Reading Assignment

Fischer & LeBlanc

Chapter 10

Chapter 11

Run Time Storage Organization

- The implementation of a programming language must include a plan for how the program and its data will be represented during execution of the program.
- Program Data Issues
  - Allocating storage for variable local to procedures and functions.
  - Addressing of procedure and function local storage.
  - Allocating storage for dynamically allocated variables.
  - Managing storage used for dynamically allocated variables.
- Types of program data
  - Global and static data
  - Procedure and function local data
  - Dynamically allocated data
- Program code representation issues will be discussed later.

Addressing of Data

- Most modern computers provide several different *addressing modes*, some possibilities are listed below

Absolute	main memory address
Register	memory address in hardware register
Register + displacement	Effective address is sum of memory address in register and a constant displacement.
Indexed	Contents of index register is added to register + displacement address
Indirect	memory contains *address* of data

- The language implementor must use the addressing modes on the target machine to provide access to the program and its data.
- Register + displacement, perhaps with indexing is the most common form of addressing.

Global and Static Storage

- Storage for global (main program) data is allocated once at the start of program execution.
- Some languages allow (e.g. C, Pl/I) or require (e.g. Fortran) data to be given the *static* storage attribute.
  Storage for static data is also allocated once at the start of program execution.
- Compiler generated tables on constants and initial values for variables may also be treated as static data.
- If the available addressing modes allow it (i.e. maximum allowable displacement is large enough) one hardware register might be used to provide addressability for global data.

Local Storage

- The variables declared in a procedure or function are the *local storage* of the procedure or function.
- In most modern languages, local storage is allocated when the procedure or function is called and deallocated when the procedure or function returns.
- In programming languages that allow recursion, there may be many instances of the local storage for a procedure or function allocated at a given point in the programs execution.
  Each instance is associated with a particular call of the procedure or function.
- In almost all languages, storage allocation for local variables follows a *stack-like* (LIFO) discipline, i.e. the most recently allocated storage is the first to be deallocated.
- Therefore, conceptually at least, a stack is used to manage the allocation and deallocation of local storage.

Storage Allocation for Variables

- Variables declared in a procedure or function are usually allocated storage in the local storage of the procedure or function in the order that they are declared.
- Scalar variables are simply allocated storage based on the underlying hardware container (byte, word, doubleword) that is used to implement the variables type.
- More complicated data structures (arrays, records, unions) require the complier to provide a mapping algorithm to for access to information stored in the data structure (e.g. elements of an array, fields in a record).

Array Storage Access

- In most modern languages (except Fortran), arrays are laid out in memory in *row major order*, i.e. the rightmost subscript of consecutive array elements varies most rapidly.
- Because arrays can be passed as parameters and/or allocated in dynamic storage, the compiler needs to be able to calculate the address of an arbitrary element of an array given only the base address of the array and information from the declaration of the array.
- Generic array declaration:
  $typeA[\ L_1 : U_1\ ,\ L_2 : U_2\ ,\ \ldots\ L_n : U_n\ ]$
- Generic array reference:
  $A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]$

Array Storage Access

- Define: *base address* of an array
  $base = addr(A[\ L_1\ ,\ L_2\ ,\ \ldots\ L_n\ ]$
- Define: *stride* in the $i_t h$ dimension
  $stride_i = U_i\ -\ L_i\ +\ 1$
- Then that address of an arbitrary element $A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]$ is

$$addr(A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]) = base + \sum_{i=1}^{n}((E_i - L_i) \cdot \prod_{j=i+1}^{n} stride_j)$$

- To simplify, let: $mult_i\ =\ \prod_{j=i+1}^{n} stride_j\ =\ stride_{i+1} \cdot mult_{i+1}$
  where $mult_n$ is the size of an array element (in address units)
  and $mult_0$ is the total size of the array, then

$$addr(A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]) = base + \sum_{i=1}^{n}((E_i - L_i) \cdot mult_i)$$

