Intermediate Code Generation

ALSU Textbook Chapter 6.1–6.4, 6.5.1–6.5.3, 6.6–6.8

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

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

Intermediate code generation

- **Compiler usually generates 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:* $x := [y]\ [\mathbf{op}]\ z,$ *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
```

Declarations: storage addresses (1/2)

- **The storage space for variables with the same scope is "usually" allocated together.**
- **Examples:**
  - **Example 1:**
    - *Static data area: for global data.*
    - *Allocated when the program starts and remains to be so for the entire execution.*
  - **Example 2:**
    - *So called* ***activation record*** *(A.R.) when a procedure is invoked.*
    - *This area holds all data that are local to this procedure.*
    - *This area is active only when the associated procedure is called.*
    - *May have multiple copies when recursive calls are allowed.*

Declarations: storage addresses (2/2)

- **Storage addresses for variables are thus two-tuples.**
  - **Class of variables: determine which area.**
  - **Offset: the relative address within this area.**
- **Example:**
  - **A is a global variable with the offset 8.**
  - **Meaning: the storage address of A is 8 plus the starting of the static data area.**
- **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.**
- **Need to maintain an offset for each scope that is not closed.**

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(**$table$,$width$**):**
    - ▷ ***record the total data size used by the symbol table*** $table$ ***is*** $width$***.***
  - **enterproc(**$table$, $name$, $newtable$**):**
    - ▷ ***insert a procedure*** $name$ ***into*** $table$***;***
    - ▷ ***the symbol table of the procedure*** $name$ ***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: stacks for different objects such as offsets, and symbol tables.**
  - $offset$: **the amount of storage used in this scope**
  - $tblptr$: **the symbol stable used in this scope**
- **Operations.**
  - **push(**$object$,$stack$**)**
  - **pop(**$stack$**)**
  - **top(**$stack$**): top of stack element**

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:**
  - $Block \rightarrow begin\ M_4\ Declarations\ Statements\ end$
    - ***{ /* a scope is closed */***
    - ***pop(tblptr);***
    - ***pop(offset); }***
  - $M_4 \rightarrow \epsilon$
    - ***{ /* enter a new block or open a new scope */***
    - ***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.; }***
- **There is a better way to handle nested procedures.**
  - **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:**
  - **gen([address #1], [assignment], [address #2], operator, address #3);**
    - ▷ *Use switch statement to actually print out the target code;*
    - ▷ *Can have different gen() for different target codes;*
- **Variable accessing: depend on the type of [address #i], generate different codes.**
  - **Watch out the differences between l-address and r-address.**
  - **Types of [address #i]:**
    - ▷ *Local temp space.*
    - ▷ *Parameter.*
    - ▷ *Local variable.*
    - ▷ *Non-local variable.*
    - ▷ *Global variable.*
    - ▷ *Registers, constants, . . .*
- **Run-time memory management, allocating of memory spaces for different types of variables during run time, is an important issue and will be discussed in the next set of slides.**

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 generating 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 gen(p, ":=",E.place);***
    else error_msg("var undefined",id.name); }
- $E \rightarrow E_1 + E_2$
  - ***{E.place := newtemp();***
  - ***gen(E.place, ":=",E_1.place,"+",E_2.place);***
  - ***freetemp(E_1.place);freetemp(E_2.place);}***
- $E \rightarrow -E_1$
  - ***{E.place := newtemp();***
  - ***gen(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.place***
    else error_msg("var undefined",id.name);}

Type conversions (1/2)

- **Assume there are only two data types, namely integer and float.**
- **Assume automatic type conversions by widening.**
  - **May have different rules such as disallowing conversion.**
- $E \rightarrow E_1 + E_2$
  - **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*
      $gen(temp_1, \text{“:=”}, \textit{int-to-float}, E_1.place);$
      $gen(E, \text{“:=”}, temp_1, \text{“+”}, E_2.place);$
    - *else*
      $gen(temp_1, \text{“:=”}, \textit{int-to-float}, E_2.place);$
      $gen(E, \text{“:=”}, temp_1, \text{“+”}, E_1.place);$
    - $freetemp(temp_1);$

Type conversions (2/2)

- **Assume there are only two data types, namely integer and float.**
- **Checking of matched data types in the assignment phase.**
- $S \rightarrow id := E$
  - {
  - **p := lookup(id.name,top(tblptr));**
  - **if p is null then error_msg(“var undefined”,id.name);**
  - **else**
    - ***if*** $id.type = E.type$ ***then gen(p, “:=”,E.place);***
    - ***else // generate type conversion assignment or***
    - ***// generate error message if conversion is not allowed***
  - }

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 during the data declaration phase, and then stored at the symbol table.***
- **PASCAL uses**

