Code Generation

ASU Textbook Chapter 8

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

`http://www.iis.sinica.edu.tw/~tshsu`

Code generation

- **Compiler usually generates intermediate codes:**
  - **Ease of re-targeting different machines.**
  - **Perform machine-independent code optimization.**
- **Intermediate language:**
  - **Postfix languages: stack machine language.**
  - **Syntax tree: graphical representation.**
  - **Three-address code: a statement containing at most 3 addresses or operands.**
    - ***A sequence of statements of the general form:*** $x := y$ ***op*** z.
    - ***A linearized representation of a binary syntax tree.***
    - ***Consists of at most 3 addresses for each statement.***
    - x ***is the result, and*** y ***and*** z ***are operands.***
    - ***"op" is an operator.***

Types of three-address statements

- **Assignment**
  - **Binary:** $x := y$ **op** z
  - **Unary:** $x :=$ **op** y
  - **"op" can be any reasonable arithmetic or logical operator.**
- **Copy**
  - **Simple:** $x := y$
  - **Indexed:** $x := y[i]$ **or** $x[i] := y$
  - **Address and pointer manipulation:**
    - ▷ $x := \&y$
    - ▷ $x := *y$
    - ▷ $*x := y$
- **Jump**
  - **Unconditional: goto** L
  - **Conditional: if** x $relop$ y **goto** L_1 **[else goto** L_2**, where** $relop$ **is** $<, =, >, \geq, \leq$ **or** $\neq$**.**
- **Procedure call**
  - **Call procedure** $P(X1, X2, \ldots, Xn)$

```
PARAM X1
PARAM X2
...
PARAM Xn
CALL P,n
```

Implementation of three-address codes (1/2)

- **Three different implementations depending on how much indirection is presented in the representation:**
  - **Quadruples.**
  - **Triples.**
  - **Indirect triples.**
- **Quadruples**

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

Implementation of three-address codes (2/2)

- **Triples**

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	$+$	(1)	(3)
(5)	assign	a	(4)

- **Indirect triples**

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	$+$	(15)	(17)
(19)	assign	a	(18)

  - Statements can be reused.

Declarations

- **Global data: generate address in the static data area.**
- **Local data in a procedure or block:**
  - **Create a symbol table entry with**
    - *data type;*
    - *relative address within the A.R. (offset).*
  - **Depend on the target machine, determine data alignment.**
    - *For example: if a word has 2 bytes and an integer variable is represented with a word, then we may require all integers to start on even addresses.*

Declarations — examples

- $P \rightarrow MD$
  - $M \rightarrow \epsilon$
    {offset := 0}
- $D \rightarrow D; D$
- $D \rightarrow id : T$
  - **{ enter_symbol_table(id.name,T.type,offset);**
  - **offset := offset + T.width }**
- $T \rightarrow integer$
  - **{ T.type := integer;**
  - **T.width := 4 }**
- $T \rightarrow real$
  - **{ T.type := real;**
  - **T.width := 8 }**
- $T \rightarrow *T_1$
  - **{ T.type := pointer(**T_1**.type);**
  - **T.width := 4 }**
- $T \rightarrow$ **array** $[num]$ **of** T_1
  - **{ T.type := array(num.val,**T_1**.type);**
  - **T.width := num.val *** T_1**.width; }**

Symbol table operations

- **Treat symbol tables as objects:**
  - **mktable(previous):**
    - ▷ *make a new table, offset in the new table is 0,*
    - ▷ *link it to the previous symbol table, and*
    - ▷ *make the new table the current working table.*
  - **deltable(current): return the previous symbol table.**
  - **enter(table,name,type,offset): insert a new identifier.**
  - **addwidth(table,width): increase the size of the current A.R. by width.**
  - **enterproc(table, name,newtable): create a procedure with its symbol table being "new table."**
- **Keeping track of scope information.**
  - **When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended.**
  - $P \rightarrow D$
  - $D \rightarrow D; D \mid id : T \mid M_1\ proc\ id; D; D\ M_2$
  - $M_1 \rightarrow \epsilon$ **{ start a new symbol table here }**
    - ▷ *table = mktable(table)*
  - $M_2 \rightarrow \epsilon$ **{ return to old symbol table }**
    - ▷ *table = deltable(table)*

