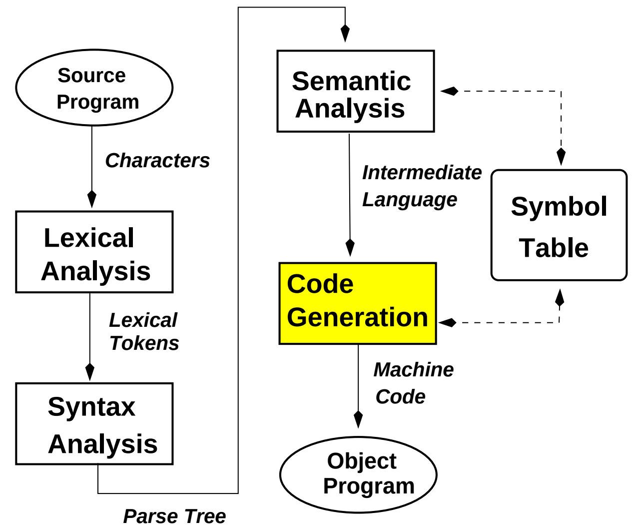
CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the
Winter 2015/2016 term at the University of Toronto



Reading Assignment

```
Fischer, Cytron and LeBlanc

Chapter  11
```

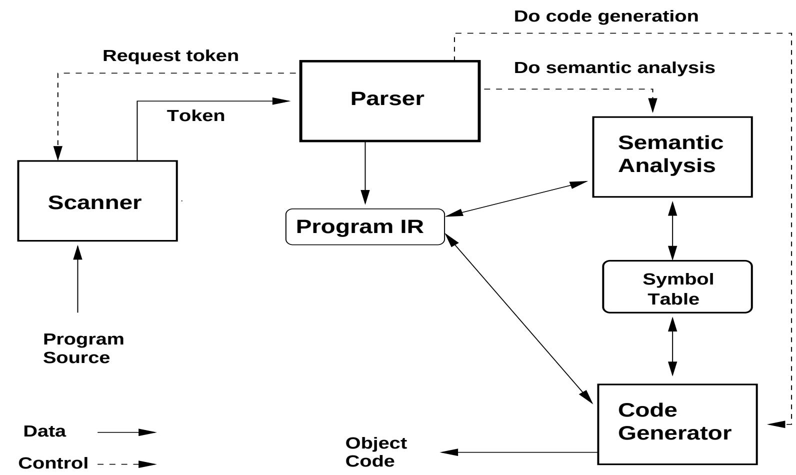

Translation and Code Generation

- The ultimate goal of a compiler is to transform a program in some source language into machine instructions for some target machine.
- In many compilers this transformation is a two step process
  1. *Translate* the program from its syntactic representation (parse tree) into some easy to generate intermediate representation (IR).
  2. *Generate Code* for the target machine from the intermediate representation.
- There are a number of advantages to this two phase approach
  - The IR is usually very easy to generate and manipulate.
  - Generating the IR can be done without worrying about constraints imposed by the target machine (e.g. a limited number of registers).
  - It is often convenient to perform machine-independent optimization by manipulating the IR.
  - One IR can serve as an interface to code generators for different hardware (e.g. gcc)

Translation of Programs

- Translation is the process of transforming a program into some intermediate representation.
- Input to the translation process is the representation of the program as produced by the parser *after* it has been subject to semantic analysis.
- Conceptually translation is performed on a parse tree for the program
- Major translation issues are expressions and control structures.
- Translation builds the programs control flow graph.

Single Pass Compiler Architecture

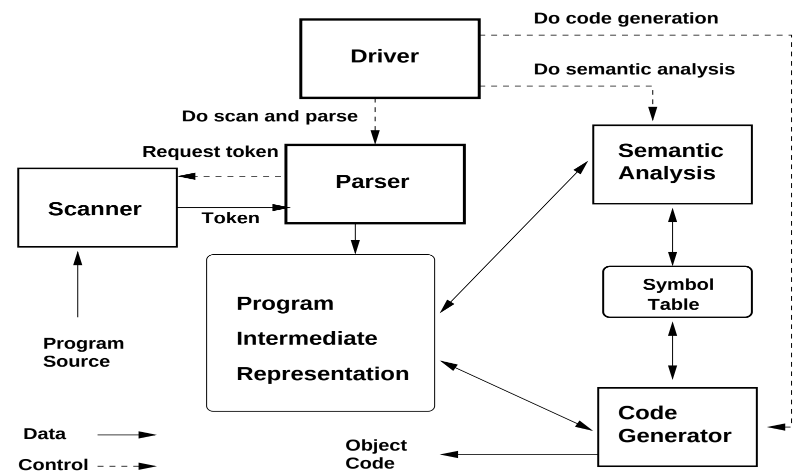


Single Pass Syntax Directed Translation

- Translation of the program is driven from the parser.
  This is the typical approach for a *one-pass compiler*.
- Syntax rules are designed to facilitate translation by breaking constructs into convenient pieces.
- Data stacks that run parallel to the parse stack are used to store the information used for semantic analysis and code generation.
  (See Slides 237 and 238)

Example	ifStatement	:	ifHead truePart elsePart **fi**
			/* Patch address of BRANCH FALSE from ifHead to elsePart */
			/* Patch branch at end of truePart to next address */
	truePart	:	**then** statement
			/* generate forward BRANCH after statement */
	elsePart	:	**else** statement
		|	λ
	ifHead	:	**if** expression
			/* generate BRANCH FALSE after expression */

Multi Pass Compiler Architecture



Abstract Syntax Tree Directed Translation

- An alternative to syntax directed translation is to first transform the program into an *Abstract Syntax Tree (AST)* and then perform semantic analysis and code generation by walking the AST.
  This approach is typical of multipass compilers and compilers for difficult languages.
- Syntax Directed Translation and Parse Tree Directed Translation are functionally equivalent and can implement the same semantic analysis and code generation actions.

