Intermediate Code Generation

Big Picture

Generation of intermediate code (ICode) is used to simplify the conversion from source code (kxi) to target code (assembly). ICode generation will accomplish this by focusing on converting high-level language statements into low-level language statements without addressing memory allocation issues.

How will this work?

If you have implemented your Symbol Table as I have requested it contains every Symbol needed by a kxi program whether defined in the kxi source code or temporary as created when evaluating expressions such as “a+b*c/d”.

Now just create ICode which will look like assembly statements for each high-level kxi statement which reference Symbols in your Symbol Table via their Symbol Id.

Quad

A Quad is an assembly-like instruction with one operator and up to three operands. Our intermediate code will combine assembly style operators with high-level language type operands.

Back Patching

Back patching is a technique used in Compiler construction to fix already generated code. A simple example of back patching, which will be shown below, is to replace label “xxx” with label “yyy” in all instructs where the string “xxx” appears in the code. Given code is generated from top to bottom, start at the current Quad and work our way backwards to the first Quad generated to ensure that Back Patching is complete. It has been found simplest to implement this strategy, such that each time a label is to replace an existing label to back patch the original label with the newly generated label.

- Note: Make sure you back patch *each and every time* a label is replaced on a line to ensure that all labels are correctly updated in the code. You are required to use back patching in your compiler, when multiple labels are to be produced for a single instruction.
- Note: While it is your options to use this specific Intermediate Code, Back Patching is not Optional. Do NOT create or use

some type of solution that avoids back patching where it is needed.

Identifiers - Address vs. Value & RHS vs. LHS

The following is a Target Code issue, but it can be confusing during Intermediate Code if you focus too much on Target Code issues. So lets look at the difference between an identifier value vs. identifier address now to try an avoid some confusion later.

As a general rule each identifier will be resolved down to the address of the identifier (i.e., not done until Target Code Generation). Then based upon the operation being performed (e.g., Intermediate Code instruction) you will either use the address or access the value at the address.

Example: x + y, you need the address of both x and y, then because this is an add operation you need the value at the address of both x and y.

Example: x = y, you need the address of both x and y, then because this is an assignment operation you only need the address of the LHS x and the *value of the RHS y*.

LHS = RHS

LHS	*Compute the address of the LHS*
RHS	*Compute the address of the RHS then access the value at that location*
=	*Assign the value from the RHS to the address of the LHS*

Indirect Addressing Mode

Indirect addressing is the most complex single concept in iCode generation. After using various alternatives the best solution I and my students have found is *to mark variables created for indirect addressing* in the symbol table but to then otherwise ignore it for now. This had made iCode generation very simple and multiple students have claimed to have finished it in a single week and a few even claimed to have finished in a single day.

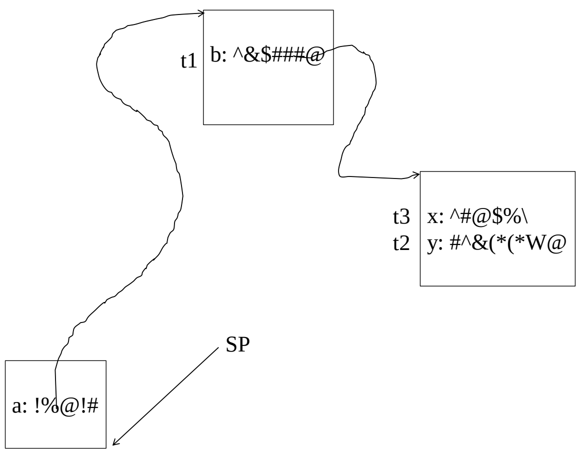
Location	LHS	Location	RHS
		Global	‘r’
		Global	123
		Global	true
Heap	this.a	Heap	this.b
Heap	c.a	Heap	c.b
Heap	a[i]	Heap	b[i]
Heap	c.a[i]	Heap	c.b[i]
Stack	a	Stack	b

Notice that every variable that can be defined in kxi can be thought of as being accessed via indirect address including a Global (Base Address + 0). Using this concept we must always compute the address of a variable before accessing its value (as described above, LHS needs only an address the RHS needs both address and value).

