Code Generation

ASU Textbook Chapter 8, Chapter 7.5

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

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

Code generation

- **Compiler usually generate intermediate codes.**
  - **Ease of re-targeting different machines.**
  - **Perform machine-independent code optimization.**
- **Intermediate language:**
  - **Postfix language: a stack-based machine-like language.**
  - **Syntax tree: a 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, ***where "op" is an operator,*** x ***is the result, and*** y ***and*** z ***are operands.***
    - ▷ ***Consists of at most 3 addresses for each statement.***
    - ▷ ***A linearized representation of a binary syntax tree.***

Types of three-address statements

- **Assignment**
  - **Binary:** $x := y$ **op** z
  - **Unary:** $x :=$ **op** y
  - **“op” can be any reasonable arithmetic or logic 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
```

Symbol table operations

- **Treat symbol tables as objects.**
  - **Accessing objects by service routines.**
- **Symbol tables: assume using a multiple symbol table approach.**
  - **mktable**($previous$)**:**
    - ▷ ***create a new symbol table.***
    - ▷ ***link it to the symbol table*** $previous$.
  - **enter**($table$,$name$,$type$,$offset$)**:**
    - ▷ ***insert a new identifier*** $name$ ***with type*** $type$ ***and*** $offset$ ***into*** $table$;
    - ▷ ***check for possible duplication.***
  - **addwidth**($table$,$width$)**:**
    - ▷ ***increase the size of the symbol table*** $table$ ***by*** $width$.
  - **enterproc**($table$, $name$, $newtable$)**:**
    - ▷ ***insert a procedure*** $name$ ***into*** $table$;
    - ▷ ***the symbol table of*** $name$ ***is*** $newtable$.
  - **lookup**($name$,$table$)**:**
    - ▷ ***check whether*** $name$ ***is declared in symbol table*** $table$,
    - ▷ ***return the entry if it is in*** $table$.

Stack operations

- **Treat stacks as objects.**
- **Stacks: many stacks for different objects such as offset, and symbol table.**
  - **push(**$object$**,**$stack$**)**
  - **pop(**$stack$**)**
  - **top(**$stack$**): top of stack element**

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, i.e., offset, within the A.R.***
  - **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

- $Declaration \rightarrow M_1 \quad D$
- $M_1 \rightarrow \epsilon$
  - ▷ ***{top(offset) := 0;}***
- $D \rightarrow D; D$
- $D \rightarrow id : T$
  - ▷ ***{enter(top(tblptr),id.name,T.type,top(offset));***
  - ▷ ***top(offset) := top(offset) + T.width; }***
- $T \rightarrow integer$
  - ▷ ***{ T.type := integer;***
  - ▷ ***T.width := 4; }***
- $T \rightarrow double$
  - ▷ ***{ T.type := double;***
  - ▷ ***T.width := 8; }***
