CMPS 104A Assignment 5: Code Generation

Due on Friday Dec 6, 23:59:59.

1. Overview

Given, the annotated abstract syntax tree, it is now possible to generate the target code. The target language is usually an intermediate language which 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.

SYNOPSIS

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

All of the requirements of previous projects also apply to this project. The code generation will be done after the type checking and only if there were no type errors. For any input file called *program*`.oc` generate an intermediate language file called *program*`.oil` and also compile this file using `gcc` to get an executable called `program`.

2. The `oil` 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.

2.1 Types in the intermediate language

Unlike most assembly languages, `oil` is typed as to basic data types: 8-bit unsigned byte (`ubyte`); 32-bit two's complement signed integer (`int`); and a pointer type which is denoted with a star (`*`) and thus compatible to C. The types in `oc` and their corresponding types in `oil` are shown in Table 2.

oc type	oil type
bool	ubyte
char	ubyte
int	int
string	ubyte *
TYPEID	struct TYPEID *
bool[]	ubyte *
char[]	ubyte *
int[]	int *
string[]	ubyte **
TYPEID[]	struct TYPEID **

Table 1: Types in oc and their corresponding types in oil.

2.2 Naming

Unlike oc, the intermediate language does not allow support scopes or shadowing. Assembly languages generally use a completely flat symbol table, so names need to be mangled in order to avoid clashing global names with various local scopes and other external code. Table **??** shows the fprintf format strings and arguments to be used to mangle a name.

Category	Format string	Arguments	Example
Global names	"__%s"	identifier	__global
Local names	"_%d_%s"	blocknr, identifier	_1_var
Control labels	"%s_%d"	keyword, counter	while_23
Temporary variables	"%c%d"	typechar, counter	i1, b2, p3

Table 2: Summary of name mangling including fprintf format strings and arguments. *typechar* is b for a *ubyte*, i for an *int*, p for a data pointer.

2.3 Expressions

Each node which represents an expression, like binop, call, etc. needs to be executed as a separate instruction/statement. Use temporary variables to store the intermediate results of each expression. For simple variables and constants, it is not strictly necessary to use a temporary variable. The constants null and false are emitted as 0, and true is emitted as 1.

The type of a temporary variable should be available as result of the type checking in the previous assignment. Either call your type checker again or save the computed type at each node during the type checking run.

```
int i;
char c;
i = 42 - 23 * 2;
c = "abc"[i+5];
```

Listing 1: Example expression in `oc`

```
int i1 = 23 * 2;
int i2 = 42 - i1;
__i = i2;
ubyte* p1 = "abc";
int i3 = __i + 5;
char c1 = p1[i3];
__c = c1;
```

Listing 2: `oil` code for the `oc` program in Listing 1

2.4 Control Structures

In low level languages, `goto` is used instead of control structure. The targets of these `goto`s are automatically generated labels using a running number, e.g. `if_1: else_1: while_1:`, etc.

For `if`, the condition will be evaluated. Depending on the resulting value, there might be a jump to the label for the *else* case. Otherwise, the code for the *if* case is executed, followed by a jump to the label for the end of the conditional. If there is no *else* case, just emit the *else* label without any code.

```
if (23 < 42) {
  g = 1;
} else {
  g = 2;
}
exit();
```

Listing 3: Example conditional statement

```
        ubyte b1 = 23 < 42;
        if (!b1) goto else_1;
        i1 = 1;
        __g = i1;
        goto fi_1;
else_1:;
        i2 = 2;
        __g = i2;
fi_1:;
        exit();
```

Listing 4: `oil` code for the `oc` program in Listing 3

For `while` the transformation is very similar. First, the condition needs to be checked. If is evaluates to false then exit/skip the loop by jumping to a label right after the whole loop. Otherwise all the statements in the body are executed, followed by a jump to the very beginning of the loop.

```
int count() {
  int n = 0;
  while (n < 10) {
    n = n + 1;
  }
}
return n;
```