Examples for symbol table operations

- $P \longrightarrow M_1 D$
  - {addwidth(top(tblptr),top(offset));
  - pop(tblptr); pop(offset);}
- $M_1 \longrightarrow \epsilon$
  - {t := mktable(null);
  - push(t,tblptr); push(0,offset); }
- $D \longrightarrow D_1; D_2$
- $D \longrightarrow proc\ id; M_2 D_1; M_3 S$
  - {t := top(tblptr); /* save symbol table */ addwidth(t,top(offset));
  - generate code for de-allocating A.R.
  - pop(tblptr); pop(offset); enterproc(top(tblptr),id.name,t);}
- $D \longrightarrow id : T$
  - {enter(top(tblptr),id.name,T.type,top(offset));
  - top(offset) := top(offset) + T.width; }
- $M_2 \longrightarrow \epsilon$ /* enter a new scope */
  - {t := mktable(top(tblptr)); push(t,tblptr); push(0,offset); }
- $M_3 \longrightarrow \epsilon$
  - {generate code for allocating A.R. }

Handling names in records

- **A record declaration is treated as entering a block in terms of "offset" is concerned.**
- **Example:**
  - $T \rightarrow\ record$ **Marker** $D\ end$
    - ▷ *{ T.type := record(top(tblptr));*
    - ▷ *T.width := top(offset);*
    - ▷ *pop(tblptr);*
    - ▷ *pop(offset); }*
  - **Marker** $\rightarrow \epsilon$
    - ▷ *{ t := mktable(nulll);*
    - ▷ *push(t.tblptr);*
    - ▷ *push(0,offset);}*

Code generation routines

- **Error reporting routine:**
  - error(error message or error number);
- **Code generation:**
  - emit([address #1], [assignment], [address #2], operator, address #3);
  - Depend on [address #i], generate different codes.
    - ▷ *Local temp space: FP+tmp_start+offset.*
    - ▷ *Papameters: FP+param_start+offset.*
    - ▷ *Global variable: GDATA+offset.*
    - ▷ *Registers, . . .*
- **Temp space anagement:**
  - newtemp(): allocate a temp space.
    - ▷ *Using a bit arrary to indicate the usage of temp space.*
    - ▷ *Usually use a circular array data structure.*
  - freetemp(t): free temp space t.
- **Label anagement:**
  - newlabel(): generate a label that's never used.
- **Symbol table lookup:**
  - lookup(name,t): check whether name is declared in symbol table t, return the entry if it is in t.

Assignment statements

- $S \rightarrow id := E$
  - **{ p := lookup(id.name,symbol_table);**
  - **if p is not null then emit(p, ":=",E.place); else error("var undefined",id.name); }**
- $E \rightarrow E_1 + E_2$
  - **{E.place := newtemp(); freetemp(E_1.place);freetemp(E_2.place);**
  - **emit(E.place, ":=",E_1.place,"+",E_2.place)}**
- $E \rightarrow E_1 * E_2$
  - **{E.place := newtemp();freetemp(E_1.place);freetemp(E_2.place);**
  - **emit(E.place, ":=",E_1.place,"*",E_2.place)}**
- $E \rightarrow -E_1$
  - **{E.place := newtemp();freetemp(E_1.place);**
  - **emit(E.place, ":=", "uminus",E_1.place) }**
- $E \rightarrow (E_1)$
  - **{E.place := E_1.place}**
- $E \rightarrow id$
  - **{p := lookup(id.name,symbol_table);**
  - **if p $\neq$ null then E.place := p else error("var undefined",id.name)}**

Addressing array elements (1/2)

- **1-D array: $A[i]$. Assume**
  - **lower bound in address $= low$**
  - **element data width $= w$**
  - **starting address $= base$**
- **Address for $A[i]$**
  - $= base + (i - low) * w$
  - $= i * w + (base - low * w)$
  - **The value of $(base - low * w)$ can be computed at compile time.**
- **2-D array $A[i_1, i_2]$.**
  - **Row major:**
    - ▷ $A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], \ldots$
    - ▷ ***Advantage: A[i,j] = A[i][j].***
    - ▷ ***A[1] means the first row.***
  - **Column major:**
    - ▷ $A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], \ldots$

Addressing array elements (2/2)

- **Address for** $A[i_1, i_2]$
  - $= base + ((i_1 - low_1) * n_2 + (i_2 - low_2)) * w$
  - $= (i_1 * n_2 + i_2) * w + (base - low_1 * n_2 * w - low_2 * w)$
  - n_2 **is the number of elements in a row.**
  - low_2 **is the lower bound of the second coordinate.**
  - **The value** $(base - low_1 * n_2 * w - low_2 * w)$ **can be computed at compiler time.**
- **Similar method for multi-dimensional arrays. Address for** $A[i_1, i_2, \ldots, i_k]$
  - $= (i_1 * \Pi_{i=2}^{k} n_i i_2 * \Pi_{i=3}^{k} n_i + \cdots + i_k) * w + (base - low_1 * \Pi_{i=2}^{k} n_i * w - \cdots - low_k * w)$
  - n_i **is the number of elements in the** i**th coordinate.**
  - low_i **is the lower of the** i**th coordinate.**
  - **The values** $\Pi_{i=j}^{k}$, $2 \leq j \leq k-1$, **and** $(base - low_1 * \Pi_{i=2}^{k} n_i * w - \cdots - low_k * w)$ **can be computed at compile time.**