Quadruples Intermediate Representation

- General form of a Quadruple

  label (Operator , leftOperand , rightOperand , result)

  Assume an infinite number of temporary storage locations: R_i
  Assume a *tuple label* of the form Ti refers to tuple i .
- Quadruple example A * B + C / D - E

1	(mult	, A	, B	, R_1)	$R_1 \leftarrow A * B$
2	(divide	, C	, D	, R_2)	$R_2 \leftarrow C/D$
3	(add	, R_1	, R_2	, R_3)	$R_3 \leftarrow R_1 + R_2$
4	(subtract	, R_3	, E	, R_4)	$R_4 \leftarrow R_3 - E$

Translation of Expressions

- Translate AST (parser output) for an expression into some intermediate representation that is good for code generation.
- A Data Object is a data structure used during translation to encode the addresses of variables and the value of literal constants.
- The processing of a variable reference (Slides 245 .. 249) results in a Data Object containing the address of the variable.
  If the variable reference was a computable address (e.g. an array subscript) then IR will have been generated to calculate the array element address and the Data Object will describe the result of this computation.
- For literal constants, the Data Object contains (or points to) to the value of the constant.

Expression Processing

- Generate code for expressions with a depth first traversal of the AST that represents the expression.
- Temporary storage may be required during expression processing to hold intermediate results. Temporary storage can be
  - Hardware registers. Often assume an infinite number of pseudo registers (i.e. the R_is) and map these later to a finite number of real registers.
  - Memory locations allocated in the activation record of the current procedure or function. Reuse these locations as required in disjoint expression.
- Optimizations strategies
  - Defer evaluation of operands until an operator forces evaluation (i.e. a value is required).
  - Evaluate expressions involving only constants at compile time.
    Watch out for arithmetic faults (e.g. overflow, divide by zero)
- Compiler may need to generate conversion tuples to deal with mixed mode arithmetic (e.g. integer & float operands)

genData

During IR generation, we need to keep track of registers, tuples and lists of tuples.

```
typedef  unsigned registerNumber ;
typedef  unsigned tupleAddress ;
typedef  struct {
                tupleAddress      start ;
                tupleAddressList  true  ;
                tupleAddressList  false ;
                registerNumber    result ;
         }  genData ;
```

Invariant: for any construct, the `start` field contains the address to the first tuple in the construct.

Invariant: for any construct that produces a value, the `result` field contains the number of the register containing the result.

- These slides use the name node to refer to the parse tree node that is being created (i.e. like `RESULT` in cup rules or $$ in Bison rules).

gencode - the master code generation function

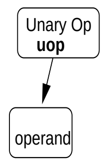
- Generate code for the parse tree starting at root.
  `genData gencode( AbstractSyntaxTree root )`

  Invariant: for any abstract syntax tree starting at root, `gencode` translates the abstract syntax tree into IR and returns a `genData` structure describing its output.
- `gencode` contains a large switch statement on the type of node at root.
- `gencode` calls itself recursively to process subtrees of root.
- Things to watch for:
  - Handling of the `genData.start` field.
  - Handling of the `genData.result` field.
- The slides that follow are a definition of the `gencode` cases for possible node types at *root*

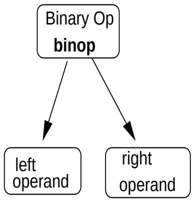
Auxiliary Translation Functions

- Emit one quadruple to the output
  ```
  genData emit( int op, int operand1,
          int operand2, int result )
  ```
- Patch the address field of the branch instruction at brat to the address addr.
  `patch( tupleAddress brat , tupleAddress addr )`
- Allocate the next available pseudo register R_i
  `registerNumber newregister()`
- Return the address of the *next* tuple Ti
  `tupleAddress nexttuple()`
- Merge two tuple address lists
  ```
  tupleAddressList merge( tupleAddressList right,
       tupleAddressList left )
  ```


Translating Arithmetic Operators

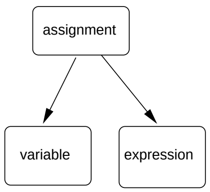


/* **uop** is + or − */
op = gencode(operand)
node.start = op.start
node.result = emit(**uop** , op.result, 0, newreg())



/* **binop** is + or − or * or / */
left = gencode(left−operand)
node.start = left.start
right = gencode(right−operand)
node.result = emit(**binop** ,left.result, right.result, newreg())

Translating assignment statements



var = gencode(variable)
node.start = var.start   variable

exp = gencode(expression)
emit(assign , exp.result , 0 , var.result)   expression

Generate code for variable address first in case evaluation of the expression has a side effect that would change the address of the variable, e.g.

```
A[ I ] = I++
```


Expression Examples

Example: X = (-A) * B + C * D

1	(uminus	, A	,	, R_1)		$R_1 \leftarrow -A$
2	(mult	, R_1	, B	, R_2)		$R_2 \leftarrow R_1 * B$
3	(mult	, C	, D	, R_3)		$R_3 \leftarrow C * D$
4	(add	, R_2	, R_3	, R_4)		$R_4 \leftarrow R_2 + R_3$
5	(assign	, R_4	,	, X)		X $\leftarrow R_4$

