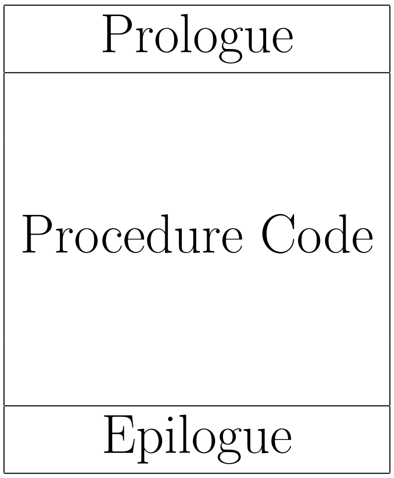
Compiled Procedure



Prologue: (or *preamble*) Save registers and return address; transfer parameters.

Epilogue: (or *postamble*) Restore registers; transfer returned value; return.

A `return` statement in a procedure is compiled to:

1. Load the returned value into a register.
2. `goto` the Epilogue.

Subroutine Call Is Expensive

The prologue and epilogue associated with each procedure are "overhead" that is necessary but does not do user computation.

- Even in scientific Fortran, procedure call overhead may account for 20% of execution time.
- Fancier languages have higher procedure call overhead.
- Relative overhead is higher for small procedures.
- Breaking a program into many small procedures increases execution time.
- A `GOTO` is much faster than a procedure call.
- Modern hardware architecture can help:
  - Parameter transfer
  - Stack addressing
  - Register file pointer moved with subroutine call

Activations and Control Stack

An *activation* is one execution of a procedure; its *lifetime* is the period during which the procedure is active, including time spent in its subroutines.

In a recursive language, information about procedure activations is kept on a *control stack*. An *activation record* or *stack frame* corresponds to each activation.

The sequence of procedure calls during execution of a program can be thought of as a tree. The execution of the program is the traversal of this tree, with the control stack holding information about the active branches from the currently executing procedure up to the root.

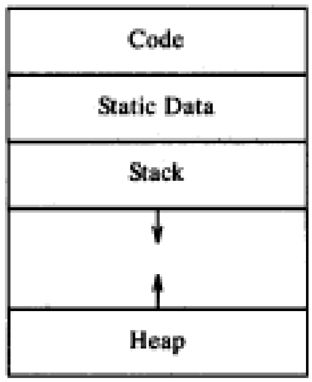
Environment

The *environment* of a procedure is the complete set of variables it can access; the *state* of the procedure is the set of values of these variables.

A *binding* is an association of a name with a storage location; we use the verb *bind* for the creation of a binding and say a variable is *bound* to a location. An environment provides a set of bindings for all variables.

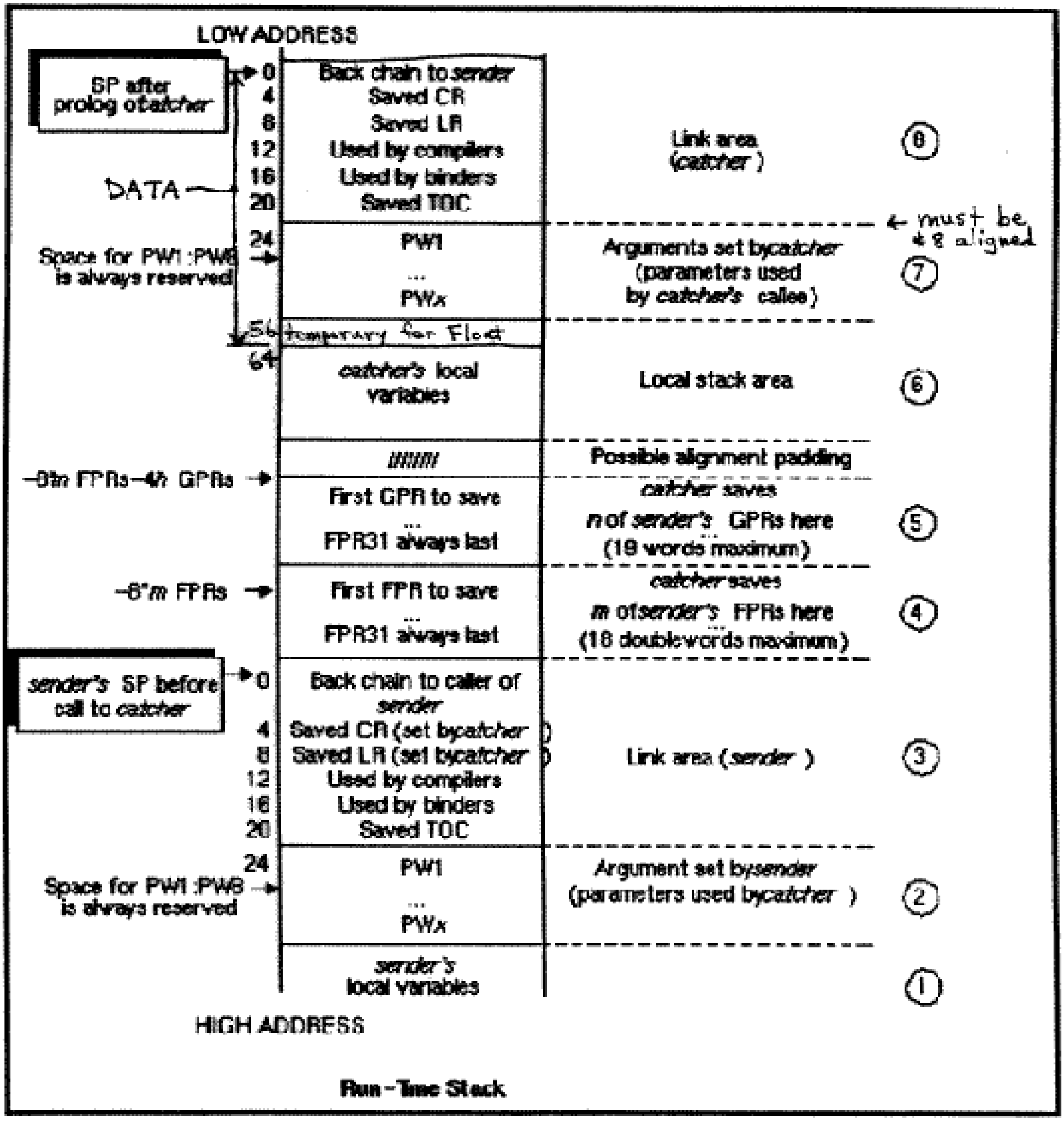
An assignment, e.g. `pi := 3.14` , changes the state of a procedure but not its environment.