- $T \rightarrow *T_1$
  - ▷ ***{ T.type := pointer(T_1.type);***
  - ▷ ***T.width := 4; }***

Handling blocks

- **Need to remember the current offset before entering the block, and to restore it after the block is closed.**
- **Example:**
  - $Block \rightarrow begin\ M_4\ Declarations\ Statements\ end$
    - ▷ ***{ pop(tblptr);***
    - ▷ ***pop(offset); }***
  - $M_4 \rightarrow \epsilon$
    - ▷ ***{ t := mktable(top(tblptr));***
    - ▷ ***push(t,tblptr);***
    - ▷ ***push(top(offset),offset);}***
- **Can also use the block number technique to avoid creating a new symbol table.**

Handling names in records

- A record declaration is treated as entering a block in terms of “offset” is concerned.
- Need to use a new symbol table.
- Example:
  - $T \rightarrow$ *record* M_5 D *end*
    - ▷ **{ *T.type := record(top(tblptr));***
    - ▷ ***T.width := top(offset);***
    - ▷ ***pop(tblptr);***
    - ▷ ***pop(offset);* }**
  - $M_5 \rightarrow \epsilon$
    - ▷ **{ *t := mktable(null);***
    - ▷ ***push(t,tblptr);***
    - ▷ ***push(0,offset);*}**

Nested procedures

- **When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended.**
  - $Proc \rightarrow procedure\ id\ ;\ M_2\ Declaration\ ;\ M_3\ Statements$
    - ▷ ***{t := top(tblptr); /∗ symbol table for this procedure ∗/***
    - ▷ ***addwidth(t,top(offset));***
    - ▷ ***generate code for de-allocating A.R.;***
    - ▷ ***pop(tblptr); pop(offset);***
    - ▷ ***enterproc(top(tblptr),id.name,t);}***
  - $M_2 \rightarrow \epsilon$
    - ▷ ***{ /∗ enter a new scope ∗/***
    - ▷ ***t := mktable(top(tblptr));***
    - ▷ ***push(t,tblptr); push(0,offset); }***
  - $M_3 \rightarrow \epsilon$
    - ▷ ***{generate code for allocating A.R.; }***
- **This is a better place to take of the case for offset initialization.**
  - **Avoid using ϵ-productions.**
    - ▷ ***ϵ-productions easily trigger conflicts.***

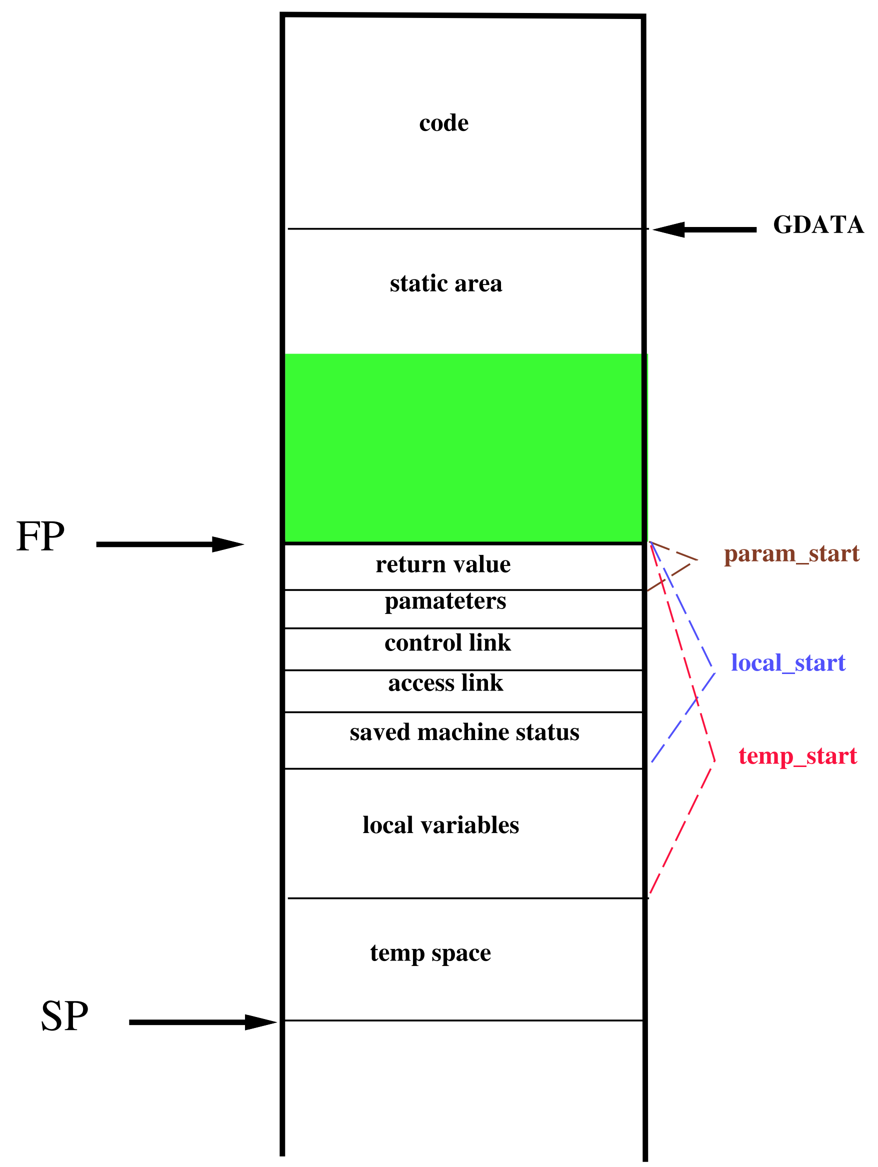
Yet another better grammar

- **Split a lengthy production at the place when in-production semantic actions are required.**
  - $Proc \rightarrow Proc\_Head\ Proc\_Decl\ Statements$
    - ▷ ***{t := top(tblptr); /∗ symbol table for this procedure ∗/***
    - ▷ ***addwidth(t,top(offset));***
    - ▷ ***generate code for de-allocating A.R.;***
    - ▷ ***pop(tblptr); pop(offset);***
    - ▷ ***enterproc(top(tblptr),id.name,t);}***
  - $Proc\_Head \rightarrow procedure\ id\ ;$
    - ▷ ***{ /∗ enter a new scope ∗/***
    - ▷ ***t := mktable(top(tblptr));***
    - ▷ ***push(t,tblptr); push(0,offset); }***
  - $Proc\_Decl \rightarrow Declaration\ ;$
    - ▷ ***{generate code for allocating A.R.; }***

Code generation routine

- **Code generation:**
  - **emit([address #1], [assignment], [address #2], operator, address #3);**