Example A[K + 1] = 3.14 * A[K] / K

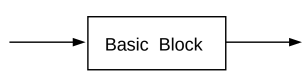
1	(add	, K	, =1	, R_1)	$R_1 \leftarrow K+1$
2	(subsadr	, A	, R_1	, R_2)	$R_2 \leftarrow \&A[R_1]$
3	(subsval	, A	, K	, R_3)	$R_3 \leftarrow A[K]$
4	(mult	, =3.14	, R_3	, R_4)	$R_4 \leftarrow 3.14 * R_3$
5	(intreal	, K	,	, R_5)	$R_5 \leftarrow intreal(K)$
6	(div	, R_4	, R_5	, R_6)	$R_6 \leftarrow R_4 / R_5$
7	(assign	, R_6	,	, R_2)	$*R_2 \leftarrow R_6$

---

subsadr - subscript and produce address
subsval - subscript and produce value
intreal - convert integer value to real value
assign - store value to address

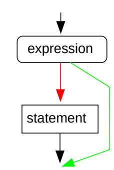
Statement Processing

- A basic block is a program is a piece of the program with one entry and one exit. Examples: an expression ; a sequence of non-branching statements (e.g. assignments).
- Branching statements cause a change in the flow of control through a program. Examples: if, for, switch .
- The control flow graph for a program describes all possible flows of control between basic blocks in the program.
  Some control is visible to the programmer, some is implicitly generated by the compiler (e.g. inside boolean expressions).
- The translation of statements involves emitting conditional and unconditional branch instructions to implement control flow graph for the program.

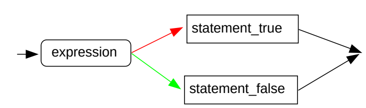


Control Flow Graph Examples

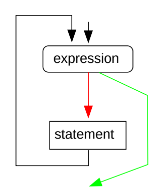
if expression **then**
statement



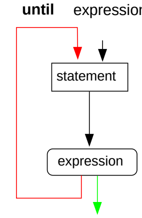
if expression **then**
statement_true
else
statement_false



while expression **do**
statement
end



repeat
statement
until expression



Forward and Backward Branches

- To generate IR to implement the control flow (loops, conditional statements, etc.) in a program the compiler generates tuple branch instructions.

  It is often the case that the compiler will not know the target address of the branch instruction at the time that it generates the instructions so some form of address patching mechanism will be required.
  Quadruple indices (i.e. Tis) may be used in branch instructions at this stage, resolve these to real machine addresses later

  Most control structures are properly nested so a *stack of branch addresses* is an appropriate mechanism for saving branch addresses.
  Or store the addresses in the AST.

  Invariant: every forward branch should be patched *exactly once*.

- Forward branches - emit a branch instruction with undefined address.
  Save the address of the instruction so that its address can be patched later.

- Backward branches - save target address of future backward branch (e.g. branch from end of loop back to start), at the appropriate point during translation. Use this address when generating the backward branch.

Boolean Expressions and Relational Operators

- The relational operators $<$, $\leq$, $=$, $\neq$, $\geq$, $>$ are usually treated as binary operators that produce a boolean result.

- Expression interpretation (PL/I) - treat boolean operators as ordinary binary operators. Implies full evaluation of boolean expressions.

- Conditional interpretation (C, Java) - only evaluate as much of a boolean expression as is required to determine its value. Generate branching code instead of arithmetic code. Definitions:

A **and** B	**if** A **then** B **else false**
A **or** B	**if** A **then true else** B
not A	Invert true and false in A

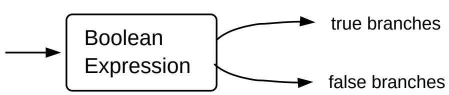
- Also need to be able to generate boolean values for assignment statements.

Translation for Boolean Expressions

- Because boolean expressions often occur inside control constructs (e.g. if statements and while loops) it is convenient to think of a boolean expression as generating branches to two targets

true target	address to branch to if boolean expression is true
false target	address to branch to if boolean expression is false

- Relational operators use compare operation followed by branch on true/false.
- Boolean variables are tested and branches are generated for true and false values of the variables.
- Things to watch for:
  - Handling of the `genData.true` field (tuple list).
  - Handling of the `genData.false` field (tuple list).



Translating Boolean Expressions

- A depth first traversal of the boolean expression tree is exactly backwards from the best order for generating branching code.

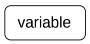
- Using depth first order, all branches will be *forward* branches.
  There may be many branches for a true or false result.

- Technique: maintain two lists of branch instruction addresses
  $node.true$ $\quad true\ list$ – branch instructions for true value
  $node.false$ $\quad false\ list$ – branch instructions for false value
  At end of expression patch all branch instructions in $true\ list$ and $false\ list$ to appropriate targets[a].

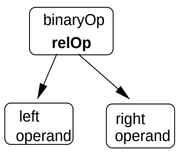
- Assume a conditional branch tuple
  (branch , value , trueAddress , falseAddress)
  Where a branch is take to the *trueAddress* if *value* is **true** and to *falseAddress* otherwise.

[a] Since each instruction is on exactly one list, the address fields of the tuples (or instructions) can be used to temporarily store the lists.

Translating Boolean variables and relational operators

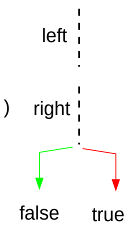


res = emit(branch, variable, ? , ?)
node.start = res.start
node.true = res.true
node.false = res.false

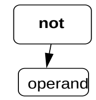


/* **relOp** is one of < <= = != >= > */

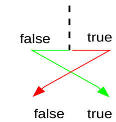
left = gencode(left−operand)
node.start = left.start
right = gencode(right−operand)
rel = emit(**relOp**, left.result, right.result, newreg())
br = emit(branch, rel.result, ? . ?)
node.true = br.true
node.false = br.false

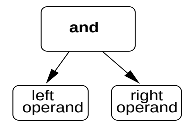


Translating and or not

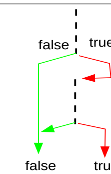


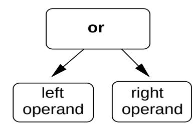
op = gencode(operand)
node.start = op.start
node.true = op.false
node.false = op.true



