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

Διάλεξη 11η
ΠΑΡΑΓΩΓΗ ΕΝΔΙΑΜΕΣΟΥ ΚΩΔΙΚΑ – III (τελευταία)

Περιεχόμενα

Εκφράσεις (1/4)

- Αριθμητικών τελεστών
- Συσχετιστικών τελεστών
- Λογικών τελεστών

Εκφράσεις (2/4)

Αριθμητικών τελεστών

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

expr → $expr_1$ *arithop* $expr_2$
{
$*expr* = *newexpr*(*arithexpr_e*);
$*expr*->*sym* = *newtemp*();
emit($*arithop*, $$expr_1$, $$expr_2$, $*expr*);
}

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

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 $expr γίνεται και αυτό τύπου *constnum_e*.

Εκφράσεις (3/4)

Συσχετιστικών τελεστών

relop → > { $*relop* = **if_greater**; }
relop → >= { $*relop* = **if_greatereq;** }
relop → < { $*relop* = **if_less**; }
relop → <= { $*relop* = **if_lesseq**; }
relop → == { $*relop* = **if_eq**; }
relop → != { $*relop* = **if_noteq**; }

expr → $expr_1$ *relop* $expr_2$
{
$*expr* = *newexpr*(*boolexpr_e*);
$*expr*->*sym* = *newtemp*();

emit($*relop*, $$expr_1$, $$expr_2$, *nextquad*()+3);
emit(**assign**, *newexpr_constbool*(0), $*expr*);
emit(**jump**, *nextquad*()+2);
emit(**assign**, *newexpr_constbool*(1), $*expr*);

}

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 8 6: ASSIGN _t3 FALSE 7: JUMP 9 8: ASSIGN _t3 TRUE 9: ASSIGN _t3 b

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).

Εκφράσεις (4/4)

Λογικών τελεστών (ολική αποτίμηση)

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

a = x || y && z;	1: AND y z _t1 2: OR _x _t1 _t2 3: ASSIGN _t2 a

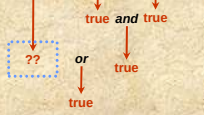
expr → $expr_1$ *boolop* $expr_2$
{
$*expr* = *newexpr*(*boolexpr_e*);
$*expr*->*sym* = *newtemp*();
emit($*boolop*, $$expr_1$, $$expr_2$, $*expr*);
}

Η μέθοδος μερικής αποτίμησης είναι ιδιαίτερα πολύπλοκη

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

```
function f(){}
a = (t=[]) || f && print;
```

```
1: NEWTABLE t
2: ASSIGN TRUE a
```

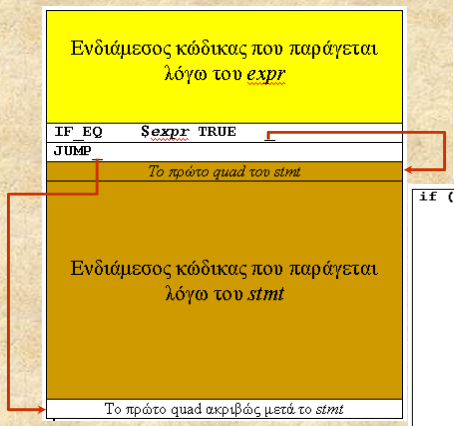


Θα μπορούσαμε να τροποποιήσουμε την υλοποίηση του assignment ώστε εάν γνωρίζουμε τον τύπο του r-value να τον διοχετεύσουμε και στο assign expression (το οποίο τώρα φέρει τύπο απλώς *assignexpr_e*). Έτσι θα μπορούσαμε να γνωρίζουμε ότι το (t=[]) είναι δυναμικός πίνακας που είναι true.

Περιεχόμενα

Εντολή διακλάδωσης (1/4)

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



•Εναλλακτικά 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 μπορέσουμε 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*.

```
if (x+y >= z-w)          1: ADD x y _t1
    f(x,y,z);            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
                                              go if
                         9: PARAM x
                         10:PARAM y
                         11:PARAM Z
                         12:CALL f
                         13:GETRETVAL _t4
                                              skip if
                         14:
```


Εντολή διακλάδωσης (2/4)

if (*expr*) *stmt* (2/2)

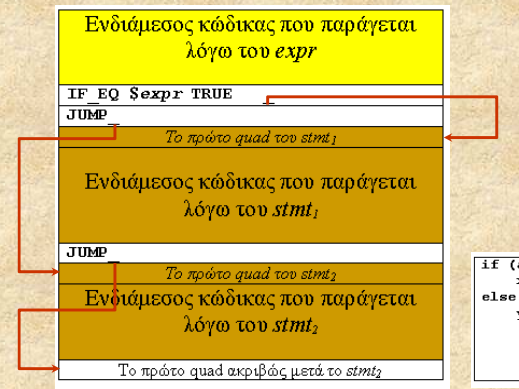
ifprefix → **if** (*expr*)
```
{
    emit(if_eq, $expr, newexpr_constbool(1), nextquad()+2);
    $ifprefix = nextquad();
    emit(jump, _);
}
```
if → *ifprefix stmt*
```
{
    patchlabel($ifprefix, nextquad());
}
```

•Βλέπουμε ότι λύνουμε εύκολα το πρόβλημα παραγωγής κώδικας 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 κώδικας για το *stmt* (καθώς το *ifprefix* γίνεται reduced πριν καν γίνει parsed οτιδήποτε παράγεται από το *stmt*).