Horner's Rule and Array Subscripts

- Given

$$addr(A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]) = base + \sum_{i=1}^{n}((E_i - L_i) \cdot \prod_{j=i+1}^{n} stride_j)$$

  Apply Horner's rule for evaluating polynomials and factor

$$constPart = ((\ldots((L_1 \cdot stride_2 + L_2) \cdot stride_3 + \ldots L_{n-1}) \cdot stride_n + L_n$$

$$varPart = ((\ldots((E_1 \cdot stride_2 + E_2) \cdot stride_3 + \ldots E_{n-1}) \cdot stride_n + E_n$$

  then

$$addr(A[\ E_1\ ,\ E_2\ ,\ \ldots\ E_n\ ]) = base - constPart + varPart$$

- If the L_i are compile time constants then $constPart$ can be calculated at compile time and subtracted from $base$ when addressing code is generated.

Reading Assignment

```
Fischer & LeBlanc

     Chapter  12
```


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).
     See Fischer & LeBlanc Chapters 11, 12 and 13.
  2. *Generate Code* for the target machine from the intermediate representation.
     See Fischer & LeBlanc Chapters 9 and 15.
- There are a number of advantages to this two phase approach
  - The IR is usually very easy to generate.
  - 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 optimization on the IR.
  - The IR is easy to store and manipulate in the compiler.

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, but many compilers don't actually build the tree.
- Major translation issues are expressions and control structures.
- Translation effectively transforms the programs parse tree into the programs control flow graph.

Syntax Directed Translation

- Translation of the program is driven from the parser.
  Typical approach for a one-pass compiler.
- Syntax rules are designed to facilitate translation by breaking constructs into convenient pieces.
- In the examples that follow we'll assume typical syntax rules.
  For each rule we'll describe the translations performed *for that rule*.
  The name $node$ will be used to stand for the LHS of each rule.
  Subscripts on non terminal symbols will be used to indicate access to translation data structures.
  Example:

$$X = A \;\; a \;\; B$$
$$node_{foo} = A_{foo}$$
$$node_{bar} = B_{bar}$$
$$node_{baz} = B_{baz} - A_{baz}$$

Intermediate Languages

- Represent the structure of the program being compiled
  - Declaration structure.
  - Scope structure.
  - *Control flow structure.*
  - Source code structure.
- Used to pass information between compiler passes
  - Compact representation desirable.
  - Should be efficiently to read and write.

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 T_i 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 parse tree (parser output) for expression into some intermediate representation that is good for code generation.
- Conceptually translation uses a depth first traversal of the parse tree for the expression, but many compilers don't actually build a parse tree, instead they store information parallel to the parse stack.
- Data Objects[a] are used during translation to encode the addresses of variables and the value of literal constants.
- The processing of a variable reference 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.

[a] Fischer & LeBlanc page 379

Expression Processing

- Use a stack of Data Objects to keep track of operands and intermediate results. This stack can run parallel to the parse stack.
- Temporary storage may be required during expression processing to hold intermediate results. Temporary storage can be
  - Hardware registers. Often initially assume an infinite number of pseudo registers (i.e. the $R_i s$) 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)*

Intermediate Representation Generation

- We'll use quadruples to illustrate code generation. Assume an infinite number of temporary locations $R_1, .., R_\infty$
- Conceptually do a depth first left to right traversal of the expression tree. In practice use parse of expression to simulate tree traversal.
- Leaves will be variables or constants. Create a Data Object for each leaf. Data object contains pointer to type information.
- For unary operator nodes (e.g. `unaryOp operand`) generate

  k (unaryOp , operand , , R_i)
- For binary operator nodes (e.g. `operandLeft binaryOp operandRight`) generate

  k (binaryOp , operandLeft , operandRight , R_i)
- Compiler may need to generate conversion tuples to deal with mixed mode arithmetic (e.g. integer & float operands)

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)	$R_2 \leftarrow \&A[R_2]$
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

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 implictly generated by the compiler (e.g. inside boolean expressions).
- The translation of statements involves emiting conditional and unconditional branch instructions to implement control flow graph for the program.
- The examples given here assume the use of stacks parallel to the parse stack to hold translation information. An equivalent translation could be done by storing information in the nodes of a parse tree.