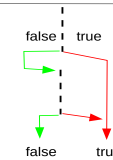


left = gencode(left−operand)
node.start = left.start
right = gencode(right−operand)
patch(left.true, right.start)
node.true = right.true
node.false = merge(left.false, right.false)



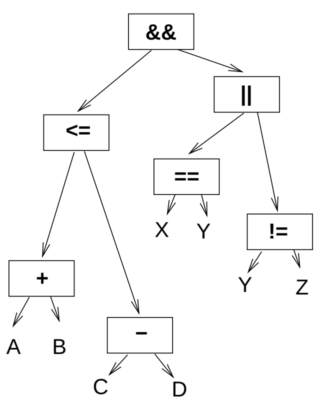


left = gencode(left−operand)
node.start = left.start
right = gencode(right−operand)
patch(left.false, right.start)
node.true = merge(left.true , right.true)
node.false = right.false



Tuple Generation Example

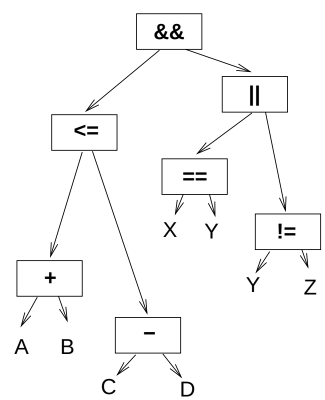
(A + B) <= (C − D) && (X == Y || Y != Z)



Compare with expression processing example in Slides 252 to 253

Tuple Generation Example

(A + B) <= (C − D) && (X == Y || Y != Z)



1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Boolean Expression Examples

Example **not** (A **or** B) **and** C **or** X $<$ Y

1	(branch	, A	, $T4$	, $T2$	)
2	(branch	, B	, $T4$	, $T3$	)
3	(branch	, C	, ?	, $T4$	)
4	(lessthan	, X	, Y	, R_1	)
5	(branch	, R_1	, ?	, ?	)

True list: $T3_{true}$, $T5_{true}$ False list: $T5_{false}$

Example A **and not** B **or not** A **and** B

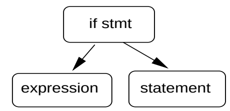
1	(branch	, A	, $T2$	, $T3$	)
2	(branch	, B	, $T3$	, ?	)
3	(branch	, A	, ?	, $T4$	)
4	(branch	, B	, ?	, ?	)

True list: $T2_{false}$, $T4_{true}$ False list: $T3_{true}$, $T4_{false}$

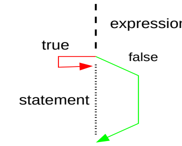
Translating if statements

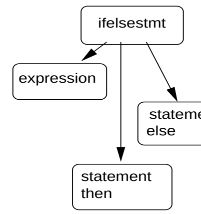
if expression **then** statement

if expression **then** statement **else** statement

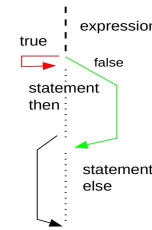


exp = gencode(expression)
node.start = exp.start
patch(exp.true, nexttuple())
gencode(statement)
patch(exp.false , nexttuple())





exp = gencode(expression)
node.start = exp.start
patch(exp.true , nexttuple())
gencode(statement−then)
trueExit = emit(branch,uncond, ? , 0)
patch(exp.false , nexttuple())
gencode(statement−else)
patch(trueExit.true , nexttuple())



if Statement Examples

Example **if** (X <= Y) X = Y

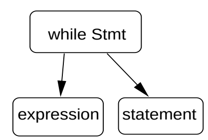
1	(lesseq	, X	, Y	, R_1	)
2	(branch	, R_1	, $T3$	, $T4$	)
3	(assign	, Y	,	, X	)

Example **if** (A[J] $<$ A[K]) X = A[J] **else** X = A[K]

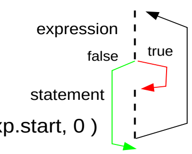
1	(subsval	, A	, J	, R_1	)	
2	(subsval	, A	, K	, R_2	)	
3	(less	, R_1	, R_2	, R_3	)	test
4	(branch	, R_3	, $T5$	, $T8$	)	branch true/false
5	(subsval	, A	, J	, R_4	)	true part
6	(assign	, R_4	,	, X	)	
7	(branch	, uncond	, $T10$	,	)	branch to end
8	(subsval	, A	, K	, R_5	)	else part
9	(assign	, R_5	,	, X	)	

Translating while and repeat statements

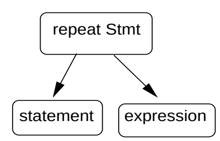
while expression **do** statement



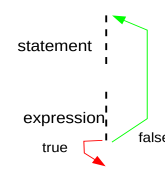
exp = gencode(expression)
node.start = exp.start
patch(exp.true , nexttuple())
st = gencode(statement)
brBack = emit(branch, uncond, exp.start, 0)
patch(exp.false , nexttuple())



repeat statement **until** expression



st = gencode(statement)
node.start = st.start
exp = gencode(expression)
patch(expr.false, st.start)
patch(exp.true, nexttuple())



while and do Loop Examples

Example **while** (K <= 100) { A[K] = K ; K ++ }

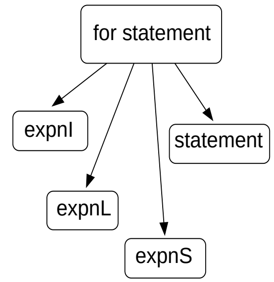
1	(lesseq	, K	, =100	, R_1)	test
2	(branch	, R_1	, $T3$	, $T8$)	branch when finished
3	(subsadr	, A	, K	, R_2)	body
4	(assign	, K	,	, R_2)	
5	(add	, K	, =1	, R_3)	
6	(assign	, R_3	,	, K)	
7	(branch	, true	, $T1$	,)	branch to test