However, there are special cases that need to be addressed in kxi, *consider the following* statements:

```
x = y;
x = a.b;
a.b = y;
a.b.c = a.b.c;
```

x = y;	The statement should be thought of as the follow alternative statement. MOV y, x Where compute the address will means find the base address and any offset and add them together to form an address. The MOV statement means compute the address of x then compute the address of y. Next find the value at the address of y and assign it to the address of x. Note even if x and y are object references this meaning does not change.
x = a.b;	This statement needs to be considered as multiple statements. REF a, b, t1 MOV t1, x The REF statement means compute the address of a then compute the offset to b relative to a. Next add the address and the offset together creating an address for 'a.b' and assign this value to t1. The *variable t1* which was *created from a REF* is very *different from x* because we must *always use it in indirect addressing*. When t1 is on the RHS of an assignment as in this example. MOV t1, a means access x as normal but for t1 *compute address of t1 then access the value at this address next use the value as an indirect address* to find the actual value. An alternative syntax which more clearly indicates this is: MOV (t1), x However because t1 was created from a REF and will be stored in the symbol table we can indicated it is an indirect memory access variable and always use indirect memory when getting its value for an assignment.
a.b = y;	This statement needs to be considered as multiple statements. REF a, b, t1 MOV y, t1 When t1 is on the LHS of an assignment as in this example we must compute the address of t1 just as before but the *value is NOT used for indirectly access memory* as before. The address computed is where the value of y is to be assigned.

a.b.x = a.b.y;	This is multiple statements as well. REF a, b, t1 REF t1, y, t2 REF t1, x, t3 MOV t2, t3 The REF statement computes addresses which are stored in t1, t2, t3. When t1 is used in a REF as in 'REF t1, y, t2' indirect memory access is not necessary but again we need to use t1 differently than a in the example 'REF a, b, t1'. Because t1 was created via a REF we need the value at the address of t1 not the base address of t1. Once again *checking the symbol table can tell us this fact even more clearly than coming up with some alternative syntax* such as.  t1 b: ^&$###@ t3 t2 x: ^#@$%\ y: #^&(*(*W@ SP a: !%@!# REF a, b, t1 REF (t1), y, t2 REF (t1), x, t3 MOV (t2), t3

Using alternative syntax may seem to make the intermediate code more readable but for whom. Your compiler will use the intermediate code not a person and the symbol table has to be used to indicate when an indirect memory access variable has been created just to create the 'REF (t1), y, t2' style statements. Thus you will always need info in the symbol table but an alternative syntax is unnecessary.

Even stack variable are really a REF.
Stack variable x.   REF FP, x, t1

Note: Something as simple as 'r1' vs. 't1' as the name of your temporary variable can be enough to indicate the variable was created from a reference for indirect memory addressing.

Intermediate Instruction

If you find it helpful create additional Intermediate Code instructions to those provided. Remember the primary purpose of intermediate code is to create an assembly like language that bridges the gap between actual assembly code and high-level language instruction.

Note: While you can add new Intermediate Code Instructions, the Target Code instructions you are allowed to use are fixed and very small. ONLY use those instructions which were provided to you at the beginning of the Semester!!!! Projects with illegal target code instructions will NOT BE GRADED!

Intermediate Code vs. Assembly Code

The main simplification intermediate code will have over actual assembly code is in the area of memory allocation.   Our intermediate code will reference variables, literals, and other structures *defined in the Symbol Table via their Symbol ID*, where as actual assembly code must allocate and access memory locations.

Mathematical Instructions

```
ADD  A, B, C   ; A + B → C
ADI  A, #, C   ; A + # → C
SUB  A, B, C   ; A – B → C
MUL  A, B, C   ; A * B → C
DIV  A, B, C   ; A / B → C
```

Boolean Instructions

```
LT   A, B, C   ; A < B → C
GT   A, B, C   ; A > B → C
NE   A, B, C   ; A != B → C
EQ   A, B, C   ; A == B → C
LE   A, B, C   ; A <= B → C
GE   A, B, C   ; A >= B → C
```

Logical Instructions

```
AND  A, B, C   ; A && B → C
OR   A, B, C   ; A || B → C
```

