```
$Id: asg5-intermed-lang.mm,v 1.4 2012-11-07 17:51:10-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments
```

1. Overview

An intermediate language is a very low level language used by a compiler to perform optimizations and other changes before emitting final assembly language for some particular machine. It generally matches common assembly language statement semantics, but in a typeful manner.

SYNOPSIS

`oc` [`-ly`] [`-@` *flag* ...] [`-D` *string*] *program*`.oc`

All of the requirements for all previous projects are included in this project. For any input file called *program*`.oc` emit an intermediate language file called *program*`.oil`, which can then be compiled into assembly language using `gcc`.

program	→	[structdef]... [stringdef]... [type *IDENT* ';']... [function]...
structdef	→	'`struct`' *TYPEID* '{' [type *IDENT* ';']... '}' ';'
stringdef	→	'`ubyte`' '`*`' *IDENT* '`=`' *STRINGCON* ';'
function	→	type *IDENT* '(' [type *IDENT* [',' type *IDENT*]...] ')' '{' [statement]... '}'
statement	→	*LABEL* ':;'
	→	['`if`' '`(`' ['`!`'] operand '`)`'] '`goto`' *LABEL* ';'
	→	'`return`' [operand] ';'
	→	[type] *IDENT* '`=`' expression ';'
	→	selection '`=`' operand ';'
	→	[[type] *IDENT* '`=`'] *IDENT* '`(`' [operand [',' operand]...] '`)`' ';'
expression	→	operand binop operand | unop operand | selection | operand
binop	→	'`+`' | '`-`' | '`*`' | '`/`' | '`%`' | '`==`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`>=`'
unop	→	'`+`' | '`-`' | '`!`' | '`(int)`' | '`(ubyte)`'
selection	→	*IDENT* '`[`' operand '`]`' | *IDENT* '`->`' *IDENT*
operand	→	*IDENT* | *INTCON* | *CHARCON*
type	→	'`void`' | '`ubyte`' ['`*`' ['`*`']] | '`int`' ['`*`'] | '`struct`' *TYPEID* '`*`' ['`*`']

Figure 1. Grammar of `oil`

`bool` *IDENT*	`ubyte` *IDENT*
`char` *IDENT*	`ubyte` *IDENT*
`int` *IDENT*	`int` *IDENT*
`string` *IDENT*	`ubyte *`*IDENT*
TYPEID IDENT	`struct` *TYPEID* `*`*IDENT*

`bool[]` *IDENT*	`ubyte *`*IDENT*
`char[]` *IDENT*	`ubyte *`*IDENT*
`int[]` *IDENT*	`int *`*IDENT*
`string[]` *IDENT*	`ubyte **`*IDENT*
TYPEID`[]` *IDENT*	`struct` *TYPEID* `**`*IDENT*

Figure 2. Type declarations in `oc` and `oil`

2. Intermediate Language

The intermediate language chosen here looks very much like C, except that, for the most part, each `oil` statement should be capable of translating into a single assembly language instruction, or only a few. Unlike most assembly languages, `oil` is typed as to basic data types, namely unsigned byte, int, and pointer. The size of the int and pointer types are dependent on the underlying architecture.

The basic grammar of `oil` is given in Figure 1, and uses the same metanotation as was used in the definition of `oc`. Figure 2 shows the relationship between types in `oc` and the corresponding C-compatible types in `oil`.

The intermediate language itself has three datatypes: 8-bit unsigned byte (`ubyte`); 32-bit two's complement signed integer (`int`); and a pointer type which is either 32 or 64 bits, depending on the architecture and compilation options such as `-m32` or `-m64`.

2.1 Program and Function Structure

An `oil` program is structured in a way similar to C, except that it looks more like assembly language. All code is either aligned with the left margin or indented by 8 spaces.