Εντολή διακλάδωσης (3/4)

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



•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*.

```
if (a || b)          1: OR a b _t1
    x = y;           2: IF_EQ TRUE _t1 4
else                 3: JUMP 6
    y = z;           4: ASSIGN y x
                     5: JUMP 7
                     6: ASSIGN z y
                     7:
```


Εντολή διακλάδωσης (4/4)

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

elseprefix → **else**
```
{
    $elseprefix = nextquad();
    emit(jump, _);
}
```
if → *ifprefix* $stmt_1$ *elseprefix* $stmt_2$
```
{
    patchlabel($ifprefix, $elseprefix+1);
    patchlabel($elseprefix, nextquad());
}
```

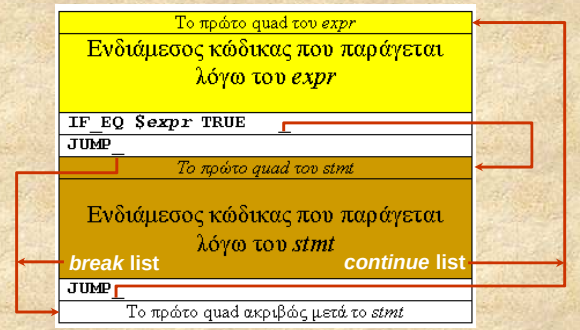
•Η τιμή του ***elseprefix*** είναι ο αριθμός του quad ακριβώς πριν το 1ο quad του $stmt_2$, και αντιστοιχεί σε ένα εμβόλιμο JUMP που ακολουθεί την εκτέλεση του $stmt_1$.
•Η τιμή του *ifprefix* αντιστοιχεί στο JUMP στην περίπτωση που το if expression είναι false, άρα θα πρέπει να οδηγεί στο 1ο quad του $stmt_2$. 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 περατωθεί η εκτέλεση του $stmt_1$ θα πρέπει να γίνει patched με τον αριθμό του 1ου quad αμέσως μετά το $stmt_2$. Όμως τη στιγμή που εκτελείται ο σημασιολογικός κανόνας, η τιμή αυτή είναι ακριβώς το *nextquad*().

Περιεχόμενα

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

Ανακυκλώσεις (1/5)

while (*expr*) *stmt* **(1/2)**




```
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 > z-w)
    if (b)
        break;
    else
        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 μία λίστα ώστε να συμπληρωθούν από τον σημασιολογικό κανόνα του while statement. Προφανώς τα break πρέπει να γίνουν patched στο quad που ακριβώς έπεται του stmt, ενώ τα continue στο 1ο quad υπολογισμού του while expression.

Ανακυκλώσεις (2/5)

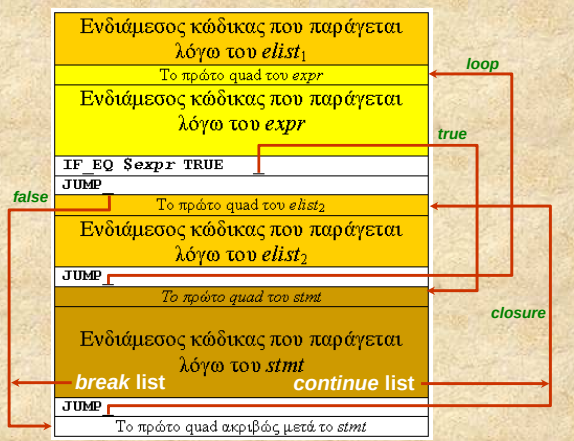
while (*expr*) *stmt* **(2/2)**

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

- Στο *whilestart* κρατάμε το 1ο quad του κώδικα υπολογισμού στης συνθήκης της εντολής while.
- Στο *whilecond* παράγουμε τον κώδικα που ελέγχει την τιμή του while expression και κάνει αντίστοιχα 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 τιμή αρχικού quad να είναι αποθηκευμένη στο γνώρισμα του *whilestart*.

Ανακυκλώσεις (3/5)

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



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

for (*elist*$_1$; *expr*; *elist*$_2$) *stmt*

elist$_1$ **while** (*expr*) **{** *stmt elist*$_2$ **}**

- 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. *elist*$_1$, *expr*, *elist*$_2$ και έπειτα *stmt*.
- Άρα θα πρέπει να φροντίσουμε και πάλι να έχουμε εμβόλιμα τα JUMP quads, ώστε να υποστηρίζεται η σημασιολογία του for loop, μέσω ειδικών γραμματικών συμβόλων και παραγωγών.

Ανακυκλώσεις (4/5)

for (*elist; expr; elist*) *stmt* **(2/3)**

N→ **{** $*N* = *nextquad*(); *emit*(**jump**, _); **}**
M→**{** $*M* = *nextquad*(); **}**

forprefix → **for (***elist*; *M expr*;
{
$*forprefix.test* = $*M*;
$*forprefix.enter* = *nextquad*();
emit(**if_eq**, $*expr*, *newexpr_constbool*(1), _);
}

for→ *forprefix* N_1 *elist*) N_2 *stmt* N_3
{
patchlabel($*forprefix.enter*, N_2+1); *true jump*
patchlabel($$N_1$$, *nextquad*()); *false jump*
patchlabel($$N_2$$, $*forprefix.test*); *loop jump*
patchlabel($$N_3$$, $$N_1$$+1); *closure jump*

patchlabel(*stmt.breaklist*, *nextquad*()); *false jump*
patchlabel(*stmt.contlist*, $$N_1$$+1); *closure jump*
}

•Ο κανόνας *N* εισάγεται για να παράγει ένα unfinished JUMP ενώ κρατάει σαν γνώρισμα το quad number αυτού του JUMP. Καθώς είναι γραμματικά της μορφής *N*→*ε*, κάνει πάντα reduce όταν τα γραμματικά σύμβολα στα αριστερά του είναι στη στοίβα.

•Καθώς χρειαζόμαστε τρία (3) τέτοια εμβόλιμα JUMPs, βάζουμε στα τρία κατάλληλα σημεία του βασικού κανόνα το γραμματικό σύμβολο *N*.

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

Ανακυκλώσεις (5/5)

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