Example **repeat** { A[K + 1] = A[K] ; K - - } **until** (K <= 0)

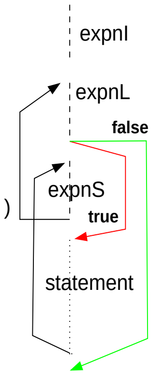
1	(add	, K	, =1	, R_1)	body
2	(subsadr	, A	, R_1	, R_2)	
3	(subsval	, A	, K	, R_3)	
4	(assign	, R_3	,	, R_2)	
5	(sub	, K	, =1	, R_4)	
6	(assign	, R_4	,	, K)	
7	(lesseq	, K	, =0	, R_5)	test
8	(branch	, R_5	, $T9$	, $T1$)	branch to body

Translating for loops

for (expnI ; expnL ; expnS **)** statement



init = gencode(expnI)
limit = gencode(expnL)
step = gencode(expnS)
brLimit = emit(branch, true, ? , 0)
stmt = gencode(statement)
retryBr = emit(branch, uncond, ? , 0)
node.start = init.start
patch(limit.true, stmt.start)
patch(brLimit.true , limit.start)
patch(retryBr.true , step.start)
patch(limit.false, nexttuple())



for Loop Example

Example **for** (J = 0 , K = N ; J < K ; J ++ , K - -)
{ T = A[K] ; A[K] = A[J] ; A[J] = T }

1	(assign	, =0	,	, J)	*expnI*
2	(assign	, N	,	, K)	
3	(less	, J	, K	, R_1)	*expnL*
4	(branch	, R_1	, $T10$	, $T18$)	branch if finished
5	(add	, J	, =1	, R_2)	*expnS*
6	(assign	, R_2	,	, J)	
7	(sub	, K	, =1	, R_3)	
8	(assign	, R_3	,	, K)	
9	(branch	, true	, $T3$	,)	branch to *expnL*
10	(subsval	, A	, K	, R_4)	body
11	(assign	, R_4	,	, T)	
12	(subsadr	, A	, K	, R_5)	
13	(subsval	, A	, J	, R_6)	
14	(assign	, R_6	,	, R_5)	
15	(subsadr	, A	, J	, R_7)	
16	(assign	, T	,	, R_7)	
17	(branch	, true	, $T5$	,)	branch to *expnS*

Translating break, continue, exit

- These statements are translated into unconditional branch instructions that go to the start or end of the loop that contains them.
- Simplest implementation uses an auxiliary *loop stack* containing two fields:
  - $loopStart$ address of the start of the loop
  - $loopExits$ list of branch instructions that exit the loop
- This stack is pushed at the start of each loop and the address of the start of the loop is recorded in $loopStart$.
- An unconditional forward branch is generated for each **break** or **exit** statement. The address of this branch is added to the $loopExits$ entry on top of the loop stack.
- At the end of each loop, all of the branches in the $loopExits$ list are fixed up to point just beyond the loop. Then the stack is popped.

Translating switch and case statements

switch (expn) **{**

case L1 : statement
break

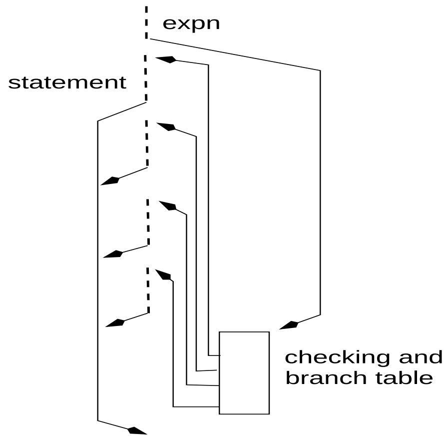
case L2 : statement
break

. . .

case Ln : statement
break

default : statement
break

}



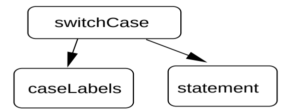
Switch/Case Translation Issues

- Use different implementation strategies depending on label set
  - *Dense Set* Use branch table indexed by expression.
  - *Sparse Set, Small* Map statement into a chain of if statements.
  - *Sparse Set, Large* Generate an internal hash table.
  - *Non Indexable* Generate an internal hash table.
- Switch/case can be compiled in one pass by recording information on each case and generating a branch table after all cases have been seen.
- Need stack or table of labels, and statement start addresses.
- Must be able to handle nested case statements.
- Should test expression limits to avoid wild jumps.

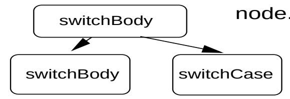
Switch Processing

- Each case in a switch statement is associated at least one (and perhaps several) case labels.
- For each case, need to keep track of label value and start of corresponding statement.
- Example that follows assumes compact and dense case label set and builds a branch table implementation.
- For convenience, branch table is built *after* the switch statement.
- Use branch generation and patching mechanisms similar to those used for **if** and **for** statements. Assume a list of (label, statement start , break statement addresses) triples. Utility functions:
  - Create one list node $makeCaseNode(label, start, breaks)$
  - Append node to list $append(list, node)$
  - Emit branching table $emitTable(list)$

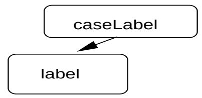
Translating switch statement body



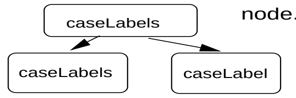
node.start = statement.start
node.list = makeCaseNode(caseLabels.list , statement.start , statement .breaks)



node.list = append(switchBody.list , switchCase.list)

