο quad υπολογισμού του *expr*.

• Επίσης, καθώς μέσα στο *stmt* μπορεί να υπάρχουν εντολές break ή continue, θα δούμε ότι αυτές προκαλούν την παραγωγή μη συμπληρωμένων jumps (unfinished) τα οποία κρατούνται σε μια λίστα ώστε να συμπληρωθούν από τον σημερινό κανόνα του while statement. Προφανώς το break πρέπει να γίνεται patched στο quad που ακριβώς έπεται του *stmt*, ενώ το continue στο 1ο quad υπολογισμού του while expression.

HY340 Α. Σαββίδης Slide 14 / 26

14

Ανακυκλώσεις (2/5)

while (expr) stmt (2/2)

```

whilestart → while
{
    $whilestart = nextquad();
}
whilecond → (expr)
{
    emit(
        if_eq,
        $expr,
        newexpr_constbool(1),
        nextquad() + 2
    );
    $whilecond = nextquad();
    emit(jump, NULL, NULL, 0);
}
while → whilestart whilecond stmt
{
    emit(jump, NULL, NULL, $whilestart);
    patchlabel($whilecond, nextquad());
    patchlist($stmt.breaklist, nextquad());
    patchlist($stmt.contlist, $whilestart);
}

```

• Στο *whilestart* κρατάμε το 1ο quad του κώδικα υπολογισμού της συνθήκης της εντολής while.

• Στο *whilecond* παράγουμε τον κώδικα που ελέγχει την τιμή του while expression και κάνει αντίστοιχο jump ή στην αρχή του *stmt* ή στο τέλος του. Καθώς ο αριθμός του quad αμέσως μετά το τέλος του *stmt* δεν είναι γνωστός, κρατάμε τον αριθμό του jump quad ως γνώρισμα του *whilecond* για να το κάνουμε patch στον βασικό κανόνα.

• Στο τέλος του while *stmt* πάντα υπάρχει JUMP στην αρχή του, με τη γνήσιου quad να είναι αποθηκευμένη στο γνώρισμα του *whilestart*.

HY340 Α. Σαββίδης Slide 15 / 26

15

Ανακυκλώσεις (3/5)

for (elist; expr; elist) stmt (1/3)

Ενδιάμεσος κώδικας που παράγεται λόγω του *elist1*

Το πρώτο quad του *expr*

Ενδιάμεσος κώδικας που παράγεται λόγω του *expr*

IF EQ *Sexpr* TRUE

JUMP

Το πρώτο quad του *elist2*

Ενδιάμεσος κώδικας που παράγεται λόγω του *elist2*

JUMP

Το πρώτο quad του *stmt*

Ενδιάμεσος κώδικας που παράγεται λόγω του *stmt*

break list

continue list

JUMP

Το πρώτο quad ακριβώς μετά το *stmt*

loop

closure

Κάθε for loop μπορεί να γραφτεί ως ένα ισοδύναμο while loop.

```

for (elist1; expr; elist2) stmt
elist1; while (expr) { stmt elist2; }

```

• Οι συνδέσεις είναι υποχρεωτικές καθώς ο κώδικας παράγεται πάντα με τη σειρά αναγνώρισης των γραμματικών συμβόλων, που είναι και η σειρά εμφάνισης στο RHS, δηλ. *elist1*, *expr*, *elist2* και έπειτα *stmt*.

• Άρα, θα πρέπει να φροντίσουμε και πάλι να έχουμε εμβόλιμα τα JUMP quads, ώστε να υποστηρίξεται η σημασιολογία του for loop, μέσω ειδικών γραμματικών συμβόλων και παραγωγών.

HY340 Α. Σαββίδης Slide 16 / 26

16



```

forprefix  $\rightarrow$  for (elist; M expr;
{
    $forprefix.test = SM;
    $forprefix.enter = nextquad();
    emit_if_eq, Sexpr, newexpr_constbool(1, 0);
}

for  $\rightarrow$  forprefix  $N_1$  elist  $N_2$  stmt  $N_3$ 
{
    patchlabel($forprefix.enter.SN;+1);  $\leftarrow$  true jump
    patchlabel(SN;+quad(0));  $\leftarrow$  false jump
    patchlabel(SN, $forprefix.test);  $\leftarrow$  loop jump
    patchlabel(SN, SN;+1);  $\leftarrow$  closure jump

    patchlist($stmt.breaklist, nextquad());
    patchlist($stmt.constlist, SN;+1);
}

```

•Επίσης, πρέπει να κρατήσουμε και τον αριθμό του quad όπου γίνεται test to for expression, καθώς πρέπει να κάνουμε patch to jump ακριβώς στο πρώτο quad του *stmt*.