(a) Structure definitions come first, with the `struct` keyword and closing brace left-aligned, and the fields indented, as in :

```
struct s_foo {
        ubyte **f_foo_string_array;
        struct s_foo *f_foo_pointer;
};
```

(b) Then all string constants are listed in the order they appear in the program. They are unlikely to be able to be used as immediate operands in assembly language. Global declarations are all left-aligned, as in :

```
ubyte *s1 = "Hello, world!\n";
```

(c) Then come all global variable declarations that appeared in the program, but without initialization, because in C, static initialization is done at pile time.

(d) Finally, functions are emitted with one parameter per line. The function name and the two braces are left aligned, as are label statements, but everything else is indented, as in :

```
int __add (
        int _1_a,
        int _1_b)
{
        int i2 = _1_a + _1_b;
        return i2;
}
```

2.2 Statements

Statements are all indented, except for labels, and all declarations of local variables must be initialized. Expressions other than calls are restricted to three-address instructions, with a destination and at most two sources.

(a) A `goto` statement may be conditional or unconditional, and if conditional is preceded by an `if` and an operand (possibly complemented) in parentheses.

(b) A `return` statement may or may not have an operand, depending on the requirements of the original `oc` program.

(c) An assignment statement takes an expression as an operand. A type is present if the identifier has not previously been declared, and absent if it has.

(d) Selection may occur on the left or right of an assignment operator, but if on the left, the right side may only be an operand, not an expression.

(e) A function call may only have operands in parentheses, but may have as many as appropriate. A void function has no assignment on the left. All other functions must have it, and the result usually goes into a temporary.

2.3 Expressions

Expressions are simple and restricted to three-address code.

(a) Binary operators are the same as they are in `oc` and have already been type checked. As in C, 0 means false and anything else means true.

(b) The unary operators are also the same, with the cast `(int)` being substituted for `ord`, and `(ubyte)` being substituted for `chr`. These represent actual machine instructions that must be

issued.

(c) Address computation for indexing and field selection is taken care of by the back end, alghough a type checker would normally assign offsets of fields in structures and offsets of local variables from the frame pointer.

(d) Figure 2 show the types in **`oc`** and their corresponding types in **`oil`**. Note that the asterisk representing a pointer cuddles up against the following identifier, not against the type.

(e) There is no **`bool`** data type, it being replaced by **`ubyte`**, as was done with **`char`**.

Global names	`"__%s"`	identifier
Local names	`"_%d_%s"`	block, identifier
Statement labels	`"%s_%d_%d_%d"`	keyword, file, line, offset
Structure typeids	`"s_%s"`	structure_typeid
Field names	`"f_%s_%s"`	structure_typeid, field_name
Virtual registers	`"%c%d"`	register_category, register_number

Figure 3. Summary of name mangling

2.4 Name Mangling

Assembly language generally uses a completely flat symbol table, so names need to be mangled in order to avoid clashing global names with various local scopes and the C library to which a program is linked. Figure 3 shows the **`fprintf`**(3) format items and arguments to be used to mangle a name.

(a) All global names, functions and global variables will be prefixed with a double underscore (__), e.g., "**`__foo`**".

(b) Local names will be prefixed with a double underscore with a block number between the underscores, as in "**`_1_n`**".

(c) Statement labels are emitted with a keyword from the language followed by the file number, line number, and offset from the token. The following keywords are used : "**`while`**", before a while expression ; "**`break`**", the statement to transfer to for a false test ; "**`else`**", to branch to if the if-test is false and there is an **`else`** ; and "**`fi`**", to skip the else part.

(d) Structure names all have file scope and are just prefixed with "**`s_`**".

(e) Field names are prefixed with "**`f_`**" and the name of the structure to which they belong.

(f) Virtual registers are just assigned numbers in sequence throughout the entire program starting with 1, 2, 3, etc. The letter used is **`b`** for a **`ubyte`**, **`i`** for an **`int`**, **`p`** for a data pointer, and **`s`** for a *STRINGCON* address. E.g., **`s1`**, **`b2`**, **`i3`**, **`p4`**.

```
1    #!/bin/sh -x
2    # $Id: ocl,v 1.2 2011-11-17 21:40:38-08 - - $
3    for file in $*
4    do
5       gcc -g -o $file -x c $file.oil oclib.c
6    done
```

Figure 4. `oil-examples/ocl`

3. Code Emission

Code is emitted from the AST with the benefit of the the symbol table and type checker. Since the suffix **`oil`** is not recognized by **`gcc`**, in order to make it compile the code, the **`-x`** option must be used. Figure 4 shows a shell script that will compile into an executable, given the name of the executable.