Statement Processing Data Structures

During that translation of statements we assume the stacks listed below which operate in parallel with the parse stack

$start$	The address of the first tuple that was generated for the corresponding parse stack item. Written as $node_{start}$
$true$	A list of tuples containing branch instructions that should be taken if the value of a boolean expression is true. Written as $node_{true}$.
$false$	A list of tuples containing branch instructions that should be taken if the value of a boolean expression is false. Written as $node_{false}$.
$list$	A list of data structures used to implement switch statements.
$result$	The number of the tuple that contains the final result of an expression node.

We assume that the processing of expressions and statements always sets $expn_{start}$, $expn_{result}$ and $statement_{start}$ to the correct value.

Translation Data Example

Stacks parallel to parse stack

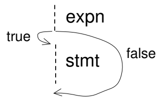
Parse	start	true	false	list	result
fi					
statement	T_{56}				
else					
statement	T_{28}	T_{55}			
then					
boolExpr	T_{23}	T_{23}, T_{25}	T_{24}, T_{27}		R_{47}
if					

Branches and Control Flow

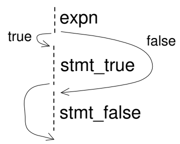
- To generate IR to implement the control flow (loops, conditional statements, etc.) in a program the compiler will need to generate tuple branch instructions.
- It is often the case that the compiler will not know the target address of the branch instruction at the time that it generates the instructions so some form of *address patching* mechanism will be required. Tuple indices (i.e. T_is) may be used in branch instructions at this stage, resolved to real machine addresses later. Fischer & LeBlanc call this *backpatching*.
- *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 code generation. Use this address when generating the backward branch.
- Most control structures are properly nested so a stack of branch addresses is an appropriate mechanism for saving branch addresses.

Translating Do, If and While Statements

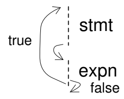
if (expn)
statement



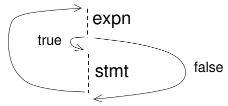
if (expn)
statement_true
else
statement_false



do
statement
while (expn)



while (expn)
statement



Translating if statements

if without else $statement \;=$ **if** ($expn$) $statement$

$node_{start} = expn_{start}$

patch($expn_{true}$, $statement_{start}$)

patch($expn_{false}$, $nextcode$)

if with else

$statement \;=$ **if** ($expn$) truePart **else** $statement$

$node_{start} = expn_{start}$

patch($expn_{true}$, $truePart_{start}$)

patch($expn_{false}$, $statement_{start}$)

patch($truePart_{true}$, $nextcode$)

$truePart \;=\; statement$

$node_{start} = statement_{start}$

$node_{true} = nextcode_{true}$

emit(branch, true, ? , unused)

if Statement Examples

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

1	lesseq	X	Y	R_1	
2	branch	R_1	T_3	T_4	
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	T_5	T_8	branch true/false
5	subsval	A	J	R_4	true part
6	assign	R_4		X	
7	branch	true	T_{10}		branch to end
8	subsval	A	K	R_5	else part
9	assign	R_5		X	

Translating while and do statements

While statement

$statement \;=$ **while** ($expn$) $statement$

emit(branch , true , $expn_{start}$, unused)

$node_{start} = expn_{start}$

patch($expn_{true}$, $statement_{start}$)

patch($expn_{false}$, $nextcode$)

Do while statement

$statement \;=$ **do** $statement$ **while** ($expn$)