Run-time Memory Organization



[Aho, Sethi, and Ullman, *Compilers*, Fig. 7.7.]

PowerPC Stack Frame Layout



Global Variable References

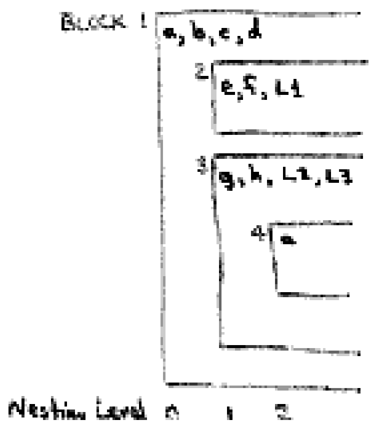
```
BEGIN REAL y, x ;
    PROCEDURE P(...)
        BEGIN
        :
        x := x+y ;
        :
        END
    :
    BEGIN REAL x, y ;
        :
      L1: P(...);
        :
    END
    :
    BEGIN INTEGER x, y ;
        :
      L2: P(...);
        :
    END
    :
END
```

The global variables referenced are those of the block in which the procedure appears in the source program (known at compile time).

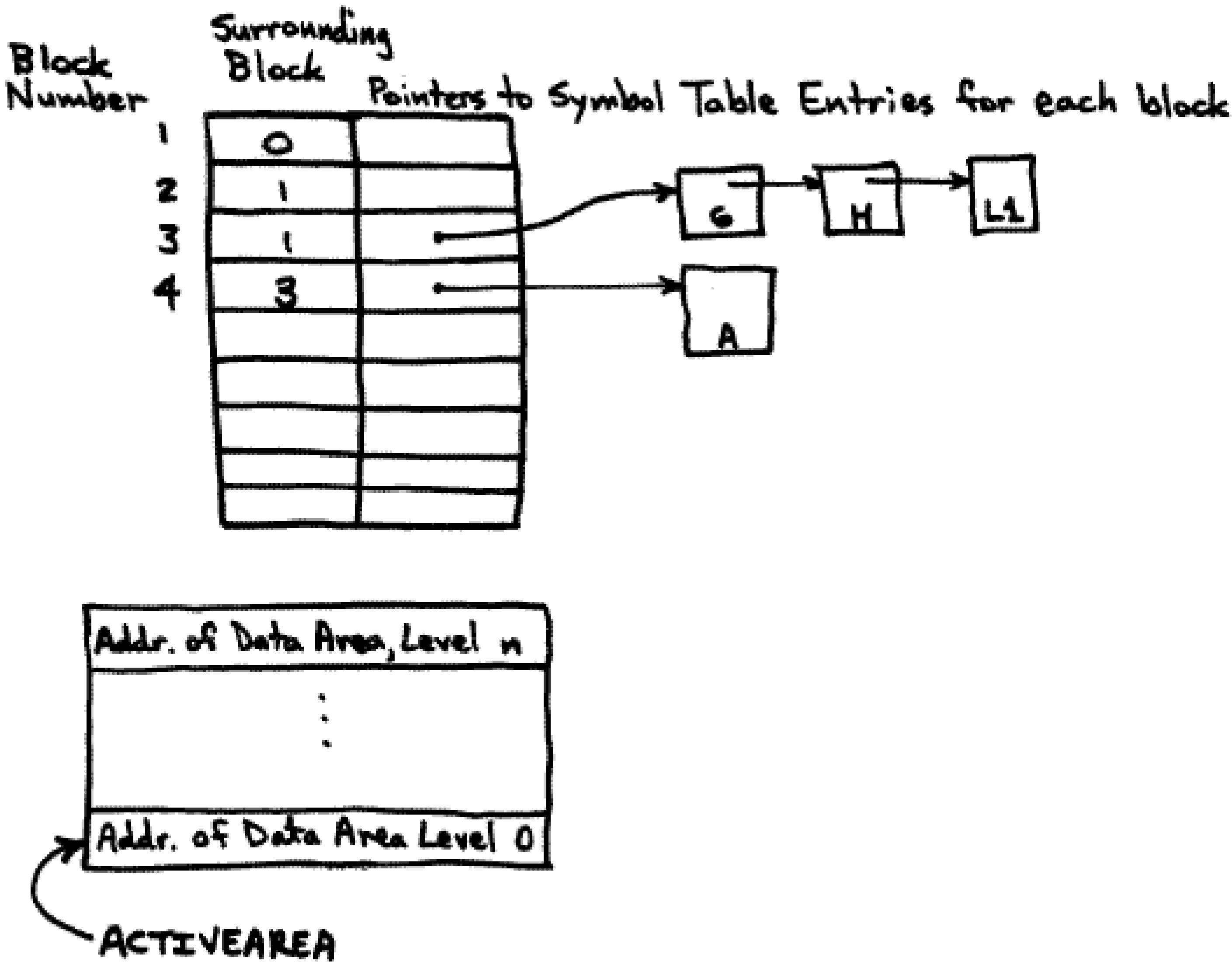
[Pratt Fig. 6.8]