Control Flow Instructions

```
BF   A, LABEL  ; Branch to LABEL if A is FALSE
BT   A, LABEL  ; Branch to LABEL if A is TRUE
JMP  LABEL     ; Jump to LABEL
```

Run-Time Stack Instructions

```
PUSH A         ; Push A on to the top of the Stack
POP  A         ; Pop the top of the Stack into A
PEEK A         ; Get the top of the Stack into A
               ; without adjusting the top of Stack
```

Function Invocation Instructions

```
FRAME F, X     ; Create a frame for function F
               ; Pass X as the instance object (this)
CALL F         ; Invoke function F
RTN            ; Use the Return Address in the current
               ; activation record to return from a
               ; function call
RETURN A       ; use the Return Address in the current
               ; Activation record to return from a
               ; function call while also return the
               ; value of A
FUNC F         ; Handle any initialization for
               ; function F, such as allocating space
               ; for any temporary and local variables
```

Memory Allocation Instructions

```
NEWI #, A      ; Allocate # bytes on the heap and
               ; place the starting address in A
NEW R, A       ; Allocate the number of bytes in
               ; variable R on the heap and
               ; place the starting address in A
```

Miscellaneous (Other) Instructions

```
MOV  A, B       ; A → B
MOVI #, A       ; # → A
WRITE A         ; Write the value of A to the Screen
READ A          ; Read from the screen into A
```

You might want to create specific read and write instructions for each data type you can read and write, rather than having to look in the system table at A to determine its type.

```
WRTC A          ; Write Char A
WRTI A          ; Write Int A
RDC  A          ; Read Char into A
RDI  A          ; Read Int into A

REF A, B, C     ; Variable C becomes an alias
                ; for the reference A.B
                ;
                ; Variable C is a
                ; temp variable and should be
                ; stored in the Symbol Table.
                ; You will find it helpful in
                ; Target Code Generation if you
                ; will also indicate in some way
                ; this is a ref variable!
                ;
                ; Using A as a Base Address add to it
                ; the offset to B and assign the
                ; address to C.

AEF A, B, C     ; Very similar to REF command only
                ; used for accessing an array.
                ; Variable C is a temp variable a
                ; should be stored in the Symbol Table.
                ; You should indicate that temp
                ; variable C was created via an AEF.
                ;
                ; A is the base address of the array
                ; and B is an index into the array A.
                ; A[B] thus C is the address computed
                ; by A + [B * sizeof(type(A))]
                ; Using this icode macro has proven to
                ; be more simple than coding the
                 Computation directly.
```

Example Code Conversions

What follows are examples of kxi code and how they can be transformed into intermediate code. As you look at the longer examples don't focus on the complexity of the entire example rather focus on how the individual expressions and statements are converted the same way each time.

Note: As you look at the more complex examples in this section also look back at the Semantic Action Slides you have been provide. Intermediate Code is created for high-level statements immediately after the Semantic Routines are executed in your Compiler.

Note: Remember this problem is a recursive problem, I have asked you to make your Compiler recursive. The solution that will best solve this problem will mesh recursively with your existing Recursive Descent Parser and the Semantic Routines that have been added to your Compiler.

Note: The infix to postfix conversion determines the order in which the operands must be executed.

Expressions

x + y;

```
ADD  x, y, t5  ; x + y → t5
```

Note: t5 is a temporary variable created during Semantic Analysis to hold the result of x + y.

x + y * z;

```
MUL  y, z, t3  ; y * z → t3
ADD  x, t3, t4 ; x + t3 → t4
```

Note: Infix to Postfix conversion using operator precedence indicates that multiplication occurs before addition.

x < y;

```
LT   x, y, t2  ; x < y → t2
```

(x < y) && (z != 3);

```
LT   x, y, t2      ; x < y → t2
NE   z, c1, t4     ; z != c1 → t4, c1=3
AND  t2, t4, t5    ; t2 && t4 → t5
```

Assignment Statements

Note: Don't worry if the sequences of Intermediate Code statements are not optimal. Creating optimal code is an Optimization, and would be handled in Intermediate Code Optimization NOT DURING Intermediate Code Generation.