Code generation for arrays

- $L \rightarrow Elist\ ]$
  - **{L.place := newtemp();**
  - **L.offset := newtemp();**
  - **emit(L.offset,":=",Elist.elesize,"*",Elist.place);}**
- $L \rightarrow id$
  - **{L.place := id.place; L.offset := null}**
- $Elist \rightarrow Elist_1, E$
  - **{ t := newtemp();**
  - **m :=** $Elist_1$**.ndim + 1;**
  - **emit(t, ":=",**$EList_1.place$**,"*",limit(**$Elist_1$**.array,m));**
  - **emit(t,":=",t,"+",E.place);**
  - **Elist.array :=** $Elist_1$**.array;**
  - **Elist.place := t;**
  - **EList.ndim := m; }**
- $Elist \rightarrow id\ [\ E$
  - **{Elist.place := E.place;**
  - **Elist.ndim := 1; p := lookup(id.name,symbol_table);**
  - **Elist.elesize := p.size;**
  - **EList.array := p.place}**

Type conversions

- $E \longrightarrow E_1 + E_2$
  - **if** $E_1.type == E_2.type$ **then**
    - ▷ ***generate no conversion code***
    - ▷ E***.type*** $=$ E_1***.type***
  - **else**
    - ▷ E***.type*** $=$ ***float***
    - ▷ $temp_1$ $=$ ***newtemp();***
    - ▷ ***if*** E_1***.type*** $==$ ***integer then***
      emit($temp_1$***,":=", int-to-float,***E_1***.place);***
      emit(E,":=",$temp_1$***,"+",***E_2***.place);***
    - ▷ ***else***
      emit($temp_1$***,":=", int-to-float,***E_2***.place);***
      emit(E,":=", $temp_1$***,"+",***E_1***.place);***
    - ▷ ***freetemp(***$temp_1$***);***

Boolean expressions

- **Two choices for implementation:**
  - **Encode true and false values numerically, evaluate analogously to an arithmetic expression**
    - 1: *true;* 0: *false.*
    - $\neq 0$: *true;* 0: *false.*
  - **Flow of control: representing the value of a boolean expression by a position reached in a program.**
- **Short-circuit code.**
  - **Generate the code to evaluate a boolean expression in such a way that it is not necessary for the code to evaluate the entire expression.**
  - **if** a_1 **or** a_2
    - a_1 *is true, then* a_2 *is not evaluated.*
  - **Similarly for "and".**
  - **Side effects in the short-circuited code are not carried out.**

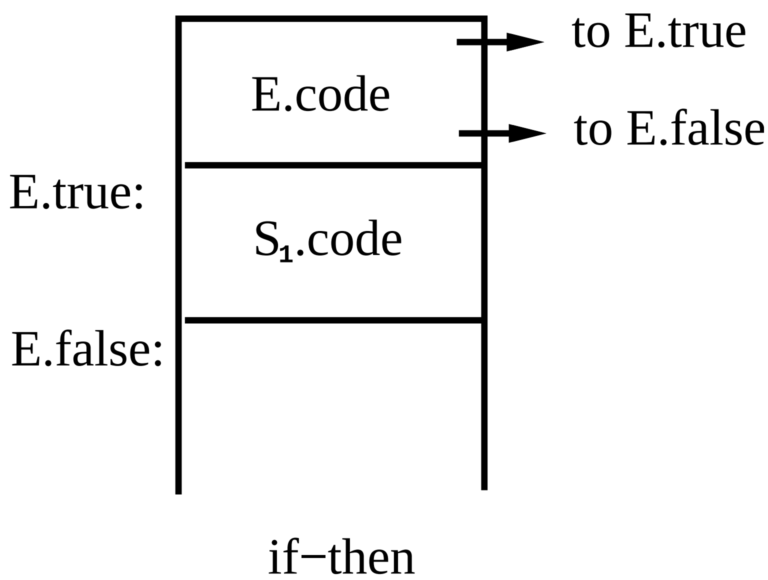
Numerical representation

- $E \rightarrow id_1 \ relop \ id_2$
  - **{E.place := newtemp();**
  - **emit("if", id_1.place,relop.op, id_2.place,"goto",nextstat+3);**
  - **emit(E.place,":=","0");**
  - **emit("goto",nextstat+2);**
  - **emit(E.place,":=","1");}**
- **Example for translating ($a < b$ or $c < d$ and $e < f$):**

```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1 /* true */
104: if c < d goto 107
105: t2 := 0 /* false */
106: goto 108
107: t2 := 1
108: if e < f goto 111
109: t3 := 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
113: t3 := t1 or t4
```