Slide 17 / 26

17



Ανακυκλώσεις (5/5)

```

      elist1      expr      elist2
for ( i = 0; i < N; ++i )
    print( "*" );
      stmt

```

```

graph TD
    n1["1: ASSIGN 0 i"] --> n2["2: IF LESS i N 5"]
    n2 --> n3["3: ASSIGN FALSE t1"]
    n3 --> n4["4: JUMP 6"]
    n4 --> n5["5: ASSIGN TRUE t1"]
    n5 --> n6["6: IF EQ TRUE t1 10"]
    n6 -- "true jump" --> n7["7: JUMP 14"]
    n6 -- "false jump" --> n8["8: ADD 1 i"]
    n8 -- "loop jump" --> n9["9: JUMP 2"]
    n9 --> n2
    n9 --> n10["10: PARAM \"*\""]
    n10 --> n11["11: CALL \"print\" t2"]
    n11 --> n12["12: GETRIVAL t2"]
    n12 -- "closure jump" --> n13["13: JUMP 6"]
    n13 --> n6
    n13 --> n14["14:"]
  
```

The diagram illustrates a control flow graph with the following nodes and edges:

- Node 1:** `1: ASSIGN 0 i`
- Node 2:** `2: IF LESS i N 5`
- Node 3:** `3: ASSIGN FALSE t1`
- Node 4:** `4: JUMP 6`
- Node 5:** `5: ASSIGN TRUE t1`
- Node 6:** `6: IF EQ TRUE t1 10`
 - Edge to Node 7: `7: JUMP 14` (labeled **true jump**)
 - Edge to Node 8: `8: ADD 1 i` (labeled **false jump**)
- Node 7:** `7: JUMP 14`
- Node 8:** `8: ADD 1 i`
- Node 9:** `9: JUMP 2` (labeled **loop jump**)
- Node 10:** `10: PARAM "*"`
- Node 11:** `11: CALL "print" t2`
- Node 12:** `12: GETRIVAL t2`
- Node 13:** `13: JUMP 6` (labeled **closure jump**)
- Node 14:** `14:`

Slide 18 / 26

18



Περιεχόμενα

- Εκφράσεις
- Εντολή διακλάδωσης
- Ανακυκλώσεις
- **Ειδικές εντολές**

Slide 19 / 26

19



Ειδικές εντολές (1/6)

- **break** και **continue**
- **return** [*expr*]

Slide 20 / 26

20



Ειδικές εντολές (2/6)

- Το **break** πρέπει να προκαλεί αλλαγή ροής ελέγχου ώστε να έχουμε άμεση έξοδο από το εκάστοτε loop.
- Το **continue** πρέπει να προκαλεί άμεση κλείσιμο της παρούσας ανακύκλωσης και έναρξη της επόμενης στο εκάστοτε loop.
- Καθώς και τα δύο χρησιμοποιούνται μόνο μέσα σε loop θα πρέπει να μπορείτε να ελέγχετε εάν βρίσκονται μέσα σε κώδικα κάποιου loop.
- Καθώς και τα δύο κάνουν jumps σε σημεία (quads) του ενδιαμέσου κώδικα που δεν είναι γνωστά μέσα στον ίδιο τον γραμματικό τους κανόνα, γίνονται emit ως unfinished jumps.

HY340

A. Σαββίδης

Slide 21 / 26

21



Ειδικές εντολές (3/6)

```
loopstart → ε { ++loopcounter; }
loopend → ε { --loopcounter; }
loopstmt → loopstart stmt loopend { $ loopstmt = $stmt; }
```

```
whilestmt → while ( expr ) loopstmt
forstmt → for ( elist; expr; elist ) loopstmt
```

```
funcblockstart → ε { push(loopcounterstack, loopcounter); loopcounter=0; }
funcblockend → ε { loopcounter = pop(loopcounterstack); }
```

```
funcdef → function [id] (idlist) funcblockstart block funcblockend
```

Κώδικας	loopcounter stack και current value
x = f();	[0]
while (x) {	[8]
function g() {	[1][0], push, reset
break;	[1][0], Άρα compile error ("not in a loop")
while (true) {	[1][1]
continue;	[1][1], Νόημα, μέση σε loop
}	[1][0]
break;	[1], pop, restore
}	[1], Νόημα, μέση σε loop
	[0]

• Δείχνουμε μόνο το τμήμα των σημασιολογικών κανόνων για τη διαχείριση του loopcounter.