    - ▷ *Use switch statement to actually print out the target code;*
    - ▷ *Can have different emit() for different target codes;*
- **Variable accessing: depend on type of [address #i], generate different codes.**
  - **Watch out the differences between l-address and r-address.**
  - **Local temp space: FP+temp_start+offset.**
  - **Parameter: FP+param_start+offset.**
  - **Local variable: FP+local_start+offset.**
  - **Non-local variable:**
    - ▷ *trace the access link a suitable number of times to get the FP of the declaring procedure;*
    - ▷ *then use the formula for local variable.*
  - **Global variable: GDATA+offset.**
  - **Registers, constants, . . .**

Example for memory management



Code generation service routines

- **Error handling routine: error_msg(error information);**
  - **Use switch statement to actually print out the error message;**
  - **The messages can be written and stored in other file.**
- **Temp space management:**
  - **This is needed in generating code for expressions.**
  - **newtemp(): allocate a temp space.**
    - ▷ *Using a bit array to indicate the usage of temp space.*
    - ▷ *Usually use a circular array data structure.*
  - **freetemp(t): free t if it is allocated in the temp space.**
- **Label management:**
  - **This is needed in generated branching statements.**
  - **newlabel(): generate a label in the target code that has never been used.**

Assignment statements

- $S \rightarrow id := E$
  - ▷ ***{ p := lookup(id.name,top(tblptr));***
  - ▷ ***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();***
  - ▷ ***emit(E.place, ":=",E_1.place,"+",E_2.place);***
  - ▷ ***freetemp(E_1.place);freetemp(E_2.place);}***
- $E \rightarrow -E_1$
  - ▷ ***{E.place := newtemp();***
  - ▷ ***emit(E.place, ":=","uminus",E_1.place);***
  - ▷ ***freetemp(E_1.place);}***
- $E \rightarrow (E_1)$
  - ▷ ***{E.place := E_1.place;}***
- $E \rightarrow id$
  - ▷ ***{p := lookup(id.name,top(tblptr));***
  - ▷ ***if p $\neq$ null then E.place := p else error("var undefined",id.name);}***

Type conversions

- **Assume there are only two data types, namely integer and float.**
- **Assume automatic type conversions.**
  - **May have different rules.**
- $E \rightarrow E_1 + E_2$
  - **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(***$temp_1$***);***

Addressing 1-D array elements

- **1-D array:** $A[i]$.
  - **Assumptions:**
    - ▷ ***lower bound in address*** $= low$
    - ▷ ***element data width*** $= w$
    - ▷ ***starting address*** $= start\_addr$
  - **Address for** $A[i]$
    - ▷ $= start\_addr + (i - low) * w$
    - ▷ $= i * w + (start\_addr - low * w)$
    - ▷ ***The value, called*** $base$, $(start\_addr - low * w)$ ***can be computed at compile time.***
- **PASCAL use**