$node_{start} = statement_{start}$

patch($expn_{true}, statement_{start}$)

patch($expn_{false}, nextcode$)

while and do Loop Examples

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

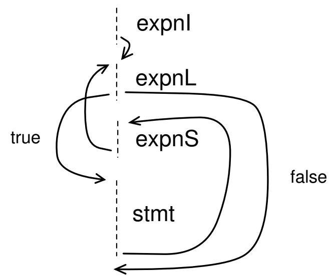
1	lesseq	K	=100	R_1	test
2	branch	R_1	T_3	T_8	branch when finished
3	subsadr	A	K	R_2	body
4	assign	K		R_2	
5	add	K	=1	R_3	
6	assign	R_3		K	
7	branch	true	T_1		branch to test

Example **do** { A[K + 1] = A[K] ; K - - } **while** (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	greateq	K	=0	R_5	test
8	branch	R_5	T_1	T_9	branch to body

Translating for Loops

for (expnI ; expnL ; expnS)
statement



Translating for Loops

For statement

$statement \;=\;$ **for** ($expnI$; $expnL$; $stepExpn$) $statement$

$node_{start} = expnI_{start}$

patch($expnL_{true}, statement_{start}$)

emit(branch , true, $stepExpn_{start}$, unused)

patch($stepExpn_{true}, expnL_{start}$)

patch($expnL_{false}, nextcode$)

$stepExpn \;=\; expnS$

$node_{start} = expnS_{start}$

$node_{true} = nextcode_{true}$

emit(branch, true, ?, unused)

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	, T_{10}	, T_{18}	, R_1)	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	, T_3	,)	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	, T_5	,)	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 auxilliary *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.

Boolean Expressions and Relational Operators

- Boolean operators **and**, **or**, **not**
- 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, Pascal) - 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.

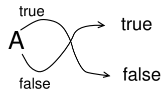
Translation for Boolean Expressions

- Because boolean expressions often occur inside control constructs (e.g. if statements and while loops) it is more 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

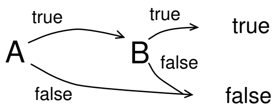
- Relational operators use compare operation followed by branch on true or false.
- Boolean variables are tested and branches are generated for true and false values of the variables.
- The **not** operator is implemented by swapping the true and false branch targets of its operand.

not A

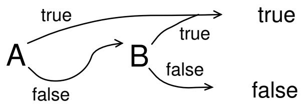


Translation for and and or

A **and** B



A **or** B



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*

$true\ list$	branch instructions for true value
$false\ list$	branch instructions for false value

  At end of expression patch all branch instructions in $true\ list$ and $false\ list$ to appropriate targets.
  Very convenient for implementing control constructs like **if** statements.
- Implementing **and**, **or**, **not** is done by manipulation of the true and false address lists.

Translating Boolean Expressions

- Operations on branch lists
  - Keep track of the start of a boolean expression $node_{start}$
  - Generate tuples $emit(op, operand, operand, result)$
  - Patch all branches on a list to a specified address $patch(list, addr)$
  - Merge two branch lists $merge(list, list)$
  - Get tuple address (T_i) of next tuple *to be generated* $nextcode$
  - Assume a conditional branch tuple

    (branch , value , trueAddress , falseAddress)
- When processing operators, assume that the $true\ list$ and $false\ list$ for its operands have already been generated (depth first order).
  The processing for each operand generates a new $true\ list$ and $false\ list$ that describe the result of the operation.
- Since each instruction is on exactly one list at any point in time, the address fields of the tuples (or instructions) can be used to temporarily store the lists.

Boolean Variables and Relations

Boolean variable $primary \ = \ variable$

$node_{start} = nextcode$
$node_{true} = nextcode_{true}$
$node_{false} = nextcode_{false}$
emit(branch , $variable$, ? , ?)

Relational Operator

$compare \ = \ leftOperand$ relOp $rightOperand$
$node_{start} = leftOperand_{start}$
emit(relOp , $leftOperand$, $rightOperand$, R_i)
$node_{true} = nextcode_{true}$
$node_{false} = nextcode_{false}$
emit(branch , R_i , ? , ?)