• Προφανώς η λογική που παρουσιάζεται πρέπει να αναμορφωθεί στους κανόνες που έχουμε ήδη παρουσιάσει για τις ανακυκλώσεις.

HY340

A. Σαββίδης

Slide 22 / 26

22



Ειδικές εντολές (4/6)

```
break → break;
{ make_stmt(&$break);
  $break.breaklist = newlist(nextquad()); emit(jump, NULL, NULL, 0);
}
continue → continue;
{ make_stmt(&$continue);
  $continue.contlist = newlist(nextquad()); emit(jump, NULL, NULL, 0);
}
stmts → stmt { $stmts = $stmt; }
stmts → stmts, stmt
{ $stmts.breaklist = mergelist($stmts.breaklist, $stmt.breaklist);
  $stmts.contlist = mergelist($stmts.contlist, $stmt.contlist);
}
```

• Για την υποστήριξη των break και continue, εισάγουμε δύο γνωρίσματα στο γραμματικό σύμβολο stmt: (α) την λίστα breaklist, που περιέχει τους αριθμούς όλων των quads που καθενα αντιστοιχεί σε unfinished jump λόγω κάποιου break που περιέχεται στο stmt, και (β) το contlist αντίστοιχα, με τους αριθμούς όλων των quads που καθενα αντιστοιχεί σε unfinished jump λόγω κάποιου continue που περιέχεται στο stmt.

• Για τον διπλό πηγάιο κώδικα, και για το stmt που αντιστοιχεί σε όλο το block του loop, σκιαγραφούμε τις εντολές που θα δώσουν quad indices που τελικά θα περιλαμβάνονται στις δύο αυτές λίστες.

```
for (i=0; i<N; ++i)
{
  if (f(i) > y)
    break;
  else {
    x = g(y);
    continue;
  }
  h(i);
}
```

HY340

A. Σαββίδης

Slide 23 / 26

23



Ειδικές εντολές (5/6)

Παράδειγμα για παραγωγή και συμπλήρωση των jumps για break και continue

```
while (a) {
  1: IF_EQ TRUE a 3
  2: JUMP 15 // go exit

  a = f(a);
  3: PARAM a
  4: CALL f
  5: GETRETVL _t1
  6: ASSIGN _t1 a

  if (a)
  7: IF_EQ a TRUE 9 // go if
  break;
  8: JUMP 11 // go else
else
  9: JUMP 15 // break
  continue;
  10: JUMP 12 // skip else
  11: JUMP 1 // continue

  continue;
  12: JUMP 1 // continue
  break;
  13: JUMP 15 // break

  14: JUMP 1 // go loop
  15: JUMP 1
```

HY340

A. Σαββίδης

Slide 24 / 26

24



Ειδικές εντολές (6/6)

- Το **return** πρέπει να επιτρέπεται μόνο μέσα σε συναρτήσεις, άρα απαιτείται και κάποιος *infunction* μετρητής που τον διαχειρίζεστε ώστε να είναι >0 εάν το παρόν return stmt είναι μέσα στο block κάποιας συνάρτησης.

```
returnstmt → return e ;  
{ emit(return, null); }  
returnstmt → return expr ;  
{ emit(return, $expr); }
```

function f() {	1: FUNCSTART f
return;	2: RETURN
return local x;	3: RETURN x
return nil;	3: RETURN nil
}	4: FUNCEND f

HY340

Α. Σαββίδης

Slide 25 / 26

25



Quad lists

```
struct stmt_t {  
    int breakList, contList;  
};  
  
void make_stmt (stmt_t* s)  
{ s->breakList = s->contList = 0; }  
  
int newList (int i)  
{ quads[i].label = 0; return i; }
```



```
int mergelist (int l1, int l2) {  
    if (!l1)  
        return l2;  
    else  
        return l1;  
    if (!l2)  
        return l1;  
    else {  
        int i = l1;  
        while (quads[i].label)  
            i = quads[i].label;  
        quads[i].label = l2;  
        return l1;  
    }  
}  
  
void patchlist (int list, int label) {  
    while (list) {  
        int next = quads[list].label;  
        quads[list].label = label;  
        list = next;  
    }  
}
```

HY340

Α. Σαββίδης

Slide 26 / 26

26