

**Intermediate Representation With Interesting Name  
(IRWIN)**

Nick Forrette – CS490  
10-17-99 Rev C

## Introduction

IRWIN (Intermediate Language With Interesting Name) is the language used in the CS490 Tiger language compiler (TC). The target of TC is assembly language for i386 Linux. IRWIN is used as an intermediate step in the process of compilation. IRWIN is designed to more closely resemble i386 assembly than does Tiger. IRWIN is also free-form. This means that there is no special termination character (like the semi-colon in C/C++).

IRWIN code is generated by traversing Tiger abstract syntax tree (AST). The IRWIN file is then parsed, and a new syntax tree is generated. This new syntax tree is designed to undergo further optimizations before the generation assembly code.

An IRWIN program executes sequentially, starting at the first statement in the program. Comments, blank lines, and code in subroutines are ignored. Jumps and subroutine calls can alter the flow of control of the program.

## Design

The design of IRWIN was influenced primarily by the needs of the Tiger language, and the restrictions of i386 assembly. IRWIN is designed to be garbage collected, as is Tiger. IRWIN supports an unlimited number of temporary variables, each of which are preserved across function calls.

## Memory / Garbage Collection

IRWIN has three separate memory spaces. A garbage collected heap, a static heap, and a stack. As the name suggests, the garbage collected heap is garbage collected and the static heap is not.

There are two kinds of heap objects: object data and raw data. Object data is capable of holding pointers and integers only. Raw data can hold any arbitrary data, but dynamic heap pointers. The reason for the limitation is to assist the garbage collection process. Pointers in object data blocks are followed/updated by the garbage collector when it is collecting. Information in raw data blocks are left untouched.

## Temporaries / Namespaces

An IRWIN temporary is capable of holding one integer, or one pointer value. Unsigned IRWIN temporaries range in value from 0 to  $2^{31} - 1$ , and signed integers from  $-2^{30}$  to  $2^{30} - 1$ . An IRWIN word is 32 bits. A byte is the standard 8 bits.

Since IRWIN supports an unlimited number of temporary variables, you must declare them before use. IRWIN has five distinct namespaces. One for temporaries, one for jump labels, one for subroutines, one for raw data, and one for object data.

All temporaries are preserved across function calls. However, each call to a function gets its own copy of these temporaries. I.E., function *foo* has variable *B*. *Foo* recursively calls itself. The second call to *foo* has its own independent copy of variable *B* to use.

Temporaries may be declared in top-level code, or inside of subroutines. If declared at the top level, they have global scope. If declared inside of a subroutine, they are scoped to that subroutine. It is possible to use access links to implement up-level variables, if needed. Raw data and object data can also be declared at any point in an

IRWIN program, however, they all act as if they were grouped together at the beginning of the program. In other words, raw data and object data are always globally scoped. It is illegal to declare subroutines inside of other subroutines. It is also illegal to jump into or out of subroutines. Always use appropriate subroutine calls and return statements.

### **Other illegal constructs.**

It is illegal for a pointer to point to the middle of an object or raw data (This will confuse the garbage collector). Instead, point to the object itself, and access the needed information based on its offset.

It is illegal to drop off the end of a function. Instead, always use an explicit return statement.

It is illegal to reference temporaries, object data, or raw data before they are declared. However, it is legal to call a subroutine before it is defined.

It is illegal to use compound statements. I.E.: *local A = 1* is illegal. It mixes assignment with declaration of a local. The proper syntax is *local A A = 1*. The declaration and initialization are split in to two statements.

It is illegal to mix integer and pointer data with string data in raw data blocks. Single quoted characters get sign extended to their equivalent integer value.

### **Statements**

Statements in IRWIN have no special end of statement character (such as ; in C). As such, it is a free form language. This means that multiple statements can be on the same line, without an explicit separator. Comments are indicated by either the # or ; characters and extend to the end of the line. Compound statements are illegal.