```
        array [-8 .. 100] of integer
```

to declare an integer array in the range of [-8], [-7], [-6] , . . . , [-1], [0], [1], . . . , [100].

Addressing 2-D array elements

- **2-D array** $A[i_1, i_2]$.
  - **Row major: the preferred mapping method.**
    - ▷ $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], \ldots$
    - ▷ ***A[i] means the ith row.***
    - ▷ ***Advantage: A[i,j] = A[i][j].***
  - **Column major:**
    - ▷ $A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], \ldots$
- **Address for** $A[i_1, i_2]$
  - $= start\_addr + ((i_1 - low_1) * n_2 + (i_2 - low_2)) * w$
  - $= (i_1 * n_2 + i_2) * w + (start\_addr - low_1 * n_2 * w - low_2 * w)$
    - ▷ n_2 ***is the number of elements in a row.***
    - ▷ low_1 ***is the lower bound of the first coordinate.***
    - ▷ low_2 ***is the lower bound of the second coordinate.***
  - **The value, called** $base$, $(start\_addr - low_1 * n_2 * w - low_2 * w)$ **can be computed at compiler time.**

Addressing multi-D array elements

- **Similar method for multi-dimensional arrays.**
- **Address for** $A[i_1, i_2, \ldots, i_k]$
  - $= (i_1 * \Pi_{i=2}^{k} n_i + i_2 * \Pi_{i=3}^{k} n_i + \cdots + i_k) * w + (start\_addr - low_1 * w * \Pi_{i=2}^{k} n_i - low_2 * w * \Pi_{i=3}^{k} n_i - \cdots - low_k * w)$
    - ▷ n_i ***is the number of elements in the*** i***th coordinate.***
    - ▷ low_i ***is the lower of the*** i***th coordinate.***
  - **The value** $(i_1 * \Pi_{i=2}^{k} n_i + i_2 * \Pi_{i=3}^{k} n_i + \cdots + i_k)$ **can be computed incrementally in grammar rules.**
    - ▷ $f(1) = i_1$***;***
    - ▷ $f(j) = f(j-1) * n_j + i_j$***;***
    - ▷ $f(k)$ ***is the value we want;***
  - **The value, called** $base$**,** $(start\_addr - low_1 * w * \Pi_{i=2}^{k} n_i - low_2 * w * \Pi_{i=3}^{k} n_i - \cdots - low_k * w)$ **can be computed at compile time.**

Code generation for arrays

- $Array \rightarrow Elist\ ]$
  - ▷ {L.offset := newtemp(); freetemp(Elist.place);
  - ▷ emit(L.offset,":=",Elist.elesize,"$*$",Elist.place);
  - ▷ emit(L.offset,":=",L.offset,"+",Elist.base);}
- $Elist \rightarrow Elist_1, E$
  - ▷ { t := newtemp(); m := $Elist_1$.ndim + 1;
  - ▷ emit(t, ":=",$EList_1.place$,"$*$",limit($Elist_1$.array,m));
  - ▷ emit(t,":=",t,"+",E.place); freetemp(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,top(tblptr)); check for id errors;
  - ▷ Elist.elesize := p.size; Elist.base := p.base;
  - ▷ EList.array := p.place;}
- $E \rightarrow id$
  - ▷ {p := lookup(id.name,top(tblptr)); check for id errors;
  - ▷ E.place := p.place;}

Boolean expressions

- **Two choices for implementation:**
  - **Numerical representation: 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

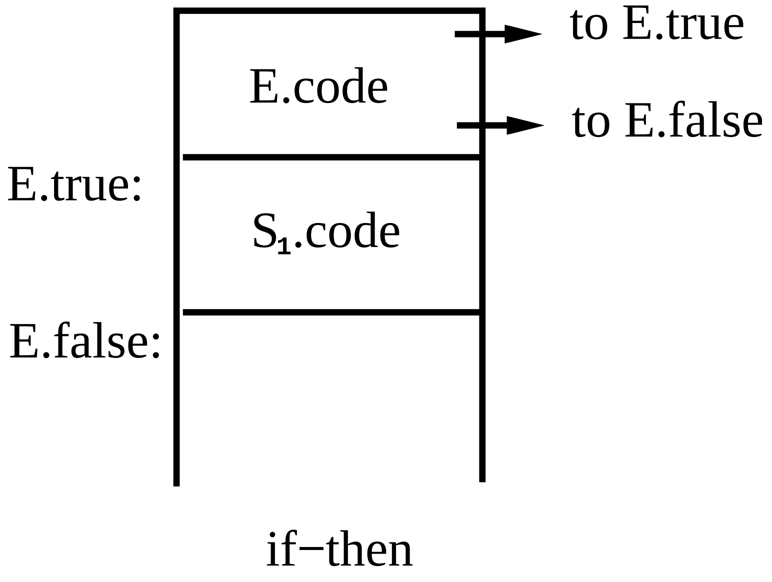
- $E \rightarrow id_1\ relop\ id_2$
  - {E.place := newtemp();
  - emit("if", id_1.place,relop.op, id_2.place,"goto",nextstat+3);
  - emit(E.place,":=","0");
  - emit("goto",nextstat+2);
  - emit(E.place,":=","1");}
- **Example for translating ($a < b$ or $c < d$ and $e < f$) using no short-circuit evaluation.**