```
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 during the data declaration phase, and then stored at the symbol table.**

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 compiler time during the data declaration phase, and then stored at the symbol table.**

YACC code for addressing array elements

- **Attributes needed during computation:**
  - **elesize: size of each element in the array.**
  - **array: a pointer to the symbol table entry containing information about the array declaration.**
  - **ndim: the current dimension index**
  - **base: $base$ address of this array**
  - **offset: the address of the array element**
    - ▷ ***this value is null for a simple variable***
  - **place: where a variable is stored**
    - ▷ ***type of variables: global, local, temp, ...***
    - ▷ ***the offset in the area of the type of variables***
    - ▷ ***Example: the 3rd (offset) temp (type) variable***
- **$limit(array, m) = n_m$ is the number of elements in the mth coordinate of the array $array$.**

Simple 1-D array addressing

- $L \rightarrow id\ [\ E\ ]$
  - ***{***
  - ***p := lookup(id.name,top(tblptr));***
  - ***check for id errors;***
  - ***make sure id is declared as a 1-D array;***
  - ***L.offset := newtemp();***
  - ***gen(L.offset,":=",p.elesize,"$*$",E.place);***
  - ***freetemp(E.place);***
  - ***gen(L.offset,":=",L.offset,"+",p.base);***
  - ***L.place = p.place;***
  - ***}***

Compute addresses for array elements

- $L \rightarrow Elist\ ]$
  - **{L.offset := newtemp();**
  - **gen(L.offset,":=",$Elist$.elesize,"*",$Elist$.place); freetemp($Elist$.place);**
  - **L.place := $Elist$.base;}**
- $Elist \rightarrow Elist_1, E$
  - **{ t := newtemp(); m := $Elist_1$.ndim + 1;**
  - **gen(t, ":=",$Elist_1$.place,"*",limit($Elist_1$.array,m));**
  - **gen(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;}**

Parsing multi-dimensional array

- **Input:** **a[i,j]**
- $L \Longrightarrow Elist\ ]$
  - ***Generate code for*** $base$ ***...***
- $\Longrightarrow Elist\ ,\ E\ ]$
  - ***Generate code for first dimension times the limit of the second dimension***
  - ***Generate code for adding the second index***
- $\Longrightarrow Elist\ ,\ j\ ]$
  - ***process the array variable***
  - ***Generate code for the first index***
- $\Longrightarrow a\ [\ E\ ,\ j\ ]$
  - ***Know the first index***
- $\Longrightarrow a\ [\ i\ ,\ j\ ]$

Grammar for accessing r-values

- **For simple variables**
  $L \rightarrow id$
  - ▷ *{p := lookup(id.name,top(tblptr));*
  - ▷ *if p ≠ null then*
  - ▷ *begin*
  - ▷ *L.place := newtemp();*
  - ▷ *gen(L.place,":=",p.place);*
  - ▷ *L.offset := null; // it is a simple variable*
  - ▷ *end*
  - ▷ *else error_msg("var undefined",id.name);}*
- **For indexed variables**
  $E \rightarrow L$
  - ▷ *{if L.offset = null then /* L is a simple id */*
  - ▷ *E.place := L.place;*
  - ▷ *else begin /* L is an indexed variable */*
  - ▷ *E.place := newtemp();*
  - ▷ *gen(E.place,":=",L.place,"[",L.offset,"]");*
  - ▷ *end}*
    - **Need a special 3-address code for indirect addressing above.**

Boolean expressions

- **Two choices for implementation:**
  - **Numerical representation: encode true and false values numerically, and then 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.**
    - ▷ *Example:* $(a > 1)$ *and* $(p\_function(\cdots) > 100)$
    - ▷ *if the calling of* $p\_function()$ *creates some side effects, then this side effect is not carried out in the case of* $(a > 1)$ *being false.*

Numerical representation

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

```
100: if a < b goto 103       107: temp2 := 1
101: temp1 := 0              108: if e < f goto 111
102: goto 104                109: temp3 := 0
103: temp1 := 1 /* true */   110: goto 112
104: if c < d goto 107       111: temp3 := 1
105: temp2 := 0 /* false */ 112: temp4 := temp2 and temp3
106: goto 108                113: temp3 := temp1 or temp4
```

