ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 11a

Intermediate Code Generation

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Declarations

- For each local name
  - – Creation of symbol-table entry
  - – Add type and relative address of storage
- `enter(name, type, offset)`
  - – name of variable
  - – type of variable
  - – offset address relative to the current block

Translation

```
P → D        { offset := 0 }
    D
D → D; D
D → id: T    { enter(id.name, T.type, offset);
               offset := offset + T.width }
T → integer  { T.type := integer;
               T.width := 4}
T → real     { T.type := real;
               T.width := 8}
T → array [num] of $T_1$ { T.type := array(num.val, $T_1$.type);
                           T.width := num.val x $T_1$.width; }
T → ^$T_1$              { T.type := pointer($T_1$.type);
                          T.width := 4 }
```

Scope

- Each procedure is associated with a symbol table,
  - A new symbol table is created when a production is seen: $D \rightarrow$ **`proc id`** D_1**;** S
- Local variables are placed to the relevant symbol table.
- Symbol tables are linked with each other, according to how procedures are called.

Example (source code)

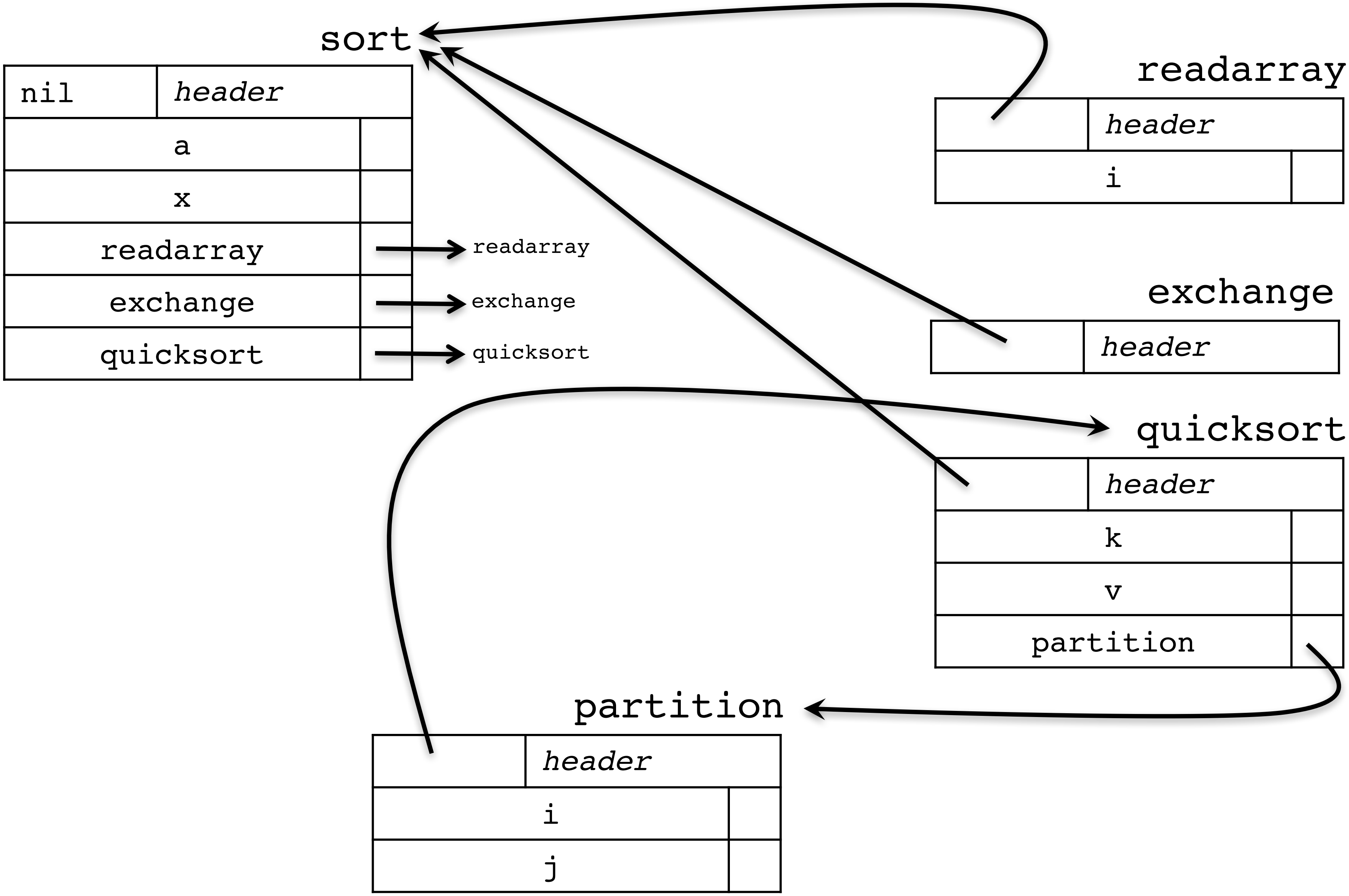
```
program sort(input, output);
    var a: array[0..10] of integer;
        x: integer;

procedure readarray;
    var i: integer;  begin ... end;

procedure exchange(i, j: integer);
    begin ... end;

procedure quicksort(m, n: integer);
    var k, v: integer;
    function partition(y, z: integer): integer;
        var i, j: integer;
        begin ... end {partition};
    begin ... end {quicksort};
begin ... end {sort}.
```