### **Identifiers**

Identifier names are made of the letters A–Z, the numbers 0–9, and the underscore character ("\_"). They may not begin with number and are case sensitive. All temporaries share the same namespace, thus must be unique across the entire program. There are a few reserved words in IRWIN. They are not to be used for temporary names or jump labels. See the following table for a listing.

### **Language Summary**

<b>Reserved word summary</b>	<b>Description</b>
alloc	Used to dynamically allocate memory on GCed heap.
call	Used to transfer control of a program to a subroutine.
data	Used to declare global, raw data memory blocks on the static heap and to take the address of <b>data</b> declared temporaries.
end	Used to mark the end of a subroutine.
exit	Used to end an IRWIN program and pass a return value back to operating system.
goto	Used to alter the flow of control of the program. Both conditional and unconditional jumps use this keyword.
if	Used in conjunction with <b>goto</b> to implement a conditional jump.

Reserved word summary	Description
<code>isint</code>	Tests to see if a local is holding an integer value.
<code>isptr</code>	Tests to see if a local is holding a pointer value.
<code>label</code>	Used to take the address of jump labels. Useful for initializing jump tables.
<code>local</code>	Declares a new temporary, and takes the address of a temporary.
<code>object</code>	Used to declare global scoped objects on the static heap, and to take the address of <b>object</b> declared temporaries.
<code>return</code>	Used to return control back to a subroutines caller.
<code>size</code>	Returns the size of heap allocated objects.
<code>sub</code>	Used to declare subroutines and to get the address of a subroutine.

L-values	Description
<code>local A</code> <code>A = R-value</code>	Temporary variables are legal l-values.
<code>data B = 1,2,3</code> <code>data [B]{0}</code>	Indirect <b>data</b> accesses. The <code>{}</code> syntax accesses raw data blocks a byte at a time. If assigning to or from a local, only the bottom 8 bits are of concern. The upper bits are ignored.
<code>object B = 1,2,3</code> <code>object [B][0]</code>	Indirect <b>object</b> accesses. The <code>[]</code> syntax accesses object data one field at a time.
<code>object A = 1,2,3</code> <code>local B</code> <code>B = object [A]</code> <code>B[0] = R-value</code>	Indirect through a pointer stored in a temporary. The pointer could be to an object or raw data block. However, make sure that you use the proper <code>{}</code> or <code>[]</code> . The <code>{}</code> are for byte-at-a-time raw data access. The <code>[]</code> are for field-at-a-time object accesses.

Compile-time constant expressions	Description
<code>0x12AB</code>	<b>Hexadecimal constant:</b> Numeric data prefixed with a <b>0x</b> prefix is assumed to be hexadecimal (base 16).
<code>-1234</code> <code>0987</code>	<b>Decimal constant:</b> Numeric data without a prefix is assumed to be decimal (base 10). Note that 0 prefixed numbers are also assumed to be decimal, not octal. A prefixed unary minus indicates the negation of the integer value.
<code>'a'</code> <code>'\8a'</code>	<b>Character constant:</b> One byte character constant. In many circumstances, the value will be sign extended to fill a word. The exception to this rule is when used to initialize global data with the <b>data</b> directive. The backslash character <b>'\XX'</b> can be used as an escape character, where XX represents a two digit hexadecimal number that is the ASCII code for the desired character.
<code>"abcde"</code>	<b>String constant:</b> A string constant. The length is fixed at one byte per character in the string. They do not dynamically grow or shrink. The length of strings in IRWIN are tracked internally by IRWIN, thus are not NULL terminated. The <code>\</code> character can also be used as an escape character like in character constants.
<code>V = object [A]</code> <code>W = data [B]</code> <code>X = label [C]</code> <code>Y = local [D]</code> <code>Z = sub [E]</code>	Only <b>object</b> , and <b>data</b> return compile time constants. <b>Sub</b> and <b>label</b> require run time computation, therefor can not be used to initialize static data. <b>Object</b> , <b>data</b> , <b>local</b> , and <b>sub</b> are overloaded for declaring their respective types.  <i>See memory/pointers below for more information.</i>