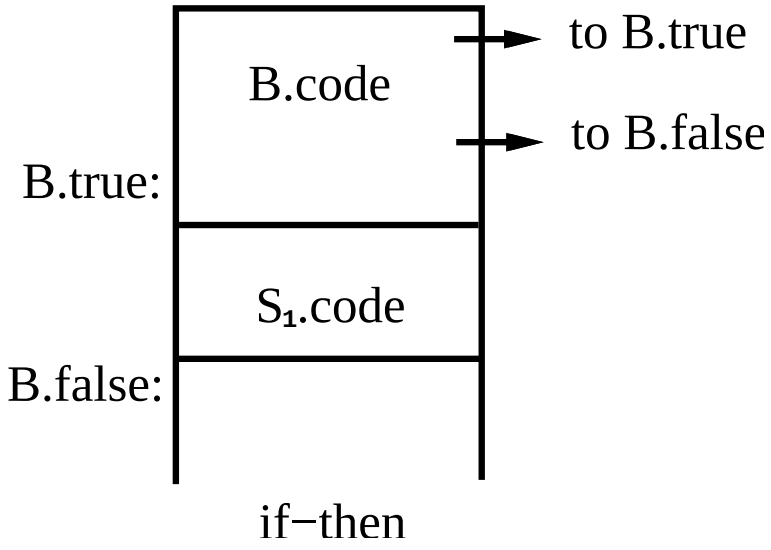
Flow of control representation

Production	Semantic actions
$B \rightarrow\ id_1\ relop\ id_2$	B.true := newlabel(); B.false := newlabel(); B.code := gen("if",id_1,relop,id_2,"goto", B.true,"else","goto",B.false) || gen(B.true,":")
$S \rightarrow\ \text{if}\ B\ \text{then}\ S_1$	S.code := B.code || S_1.code || gen(B.false,":")

- **|| is the code concatenation operator.**
- **Uses only S-attributed definitions.**

If-then: YACC implementation

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



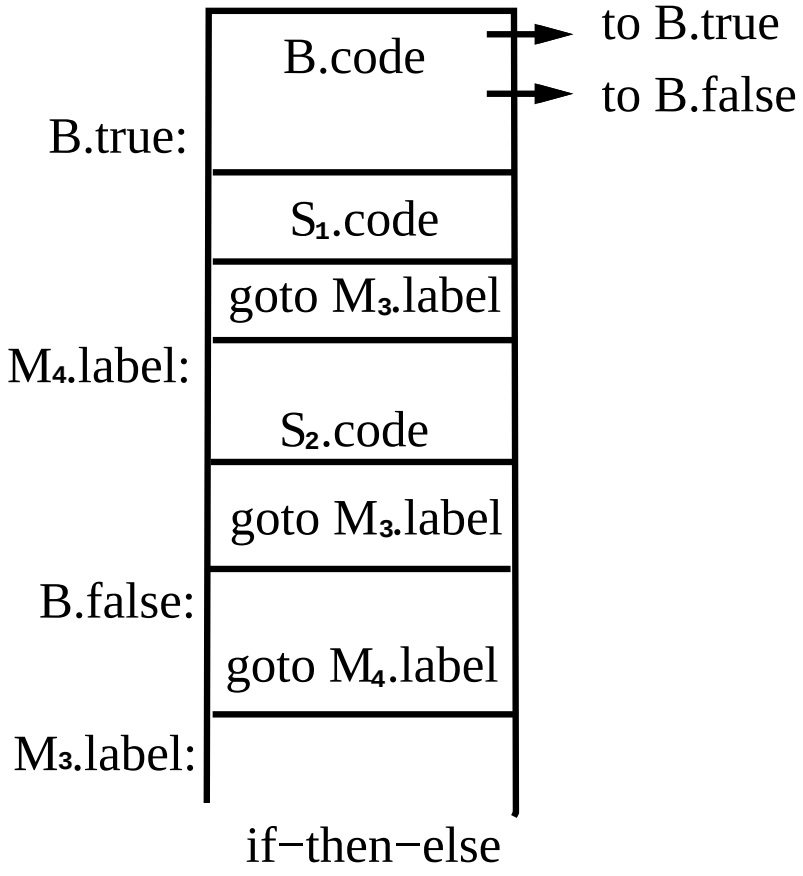
If-then-else: L-attributed defs.

Production	Semantic actions
$P \rightarrow S$	S.next := newlabel(); P.code := S.code || gen(S.next,":")
$B \rightarrow\ id_1\ relop\ id_2$	B.true := newlabel(); B.false := newlabel(); B.code := gen("if",id_1,relop,id_2,"goto", B.true,"else","goto",B.false) || gen(B.true,":")
$S \rightarrow$ if B then S_1 else S_2	S_1.next := S.next; S_2.next := S.next; S.code := B.code || S_1.code || gen("goto",S.next) || gen(B.false,":") || S_2.code

- **Need to use inherited attributes of S to define the attributes of S_1 and S_2.**
- **Solutions: Back-patching or rewriting syntax directed defs.**

If-then-else: S-attributed defs. (1/2)