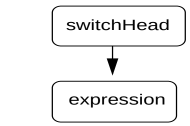


node.list = makeNode(label.value , 0 , 0)

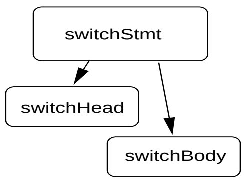


node.list = append(caseLabels.list , caseLabel.list)

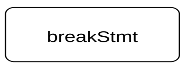
Translate switch statement control



exp = gencode(expression)
node.start = exp.start
sub = emit(subsaddr , ? , exp.result , 0)
br = emit(branch, uncond , sub.result , 0)
node.true = sub.true



node.start = switchHead.start
tbl = emitTable(switchBody)
patch(switchHead.true ,. tbl.start)
patch(switchBody,breaks , nexttuple())



br = emit(branch, uncond, ? , 0)
node.breaks = br.true
node.start = br.start

switch Example

switch(I + 1) {	1	(add	, I	, =1	, R_1)
	2	(subsadr	, $T13$	, R_1	, R_2)
	3	(branch	, uncond	, R_2	,)
case 1 : J += 2 ;	4	(add	, J	, =2	, R_3)
	5	(assign	, R_3	,	, J)
break ;	6	(branch	, uncond	, $T17$	,)
case 3: J = J * 4 ;	7	(mult	, J	, =4	, R_4)
	8	(assign	, R_4	,	, J)
break ;	9	(branch	, uncond	, $T17$	,)
case 2 :					
case 4 ; J = J - I ;	10	(sub	, J	, I	, R_5)
	11	(assign	, R_5	,	, J)
break ;	12	(branch	, uncond	, $T17$	,)
}	13	(branch	, uncond	, $T4$	,)
	14	(branch	, uncond	, $T10$	,)
	15	(branch	, uncond	, $T7$	,)
	16	(branch	, uncond	, $T10$	,)

Translating Procedure and Function Declarations[a]

- Perform semantic analysis on routine header.
- Record formal parameter declarations in symbol/type table.
- Record type of return value for a function.
- Emit branch around the routine.
- Emit prologue code to set up run time environment for the routine.
- Process declarations for local variables in the body of the routine.
  Lay out the routines activation record.
  Translate statements in the body of the routine.
- Emit epilogue code as required.
- Record entry point address and parameter information in symbol table entry for the routine *in its parent's scope*.

[a] To simplify our discussion the term *routine* will be used to refer to procedures and functions in situations where the difference between the two is irrelevant.

Prologues and Epilogues

- The prologue code is typically emitted at the head of each routine. It is usually responsible for
  - Saving any registers modified by the routine.
  - Saving the return address for the call.
  - Updating the display to the correct addressing environment
  - Allocating storage for the (rest of) the routines activation record.
- The epilogue code is invoked to effect a return from the routine. It is usually responsible for
  - Deallocating all local storage for the routine.
  - For functions, making sure the return value is passed back to the caller.
  - Restoring the registers that were saved in the prologue.
  - Restoring the display to the callers addressing environment.
  - Branching to the return address for the routine.

Complete Activation Record

temporaries and data with dynamic size
local variables
nth parameter
...
1st parameter
register save area
static link
dynamic link
return address
return value

Returning

- In most implementations the **return** statement is translated into a branch to the epilogue code which performs any cleanup that is required and then branches to the routines return address.
- For **return** *expression* if the function's return type is a scalar value, it is usually returned in a hardware register or on top of a stack.
- If a function returns a large object (e.g. array or record) a buffer has to be allocated to hold the object. This can either be done in the caller, or it can be done in the function. In most cases the caller passes a pointer to a buffer to hold the result. The buffer must eventually be freed to avoid memory leaks.

X = bigReturner(...)	Pass address of X to bigReturner
Y = bigReturner(...).fieldName	Pass address of tempBuffer to bigReturner. Free tempBuffer later

Caller vs. Callee

- The actions required to call and return from a routine can be performed in the calling program or in the called routine. These actions include
  - Saving and restoring registers.
  - Display saving and update.
  - Handling return values.
  - Allocating space for the block mark.
  - Copying of value arguments as required.
- Example - saving and restoring registers.
  - *Caller Save* - The calling program is responsible for saving and restoring registers. It only needs to save the registers that it knows are busy. Costs some code space at every call.
  - *Callee Save* - The called routine is responsible for saving and restoring registers. It only needs to save those registers that is actually uses. Usually save space since there is only one copy of the save/restore code.

Function Declaration Example

	1	(branch	, true	, $T19$,	)
int F(int *A, B[]) {	2	(savereg	,	, ,	)
	3	(display	, 1	, ,	)
int C, D ;	4	(local	, =2	, =0 ,	)
C = *A ;	5	(deref	, A	, ,	R_1)
	6	(assign	, R_1	, ,	C)
D = B[1] ;	7	(subsval	, B	, =1 ,	R_2)
	8	(assign	, R_2	, ,	D)
if(C $>$ D)	9	(greater	, C	, D ,	R_3)
	10	(branch	, R_3	, $T11$,	$T13$)
return C	11	(assign	, C	, ,	$retVal$)
else	12	(branch	, true	, $T16$,	)
return D + C ;	13	(add	, D	, C ,	R_4)
	14	(assign	, R_4	, ,	$retVal$)
	15	(branch	, true	, $T16$,	)
	16	(unlocal	, =2	, =0 ,	)
	17	(undisplay	, 1	, ,	)
}	18	(return	, $retVal$	, ,	)

Translating Procedure and Function Calls

- Lookup the routine in the symbol table.
- Process argument list (if any)
  - Type check each argument against the corresponding formal parameter.
  - Generate code to pass the argument to the routine. Allocate temporary storage for the argument if required.
- Generate call to the routine.
- Generate any required post-call cleanup code.