The file **`oclib.oh`** is bilingual for both **`gcc`** and **`oc`**, and mangles names correctly when **`__OCLIB_C__`** is

defined. Figure 5 shows the raw file, Figure 6 show the file preprocessed for `oc`, and Figure 7 shows it preprocessed for `gcc`. Figures 8 and 9 show the code for `oclib.c`, the library written in C.

3.1 Top Level Description

First, a prolog is emitted, then the children of the root are traversed in their own sequence to build the various parts of the program.

(a) The prolog is first generated :
```
#define __OCLIB_C__
#include "oclib.oh"
```

(b) Then all structure definitions are traversed and output as described above, with names mangled, and fields properly indented.

(c) Next, all string constants are given names as described above. This should not require a complete tree traversal. Retrofit either the construction of the AST or the type checking module to put all *STRINGCON* AST nodes into a queue.

(d) Then all global variables, immediate children of the root, and output, but without any initialization given to them by the source.

(e) Then all functions are output, as described below. Finally a single function with the signature
`void __ocmain ()`
is output, containing all global initializations and statements.

3.2 Emitting Functions and Statements

Function emission is done by a complete depth-first mostly post-order traversal of the AST for that function. Prototypes need not be emitted. They are used only in type checking.

(a) Output the function's return type, mangled name, and mangled parameter names. Mangle the parameter names using the block number the function created.

(b) Following that, indented, are the declarations of all of the parameters.

(c) An opening brace, unindented, at the start of the function's body, and a closing brace, unindented, after the function is finished.

(d) A block simply has its children traversed.

(e) A `while` has two children. For the `while` and `break` labels, both use the serial number of the `while` token.
  (i) Emit : `while_%d_%d_%d:;` unindented.
  (ii) Emit the first child, unless it is an operand.
  (iii) Emit : `if (!`*operand*`) goto break_%d_%d_%d;`
  (iv) Emit the statement.
  (v) Emit : `goto while_%d_%d_%d;`
  (vi) Emit : `break_%d_%d_%d:;` unindented.

(f) For an `if-else` statement, do the following :
  (i) Emit the first expression, unless it is an operand.
  (ii) Emit : `if (!`operand`) goto else_%d_%d_%d;`
  (iii) Emit the second child (consequent statement).
  (iv) Emit : `goto fi_%d_%d_%d;`
  (v) Emit : `else_%d_%d_%d:;` unindented.
  (vi) Emit the third child (alternate statement).
  (vii) Emit `fi_%d_%d_%d:;` unindented.

(g) For an `if` with no `else`, do the following :
  (i) Emit the first expression, unless it is an operand.
  (ii) Emit : `if (!`operand`) goto fi_%d_%d_%d;`
  (iii) Emit the second child (consequent statement).
  (iv) Emit `fi_%d_%d_%d:;` unindented.

(h) If a `return` statement has an expression, emit the expression, unless it is an operand, then return the operand. If not, just `return`.

3.3 Emitting Expressions

Expressions are always emitted using a strict depth-first post-order traversal, accumulating values into virtual registers. This results in some redundant virtual registers, but the back end can use copy propagation to remove them during register allocation.

(a) In each case of generating code into an operand, generate a declaration of a virtual register with the appropriate letter, `b`, `i`, or `p`, and assign it the value of the expression thus evaluated.

(b) The binary operators are the same in both `oc` and `oil`, and after emitting an instruction, record the register number in the AST node itself. This will require adding yet another field. Example :

```
int i5 = _3_n + 1;
```

(c) Similarly, handle the unary operators, except that `ord` becomes `(int)` and `chr` becomes `(ubyte)`.