- $B \rightarrow\ id_1\ relop\ id_2$
  - { B.**true := newlabel();**
  - B.**false := newlabel();**
  - **gen(“if”,**id_1**,relop,**id_2**,“goto”,** B**.true,“else”,“goto”,**B**.false);**
  - **gen(**B**.true,“:”);}**
- $S \rightarrow$ **if** B **then** $S_1\ M_3$ **else** $M_4\ S_2$
  - {
  - **gen(“goto”,**M_3**.label);**
  - **gen(**B**.false,“:”);**
  - **gen(“goto”,**M_4**.label);**
  - **gen(**M_3**.label,“:”);}**
- $M_3 \rightarrow \epsilon$
  - {M_3**.label := newlabel();**
  - **gen(“goto”,**M_3**.label);}**
- $M_4 \rightarrow \epsilon$
  - {M_4**.label := newlabel();**
  - **gen(**M_4**.label,“:”);}**



If-then-else: S-attributed defs. (2/2)

- The code is ugly and slow.
- Use a post-processing optimization package to rewrite these ugly and slow codes.
  - If the next instruction after a label is a goto, then de-referencing this label.
  - That is, the followings are equivalent.

```
▷         goto label1;
          ...
  label1:
          goto label2;
▷         goto label2;
          ...
  label1:
          goto label2;
```

- Another form of back-patching.

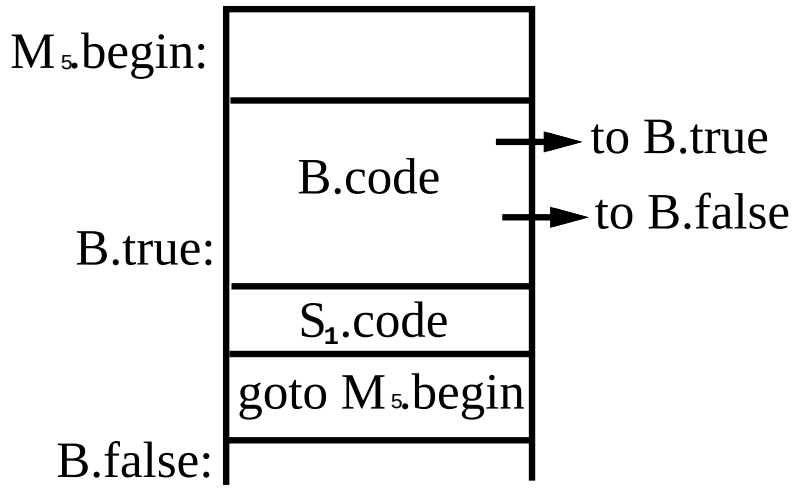
While loop

Production	Semantic actions
$B \rightarrow\ id_1\ relop\ id_2$	B.true := newlabel(); B.false := newlabel(); B.code := gen("if",id_1,relop,id_2,"goto", B.true,"else","goto",B.false) || gen(B.true,":")
$S \rightarrow\ \text{while}\ B\ \text{do}\ S_1$	S.begin := newlabel(); S.code := gen(S.begin,":") || B.code || S_1.code || gen("goto",S.begin) || gen(B.false,":")

- **Uses only S-attributed definitions.**

While loop: YACC implementation

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

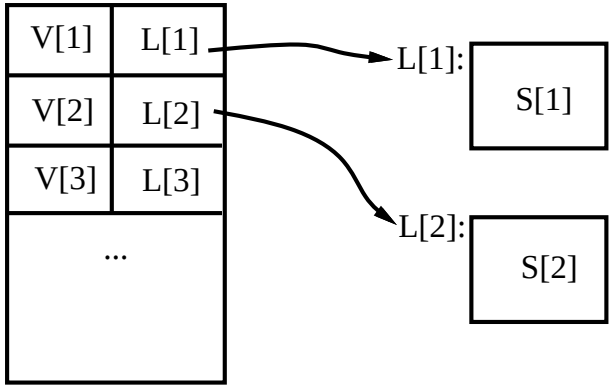
```
code to evaluate E into t
goto test
L[1]: code for S[1]
    goto next
    ...
L[k]: code for S[k]
    goto next
L[d]: code for S[d]
    goto next
test:
    if t = V[1] goto L[1]
    ...
    if t = V[k] goto L[k]
    goto L[d]
next:
    ...
```

Can easily be converted into a lookup table!

```
    code to evaluate E into t
    if t <> V[1] goto L[1]
    code for S[1]
    goto next
L[1]: if t <> V[2] goto L[2]
    code for S[2]
    goto next
    ...
L[k-1]: if t <> V[k] goto L[k]
    code for S[k]
    goto next
L[k]: code for S[d]
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***
    - ▷ ***gen("PARAM", q);***
    - ▷ ***gen("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.**