Conjunction, Disjunction and Negation[a]

Conjunction $conjunct \ = \ leftOperand$ **and** $rightOperand$

$node_{start} = leftOperand_{start}$
patch($leftOperand_{true}$, $rightOperand_{start}$)
$node_{true} = rightOperand_{true}$
$node_{false} =$ merge($leftOperand_{false}$, $rightOperand_{false}$)

Disjunction $disjunct \ = \ leftOperand$ **or** $rightOperand$

$node_{start} = leftOperand_{start}$
patch($leftOperand_{false}$, $rightOperand_{start}$)
$node_{true} =$ merge($leftOperand_{true}$, $rightOperand_{true}$)
$node_{false} = rightOperand_{false}$

Negation $negation \ = \$ **not** $Operand$

$node_{start} = Operand_{start}$
$node_{true} = Operand_{false}$
$node_{false} = Operand_{true}$

[a] See Slides 208 and 209

Boolean Expression Examples

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

1	(branch	, A	, T_4	, T_2	)
2	(branch	, B	, T_4	, T_3	)
3	(branch	, C	, ?	, T_4	)
4	(lessthan	, X	, Y	, R_1	)
5	(branch	, R_1	, ?	, ?	)

True list: $T_{3_{true}}$, $T_{5_{true}}$ False list: $T_{5_{false}}$

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

1	(branch	, A	, T_2	, T_3	)
2	(branch	, B	, T_3	, ?	)
3	(branch	, A	, ?	, T_4	)
4	(branch	, B	, ?	, ?	)

True list: $T_{2_{false}}$, $T_{4_{true}}$ False list: $T_{3_{true}}$, $T_{4_{false}}$

Reading Assignment

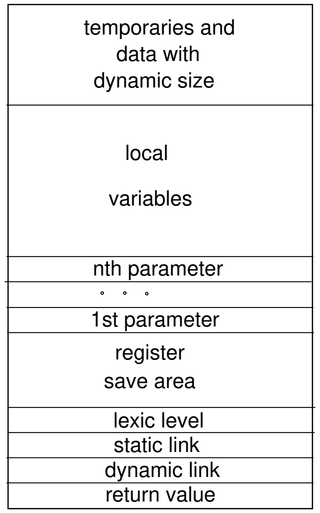
Fischer & LeBlanc

Chapter 9

Local Storage - Activation Records

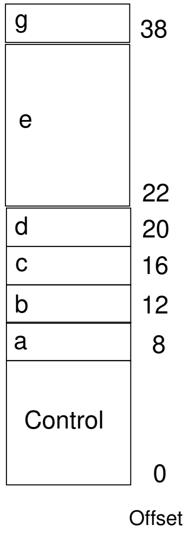
- The term *activation record* refers to the local storage that is allocated when a function or procedure is called.
- A typical activation record contains
  - The return address for the call
  - Control information required to maintain addressing of local storage.
  - Actual parameters passed to this particular call of the procedure or function
  - Storage for the local variables declared in the procedure or function.
  - Pointers to storage for variables that were dynamically allocated, .e.g arrays with dynamic bounds. Array may also be in the activation record.
  - Temporary storage required for expression evaluation.
  - Storage for variables from *micro scopes* contained in the procedure or function.