x = y;

```
MOV  x, y ; x → y
```

x = x + y;

```
ADD  x, y, t5  ; x + y → t5
MOV t5, x      ; t5 → x
```

x = y * g + f / k ;

```
MUL y, g, t1   ; y * g → t1
DIV f, k, t2   ; f / k → t2
ADD t1, t2, t3 ; t1 + t2 → t3
MOV t3, x      ; t3 → x
```

Note: Infix to postfix conversion of operators during Semantic Analysis has correctly ordered the Intermediate Code instruction in the order of their precedence.

a . x = b . y ;

```
        REF a, x, t23
        REF b, y, t24
        MOV t23, t24            ; t23 → t24
                                ; t23 and t24 point to
                                ; something (a location) on
                                ; the heap
```

Note: This assignment statement seems more complex that "x = y", but when handled correctly it is not that much more complex. First, focus on the fact that the semantic action #rExist will determine if the reference "a.x" and "b.y" exist and replace them with a single temporary variable. Thus in the end, this is really just an assignment of two temporary variables, t23 and t24. The main difference is that an indirect assignment will be necessary as both t23 and t24 contain an address that points at the data not the actual data. But that will be handled later in target code.

Note: The actual meaning (implementation) of REF does not have to be set yet, this is a Target Code detail. However, to ease your mind consider the following:

"a" must contain a pointer to an object on the heap.

"x" must be an element of "a", thus "x" is simply an offset from the start of "a".

REF will need to add the offset of "x" to the base address of "a", and place the result into temporary variable t23 (Effective Address = Base Address + Offset).

a.k.x = b.g.y;

Note: As before the assignment statement operates on only two identifiers and as before all references are resolved using the REF instruction.

```
        REF a, k, t25           ; a + offset(k) → t25
        REF t25, x, t26         ; t25 + offset(x) → t26
        REF b, g, t27           ; b + offset(g) → t27
        REF t27, y, t28         ; t27 + offset(y) → t28
        MOV t28, t26            ; t28 → t26
```

Control Structure Statements

Note: When creating Labels for Control Structures, the actual Labels can be a simple string followed by a unique number, to make each unique (e.g., L1, L2, L3). In the following examples I will attempt to name the labels something more meaningful than L1, L2, etc.

if (x < y) { bla1} bla2

```
          LT x, y, t1    ; x<y → t1
          BF t1, SKIPIF  ; BranchFalse t1, SKIPIF
          bla1
SKIPIF:   bla2
```

if (x == y) { bla1 } else { bla2 } bla3

```
          EQ x,y, t1     ; x == y → t1
          BF t1, SKIPIF  ; BranchFalse t1, SKIPIF
          bla1           ; IF Code
          JMP SKIPELSE   ; Generate as Part of ELSE
SKIPIF:   bla2           ; ELSE Code
SKIPELSE: bla3
```

while (x < y) { bla1; bla2; bla3; } bla4

```
BEGIN:    LT x, y, t1         ; x<y → t1
          BF t1, ENDWHILE     ; BranchFalse t1, ENDWHILE
          bla1                ; While Code
          bla2                ; While Code
          bla3                ; While Code
          JMP BEGIN
ENDWHILE: bla4
```

if (x < y) { bla1 } else if (z != 3) { bla2 } bla3

Note: The use of nested if-else-if-statements will require Back patching.

Back patching has been used to correct the problem where labels SKIPELSE1 and SKIPIF2 were to be placed on the same Quad (bla3).

- *bla3 is the destination*
    - *after bla1is executed (SKIPELSE1)*
    - *if (z != 3) is false (SKIPIF2)*

- *This will mean every time you place a label on a Quad, you should first check if an existing label already exists on the Quad. If one does, replace it when the newer label and back patch the code.*

When the label SKIPIF2 replaces label SKIPELSE1 on the Quad, the Compiler must back patch all previous instructions to replace "SKIPELSE1" with "SKIPIF2".