Global Variables in Algol, Pascal, PL/I

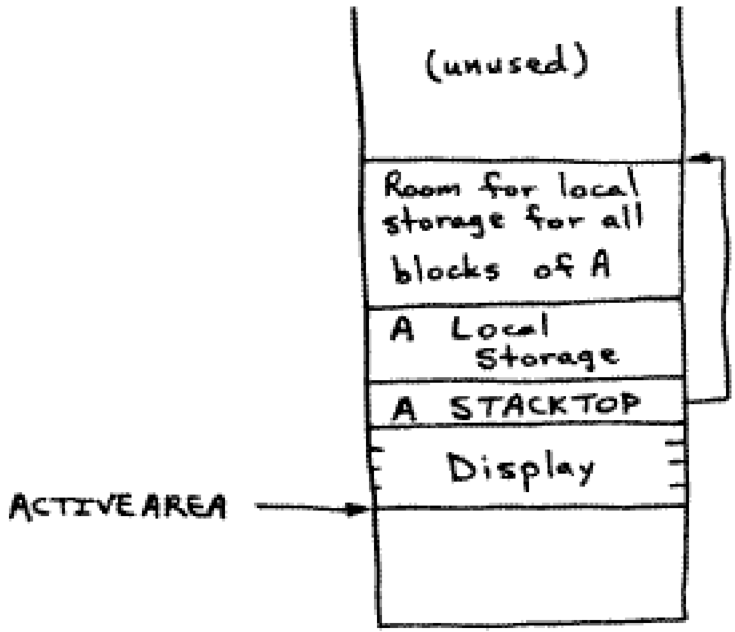
```
BEGIN  REAL a,b,c,d;
   :
   BEGIN REAL e,f;
      L1: ...
   END;
   BEGIN REAL g,h;
      :
      L2: BEGIN REAL a;
             :
          END;
      L3: ...
   END;
END;
```

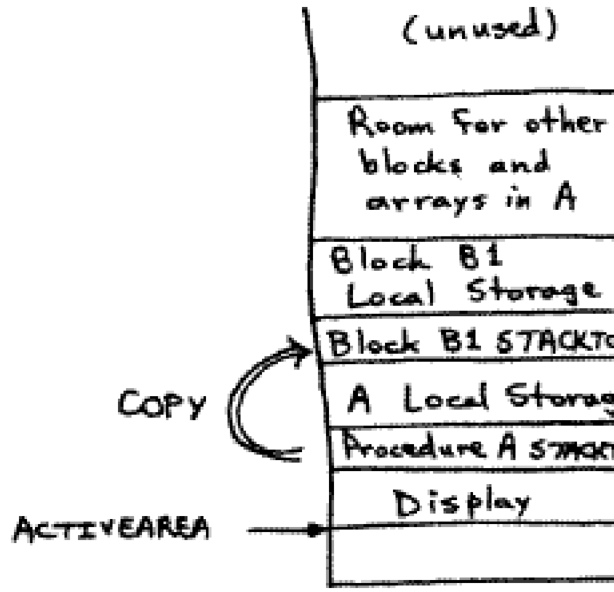


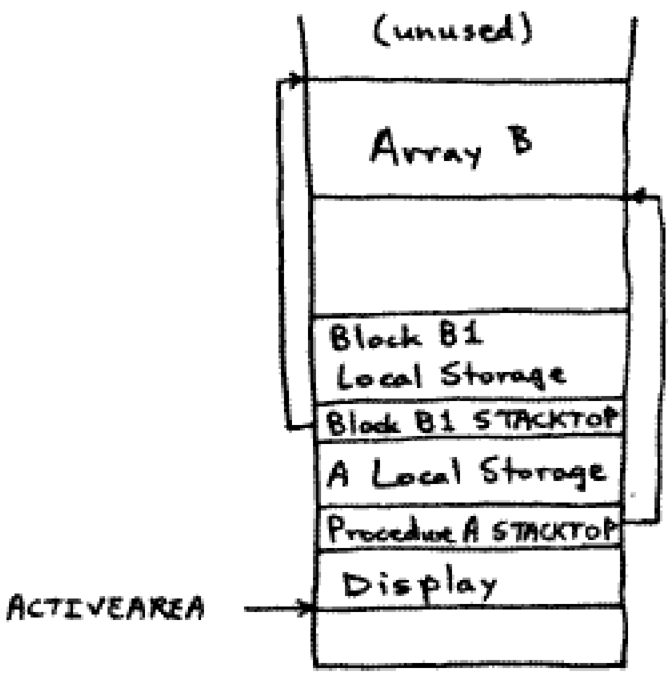
Block-structured Symbol Table



Run-time Stack for Algol







Code Generation

We assume that the input is error-free and complete, for example that any type conversion operators have already been inserted.[1]

Can generate:

- Binary
  - absolute
  - relocatable
- Assembly
- Interpreted code (e.g. Java byte codes)

Problems include:

- Instruction selection
- Register management
- Local optimization

[1]This slide was written by John Werth.