```
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
```

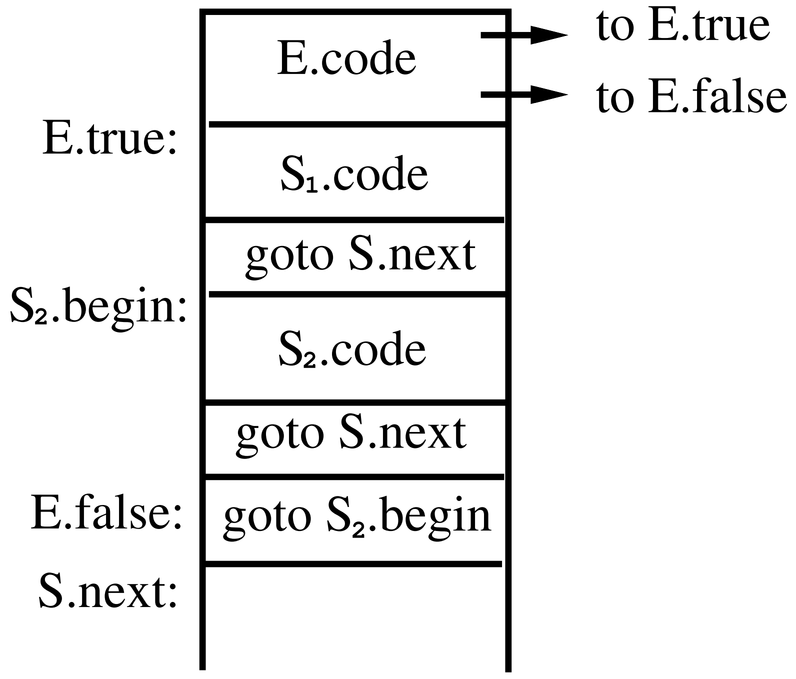
Flow of control representation

- $E \rightarrow\ id_1\ relop\ id_2$
  - **{ 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

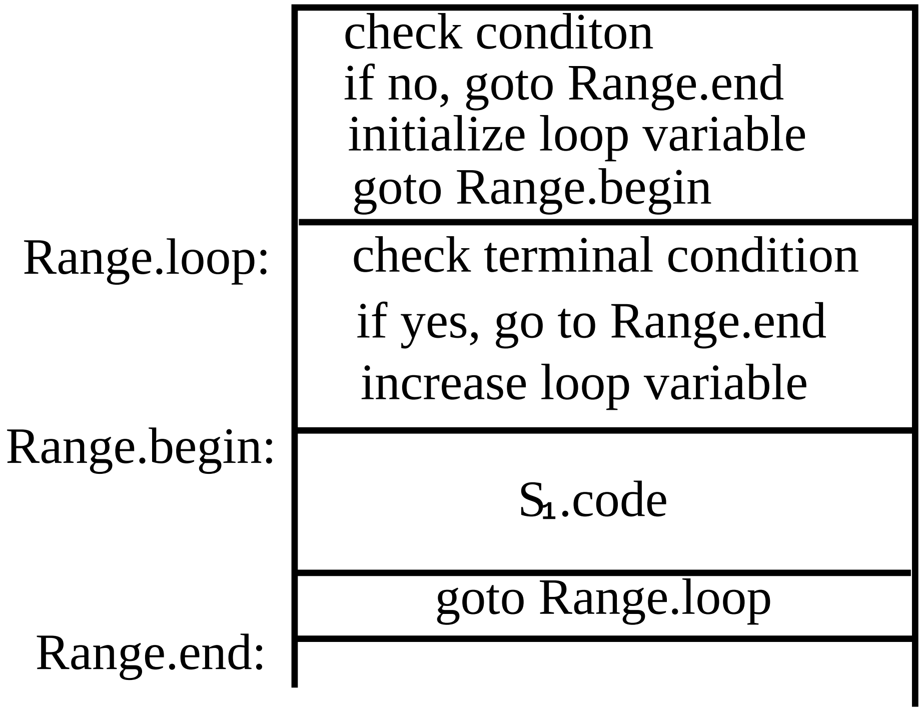
- $E \rightarrow\ id_1\ relop\ id_2$
  - **{ E.true := newlabel();**
  - **E.false := newlabel();**
  - **emit("if",id_1,relop,id_2,"goto", E.true,"else","goto",E.false);**
  - **emit(E.true,":");}**
- $S \rightarrow$ **if** E **then** $S_1\ M_3$ **else** $M_4\ S_2$
  - **{S.next = M_3.next;**
  - **emit("goto",S.next);**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



if–then–else

For loop

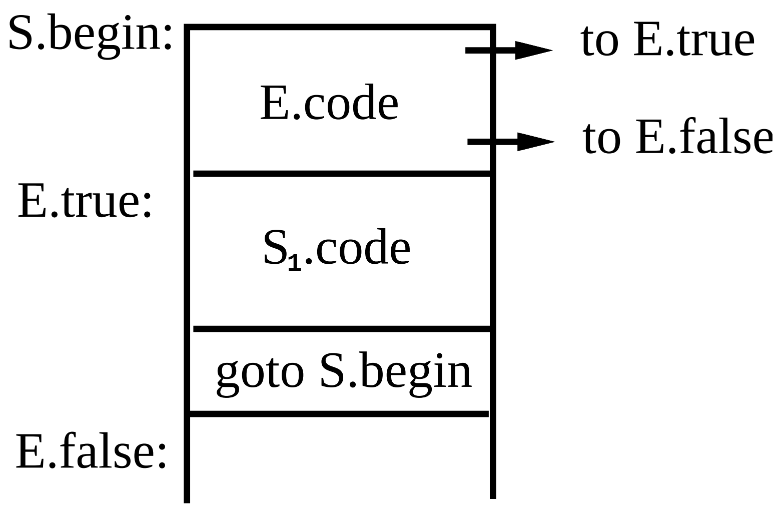
- $Range \rightarrow id\ := \ E_1\ to\ E_2$
  - { check E_1 and E_2 are integers;
  - p=lookup(id.name,top(tblptr)); check for id errors;
  - Range.end = newlabel();
  - emit("if",E_1.place,">", E_2.place,"goto",Range.end);
  - emit(p.place,":=",E_1.place);
  - Range.begin = newlabel(); emit("goto",Range.begin);
  - Range.loop; = newlabel(); emit(Range.loop,":";
  - emit("if",p.place,"==",E_2.place, "goto",Range.end);
  - emit(p.place,"++");
  - emit(Range.begin,":");}
- $S \rightarrow$ for $Range$ do S_1
  - {emit("goto",Range.loop);
  - emit(Range.end,":");}



for loop

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.
  - Back-patching.

Implementation of case statements (1/2)

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

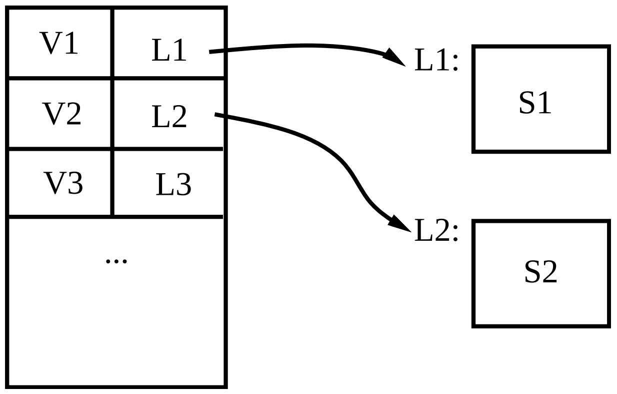
```
code to evaluate E into t
goto test
L1: code for S1
    goto next
    ...
Lk: code for Sk
    goto next