Memory / pointers	Description
<pre>data X = 'a', 'b', 5 data Y = "abcde" data Z = 123, '\AF'</pre>	<p><b>Raw data:</b> The <b>data</b> directive is used to allocate pre-initialized raw-data on the static heap. The size of the block of memory is just large enough to hold the initialization data. The initialization data must also be a compile time constant. <i>See previous section.</i></p> <p>Since any information allocated with a <b>data</b> directive is raw, the garbage collector will not follow any pointers contained therein. Garbage collection will not update pointers, therefore it is illegal to initialize raw data with dynamic pointer values.</p> <p>Note that <b>data</b> is overloaded. It is also used to take the address of <b>data</b> temporaries.</p>
<pre>object A = 1, 'A',           object [X]</pre>	<p><b>Object data:</b> The <b>object</b> directive allocates space for initialized object data. Each field may contain an integral value or a pointer. Each field must be initialized by a compile time constant. The garbage collector will follow/update pointers stored in <b>object</b> allocated blocks.</p> <p>Note that <b>object</b> is overloaded. It is also used to get the address of <b>object</b> data blocks.</p>
<pre>local B</pre>	<p><b>Temporary variable:</b> Temporaries are declared with the <b>local</b> keyword. If used at the top level of a program, they are globally scoped. If used within a subroutine, its scope is that subroutine.</p> <p>Note that <b>local</b> is overloaded. It is also used to take the address of <b>local</b> temporaries.</p>
<pre>V = object [A] W = data [B] X = label [C] Y = local [D] Z = sub [E]</pre>	<p><b>Address of:</b> The <b>object</b>, <b>data</b>, <b>label</b>, <b>local</b>, and <b>sub</b> operators are used to take the address of object data, raw data, labels, locals, or subroutines.</p> <p><i>See compile time constant above for more information.</i></p>
<pre>object B = 2, data [B] object [B][0] = 53 object [B][1] = data [A]</pre>	<p><b>Object Fields:</b> It is possible to access particular fields of an object with the <b>[]</b> syntax.</p>
<pre>data B = "asdf"</pre>	<p><b>Data Fields:</b> It is possible to access particular bytes of an object with the <b>{}</b> syntax. Data blocks should never contain dynamic pointer values.</p>
<pre>A = alloc B</pre>	<p><b>Dynamic heap storage:</b> Storage on the GCed heap may be allocated with the <b>alloc</b> keyword.</p>
<pre>A = size B</pre>	<p><b>Size of heap object:</b> The <b>size</b> operator expects a pointer to a heap allocated memory block and yields the size in bytes of the that memory block.</p>
<pre>A = foo@bar</pre>	<p><b>Stack offset:</b> The offset of variable bar within function foo's stack frame. Useful when using access links to implement up-level variable references sometimes found in higher-level languages.</p>
<pre>B = sub [foo] call [B](1,2,3)</pre>	<p><b>Call anonymous subroutine:</b> It is possible to call a subroutine by its address with <b>call [...]</b> syntax. There is no compile time type checking when calling subroutines this way!</p>

Flow control	Description
<pre>my_label:</pre>	<p><b>Label:</b> An identifier followed by a colon are used to declare a jump label.</p>
<pre>goto my_label</pre>	<p><b>Unconditional jump:</b> Unconditional jump to a label.</p>