Code Generation

Code generation can be broken into several steps:

1. Generate the prologue
2. Generate the program code
3. Generate the epilogue

Subroutines are provided to generate the prologue and epilogue.

The arguments to the code generator are:

```
gencode(pcode, varsize, maxlabel)

   pcode    = pointer to code:
              (program foo (progn output)
                           (progn ...))
   varsize  = size of local storage in bytes
   maxlabel = max label number used so far
```

Code Generation

A starter program `codgen.c` is furnished. A very simple program, `triv.pas`, can be compiled by `codgen.c`:

```
program graph1(output);
var i:integer;
begin   i := 3   end.
```

The result is `triv.s`:

```
.globl graph1
   .type graph1, @function
graph1:
   ...
   subq $32, %rsp      # space for stack frame
# --------- begin Your code -------
   movl $3,%eax        #  3 -> %eax
   movl %eax,-32(%rbp) #  i := %eax
# --------- begin Epilogue code ---
   leave
   ret
```

Running Generated Code

Programs can be run using `driver.c` as the runtime library:

```
% cc driver.c triv.s -lm
% a.out
calling graph1
exit from graph1
```

`driver.c` is quite simple:

```
void main()
  { printf("calling graph1\n");
    graph1();
    printf("exit from graph1\n");
  }

void write(char str[])
  {  printf("%s", str);  }

void writeln(char str[])
  {  printf("%s\n", str);  }

int round(double x)
...
```

Code Generation for Statements

The function `genc(code)` generates code for a statement. There are only a few kinds of statements:

1. `PROGN`
   For each argument statement, generate code.
2. `:=`
   Generate the right-hand side into a register using `genarith`. Then store the register into the location specified by the left-hand side.
3. `GOTO`
   Generate a Branch to the label number.
4. `LABEL`
   Generate a Label with the label number.
5. `IF`
   `(IF c p1 p2)` can be compiled as:
   `IF c GOTO L1;`
   `p2; GOTO L2; L1:  p1; L2:`
   Optimizations are discussed later.
6. `FUNCALL`
   Compile short *intrinsic* functions in-line. For others, generate subroutine calls.

Arithmetic Expressions

Code for arithmetic expressions on a multi-register machine can be generated from trees using a simple recursive algorithm.

The specifications of the recursive algorithm are:

- **Input:** an arithmetic expression tree
- **Side Effect:** outputs instructions to the output file
- **Output:** returns the number of a register that contains the result.

Basic Expression Algorithm

The basic algorithm for expressions is easy:

- Operand (leaf node): get a register; generate a `load`; return the register.
- Operator (interior node): generate operand subtrees; generate op; return result register.

```
(defun genarith (x)
  (if (atom x)                  ; if leaf,
      (genload x (getreg))      ;   load
      (genop (op x)             ; else op
             (genarith (lhs x))
             (genarith (rhs x))) ) )
```

```
>(genarith '(* (+ a b) 3))
   LOAD  A,R1
   LOAD  B,R2
   ADD   R1,R2
   LOAD  3,R3
   MUL   R2,R3
R3
```

Trace of Expression Algorithm

```
>(genarith '(* (+ a b) 3))

  1> (GENARITH (* (+ A B) 3))
    2> (GENARITH (+ A B))
      3> (GENARITH A)
        4> (GENLOAD A R1)
   LOAD  A,R1
        <4 (GENLOAD R1)
      <3 (GENARITH R1)
      3> (GENARITH B)
        4> (GENLOAD B R2)
   LOAD  B,R2
        <4 (GENLOAD R2)
      <3 (GENARITH R2)
      3> (GENOP + R1 R2)
   ADD   R1,R2
      <3 (GENOP R2)
    <2 (GENARITH R2)
    2> (GENARITH 3)
      3> (GENLOAD 3 R3)
   LOAD  3,R3
      <3 (GENLOAD R3)
    <2 (GENARITH R3)
    2> (GENOP * R2 R3)
   MUL   R2,R3
    <2 (GENOP R3)
  <1 (GENARITH R3)
R3
```

Arithmetic Expression Algorithm

The `genarith` input is a tree (operand or operator):

- Operand (leaf node):
  1. Get a register.
  2. An operand may be a variable or constant:
     (a) Variable: Generate an instruction to load the variable into the register.
     (b) Constant:
        i. Small constant: Generate an immediate instruction to load it into the register directly.
        ii. Otherwise, make a *literal* for the value of the constant. Generate an instruction to load the literal into the register.
  3. Return the register number.
- Operator (interior node):
  1. Recursively generate code to put each operand into a register.
  2. Generate the operation on these registers, producing a result in one of the source registers.
  3. Mark the other source register unused.
  4. Return the result register number.

Register Management

Issues are:[2]

- register allocation: which variables will reside in registers?
- register assignment: which specific register will a variable be placed in?

Registers may be:

- general purpose (usually means integer)
- float
- special purpose (condition code, processor state)
- paired in various ways

[2] This slide was written by John Werth.