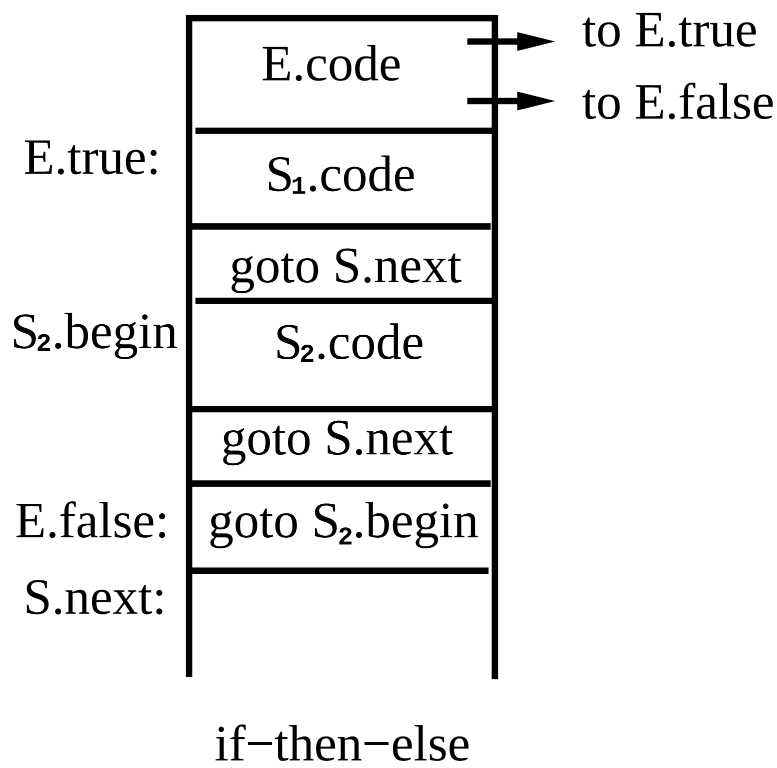
Flow of control representation

- $E \rightarrow \; id_1 \; relop \; id_2$
  - **{ E.true := newlabel();**
  - **E.false := newlabel();**
  - **emit("if",id_1,relop,id_2,"goto", E.true,"else","goto",E.false);**
  - **emit(E.true,":");}**
- $S \rightarrow$ **if** E **then** S_1
  - **{emit(E.false,":");}**



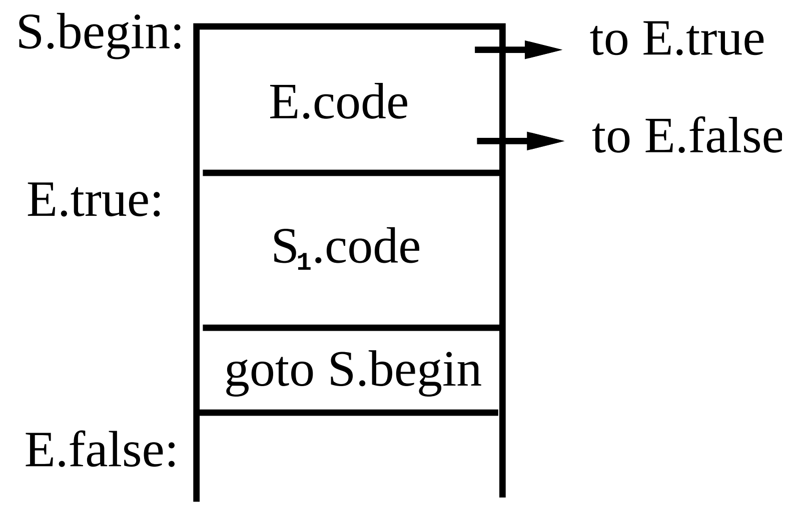
If-then-else

- $E \rightarrow\ id_1\ relop\ id_2$
  - { E.true := newlabel();
  - E.false := newlabel();
  - emit("if",id_1,relop,id_2,"goto", E.true,"else","goto",E.false);
  - emit(E.true,":");}
- $S \rightarrow$ **if** E **then** $S_1\ M_3$ **else** $M_4\ S_2$
  - {S.next = M_3.next;
  - emit("goto",S.next);
  - emit(E.false,":");
  - emit("goto",M_4.label);
  - emit(S.next,":");}
- $M_3 \rightarrow \epsilon$
  - {M_3.next := newlabel();
  - emit("goto",M_3.next);}
- $M_4 \rightarrow \epsilon$
  - {M_4.label := newlabel();
  - emit(M_4.label,":");}