- *To back patch, works backward from the current instruction to the first and replace "SKIPELSE1" with "SKIPIF2" any place that it is found.*

```
          LT x, y, t1     ; x<y → t1
          BF t1, SKIPIF1
          bla1
          JMP SKIPIF2     ; Label is back patched
SKIPIF1:  NE z, c1, t2    ; z!=c1 → t2
          BF t, SKIPIF2
          bla2
SKIPIF2:  bla3            ; Two possible labels here
                          ; SKIPF2 or SKIPELSE1
```

while (x < y) { bla1; bla2; bla3; } while (z != 3) { bla4 } bla5

Note: The use of repeated WHILE-statements will require back patching to set the labels correctly. The statement "NE z, c1, t7" could have two possible labels, "BEGIN2" or "ENDWHILE1". Since BEGIN2 was created last we use that label and back patch ENDWHILE1 everywhere that it appears.

- *while(z != 3) is the destination*
  - *if (x <y) is false (ENDWHILE1)*
  - *after bla4 is executed (BEGIN2)*

```
BEGIN1:    LT x, y, t1          ; x<y → t1
           BF t1, BEGIN2        ; Label is Back Patched
           bla1                 ; While Code
           bla2                 ; While Code
           bla3                 ; While Code
           JMP BEGIN1
BEGIN2:    NE z, c1, t7         ; z!=c1 → t7 back patch label
           BF t7, ENDWHILE2
           Bla4
           JMP BEGIN2
ENDWHILE2:bla5
```

Other Statements

cout << a;

```
WRITE a
```

Note: Type information can be imbedded into the Quad for additional clarity. This is another alternative to making new operators.
1 → Integer
2 → Char

```
WRITE 1, a
WRITE 2, a
```

cin >> a;

```
READ a
```

Note: Type information can be imbedded into the Quad for additional clarity.
1 → Integer
2 → Char

```
READ 1, a
READ 2, a
```

return;

```
RTN
```

return x;

```
RETURN x
```

Function Calls

Note: All parameters in kxi are pass-by-value, there is no syntax in kxi to allow pass-by-reference. You can pass a reference (pointer) to an object or array, but you are still passing it by value not by reference.

x.f();

```
FRAME f, x      ; Create an activation record
CALL f          ; Invoke the function
```

Note: The intermediate code above is basically equivalent to the following code below. Invoking a member function is nothing more complex than just calling a function with the referenced object as the first passed parameter on the stack. There is NO MAGIC here!!!!!!

```
FRAME f
PUSH x     ; Push x on run-time stack
CALL f
```

f();

Note: There is always an object to pass in kxi, even when not explicitly stated there is an implicit "this" object implied here. Just use the keyword "this" in your intermediate code.

```
FRAME f, this
CALL f
```

x.f(i, j);

```
FRAME f, x
PUSH j          ; Push j on run-time stack
PUSH i          ; Push i on run-time stack
CALL f
```

y = x.f();

Note: The use of instruction PEEK vs. POP, allows us to access the return value from the top of the Stack without changing the Top of Stack Pointer (i.e., SP).

```
FRAME f, x
CALL f
PEEK t5         ; x.f() → t5
MOV  t5, y      ; t5 → y
```

x = y . f(k , g) + g * r ;

Note: See the Semantic Action Slides for the infix to postfix conversion of this expression. Intermediate Code is generated as the ref_sar is popped off the SAS.

```
MUL   g, r, t4          ; g * r → t4
FRAME f, y
PUSH g
PUSH k
CALL f
PEEK t5                 ; y.f(k,g)  → t5
ADD   t5, t4, t6        ; t5 + t4 → t6
MOV   t6, x             ; t6 → x
```

r = x . f ( ) . g ( ) . y;