Complete Activation Record



Activation Record Example

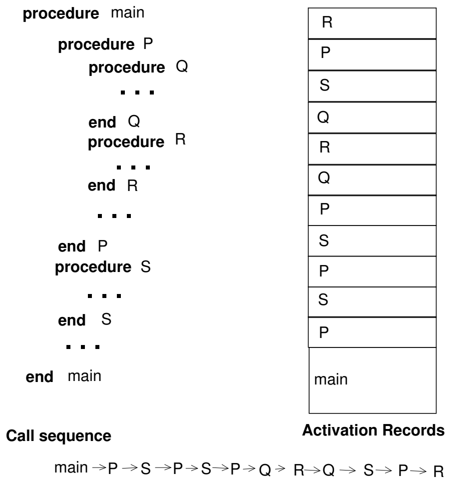
```
int F( int* a , float b ){
    int c ;
    char d[4];
    float e[4];
    int* g ;

    ...
}
```



Addressing Activation Records

- During the execution of a program each procedure needs to be able to access
  - Its own local data *corresponding to its invocation.*
  - The local data of any containing procedures.
  - Global data in the main program.
  - Any static data that it is allowed to access according to the scope and visibility rules of the language.
- An addressing mechanism called a *display* is used to describe the addressing for local storage of procedures and functions.

Activation Record Example



Fischer/Leblanc Figure 9.5

Lexical Levels

- Define: *lexical level* is the depth of static nesting of scopes (usually major scopes like procedures and functions) in a program.
- Theorem: In normal procedural languages, at any point during the execution of a program at most one activation record at each lexical level is visible .
- For the example in Slide 222

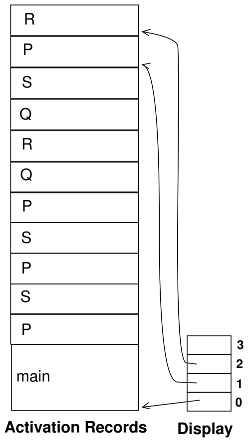
Name	Level	Visible Scopes
main	0	main
P	1	main, P
Q	2	main, P, Q
R	2	main, P, R
S	1	main, S

Displays

- Define: A *display* is a vector of addresses of activation records. The display is indexed by lexical level.
- Define: An *Order Number* is a non-negative integer that specifies the order in which objects are defined in a scope.
- If all objects are of unit size then the order number is the items relative offest in the scopes activation record.
  In most modern languages all objects are not of unit size and the order number is translated into a relative byte offset into the activation record.
- Corollary to Theorem: At any point during a programs execution, one display entry for each active lexical level is sufficient to provide addressing to all visible activation records.

Activation Records and Display

```
procedure main
    procedure P
        procedure Q
            . . .
        end Q
        procedure R
            . . .
        end R
        . . .
    end P
    procedure S
        . . .
    end S
    . . .
end main
```



Call sequence

main → P → S → P → S → P → Q → R → Q → S → P → R

Fischer/Leblanc Figure 9.5

Display Update

- To provide correct addressing during program execution, the display must be updated every time that the list of visible scopes changes.
- Micro scopes are usually subsumed in the nearest enclosing major scope (procedure or function) so that the display only needs to be updated on entry to or exit from a procedure or function.
- Types of procedure/function calls

same level	caller and callee have the same lexical level	Q calls R
up level	lexical level of callee greater than level of caller	P calls Q
down level	lexical level of caller greater than lexical level of callee	Q calls S

- The different types of calls require different amounts of the display to be saved and restored. Defer parametric calls to Slide 234.
- A algorithm for updating the display should be efficient and *correct* for all possible sequences of calls.

Display Update

- Define: A *dynamic link* is a pointer from an activation record to the previous activation record on the stack of activation records.
  The dynamic link is useful in deallocating an activation record on return from a procedure or function.
- Define: A *static link* is a pointer from an activation record to the activation record of its statically containing scope (parent).
- On procedure call to a procedure/function at level L, the static link field in the block mark is the value of display[L - 1]
- Define: The *block mark* is the control information that occurs at the start of every activation record. The block mark typically contains the return address for the invocation of the procedure or function, links for maintaining the display
- Retain dynamic link for resetting the stack pointer on return.
- Intuitively: chain down static links copying entries into display.
  Order is child, parent, grandparent, greatgrandparent .. ultimate ancestor.
- Cost depends only on depth of lexical nesting.