Ld: code for Sd
    goto next
test:
    if t = V1 goto L1
    ...
    if t = Vk goto Lk
    goto Ld
next:
    ...
```

Can easily be converted into a lookup 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.
- Back-patching:
  - 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 back-patched.
  - 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.**
  - **Sample object code:**
    - ▷ *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*

Parameter passing

- **Terminology:**
  - **procedure declaration:**
    - ▷ *parameters, formal parameters, or formals.*
  - **procedure call:**
    - ▷ *arguments, actual parameters, or actuals.*
- **The value of a variable:**
  - r**-value: the current value of the variable.**
    - ▷ *right value*
    - ▷ *on the right side of assignment*
  - l**-value: the location/address of the variable.**
    - ▷ *left value*
    - ▷ *on the left side of assignment*
  - **Example:** $x := y$
- **Four different modes for parameter passing**
  - **call-by-value**
  - **call-by-reference**
  - **call-by-value-result(copy-restore)**
  - **call-by-name**

Call-by-value

- **Usage:**
  - Used by PASCAL if you use non-var parameters.
  - Used by C++ if you use non-& parameters.
  - The only thing used in C.
- **Idea:**
  - calling procedure copies the r-values of the arguments into the called procedure's A.R.
- **Effect:**
  - Changing a formal parameter (in the called procedure) has no effect on the corresponding actual. However, if the formal is a pointer, then changing the thing pointed to does have an effect that can be seen in the calling procedure.
- **Example:**