Simple Register Allocation

Note that there may be several classes of registers, *e.g.*, integer data registers, index registers, floating point registers.

A very simple register allocation algorithm is:

1. At the beginning of a statement, mark all registers as not used.
2. When a register is requested,
   (a) If there is an unused register, mark it used and return the register number.
   (b) Otherwise, punt.

On a machine with 8 or more registers, this algorithm will almost always work. However, we need to handle the case of running out of registers.

Heuristic for Expressions

The likelihood of running out of registers can be reduced by using a heuristic in generating code for expressions:

> Generate code for the *most complicated* operand first.

The "most complicated" operand can be found by determining the size of each subtree. However, simply generating code for a subtree that is an operation before a subtree that is a simple operand is usually sufficient.

With this simple heuristic, on a machine with 8 or more registers, the compiler will never[3] run out.

If a machine allows arithmetic instructions to be used with a full address, the operation may be combined with the last load.

---

[3] Well, hardly ever.

Improving Register Allocation

The simple register allocation algorithm can be improved in two ways:

- Handle the case of running out of available registers. This can be done by storing some register into a temporary variable in memory.
- Remember what is contained in registers and reuse it when appropriate. This can save some load instructions.

Register Allocation

Used	Use Number	Token

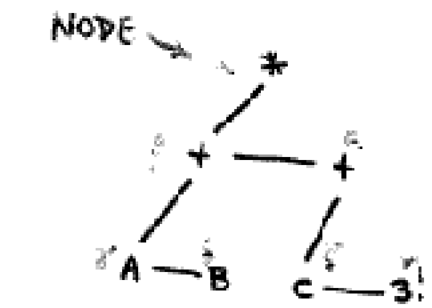
An improved register allocation algorithm, which handles the case of running out of registers, is:

1. At the beginning of a statement, mark all registers as not used; set use number to 0.
2. When an operand is loaded into a register, record a pointer to its token in the register table.
3. When a register is requested,
   (a) If there is an unused register: mark it used, set its use number to the current use number, increment the use number, and return the register number.
   (b) Otherwise, find the register with the smallest use number. Get a temporary data cell. Generate a Store instruction (*spill code*) to save the register contents into the temporary. Change the token to indicate the temporary.

Now, it will be necessary to test whether an operand is a temporary before doing an operation, and if so, to reload it. Note that temporaries must be part of the stack frame.

Example of Code Generation

Expression: (A+B)*(C+3.)



Initially:

NUSE = 0   NTEMP = 0

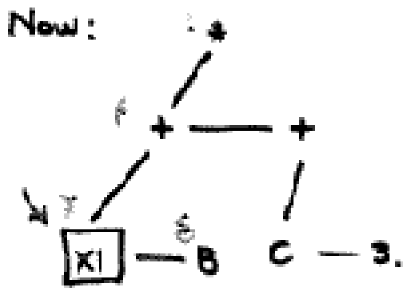
NODE = α

	Pointer	Used	Use No.
X1		0	
X2		0	
X6		0	

1. Operands of α not in registers. Push down to β.
2. Operands of β not in registers. Push down to γ.
3. γ is a variable. Get a register (X1)

   Generate   SA1 B0+A

Now:



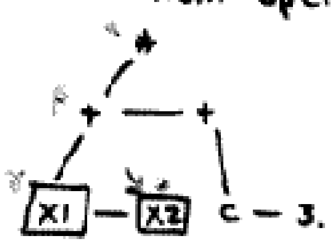
NUSE = 1

NODE = γ

	Pointer	Used	Use No.
X1	γ	1	1
X2		0	
X6		0	

4. Move to next operand (δ). Get a register (X2)

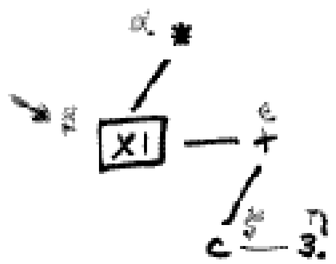
   Generate SA2 B0+B



	Pointer	Used	Use No.
X1	γ	1	1
X2	δ	1	2
X6		0	

Example (2)

5. No more operands; pop up to β.
   Operands of β are in Registers, so we can generate code.
   Assume use of first operand register for Result.



Generate RX1 X1+X2

Release Register X2

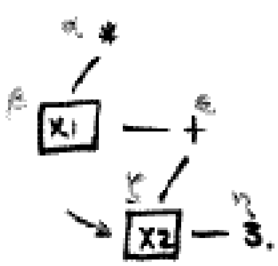
Release Nodes γ and δ

Change Node β

	Pointer	Used	Use No.
X1	β	1	3
X2		0	
X6		0	

6. Move across to next element at same level (ε).
   Its operands are not in registers, so push down to ζ.

7. ζ is a variable. Get a register (X2).
   Generate SA2 B0+C



	Pointer	Used	Use No.
X1	β	1	3
X2	ζ	1	4
X6		0	

8. Move across to element at same level (η).
   It is a constant. Look up in constant table; assume its offset in constant table is 4.
   Get a Register: No loadable register is free.
   Select register with smallest use number (X1).

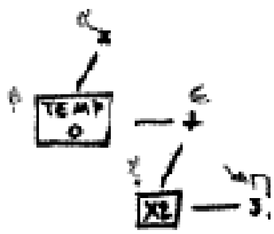
Example (3)

Generate a Store of X1: BX6 X1
SA6 B0+TS+0

(The 0 is the current value of NTEMP)

NTEMP = NTEMP +1

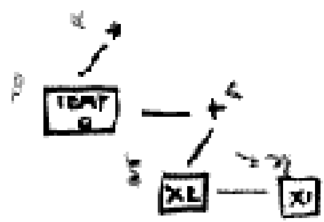
Change node β to a temporary node.