Argument-Passing Methods

Formal parameters in routine definition are just place holders, replaced by arguments during a routine call.
Parameter passing - matching of arguments with formal parameters when a routine call occurs. But... what does `A[i]` mean: a name? a value? an address?

Possible interpretations:

- call by value: pass value of `A[i]`
- call by reference: pass location of `A[i]`
- call by name: pass something[a] that calculates address of `A[i]`
- call by value-result: copy value of argument into formal parameter on entrance, copy values of formal parameters back into arguments on exit.

[a] Historically an address generating internal function called a *thunk*

Argument Passing Methods

- *By Value* - Formal parameter acts like a variable local to the routine that is initialized with the value of the argument. Programming languages vary on whether it can be modified inside the routine.
- *By Result* - Formal parameter acts like an uninitialized variable local to the routine. As the routine returns the value of this variable is assigned to the argument (which must be a variable).
- *By Value-Result* - Like By Result except the variable is initialized as in By Value.
- *By Reference (address)* - The formal parameter is assigned the *address* of the argument. It acts like a hidden pointer that is automatically dereferenced as required. By reference is the preferred method for large data items like arrays and structures.
- *By Name* - Like by reference *except* that the address of the argument is *recalculated every time the formal parameter is used* in the routine.
  Primarily of historical interest (Algol 60). In functional languages it's called *lazy evaluation*

Call by Value-Result

Same as by reference unless aliasing is used.

```
procedure f (x,y : integer);
begin
  x := x+1;
  x := a;
  y := y-1;
  write (x,y);
end;
```

```
begin (* main *)
  a = 1; b := 2;
  f (a,b);
  write (a,b);
end.  (* main *)
```

Output by reference: 2 1 2 1

Output by value-result: 1 1 1 1

Call by Name[a]

```
procedure print (j : integer, a : integer);
begin
  for j := 1 to  3 do
     write (a);
end;
begin (* main *)
  A[1] := 1;
  A[2] := 2;
  A[3] := 3;
  i := 0;
  print( i, A[i] );
end;
```

Output by name: 1 2 3

[a] For an interesting use of call by name, see the Wikipedia entry for **Jensens Device**

Argument Passing Issues

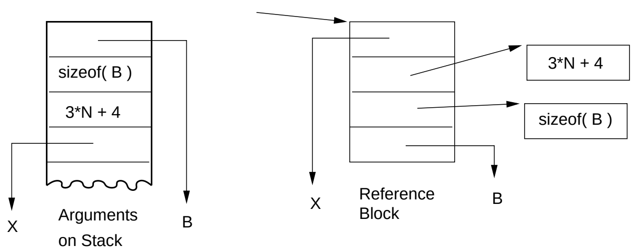
- How is the argument passing technique determined?
  - *Caller's Choice* - In some languages (e.g. PL/I, Fortran) the form of the argument determines the technique (e.g. value or reference) that is used to pass the argument.
    Usually all arguments are passed by reference with compiler generated temporary locations used for value parameters.
  - *Callee's Choice* - In most modern languages (e.g. C, Pascal) the form of the formal parameter declaration determines the technique that must be used to pass the argument.
- Most implementations try to avoid passing large objects (e.g. records, arrays) by value. If value parameters can't be modified in the routine (i.e. they behave like **consts**) then large objects can be secretly passed as reference parameters.

  If value parameters can be modified in the routine then a copy must be made by the caller or the callee. This is inefficient.

Argument Passing Implementations

procedure P (**var** I : **int** , J : **int** , **var** K [*] : **int**)
/* reference value reference */
. . .
end P
var X, N, B[128] : **int**

P(X , 3 * N + 4 , B) /* Call P */



Handling Dynamic Arguments

- Many languages allow the declaration of a routines to specify formal parameters that have dynamically specified components. Examples:
  array 1.. * **of real** **string**(*)
  Usually the *type/structure* of the parameter is fixed and what varies from one call to another is the *size* of the argument.
- Dynamic parameters are usually implemented by creating a run-time *descriptor* (a.k.a *dope vector*) which contains the information necessary to access the argument.
- Code is generated to fill in the descriptor at each call to the routine.
- All accesses to the parameter inside the routine use information from the descriptor for access to the argument. This is related to the technique for handling dynamic arrays in Slide 266

Function Call Examples

Example: X = F(IP , AR) ;

1	(scalararg	, IP	, =1	, R_1	)
2	(arrayarg	, AR	, =2	, R_2	)
3	(funcname	, F	,	, R_3	)
4	(fcall	, R_3	, =2	, R_4	)
5	(assign	, R_4	,	, X	)

Example sort(A , ASIZE, acompare)

1	(arrayarg	, A	, =1	, R_1	)
2	(scalararg	, ASIZE	, =2	, R_2	)
3	(addrarg	, acompare	, =3	, R_3	)
4	(funcname	, sort	,	, R_4	)
5	(call	, R_4	, =3	,	)

Input and Output Statements

- Input and Output statements often map directly to calls on builtin functions:
  Turing: **put** "Hello World"
  C: printf("%s\n", "Hello World") ;
  For input/output with a format, the builtin function typically has a FSM to interpret the format and transfer data.
- PL/I and Fortran have a more complicated form of formatted input and output in which the format can be determined dynamically.
  PUT EDIT(A(N), (K)F(5)) S, (DO I = 1 TO K , B(K)) ;
  Where N and K are arbitrary expressions and (K)F(5) means
  K instances of format item F(5)

  To implement this build a weave of branches between the format item list and the data list, or make the format list and the data list *coroutines*.