```
void f(int *p)
{  *p = 5;
   p = NULL;
}
```

```
main()
{int *q = malloc(sizeof(int));
*q=0;
f(q);
}
```

  - In $main$, q will not be affected by the call of f.
  - That is, it will not be NULL after the call.
  - However, the value pointed to by q will be changed from 0 to 5.

Call-by-reference (1/2)

- **Usage:**
  - Used by PASCAL for var parameters.
  - Used by C++ if you use & parameters.
  - FORTRAN.
- **Idea:**
  - Calling procedure copies the l-values of the arguments into the called procedure's A.R. as follows:
    - ▷ *If an argument has an address then that is what is passed.*
    - ▷ *If an argument is an expression that does not have an l-value (e.g., $a + 6$), then evaluate the argument and store the value in a temporary address and pass that address.*
- **Effect:**
  - Changing a formal parameter (in the called procedure) does affect the corresponding actual.
  - Side effects.

Call-by-reference (2/2)

- **Example:**

```
FORTAN quirk /* using C++ syntax */
void mistake(int & x)
{x = x+1;}
main()
{mistake(1);
cout<<1;
}
```

  - **In C++, you get a warning from the compiler because x is a reference parameter that is modified, and the corresponding actual parameter is a literal.**
  - **The output of the program is 1.**
  - **However, in FORTRAN, you would get no warning, and the output may be 2. This happens when FORTRAN compiler stores 1 as a constant at some address and uses that address for all the literal "1" in the program.**
  - **In particular, that address is passed when "mistake()" is called, and is also used to fetch the value to be written by "count". Since "mistake()" increases its parameter by 1, that address holds the value 2 when it is executed.**

Call-by-value-result

- **Usage: FORTRAN IV and ADA.**
- **Idea:**
  - **Value, not address, is passed into called procedure's A.R.**
  - **When called procedure ends, the final value is copied back into the argument's address.**
- **Equivalent to call-by-reference except when there is aliasing.**
  - **"Equivalent" in the sense the program produces the same results, NOT the same code will be generated.**
  - **Aliasing : two expressions that have the same l-value are called aliases. That is, they access the same location from different places.**
  - **Aliasing happens through pointer manipulation.**
    - ▷ ***call-by-reference with an argument that can also be accessed by the called procedure directly, e.g., global variables.***
    - ▷ ***call-by-reference with the same expression as an argument twice; e.g., $test(x, y, x)$.***

Call-by-name (1/2)

- Usage: Algol.
- Idea: (not the way it is actually implemented.)
  - Procedure body is substituted for the call in the calling procedure.
  - Each occurrence of a parameter in the called procedure is replaced with the corresponding argument, i.e., the TEXT of the parameter, not its value.