Listing 5: Example while statement

```
int
count()
        int _1_n = 0;
while_1:;
        ubyte b1 = _1_n < 10;
        if (!b1) goto break_1;
        i1 = _1_n + 1;
        _1_n = i1;
        goto while_1;
break_1:;
        i2 = _1_n;
        return i2;
```

Listing 6: `oil` code for the `oc` program in Listing 5

3. Output Format

Independent of the actual input file, every `oil` file always starts with the same preamble:

```
#define __OCLIB_C__
#include "oclib.oh"
```

Listing 7: Preamble for all `oil` files

Then all structure definitions are traversed and output as described above, with the correct `oil` types, and fields properly indented.

```
struct foo {
        ubyte **string_array;
        struct foo *pointer;
};
```

Listing 8: Example struct in `oil`

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

```
ubyte *__mystring;
int __exit_status;
```

Listing 9: Global variable declarations in `oil`

Then all functions are output with their parameters and statements properly indented as shown in Listing 10.

```
int
__add(
        int _1_a,
        int _1_b)
{
        int i1 = _1_a + _1_b;
        return i1;
}
```

Listing 10: Example function in `oil`

Finally a single function with the signature `void __ocmain ()` is output, containing all global initializations and the statements of the root program.

4. Example code

```
#define __OCLIB_C__
#include "oclib.oh"

ubyte **__argv;
int __argi;

void __ocmain ()
{
        ubyte **p1 = __getargv();
        __argv = p1;
        __argi = 1;
while_1:;
        ubyte *p2 = __argv[__argi];
        ubyte b3 = p2 != 0;
        if (!b3) goto break_2;
        ubyte b4 = __argi > 1;
        if (!b4) goto else_3;
        __putc (' ');
        goto fi_4;
else_3:;
fi_4:;
        ubyte *p4 = __argv[__argi];
        __puts (p4);
        int i5 = __argi + 1;
        __argi = i5;
        goto while_1;
break_2:;
        __endl ();
}
```

Listing 11: Expected `.oil` file for the `14-ocecho.oc` test program.

5. Compiling the intermediate code

If the code generation was done correctly, the resulting `oil` file includes valid C code which can be compiled with `gcc` and linked to the `oc` standard library which is provided at

```
:/Assignments/oil-examples/oclib.c
```

Invoking the compiler with the `system` function could be similar to

```
system("gcc -g -o program -x c program.oil oclib.c")
```

If this reports an error then there is a problem with the generated code. Otherwise, you get an executable program which corresponds to the provided `.oc` input file.

6. Grading

There are 60 points in total.

(5) Good overall coding style (identifiers, indentation, comments)
(12) Abstract syntax tree traversal code which emits code for expressions and statements
(4) Code for name mangling.
(4) Code to convert `while` and `if` statements to jumps and labels.
(3) Code generation for functions and their parameters.
(2) Code to create the `__ocmain` function.

If your program did not compile, ignore the following...

(6) Non-zero exit code for `oc` programs starting with `9x` and for `nosuch.oc`, zero exit code for all other test files (-1 pt/error)
(4) `.str`, `.tok`, `.ast`, and `.sym` files created as per previous projects
(8) `.oil` files created with assembler/SSA-style intermediate code
(2) `.oil` files do not contain `while` or `if`
(2) `.oil` files correctly indented with 8 spaces for instructions
(5) Executables (`01-hello`, `10-hundred`) created (-1 pt/missing)
(1) Running `./01-hello` prints "Hello, world!".
(1) Running `./10-hundred` counts from 1 to 101.
(1) Running `./53-insertionsort Lorem ipsum dolor sit amet` prints "Lorem amet dolor ipsum sit".

7. What to Submit

Submit `README`, `Makefile`, `scanner.l`, `parser.y`, and all of the header and C++ implementation files. Do not submit the file generated by `flex` or `bison`. To submit, log into the Linux Lab, either locally or by ssh, and

```
$ cd /path/to/my/files/
Check that README (and optionally PARTNER) exists
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.build
Check if there are any errors building oc
$ /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments/grading/mk.tests
Check if there are any unexpected results
$ submit cmps104a-wm.f13 asg5 README Makefile ... ...
```