NEWREG = X1

Now we can load the constant 3. :

SA1 B0+CS+4



	Pointer	Used	Use No.
X1	η	1	5
X2	ξ	1	4
X6		0	

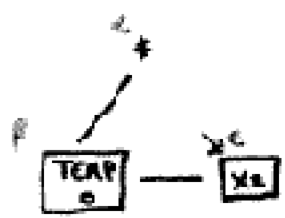
9. No more operands, pop up to ε.
Operands of ε are in registers, so we can generate code.
Assume first operand register for result.

Generate RX2 X2+X1

Release Register X1

Release Nodes ξ and η

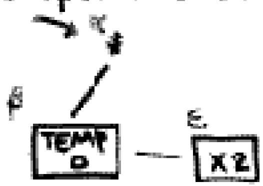
Change node ε to Register.



	Pointer	Used	Use No.
X1		0	
X2	ε	1	6
X6		0	

Example (4)

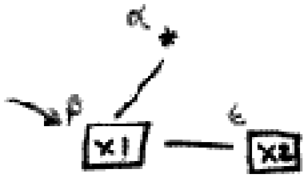
10. No more operands at this level. Pop up to Node α.



11. An operand of Node α is not a register. Push Down to β.

12. β is a Temporary. Get a Register (X1)

Generate Load: SA1 B0+T$+0

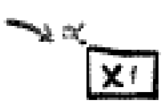


	Pointer	Used	Use No.
X1	β	1	7
X2	ε	1	6
X6		0	

13. Move to next node at same level (ε). It is a register, so no action is taken.

14. No more nodes at this level, so pop up to Node α.

15. Arguments of Node α are Registers, so we generate code.



Generate RX1 X1*X2
Release Register X2
Release Nodes β and ε
Change node α to Register.

	Pointer	Used	Use No.
X1	α	1	8
X2		0	
X6		0	

16. Top node is a register, so Stop; Result is in X1.

Reusing Register Contents

Used	Contents

Many instructions can be eliminated by reusing variable values that are already in registers:[4]

1. Initially, set the contents of each register to `NULL`.
2. When a simple variable is loaded, set the **`contents`** of the register to point to its symbol table entry.
3. When a register is requested, if possible choose an unused register that has no contents marked.
4. When a variable is to be loaded, if it is contained in an unused register, just mark the register used. This saves a Load instruction.
5. When a register is changed by an operation, set its contents to `NULL`.
6. When a value is stored into a variable, set the contents of any register whose contents is that variable to `NULL`. Then mark the register from which it was stored as containing that variable.
7. When a Label is encountered, set the contents of all registers to `NULL`.
8. The *condition code* contents can be reused also.

[4] We assume that there are no *aliases* for variables.

Register Targeting

On some machines, it is useful to be able to tell `genarith`, top-down, that its result should be produced in a certain register if possible.

Example: Suppose that a function argument should be transmitted in register `%xmm0`. If the argument can be generated in `%xmm0` directly, it will save a move instruction.

x86 Processor

We will assume a x86-64 processor. This processor has a vast number of instructions (some undocumented) and two major families of assembler syntax and calling sequence conventions. We will use the AT&T/Unix syntax and **gcc** calling conventions.

General-purpose (Integer) Registers:
32/64 bits, numbered `0 - 7` in `genasm`. We will use them in the order `%eax`, `%ecx`, `%edx`, `%ebx` since `%ebx` is callee-saved. `RBASE` to `RMAX` is the local integer register range.

Floating Point Registers:
64 bits, numbered `8 - 15` in `genasm`. `FBASE` to `FMAX` is the floating register range. These are called `%xmm0` through `%xmm7`.

Move (Load/Store) Instructions

Most of the instructions used in a computer program are instructions that move data. The x86 processor uses variable-length instructions and offers very flexible addressing options.

The Unix syntax of x86 instructions shows data movement from left to right:

```
movl   $0,%eax          #  0 -> %eax
movl   %eax,-32(%rbp)   #  %eax -> i
```

There are three data formats that we will use:

Instruction	**Terminology**	**Bits**	**Use For**
`MOVL`	long	32	Integer
`MOVQ`	quad-word	64	Pointer
`MOVSD`	signed double	64	Float

Kinds of Move Addressing

There are several addressing styles that are used with move instructions:

Constants or *immediate* values are specifies with a `$`. x86 allows even very large integer constants.

```
movl   $0,%eax            #  0 -> %eax
```

Stack Variables have negative offsets relative to `%rbp`. The offset is the offset from the symbol table minus the stack frame size.

```
movl   %eax,-32(%rbp)  #  %eax -> i
```

In this case, `i` has an offset of 16 and the stack frame size is 48.

Literals have offsets relative to `%rip`.

```
movsd   .LC5(%rip),%xmm0  #  0.0625 -> %xmm0
```

Record References have offsets relative to a register containing a pointer to the record.

```
movl   %eax,32(%rcx)  #  ^. []
```

Move with Calculated Address

x86 allows very flexible addressing:

Offset from Register

```
movl   %eax,-32(%rbp)  #  %eax -> i
```

Offset from Two Registers

```
movsd %xmm0,-1296(%rbp,%rax) #  ac[]
```

The offset and contents of the two registers are added to form the effective address.