  - Similar to macro substitution.
  - Idea: a parameter is not evaluated unless its value is needed during the computation.

Call-by-name (2/2)

- **Example:**

```
void init(int x, int y)
{ for(int k = 0; k <10; k++)
    {  x++; y = 0;}
}
```

```
main()
{ int j;
  int A[10];
  j = -1;
  init(j,A[j]);
}
```

- **Conceptual result of substitution:**

```
main()
{ int j;
  int A[10];
  j = -1;
  for(int k = 0; k<10; k++)
  { j++;        /* actual j for formal x */
    A[j] = 0; /* actual A[j] for formal y */
  }
}
```

- **Call-by-name is not really implemented like macro expansion. Recursion would be impossible, for example, using this approach.**

How to implement call-by-name?

- Instead of passing values or addresses as arguments, a function (or the address of a function) is passed for each argument.
- These functions are called thunks. , i.e., a small piece of code.
- Each thunk knows how to determine the address of the corresponding argument.
  - Thunk for j: find address of j.
  - Thunk for $A[j]$: evaluate j and index into the array A; find the address of the appropriate cell.
- Each time a parameter is used, the thunk is called, then the address returned by the thunk is used.
  - $y = 0$: use return value of thunk for y as the l-value.
  - $x = x + 1$: use return value of thunk for x both as l-value and to get r-value.
  - For the example above, call-by-reference executes $A[1] = 0$ ten times, while call-by-name initializes the whole array.
- Note: call-by-name is generally considered a bad idea, because it is hard to know what a function is doing – it may require looking at all calls to figure this out.

Advantages of call-by-value

- **Consider not passing pointers.**
- **No aliasing.**
- **Arguments are not changed by procedure call.**
- **Easier for static optimization analysis for both programmers and the complier.**
- **Example:**

```
x = 0;
Y(x);   /* call-by-value */
z = x+1; /* can be replaced by z=1 for optimization */
```

- **Compared with call-by-reference, code in the called function is faster because of no need for redirecting pointers.**

Advantages of call-by-reference

- **Efficiency in passing large objects.**
- **Only need to copy addresses.**

Advantages of call-by-value-result

- If there is no aliasing, we can implement call-by-value-result using call-by-reference for large objects.
- No implicit side effects if pointers are not passed.

Advantages of call-by-name

- **More efficient when passing parameters that are never used.**
- **Example:**

```
P(Ackerman(5),0,3)
/* Ackerman's function takes enormous time to compute */

function P(int a, int b, int c)
{ if(odd(c)){
      return(a)
  }else{ return(b)  }
}
```

- **Note: if the condition is false, then, using call-by-name, it is never necessary to evaluate the first actual at all.**
- **This saves lots of time because evaluating a takes a long time.**