Example (symbol tables)



sort

nil	*header*
a	
x	
readarray	
exchange	
quicksort	

readarray

	header
i	

exchange

	header

quicksort

	header
k	
v	
partition	

partition

	header
i	
j	

Symbol-table functions

- `mktable(previous)`
  - creates a new table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table (stored in *header*), presumably that for the enclosing procedure.
- `enter(table, name, type, offset)`
  - creates a new entry of name *name* in the symbol table pointer to by *table*.
- `addwidth(table, width)`
  - records the cumulative width of all the entries in *table* in the header associated with this symbol table.
- `enterproc(table, name, newtable)`
  - creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Translation

```
P → M D  { addwidth(top(tblptr), top(offset));
           pop(tblptr); pop(offset) }
M → ε { t := mktable(nil);
           push(t, tblptr); push(0, offset) }
D → D1 ; D2
D → proc id ; N D1 ; S { t := top(tblptr);
                         addwidth(t, top(offset));
                         pop(tblptr); pop(offset);
                         enterproc(top(tblptr), id.name, t) }
D → id: T { enter(top(tblptr), id.name, T.type, top(offset));
            top(offset) := top(offset) + T.width }
N → ε       { t := mktable(top(tblptr));
              push(t, tblptr); push(0, offset) }
```

Syntax-directed Definition for Three-address Code

PRODUCTION	SEMANTIC RULES
S → **id** := *E*	*S.code* := *E.code* || *gen*(**id**.*place* ‘:=‘ *E.place*)
E → E_1 + E_2	*E.place*:= *newtemp*; *E.code* := E_1*.code* || E_2*.code* || gen(E.place ‘:=‘ E_1*.place* ‘+’ E_2*.place*)
E → E_1 * E_2	*E.place*:= *newtemp*; *E.code* := E_1*.code* || E_2*.code* || gen(E.place ‘:=‘ E_1*.place* ‘*’ E_2*.place*)
E → -E_1	*E.place*:= *newtemp*; *E.code* := E_1*.code* || gen(E.place ‘:=‘ ‘uminus’ E_1*.place*)
E → (E_1)	*E.place*:= E_1*.place*; *E.code* := E_1*.code*
E → **id**	*E.place*:= **id**.*place*; *E.code* := ‘ ‘

Incorporating the symbol table

```
S → id := E     { p := lookup(id.name);
                  if p != nil then
                      emit(p ‘:=‘ E.place)
                  else error }
E → E1 + E2    { E.place := newtemp;
                  emit(E.place ‘:=‘ E1.place ‘+’ E2.place) }
E → E1 * E2     { E.place := newtemp;
                  emit(E.place ‘:=‘ E1.place ‘*’ E2.place) }
E → -E1         { E.place := newtemp;
                  emit(E.place ‘:=‘ ’uminus’ E1.place) }
E → ( E1 )      { E.place := E1.place) }

E → id           { p := lookup(id.name);
                  if p != nil then
                      E.place := p
                  else error }
```

Reusing Temporary Names

- Temporary variables occupy slots in the symbol table

  evaluate E_1 to t_1

  evaluate E_2 to t_2

  $t := t_1 + t_2$

- Use a counter *c*, initialize to zero.
  - Whenever a temporary name is used, decrement *c*
  - Whenever a temporary name is created, increment *c*

Example

```
x := a*b+c*d-e*f
```

STATEMENT	VALUE OF *c*
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1 ($0 was used -1, $1 was used -1, $0 is created +1)
$1 := e * f	2
$0 := $0 - $1	1 ($0 was used -1, $1 was used -1, $0 is created +1)
x := $0	0

Boolean Expressions

Λογικές Εκφράσεις

- Boolean expressions are composed by boolean operators (**and**, **or**, and **not**) and relational expressions

```
E→E or E|E and E|not E|(E)|id relop id|true|false
relop → <|<=|=|<>|>|>=
```

- Two methods for translating
  - Numerical representation (1 denotes true, 0 denotes false)
  - Flow-of-control representation

TAC for Boolean Expressions

```
a or b and not c
t_1 := not c
t_2 := b and t_1
t_3 := a or t_2
```

```
a < b
100: if a < b goto 103
101: t := 0
102: goto 104
103: t := 1
104:
```

```
a < b or c < d and e < f
100: if a < b goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: if c < d goto 107
105: t_2 := 0
106: goto 108
107: t_2 := 1
108: if e < f goto 111
109: t_3 := 0
110: goto 112
111: t_3 := 1
112: t_4 := t_2 and t_3
113: t_5 := t_1 or t_3
```