Note: This complex looking example is really handled as four different, yet simple statements, not one huge statement. Your Compiler must work recursively (recursive calls to member_refz) to correctly handle this kind of code. If the code consisted of the following statements it would be easier to see what code needed to be generated:

```
     t9   = x.f();
     t10  = t9.g();
     r    = t10.y;
```

```
FRAME f, x
CALL f                  ; x.f()  → t9
PEEK t9
FRAME g, t9
CALL g                  ; t9.g()  → t10
PEEK t10
REF t10, y, t11         ; t10 + offset(y)  → t11
MOV t11, r              ; t11 → r
                        ; t11 points at something
                        ; on the heap
```

x . f ( ) . g ( ) . y = r ;

```
FRAME f, x
CALL f
PEEK t1                 ; x.f()  → t1
FRAME g, t1
CALL g
PEEK t2                 ; t1.g()  → t2
REF t2, y, t3           ; t2+offset(y)  → t3
MOV r, t3               ; r → t3
```

f(r*3,g<k);

```
     MUL r, c9, t7         ; r * 3 → t7
     LT g, k, t8           ; g < k → t8
     FRAME f, this
     PUSH t8
     PUSH t7
     CALL f                ; f(t7, t8)
```

Accessing Array Elements

Note: An Array reference is similar to an Instance Variable reference in that we start with a Base Address and use an Offset to compute the effective address. However, when accessing Arrays the computation is a little more complex:

One dimensional array.
*Effective Address = Base Address + (Index * sizeof(Element))*

Two dimensional array. (Not used in kxi)
*Effective Address = Base Address + (ColumnIndex * sizeof(Element)) + (RowIndex*sizeof(Element)*width)*

Effective Address - is the location of the Array Element in memory.
Index - is the integer used to index into the Array.
sizeof(Element) - is the number of bytes used to store a element of the array.

The temp value created for the array address must be used as an Indirect Address vs. a specific value (ADD in this case is very much like a REF only the offset is computed by the MUL)…
*The icode instruction AEF works much like REF only it is specific to arrays. AEF takes a base address and an index then computes an address (base address + [sizeof(type) * index])*

x = c[r];
Note: SOE3 is a constant that contains the sizeof information for array c.

```
     AEF c, r, t6          ; compute address
     MOV t6, x             ; (t6) → x
or
     MUL r, soc3, t5       ; index * sizeof(Element) → t5
     ADD c, t5, t6         ; base address + t5 → (EA) t6
     MOV t6, x             ; (t6) → x
```

c[r] = x;

```
        AEF c, r, t10
        MOV x, t10            ; x → (t10)
or
        MUL r, soc3, t9
        ADD c, t9, t10
        MOV x, t10            ; x → (t10)
```

c[r+3] = c[r-5];

```
        SUB r, c10, t24       ; r - 5 → t24
        AEF c, t24, t25
        ADD r, c9, t26        ; r + 3 → t26
        AEF c, t26, t27
        MOV t26, t27          ; (t26) → (t27)
or
        ADD r, c9, t23        ; r + 3 → t23
        SUB r, c10, t24       ; r - 5 → t24
        MUL t24, sco3, t25    ; t24 * sizeof(Element) → t25
        ADD c, t25, t26       ; c + t25 → t26
        MUL t23, soc3, t27    ; t23 * sizeof(Element) → t27
        ADD c, t27, t28       ; c + t27 → t28
        MOV t26, t28          ; (t26) → (t28)
```

Creating Objects

Note: Constructors are really just function calls that are used to initialize a structure in memory (the object). We will let the instruction "NEW" create space, and the function named the same as the class (e.g., constructor) initializes the object.

*Note: The "new" needs to allocate space on the heap (e.g., malloc) for an object the size of a Cat. We will assume a Cat has 3 instance variables all of type integer (4 bytes per integer), 3*4 = 12 bytes for an instance of a Cat.*

Note: Compute the size of each Class and store it in the Symbol Table along with the class. This can be simply done as you parse the Class. Computing the size of objects can be deferred until Target Code, but this would require a more complex Compiler. Just add this to your parser.

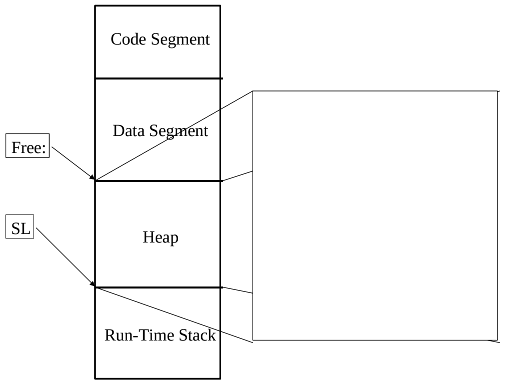
Cat k = new Cat();