Display Update Algorithm

block mark:

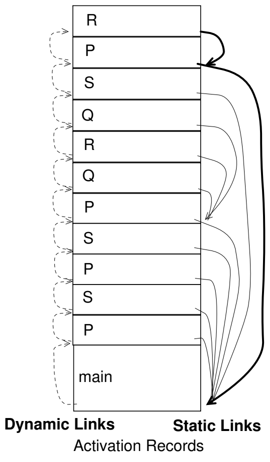
```
struct blockMark {
        void * returnAddress ;     /* return address */
        void * dynamicLink ;       /* dynamic link */
        void * staticLink ;        /* static link */
        short  lexicLevel ;        /* lexical level of this scope */
} ;
```

Algorithm:

```
P = address of topmost block mark ;
display[ * ] = empty ;       /* Neatness only */
L = P -> lexicLevel ;
while( L != 0 ) {
    display[ L ] = P ;
    P = P ->staticLink ;
    L = P -> lexicLevel ;
}
```

Static and Dynamic Links

```
procedure main
   procedure P
      procedure Q
         . . .
      end Q
      procedure R
         . . .
      end R
      . . .
   end P
   procedure S
      . . .
   end S
   . . .
end main
```



Call sequence

main → P → S → P → S → P → Q → R → Q → S → P → R

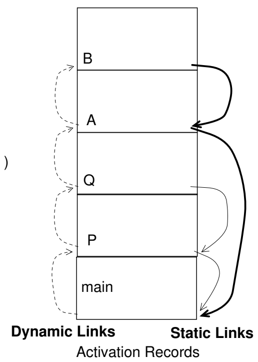
Fischer/Leblanc Figure 9.5

Parametric Procedures

- Many languages (C, Pascal, Fortran, Pl/I) allow the *name* of a function or procedure to be passed as an actual parameter to some other function or procedure. Other languages allow procedure and function *variables*.
- These constructs cause problems for display update algorithms because they allow a procedure or function to be executed far from the environment of their parent scopes.
- Intuitively, the correct display for a procedure or function passed as a parameter is the display as it existed at the point where it was originally passed as a parameter.
- Passing the entire display around with the parametric procedure or function is possible, but there are better solutions.
- The next two slides illustrate the problems caused by parametric procedures.

Parametric Procedure Example

```
procedure main
   procedure P
      procedure Q
         procedure R
            . . .
         end R
         A( R )
      end Q
      Q
   end P
   procedure A( F )
      procedure B( G )
         G
      end B
      B( F )
   end A
   P
end main
```

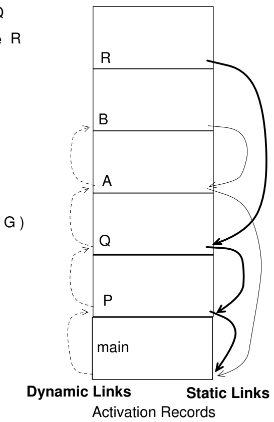


Call sequence

main → P → Q → A → B

Parametric Procedure Example - continued

```
procedure main
   procedure P
      procedure Q
         procedure R
            . . .
         end R
         A( R )
      end Q
      Q
   end P
   procedure A( F )
      procedure B( G )
         G
      end B
      B( F )
   end A
   P
end main
```



Call sequence

main → P → Q → A → B → R

Parametric Procedure Issues

- The presence of parametric procedures implies that the compiler cannot determine the to/from lexic level of calls.
- Parameter counting and/or parameter type checking may have to be done at run time.
- *Name* of parent procedure is not adequate to identify parent since multiple instances of the parents activation record may exist on the stack when the parametric procedure is called.
- Therefore , the internal representation of a parametric procedure must include
  - The code address for the entry point of the procedure.
  - some form of information that will allow the correct addressing environment for the parametric procedure to be established when it is called.

Parametric Procedures - Solutions

- There are several possible solutions for providing a correct addressing environment for parametric procedures.
  The solutions differ in how parametric procedures are represented internally.