(d) Treat selectors other than those which a left child of assignment as binary operators, and evaluate them into a virutal register of the appropriate type.

(e) An allocator uses a call to `xcalloc()` to allocate the storage, based on the type of allocator, for some structure type T and pointer number i :

(i) '`new`' *TYPEID* '`(`' '`)`' :
Emit : `struct` T `*p`i `= xcalloc (1, sizeof (struct` T`));`

(ii) '`new`' '`string`' '`(`' expression '`)`' :
Emit the expression, unless it is an operand.
Emit : `ubyte *p`i `= xcalloc (`*opnd*`, sizeof (ubyte));`

(iii) '`new`' basetype '`[`' expression '`]`' :
Emit the expression, unless it is an operand.
Emit one of the following, depending on the *basetype* :

- `ubyte *p`i `= xcalloc (`*opnd*`, sizeof (ubyte));`
- `int *p`i `= xcalloc (`*opnd*`, sizeof (int));`
- `ubyte **p`i `= xcalloc (`*opnd*`, sizeof (ubyte *));`
- `struct` T `**p`i `= xcalloc (`*opnd*`, sizeof (struct` T `*));`

(f) For a function call, evaluate each of the arguments into registers, and generate the call instruction. A `void` function is just called. A non-`void` function has its result captured into a register.

(g) An *INTCON* must be emitted with leading zeros suppressed, since 0077 means tha same thing as 77 in `oc`, but it means 63 in C. A *CHARCON* is emitted as is. The constants `null` and `false` are emitted as `0`, and `true` is emitted as `1`.

(h) Constants (except strings) and variables are never captured in registers, but instead are used as immediate operands.

```
 1  // $Id: oclib.oh,v 1.24 2011-11-18 17:34:29-08 - - $
 2
 3  #ifndef __OCLIB_OH__
 4  #define __OCLIB_OH__
 5
 6  #ifdef __OCLIB_C__
 7  #   define __(ID)         __##ID
 8  #   define NONE__         void
 9  #   define VOID__(ID)     void __##ID
10  #   define BOOL__(ID)     ubyte __##ID
11  #   define CHAR__(ID)     ubyte __##ID
12  #   define INT__(ID)      int __##ID
13  #   define STRING__(ID)   ubyte *__##ID
14  #   define STRINGS__(ID)  ubyte **__##ID
15  #   define null           0
16  #   define false          0
17  #   define true           1
18  typedef unsigned char ubyte;
19  void *xcalloc (int nelem, int size);
20  #else
21  #   define EOF            (-1)
22  #   define __(ID)         ID
23  #   define NONE__
24  #   define VOID__(ID)     void ID
25  #   define BOOL__(ID)     bool ID
26  #   define CHAR__(ID)     char ID
27  #   define INT__(ID)      int ID
28  #   define STRING__(ID)   string ID
29  #   define STRINGS__(ID)  string[] ID
30  VOID__(__assert_fail) (STRING__(expr), STRING__(file), INT__(line));
31  #endif
32
33  VOID__(putb) (BOOL__(b));
34  VOID__(putc) (CHAR__(c));
35  VOID__(puti) (INT__(i));
36  VOID__(puts) (STRING__(s));
37  VOID__(endl) (NONE__);
38  INT__(getc) (NONE__);
39  STRING__(getw) (NONE__);
40  STRING__(getln) (NONE__);
41  STRINGS__ (getargv) (NONE__);
42  VOID__(exit) (int status);
43  #define assert(expr) \
44          {if (! (expr)) __(__assert_fail) (#expr, __FILE__, __LINE__);}
45
46  #endif
47
```

Figure 5. `oc-programs/oclib.oh`

```
 1    # 1 "oc-programs/oclib.oh"
 2    # 1 "<built-in>"
 3    # 1 "<command-line>"
 4    # 1 "oc-programs/oclib.oh"
 5    # 30 "oc-programs/oclib.oh"
 6    void __assert_fail (string expr, string file, int line);
 7
 8
 9    void putb (bool b);
10    void putc (char c);
11    void puti (int i);
12    void puts (string s);
13    void endl ();
14    int getc ();
15    string getw ();
16    string getln ();
17    string[] getargv ();
18    void exit (int status);
```