Translation Scheme

```
E → $E_1$ or $E_2$  { E.place := newtemp;
                     emit(E.place ‘:=‘ $E_1$.place ‘or’ $E_2$.place) }
E → $E_1$ and $E_2$ { E.place := newtemp;
                     emit(E.place ‘:=‘ $E_1$.place ‘and’ $E_2$.place) }
E → not $E_1$       { E.place := newtemp;
                      emit(E.place ‘:=‘ ’not’ $E_1$.place) }
E → ( $E_1$ )       { E.place := $E_1$.place) }

E → $id_1$ relop $id_2$ { E.place := newtemp;
                      emit(‘if‘$id_1$.place relop.op $id_2$.place
                      ‘goto‘ nextstat+3);
                      emit(E.place ‘:=‘ ‘0‘);
                      emit(‘goto‘ nexstat+2);
                      emit(E.place ‘:=‘ ‘1‘) }
E → true            { E.place := newtemp;
                      emit(E.place ‘:=‘ ‘1‘) }
E → false           { E.place := newtemp;
                      emit(E.place ‘:=‘ ‘0‘) }
```

Syntax-directed Definition

<table>
<tr><th>PRODUCTION</th><th>SEMANTIC RULES</th></tr>
<tr><td>$E \rightarrow E_1$ <b>or</b> E_2</td><td>E_1.<i>true</i> := <i>E.true</i>;
E_1.<i>false</i> := <i>newlabel</i>;
E_2.<i>true</i> := <i>E.true</i>;
E_2.<i>false</i> := <i>E.false</i>;
<i>E.code</i>:= E_1.<i>code</i>||gen(E_1.<i>false</i>‘:‘)||E_2.<i>code</i></td></tr>
<tr><td>$E \rightarrow E_1$ <b>and</b> E_2</td><td>E_1.<i>true</i> := <i>newlabel</i>;
E_1.<i>false</i> := <i>E.false</i>;
E_2.<i>true</i> := <i>E.true</i>;
E_2.<i>false</i> := <i>E.false</i>;
<i>E.code</i>:= E_1.<i>code</i>||gen(E_1.<i>true</i>‘:‘)||E_2.<i>code</i></td></tr>
<tr><td>$E \rightarrow$ <b>not</b> E_1</td><td>E_1.<i>true</i>:=<i>E.false</i>;E_1.<i>false</i>:=<i>E.true</i>;
E.code:= E_1.<i>code</i></td></tr>
<tr><td>$E \rightarrow$ $\mathbf{id}_1$ <b>relop</b> $\mathbf{id}_2$</td><td><i>E.code</i>:=<i>gen</i>(‘if‘$\mathbf{id}_1$.<i>place</i> <b>relop</b>.<i>op</i>
$\mathbf{id}_2$. <i>place</i> ‘goto‘ <i>E.true</i>)||
gen(‘goto‘, E.false)</td></tr>
</table>

Examples

```
a < b or c < d and e < f
     if a < b goto Ltrue
     goto L1
L1:  goto L2
     goto Lfalse
L2:  if c < d goto 107
     goto Lfalse
```

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y – z
L1:  if a < b goto L2
     goto Lnext
L2:  if c < d goto L3
     goto L4
L3:  t_1 := y + 2
     x := t_1
     goto L1
L4:  t_2 := y - z
     x := t_2
     goto L1
Lnext:  goto Lfalse
```

Backpatching

- We use two passes
  - One to construct the syntax tree
  - One to traverse the syntax tree (depth-first order) and execute the translation
- Sometimes labels for booleans and flow-of-control statements are not known in advance in a single pass
- To reduce passes, we put unknown labels in a list and, once labels are known, we revisit *only* the ones that are currently unknown

Backpatching functions

- `makelist(i)`
  - creates a new list containing only `i`, an index into the array of quadruples; `makelist` returns a pointer to the list it has made
- merge(p_1, p_2)
  - concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list
- `backpatch(p, i)`
  - inserts `i` as the target label for each of the statements on the list pointed to by `p`

Translation Scheme

```
E → E1 or M E2 { backpatch(E1.falselist, M.quad);
                 E.truelist := merge(E1.truelist, E2.truelist);
                 E.falselist := E2.falselist }

E → E1 and M E2  { backpatch(E1.truelist, M.quad);
             E.truelist := E2.truelist;
             E.falselist := merge(E1.falselist, E2.falselist); }

E → not E1 { E.truelist := E1.falselist;
               E.falselist := E1.truelist; }
```

Translation

```
E → ( E1 ) { E.truelist := E1.truelist;
             E.falselist := E1.falselist; }

E → id1 relop id2 { E.truelist := makelist(nextquad);
                    E.falselist:= makelist(nextquad+1);
              emit(‘if‘id1.place relop id2.place ‘goto _‘);
              emit(‘goto _‘) }
E → true       { E.truelist := makelist(nextquad);
                 emit(‘goto _‘) }
E → false      { E.falselist := makelist(nextquad);
                 emit(‘goto _‘) }
M → ε          { M.quad := nextquad }
```