1. Save entire display when parametric procedure is passed and pass it along as part of the parametric proc.
   Trivial to set up display, makes parametric procedures large objects.
2. Save entire display in local storage of caller and pass address to save display as part of parametric procedure.
3. Represent parametric procedure as address of parents local storage, and offest in this local storage of code address for parametric procedure.
   The address of parents local store is the correct static link for the parametric procedure.

More info in tutorial.

Constant Cost Display Update

- The cost of the display update algorithm in Slide 228 varies with the difference of the lexical levels of the calling and called procedure. An alternative is the *constant cost* display update algorithm.
- Constant Cost Display Update
  - On a call from level N to level M
  - Save display[M] in the local storage of the calling procedure.
  - Called procedure sets display[M] to point at its activation record.
  - On return, restore the saved value of display[M].
- Intuitively, display[M] will be modified, so save it first.
  If this algorithm is applied consistently, display will always be correct.
- Advantages: display update code is small, fast and easy to generate.
  Downlevel calls are cheap. Can extend for parametric procedures.
- Disadvantage: logically unnecessary save/restore on uplevel calls.

Reading Assignment

```
Fischer & LeBlanc

          Chapter  13
```

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 procedure.
- Emit prologue code to set up run time environment for the routine.
- Process body of the routine. Process declarations for local variables. Lay out activation record. Translate statements in the body.
- 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.

Function Declaration Example

	1	(branch	, true	, T_{19} ,	)
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	, T_{11} , T_{13}	)
return C	11	(assign	, C	, , $retVal$	)
else	12	(branch	, true	, T_{16} ,	)
return D + C ;	13	(add	, D	, C , R_4	)
	14	(assign	, R_4	, , $retVal$	)
	15	(branch	, true	, T_{16} ,	)
	16	(unlocal	, =2	, =0 ,	)
	17	(undisplay	, 1	, ,	)
}	18	(return	, $retVal$	, ,	)

Translating Procedure and Function Calls

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

Parameter Passing Techniques

- *By Value* - Formal parameter acts like a variable local to the routine that is initialized with the value of the actual parameter. 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 actual parameter (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 actual parameter. It acts like a hidden pointer that is automatically dereferrenced 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 actual parameter is *recalculated every time the parameter is used* in the routine. Primarily of historical interest (Algol 60).

Parameter Passing Issues

- How is the parameter passing technique determined?
  - *Caller's Choice* - In some languages (e.g. PL/I, Fortran) the form of the actual parameter determines the technique (e.g. value or reference) that is used to pass the parameter.
    Usually all parameters are passed by reference with compiler generated temporary locations used for value parameters.
  - *Callee's Choice* - In most modern lanugages (e.g. C, Pascal) the form of the formal parameter declaration determines the technique that must be used to pass the parameter.
- 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. Turing does this.

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

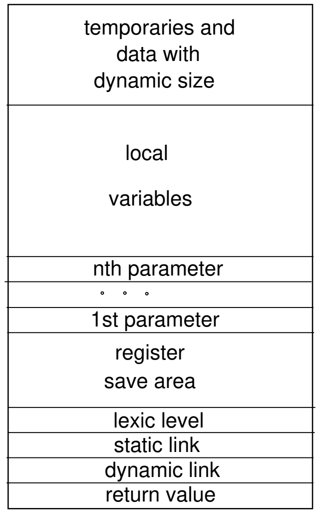
Prologues and Epilogues

- 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 routine's 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.

Returning

- In most implemetations the **return** statement is translated into a branch to the epilogue code which performs any cleanup that is required and then branches to the routine's 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 a pointer to this buffer is passed. The buffer must eventually be freed to avoid memory leaks.

Complete Activation Record



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 parameters 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.

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)) ;

  Need to build a weave of branches between the format item list and the data list.
- PL/I and Cobol have particularly ugly input/output constructs.

food for slide eater

food for slide eater