Classes and Modules

- A class-like construct is available in a number of languages.
- Classes are used to isolate some data and a collection of procedures and functions that operate on the data. i.e as an *abstract data type*
- Classes allow separate compilation in large software systems. This is the primary use of classes in C++ and modules in Modula-3
- Classes provide *information hiding* so that data internal to the class is not visible outside of the class. Information hiding is an important tool in allowing parts of a software system to be constructed and maintained separately.

Module Implementation Issues

- Import mechanisms, explicit vs. implicit. Modula-3 example:
  IMPORT Math ;
  FROM Math IMPORT sin, cos, sqrt ;
- Export mechanisms and information hiding.
  Information Hiding means that nothing outside the class can determine the representation of or modify the value of a hidden item.
- Initialization and finalization of class instances.
  - When should it happen?
  - Can it be guaranteed to happen? e.g. assignment of class variables, class instances embedded in other constructs.
  - What is the correct order of class initialization?
  - What is the correct order of class finalization?
- Nested class declarations.

Kinds of Classes

- The difficulty in implementing a class depends on the kind of class, single instance, multiple instance or template multiple instance.
- Single Instance Classes There is exactly one instance of the classes internal data and routines. Examples: modules in Turing, Modula-2, and Modula-3 , Packages in ADA.
- Multiple Instances Classes There may be multiple instances of the classes internal data but only one instance of the class member functions. Examples modules in Euclid, Classes in C++ , Classes in Java.
- Template Multiple Instance Classes. The class is parameterized by constant and type information (templates)which can be used to specialize each instance of the class.. Usually requires multiple copies of the classes data *and member functions*. Examples: parameterized modules in Euclid, template Classes in C++ and Java

Generic Class

class {
  Import declarations
  Export declarations

  Internal constants, types and variables

  Exported constants and types

  Internal functions and procedures

  Exported functions and procedures

}

Implementing Class Imports and Exports

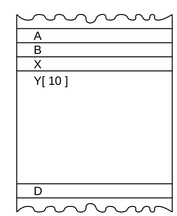
- Managing imports and exports is a symbol table management issue.
- The body of a class is a separate scope like the body of a routine.
- Create a type table entry for describing classes even if classes aren't treated like types in the language. The type table entry will be similar to the ones used for records
- The names exported by a class are linked together in the symbol table in a list that is pointed to from the classes type table entry. This list is used outside of the class to resolve references to exported items.[a]
- Symbol table entries for items imported into a class can be copied into the symbol table of the class or if a scope stack (Slide 189) is used for symbol table search the imported items can be pushed onto the scope stack

[a] This is similar to the processing required to resolve references to fields in a record (see Slides 190 .. 191).

Single Instances Class Data

- Treat class like a structure variable
- Allocate storage in enclosing major scope, i.e. class data is handled like minor scope data.
- Member functions are children of enclosing major scope
- Control access to class data via symbol table lookup management

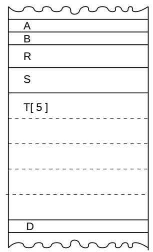
```
var  A , B :  integer
Class   C1  {
   var  X, Y[10]  :  integer
   func   F( Z  :   integer  ) {
      var     P, Q  :  boolean
        . . .
    }   % end F
    . . .
}    % end  C1
 var  D  :  boolean
```



Multiple Instances Classes

- Treat class like a structure type, Similar to a record or structure type.
- Allocate storage separately for each class instance *variable*.
- For space efficiency, generate one instance of each member function.
- Give member functions access to class instance data via an extra parameter or a hidden pointer for the class.

```
var  A , B :  integer
Class   C2  {
  var  X  :  integer
  proc  P( Y : integer )
    { X := Y }
  . . .
}  % end  C2
var  R , S  : C2
var  T[ 5] : C2
var  D :  boolean
```



Examples

proc P(Y : **integer**) { X := Y }

becomes

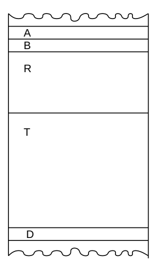
proc P(self : @C2 , Y : **integer**) { self -> X := Y }

R.P(17)	becomes	C2.P(@R , 17)
T[A].P(B)	becomes	C2.P(@T[A] , B)

Multiple Instance Template Classes

- Treat each class template instance as a distinct structure type
- Allocate storage separately for each instance variable as with multiple instance classes.
  Size of allocated storage may depend on template parameters.
- Generate an instance of each member function for each distinct class template instantiation.
- Give member functions access to class instance data via an extra parameter or hidden pointer for the class.

```
var  A , B :  integer
Class   C3 < T, N >   {
  var  X [ N ] :  T
  proc  P( Y :  T  )
    { X[ 2 ]  := Y }
  . . .
}  % end  C3
var   R : C3 < integer ,5 >
var   T : C3< C2, 8 >
var   D :  boolean
```



Class Initialization and Finalization

- Initialization is a activity that occurs when an instance of a class is being allocated. Generally initialization is responsible for initializing the local data in the instance. An example of an initializer is the Class *constructor* in C++ and Java.
- Finalization is the activity that takes place when storage for an instance of a class is about to be deallocated. Finalization is generally responsible for cleaning up local data, e.g. deallocating internal storage to prevent memory leaks. An example of a finalizer is the Class *destructor* in C++ and Java
- If class instances can be embedded within other constructs (e.g. as fields of a structure or another class) then it may take a lot of effort in the implementation to guarantee that initializers and finalizers are always invoked when they should be.
- A language design should provide clear guidance on the rules for initialization and finalization. Often this is not done.

Examples

var A , B : C2

A := B

Should A (and any classes embedded in A) be finalized before B is assigned?

Usually finalization is done in reverse order of initialization, For example:

var T[100] : C2

initialize T[1] **...** T[100]

finalize T[100] **...** T[1]

If several classes are mutually dependent, it might be necessary to apply a topological sort to determine the correct initialization order.