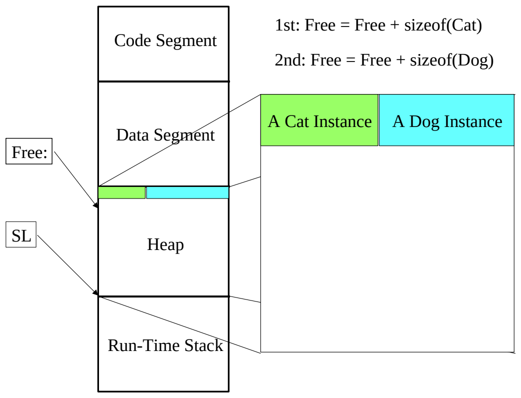
```
        NEWI 12, t24          ; malloc(sizeof(Cat)) → t24
```

```
FRAME Cat, t24
CALL Cat
PEEK t25               ; t24.Cat() → t25
MOV t25, k             ; t25 → k
```

Allocating an Object on the Heap: Target Code

Free is a variable that contains the starting address of the Heap (e.g., a pointer to free heap space).
Allocate each new object (e.g., Instance or Array) at the address of Free and the then increment Free by the size of the allocated Object.
This is not Memory Management (e.g., that is often done by the OS), this is just a simple strategy to allocate objects on the Heap.





Cat k = new Cat(7, x);

```
NEWI 12, t24            ; malloc(sizeof(Cat)) → t24
FRAME Cat, t24
PUSH x
PUSH c7                 ; 7 → c7
CALL Cat
PEEK t25                ; t24.Cat(c7, x) → t25
MOV t25, k              ; t25 → k
```

Creating Arrays

Note: We will assume that a pointer (reference to an Instance) is 4 bytes long. See how we use the size of a pointer not the size of a Cat!!! Again we could defer knowing the size of a pointer until Target Code, but it would make the Compiler more complex.

Note: While Semantic Action #arr will check that the expression that indicates the size of an array is an integer. It is outside the scope of your Compiler to perform bounds checking on an Array. Bounds checking is a Run-Time check, not a Compile-Time check!

Cat c[ ] = new Cat[3];

```
MUL c4, c3, t3          ; sizeof(pointer) * 3 → t3
NEW t3, t4              ; malloc(t3) → t4
MOV t4, c               ; t4 → c
```

Cat x[ ] = new Cat[r] ;

```
MUL c4, r, t8           ; sizeof(pointer) * r → t8
NEW t8, t9              ; malloc(t8) → t9
MOV t9, x               ; t9 → x
```

int a[] = new int[r*k];

```
MUL r, k, t11           ; r * k → t11
MUL c2, t11, t12        ; sizeof(int) * t11 → t12
NEW t12, t13            ; malloc(t12) → t13
MOV t13, a              ; t13 → a
```

Instance vs. Stack Variables

Note: All variables except those that are GLOBAL will be accessed using a Base Address ± an Offset.

Note: Compute this during parsing and store in the Symbol Table.

Instance Variable will be accessed as:

(Address of this) + (Variable Offset)

Function (Stack) Variables will be accessed as:

(Address of the Current Stack Frame) – (Variable Offset)

}

Constructors & Static Initializers

```
    class Cat {
        int x = 7;
        int i = 5;
        int y;

        Cat(int a) {
            y = a;
        }

        // kxi only allows one Constructor
        Cat() {
            y = 0;
        }
    }
```

Note: Constructors are just function calls.
Note: The Static Initializers x = 7 & i = 5 are also part of a function call.

All Constructors of a Class invoke the static initializer function before executing the code of the Constructor. Static Initializers must be done by Code. Static Initialization can not be set at Compile-Time because the Objects are created at Run-Time

Static Initializers as Function Calls: What the Code Means!

```
class Cat {
    int x;
    int I;
    int y;

    private void CatStaticInit() {
        x = 7;
        i = 5;
    }

    Cat(int a) {
        CatStaticInit();
        y = a;
    }

    // kxi only allows one Constructor
    Cat() {
        CatStaticInit();
        y = 0;
    }
}
```