Offset from Two Registers with Multiplier

```
movsd %xmm0,-1296(%rbp,%rax,8) #  x[]
```

In this case, the second register is multiplied by 2, 4, or 8 before being added. This can allow many aref expressions to be done in a single instruction.

Literals

A *literal* is constant data that is assembled as part of the compiled program. Literals must be made for large integers, all floats, and most strings.

There are three programs that make literals; each is called with a literal value and a label number:

- `makeilit(i,label)` : integer (not needed for x86)
- `makeflit(i,label)` : float
- `makeblit(i,label)` : byte (string)

A literal is accessed relative to the Instruction Pointer:

```
movsd   .LC4(%rip),%xmm1
```

Literals are saved in tables and output at the end of the program.

```
        .align  8
.LC4:
        .long   0
        .long   1078001664
```

Integer Arithmetic Instructions

These instructions operate on registers or memory. `S,D` represent source and destination.

`addl`	`S,D`	$D + S \rightarrow D$
`subl`	`S,D`	$D - S \rightarrow D$
`imull`	`S,D`	$D * S \rightarrow D$
`ldiv`	`S,D`	$D/S \rightarrow D$
`cmpl`	`S,D`	$compare\ D - S, set\ condition$
`andl`	`S,D`	$D \wedge S \rightarrow D$
`orl`	`S,D`	$D \vee S \rightarrow D$
`notl`	`D`	$\neg D \rightarrow D$
`negl`	`D`	$-D \rightarrow D$

Note that arithmetic can be done directly on memory: `i := i + 1` can be one instruction:

```
addl $1, -4(%rbp)
```

Compare and Jump

A *compare* is a subtract that does not store its results; however, the results set the *condition code*, which can be tested by jump instructions.

`cmpl`	`S,D`	$compare\ D - S, set\ condition$, integer
`cmpsd`	`S,D`	$compare\ D - S, set\ condition$, float

The jump instructions test the condition code:

`jmp`	Jump always.
`jle`	Jump if $D \leq S$
`je`	Jump if $D = S$
`jne`	Jump if $D \neq S$
`jge`	Jump if $D \geq S$
`jl`	Jump if $D < S$
`jg`	Jump if $D > S$

Floating Point

These instructions operate on registers or memory. `S,D` represent source and destination.

`addsd`	`S,D`	$D + S \rightarrow D$
`subsd`	`S,D`	$D - S \rightarrow D$
`mulsd`	`S,D`	$D * S \rightarrow D$
`divsd`	`S,D`	$D/S \rightarrow D$
`cmpsd`	`S,D`	$compare\ D - S, set\ condition$

Routine are provided to generate the instruction sequences for `fix`, `float` and `negate` operations.

Intrinsic Functions

Some things that are specified as functions in source code should be compiled in-line. These include:

1. Type-change functions that do no computation: `boole`, `ord`, `chr`.
2. Functions that are only a few instructions: `pred`, `succ`, `abs`.
3. Functions that are implemented in hardware: `sqrt` may be an instruction.

Function Calls

For external functions, it is necessary to:

1. Set up the arguments for the function call.
2. Call the function.
3. Retrieve the result and do any necessary final actions.

A function call involves the following:

1. Load arguments into registers:
   - For string literals, address in `%edi`:

     ```
     movl   $.LC12,%edi  #  addr of literal .LC1
     ```

   - For floating arguments, `%xmm0`
2. Execute a `call` instruction:

   ```
   call   sin
   ```

3. Floating results are returned in `%xmm0`. Integer results are returned in `%eax`.

Volatile Registers

By convention, some registers are designated volatile or caller-saved, i.e. destroyed by a subroutine call. Other registers are designated non-volatile or callee-saved and must be preserved (or not used) by a subroutine.

We will try to use only the registers `%eax`, `%ecx`, and `%edx`, since `%ebx` is callee-saved.

Any floating values that need to be preserved across a call must be saved on the stack prior to the call and restored afterwards. Routines are provided to save one floating register on the stack and restore it.

Details of Function Call

1. For each argument, use `genarith` to compute the argument. If needed, move the result from the register returned by `genarith` to `%xmm0` and mark the `genarith` register unused.
2. For each volatile register that is in use, save it
3. Call the function
4. For each volatile register that is in use, restore it
5. Return the function result register (`%xmm0` or `%eax`) as the result of `genarith`.

IF Statement Generation

Code for an intermediate code statement of the form `(if c p1 p2)` can be generated as follows:

1. Generate code for the condition `c` using the arithmetic expression code generator. Note that a `cmp` instruction should be generated for all comparison operators, regardless of which comparison is used.
2. Generate the appropriate jump-on-condition instruction, denoted `jmp c` below, by table lookup depending on the comparison operator.

```
    jmp c    .L1
    p2                  ! "else"
    jmp      .L2
.L1:
    p1                  ! "then"
.L2:
```

The following jump table can be used:

op	$=$	$\neq$	$<$	$\leq$	$\geq$	$>$
c	je	jne	jl	jle	jge	jg
-c	jne	je	jge	jg	jl	jle