Flow control	Description
<code>B = label [my_label]</code> <code>goto [B]</code>	<b>Unconditional jump:</b> Unconditional jump through a pointer to a label.
<code>if A goto foo</code> <code>if A goto [B]</code>	<b>Conditional jump:</b> Jump to label A (or through a pointer to a label) if temporary A is not zero.
<code>call foo()</code> <code>call bar(1,2,3)</code> <code>call bar(A,B,C)</code>	<b>Subroutine call:</b> Calls to subroutines are done with the <b>call</b> keyword. They are type checked to ensure that the number of parameters passed to the named sub match the number of parameters in its declaration.  Note: It is legal to call subroutines before they are declared!
<code>B = sub [foo]</code> <code>call [B](1,2,3)</code>	<b>Call anonymous subroutine:</b> <i>See memory/pointer table for more information.</i>
<code>return</code> <code>return ()</code> <code>return (X)</code>	<b>Subroutine termination:</b> It is illegal to drop off the bottom of IRWIN subroutines. If no value is to be returned to the calling function, use one of the first two versions of <b>return</b> . If returning a value, use the third version of <b>return</b> .  Note: Do NOT mix void and non-void <b>return</b> statements in the same subroutine!
<code>sub foo(A,B,C)</code> <code># ...</code> <code>end sub foo</code>	<b>Subroutine declaration:</b> Subroutines may only be declared at the top level of an IRWIN program. Their length is unlimited. The parameters in the parameter list act as if each has been declared as <b>local</b> in the body of the subroutine. The names of the parameters share the same namespace as other temporaries.

Arithmetic operators	Description
<code>A = B + C</code>	<b>Addition:</b> Only legal with integer values! Behavior with pointer operands is undefined. Use [...] Indirection syntax instead.
<code>A = B - C</code>	<b>Subtraction:</b> <i>See Addition.</i>
<code>A = B * C</code>	<b>Multiplication:</b> <i>See Addition.</i>
<code>A = B / C</code>	<b>Signed Division:</b> <i>See Addition.</i>
<code>A = B \ C</code>	<b>Unsigned Division:</b> <i>See Addition.</i>
<code>A = B /% C</code>	<b>Signed Mod:</b> <i>See Addition.</i>
<code>A = B \% C</code>	<b>Unsigned Mod:</b> <i>See Addition.</i>
<code>A = B &gt;&gt; C</code>	<b>Right shift:</b> <i>See Addition.</i>
<code>A = B &gt;&gt;&gt; C</code>	<b>Sign extended right shift:</b> <i>See Addition.</i>
<code>A = B &lt;&lt; C</code>	<b>Left shift:</b> <i>See Addition.</i>
<code>A = B &amp; C</code>	<b>Bitwise AND</b>
<code>A = B   C</code>	<b>Bitwise OR</b>
<code>A = B ^ C</code>	<b>Bitwise XOR:</b> Note that negation can be expressed as $A \wedge \sim 1$
<code>A = B &gt; C</code> <code>A = B &lt; C</code> <code>A = B &lt;= C</code> <code>A = B &gt;= C</code> <code>A = B == C</code> <code>A = B != C</code>	<b>Comparison operators:</b> All have their traditional mathematical meanings. Only the two equality operators ( == and != ) with pointers. The garbage collector may reorder the layout of heap memory at any time. Only equality is guaranteed to be preserved by the collector.

## Sample Program #1

```
# Title : Sample IRWIN program #1
# Author: Nick Forrette

local A      # Declares global scoped temporary A.
A = 5        # Set temporary A's value to integer 5

local B B = 5 # Same as above but takes advantage of IRWIN's free form

local C C     # Also the same as above. Multiple statements can
=           # on the same line without a special separator. They can
0           # also span lines, or a combination of the two.

local D      # Declares a temporary without giving it an initial value
            # IRWIN will give it the value 0.

C = call mult(A, B) # Call sub mult before it is declared
                  # C gets return value of the mult sub. In this case,
                  # C will be assigned 25

D = call mult(C, 2) # Another call to mult passing a compile time constant
                  # and the value of temporary C.

#####
# Subroutine declaration
# Takes two values (assumes they are integers) and returns their product.
#####
sub mult(mult_A, mult_B)    # mult_A and mult_B are just like local
                          # variables whose scope is sub mult.

    # Another example of free form IRWIN.
    # The following line is TWO separate statements.
    # They are underlined to distinguish them
    local mult_C mult_C = mult_A * mult_B

    # Return the value of mult_C to the caller.
    return (mult_C)

end sub mult
```