While loop

- $E \rightarrow\ id_1\ relop\ id_2$
  - **{ E.true := newlabel();**
  - **E.false := newlabel();**
  - **emit(“if”,id_1,relop,id_2,“goto”, E.true,“else”,“goto”,E.false);**
  - **emit(E.true,“:”);}**
- $S \rightarrow$ **while** $M_5\ E$ **do** S_1
  - **{S.begin =** M_5**.begin;**
  - **emit(“goto”,S.begin);**
  - **emit(E.false,“:”);}**
- $M_5 \rightarrow \epsilon$
  - **{**M_5**.begin := newlabel();**
  - **emit(**M_5**.begin,“:”);}**



while loop

Case/Switch statement

- **C-like syntax:**
  - switch expr{
  - case V_1: S_1
  - $\cdots$
  - case V_k: S_k
  - default: S_d
  - }
- **Translation sequence:**
  - Evaluate the expression.
  - Find which value in the list matches the value of the expression, match default only if there is no match,
  - Execute the statement associated with the matched value.
- **How to find the matched value:**
  - Sequential test.
  - Look-up table.
  - Hash table.
  - Backpatching.

Implementation of case statements (1/2)

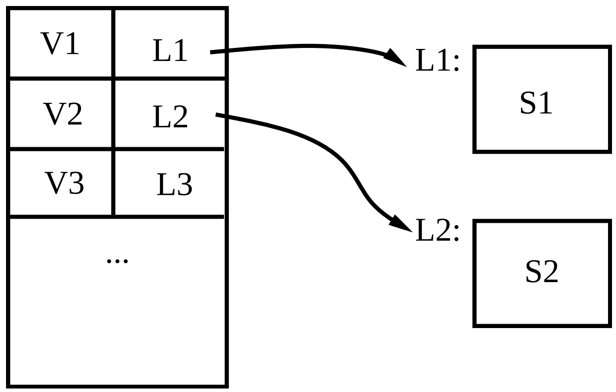
- **Two different translation schemes for sequential test.**

```
code to evaluate E into t
goto test
L1: code for S1
    goto next
    ...
Lk: code for Sk
    goto next
Ld: code for Sd
    goto next
test:
    if t = V1 goto L1
    ...
    if t = Vk goto Lk
    goto Ld
next:
    ...
Can easily be used into a loopkup table!
```

```
code to evaluate E into t
    if t <> V1 goto L1
    code for S1
    goto next
L1: if t <> V2 goto L2
    code for S2
    goto next
    ...
L(k-1): if t <> Vk goto Lk
    code for Sk
    goto next
Lk: code for Sd
next:
```

Implementation of case statements (2/2)

- **Use a table and a loop to find the address to jump.**



- **Hash table: when there are more than 10 entries, use a hash table to find the correct table entry.**
- **Backpatching:**
  - **Generate a series of branching statements with the targets of the jumps temporarily left unspecified.**
  - **To-be-determined label table: each entry contains a list of places that need to be backpatched.**
  - **Can also be used to implement labels and goto's.**

Procedure calls

- **Space must be allocated for the A.R. of the called procedure.**
- **Arguments are evaluated and made available to the called procedure in a known place.**
- **Save current machine status.**
- **When a procedure returns**
  - **place return value in a known place**
  - **restore A.R.**

Example for procedure call

- **Example:**
  - $S \rightarrow$ **call** $id(Elist)$
    - ▷ ***{for each item p on the queue Elist.queue do***
    - ▷ ***emit("PARAM", q);***
    - ▷ ***emit("call", id.place);}***
  - $Elist \rightarrow Elist, E$
    - ▷ ***{append E.place to the end of Elist.queue}***
  - $Elist \rightarrow E$
    - ▷ ***{initialize $Elist.queue$ to contain only E.place}***
- **Idea:**
  - **Use a queue to hold parameters, then generate codes for parameters.**
  - **May have code like:**
    - ▷ ***code for E_1, store in t_1***
    - ▷ $\cdots$
    - ▷ ***code for E_k, store in t_k***
    - ▷ ***PARAM t_1***
    - ▷ $\cdots$
    - ▷ ***PARAM t_k***
    - ▷ ***call p***