```
for (i=0; i<N; ++i)
    print("*")
```

elist$_1$
expr
elist$_2$
stmt

```
1: ASSIGN 0 i

2: IF_LESS i N 5
3: ASSIGN FALSE _t1
4: JUMP 6
5: ASSIGN TRUE _t1

6: IF_EQ TRUE _t1 10      true jump
7: JUMP 14                false jump

8: ADD i 1 i
9: JUMP 2                 loop jump

10: PARAM "*"
11: CALL "print"
12: GETRETVAL _t2
13: JUMP 8                closure jump

14:
```


Περιεχόμενα

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

Ειδικές εντολές (1/6)

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

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 ενδιάμεσου κώδικα που δεν είναι γνωστά μέσα στον ίδιο τον γραμματικό τους κανόνα, γίνονται emit ως unfinished jumps.

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*

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

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

Κώδικας	***loopcounter*** stack και current value
`x = f();`	[0]
`while (x) {`	[1]
`  function g() {`	[1][0], ***push, reset***
`  break;`	[1][0], Άρα compile error (“***not in a loop***”)
`  while(true) {`	[1][1]
`  continue;`	[1][1], Νόμιμο, μέσα σε loop
`  }`	[1][0]
`  }`	[1], ***pop, restore***
`  break;`	[1], Νόμιμο, μέσα σε loop
`}`	[0]

Ειδικές εντολές (4/6)

break→**break ;**
{ $*break.breaklist* = *newlist*(*nextquad*()); *emit*(**jump**, _); }
continue→**continue;**
{ $ *continue.contlist* = *newlist*(*nextquad*()); *emit*(**jump**, _); }

stmts→ *stmt* { $*stmts* = $*stmt*; }
stmts→ $stmts_1$ *stmt*
{
$*stmts.breaklist* = *merge*($stmts_1$.*breaklist*, $*stmt.breaklist*);
$*stmts. contlist* = *merge*($stmts_1$. *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 καθένα αντιστοιχεί σε unfinished jump λόγω κάποιου break που περιέχεται στο *stmt*, και (β) το *contlist* αντίστοιχα, με τους αριθμούς όλων των quads που καθένα αντιστοιχεί σε unfinished jump λόγω κάποιου continue που περιέχεται στο *stmt*.

•Για τον διπλανό πηγαίο 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(x)[i] > y)
        break;
    else {
        z = g(y);
        continue;
    }
    h();
}
```


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:  GETRETVAL _t1
                    6:  ASSIGN _t1 a

   if (a)           7:  IF_EQ 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:
```

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** *ε* **;**
{ *emit*(**return**, *null*); **}**
returnstmt → **return** *expr***;**
{ *emit*(**return**, *$expr*); **}**

```
function f() {
    return;
    return local x;
    return nil;
}
```

```
1:  FUNCSTART f
2:  RETURN
3:  RETURN x
3:  RETURN nil
4:  FUNCEND f
```