Figure 6. `cpp oc-programs/oclib.oh`

```
 1    # 1 "oc-programs/oclib.oh"
 2    # 1 "<built-in>"
 3    # 1 "<command-line>"
 4    # 1 "oc-programs/oclib.oh"
 5    # 18 "oc-programs/oclib.oh"
 6    typedef unsigned char ubyte;
 7    void *xcalloc (int nelem, int size);
 8    # 33 "oc-programs/oclib.oh"
 9    void __putb (ubyte __b);
10    void __putc (ubyte __c);
11    void __puti (int __i);
12    void __puts (ubyte *__s);
13    void __endl (void);
14    int __getc (void);
15    ubyte *__getw (void);
16    ubyte *__getln (void);
17    ubyte **__getargv (void);
18    void __exit (int status);
```

Figure 7. `cpp -D__OCLIB_C__ oc-programs/oclib.oh`

```
 1  // $Id: oclib.c,v 1.44 2011-11-18 18:44:19-08 - - $
 2
 3  #include <ctype.h>
 4  #include <libgen.h>
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include <string.h>
 8
 9  #define xfflush  (void) fflush
10  #define xfprintf (void) fprintf
11  #define xprintf  (void) printf
12
13  #define __OCLIB_C__
14  #include "oclib.oh"
15
16  ubyte **oc_argv;
17
18  void ____assert_fail (char *expr, char *file, int line) {
19     xfflush (NULL);
20     xfprintf (stderr, "%s: %s:%d: assert (%s) failed.\n",
21               basename ((char *) oc_argv[0]), file, line, expr);
22     xfflush (NULL);
23     abort ();
24  }
25
26  void *xcalloc (int nelem, int size) {
27     // LINTED (sign extension from 32-bit to 64-bit integer)
28     void *result = calloc (nelem, size);
29     assert (result != NULL);
30     return result;
31  }
32
33  void __ocmain (void);
34  int main (int argc, char **argv) {
35     argc = argc; // warning: unused parameter 'argc'
36     oc_argv = (ubyte **) argv;
37     __ocmain ();
38     return EXIT_SUCCESS;
39  }
40
```

Figure 8. **`oc-programs/oclib.c`, part 1**

```
41    ^L
42    ubyte *scan (int (*skipover) (int), int (*stopat) (int)) {
43       int byte;
44       do {
45          byte = getchar ();
46          if (byte == EOF) return NULL;
47       } while (skipover (byte));
48       ubyte buffer[0x1000];
49       ubyte *end = buffer;
50       do {
51          // LINTED (assignment of 32-bit integer to 8-bit integer)
52          *end++ = byte;
53          assert (end < buffer + sizeof buffer);
54          *end = '\0';
55          byte = getchar ();
56       }while (byte != EOF && ! stopat (byte));
57       ubyte *result = (ubyte *) strdup ((char *) buffer);
58       assert (result != NULL);
59       return result;
60    }
61
62    int isfalse (int byte)   { return 0 & byte; }
63    int isnl (int byte)      { return byte == '\n'; }
64    void __putb (ubyte byte) { xprintf ("%s", byte ? "true" : "false"); }
65    void __putc (ubyte byte) { xprintf ("%c", byte); }
66    void __puti (int val)    { xprintf ("%d", val); }
67    void __puts (ubyte *str) { xprintf ("%s", str); }
68    void __endl (void)       { xprintf ("%c", '\n'); xfflush (NULL); }
69    int __getc (void)        { return getchar (); }
70    ubyte *__getw (void)     { return scan (isspace, isspace); }
71    ubyte *__getln (void)    { return scan (isfalse, isnl); }
72    ubyte **__getargv (void) { return oc_argv; }
73    void __exit (int status) { exit (status); }
74
```

Figure 9. **`oc-programs/oclib.c`, part 2**