IF Statement Optimization

Special cases of `IF` statements are common; these can be compiled as shown below, where `jmp c` represents a jump on condition and `jmp -c` represents a jump on the opposite of a condition.

```
(if c (goto l))              jmp c   l

(if c (progn) (goto l))      jmp -c  l

(if c p1 (goto l))           jmp -c  l
                             p1

(if c (goto l) p2)           jmp c   l
                             p2

(if c p1)                    jmp -c  L1
                             p1
                         L1:

(if c (progn) p2)            jmp c   L1
                             p2
                         L1:
```

Array References

Suppose the following declarations have been made:

`var i: integer; x: array[1..100] of real;`

This would give `i` an offset of `4` and `x` an offset of `8` (the initial offset is `4`; since `x` is `double`, its offset must be 8-aligned.). The total storage is `808`. A reference `x[i]` would generate the code:

```
(AREF X (+ -8 (* 8 I)))
```

The effective address is: `%rbp`, minus stack frame size, plus the offset of `x`, plus the expression `(+ -8 (* 8 I))`.

Easy Array References

```
(AREF X (+ -8 (* 8 I)))
```

One way to generate code for the array reference is to:

- use **genarith** to generate `(+ -8 (* 8 I))` in register (`%eax`) (move the result to `%eax` if necessary).
- Issue the instruction `CLTQ`, which sign-extends `%eax` to `%rax`.
- access memory from the offset and sum of the registers.

```
movsd   %xmm0,-1296(%rbp,%rax)  #  ac[]
```

This is easy from the viewpoint of the compiler writer, but it generates many instructions, including a possibly expensive multiply.

Better Array References

```
(AREF X (+ -8 (* 8 I)))
```

A better way generate the array reference is to:

1. combine as many constants as possible
2. replace the multiply by a shift

Note that in the expression `(+ -8 (* 8 I))` there is an additive constant of `-8` and that the multiply by `8` can be done in the x86 processor by a shift of 3 bits.

This form of code is ony one instructions on x86:

```
movsd  %xmm0,-208(%rbp,%rax,8)
```

Pointer References

A pointer operator specifies indirect addressing. For example, in the test program, the code `john^.favorite` produces the intermediate code:

```
(aref (^ john) 32)
```

Note that a pointer operator can occur *only* as the first operand of an `aref`, and the offset is usually a constant. Compiling code for it is simple: the address is the sum of the pointer value and the offset:

```
movq -1016(%rbp),%rcx  #  john -> %rcx
movl %eax,32(%rcx)     #  ^. []
```

switch Statement

```
int vowel(ch)
  int ch;
  {  int sw;
     switch ( ch )
       {  case 'A':  case 'E':  case 'I':
          case 'O':  case 'U':  case 'Y':
            sw = 1; break;
          default: sw = 0; break;
          }
     return (sw);
   }
```

switch Statement Compiled

```
vowel:
        save    %sp,-104,%sp
        st      %i0,[%fp+68]
.L14:
        ba      .L16
        nop
.L17:
.L18:
.L19:
.L20:
.L21:
.L22:
        mov     1,%o0
        ba      .L15
        st      %o0,[%fp-8]
.L23:           !    default: sw = 0; break;
        ba      .L15
        st      %g0,[%fp-8]
.L16:
        ld      [%fp+68],%o0
        cmp     %o0,79
        bge     .L_y0
        nop
        cmp     %o0,69
        bge     .L_y1
        nop
        cmp     %o0,65
        be      .L17
        nop
        ba      .L23
        nop
.L_y1:
        be      .L18
        nop
        ... 20 more instructions
.L24:
.L15:
        ld      [%fp-8],%i0
        jmp     %i7+8
        restore
```

`switch` Statement Compiled `-O`

```
[ ... big table constructed by the compiler ... ]
vowel:
        sub     %o0,65,%g1
        cmp     %g1,24
        bgu     .L77000008
        sethi   %hi(.L_const_seg_900000102),%g2
.L900000107:
        sll     %g1,2,%g1
        add     %g2,%lo(.L_const_seg_900000102),%g2
        ld      [%g1+%g2],%g1
        jmpl    %g1+%g2,%g0
        nop
.L77000007:
        or      %g0,1,%g1
        retl    ! Result =  %o0
        or      %g0,%g1,%o0
.L77000008:
        or      %g0,0,%g1
        retl    ! Result =  %o0
        or      %g0,%g1,%o0
```

Table Lookup

```
static int vowels[]
   = {1,0,0,0,1,0,0,0,1,0,0,0,0,
      0,1,0,0,0,0,0,1,0,0,0,1,0};

int vowel(ch)
  int ch;
  {  int sw;
     sw = vowels[ch - 'A'];
     return (sw);
   }
```

Table Lookup Compiled

```
vowel:
        save    %sp,-104,%sp
        st      %i0,[%fp+68]
.L15:
        ld      [%fp+68],%o0
        sll     %o0,2,%o1
        sethi   %hi(vowels-260),%o0
        or      %o0,%lo(vowels-260),%o0
        ld      [%o1+%o0],%i0
        st      %i0,[%fp-8]

        jmp     %i7+8
        restore
```

Table Lookup Compiled -O

```
vowel:
        sll     %o0,2,%g1
        sethi   %hi(vowels-260),%g2
        add     %g2,%lo(vowels-260),%g2
        retl                       ! Result =  %o0
        ld      [%g1+%g2],%o0      ! volatile
```

Bottom Line:

`switch`	46
`switch -O`	15
Table Lookup	10
Table Lookup `-O`	5

Table Lookup beats the `switch` statement in code size and performance; it is also better Software Engineering.