CSE 302: Compilers | Lab 2

Generating Three-Address Code

Out: `2020-09-17` **Due: `2020-09-24 23:59:59`**

1 INTRODUCTION

In this lab you will go from the parse trees you built in lab 1 for the `BX0` language to the TAC intermediate language. You will use one (or both) of the *maximal munch* algorithms that you saw in lecture 2.

This lab will be assessed. It is worth 10% of your final grade. You must work in groups of 2 or 3.

Covid-19

Observe all health guidelines—particularly the interpersonal distancing guidelines—when working in groups. Use a video-conferencing platform such as *Zoom* or *Google Meet* to collaborate.[1] If a physical meeting is unavoidable, try to do it outdoors, keep it short, and wear masks.

2 THREE-ADDRESS CODE (TAC)

The target of your compiler for lab 2 is the *three-address code* (TAC) language that will be specified in this section. TAC comes equipped with an input syntax that will be described in sec. 2.2. You will be given a standalone interpreter for TAC, written in Python, that can execute a file containing a sequence of TAC instructions in this syntax. The interpreter is also usable as a library from within Python, without needing to generate textual TAC code and re-parsing it. The library is described in sec. 2.3.

2.1 TAC *Building Blocks: Temporaries and Instructions*

TEMPORARIES The equivalent of variables of `BX0` in TAC are *temporaries*. All temporaries represent 64-bit integers in two's complement representation. In this lab we will only use *anonymous temporaries*, which are temporaries of the form `%`n where n is a natural number (i.e., $n \in \{0, 1, 2, \dots\}$). Some examples: `%1`, `%42`, `%999`. TAC also supports *named temporaries*, but we will not use them for now.

Temporaries do not have to be declared, and each temporary may be read from and written to in arbitrarily many instructions. If a temporary is read from before being written to at least once, it is a runtime error which will abort the execution of the TAC program with an error message.

INSTRUCTIONS Every instruction of TAC (with one exception) contains the following components:

- An *opcode*, which is unique for every instruction type. The full list of TAC opcodes is given in figure 1, together with their interpretations.
- Zero, one, or two *argument temporaries*, which is/are read from *before* the instruction is executed.
- Zero or one *result temporary*, which is/are written to *at the end* of the instruction. If the instruction triggers a runtime error, then the result temporary is not written to — this is not relevant for this lab, but will become important eventually.

[1] *Google Meet*, unlike *Zoom*, has no duration restrictions for meetings with multiple people.

Instruction	Description
`Td = const K;`	Set temporary `Td` to the value `K`.
`Td = copy Ts;`	Copy the value of `Ts` to `Td`.
`Td = binop Tl, Tr;`	Compute the value of binary operator $binop \in \{$`add`, `sub`, `mul`, `div`, `mod`, `and`, `or`, `xor`, `shl`, `shr`$\}$ applied to the values of `Tl` (left operand) and `Tr` (right operand) and store the result in `Td`.
`Td = unop Ts;`	Compute the value of a unary operator $unop \in \{$`neg`, `not`$\}$ applied to the value of `Ts` and store the result in `Td`.
`print Ts;`	Print the value of `Ts` to the standard output.
`nop;`	No effect; doesn't read from or write to any temporaries.

Figure 1: The TAC Instructions. Here, `Ts`, `Td`, `Tl`, and `Tr` stand for TAC temporaries and `K` stands for a 64-bit signed integer constant.

PROGRAMS A TAC program is an ordered sequence of TAC instructions. It's execution model is very simple: each instruction in sequence, starting from the first instruction, is executed serially. The execution of the program ends successfully if every instruction is successfully executed.

2.2 TAC *Syntax and the* TAC *Interpreter (`tac.py`)*

The purpose of this lab is to compile a `BX0` program to a TAC program. While debugging your compiler, you may also find it useful to hand-write some TAC programs to test parts of your compiler such as the optimization passes described in section 3.3. The lexical tokens and grammar of TAC are shown in fig. 2. A TAC ⟨program⟩ is a sequence of zero or more TAC ⟨instr⟩uctions. Although whitespace is ignored, it is a good convention to keep each individual instruction on a line by itself.

You will be given a TAC interpreter, `tac.py`, that can read and execute TAC program files written in the above syntax. Here is a sample TAC program and the way to execute it with `tac.py`:

```
// In file: test.tac
%0 = const 10;
%2 = const 2;
%3 = mul %2, %0;
%5 = mul %3, %3;
%6 = const 2;
%7 = div %5, %6;
%8 = const 9;
%9 = mul %8, %7;
%10 = mul %9, %7;
%11 = const 3;
%12 = mul %11, %7;
%13 = add %10, %12;
%14 = const 8;
%15 = sub %13, %14;
print %15;
```

```
$ python3 tac.py -v test.tac            # each -v increases verbosity
// %0 = const 10;
// %2 = const 2;
// %3 = mul %2, %0;
// %5 = mul %3, %3;
// %6 = const 2;
// %7 = div %5, %6;
// %8 = const 9;
// %9 = mul %8, %7;
// %10 = mul %9, %7;
// %11 = const 3;
// %12 = mul %11, %7;
// %13 = add %10, %12;
// %14 = const 8;
// %15 = sub %13, %14;
// print %15;
360592
```

2.3 *Using the* TAC *Interpreter from Python*

The TAC interpreter is also usable as a library from within Python. To load it, simply state:

```
TEMP   ::= %(0|[1-9][0-9]*|[A-Za-z][A-Za-z0-9_]*)
NUM64  ::= 0|-?[1-9][0-9]*                                  (numerical value ∈ [−2^63, 2^63))
BINOP  ::= (add|sub|mul|div|mod|and|or|xor|shl|shr)
UNOP   ::= (neg|not)
PRINT  ::= 'print'     CONST ::= 'const'     COPY ::= 'copy'     NOP ::= 'nop'
COMMA  ::= ','         EQ    ::= '='         SEMICOLON ::= ';'
comment ::= //[^\n]*\n?     whitespace ::= [ \t\f\v\r\n]                    (ignored)
```

⟨program⟩ ::= ϵ | ⟨instr⟩ SEMICOLON ⟨program⟩

⟨instr⟩ ::= TEMP EQ CONST NUM64 | TEMP EQ COPY TEMP
| TEMP EQ UNOP TEMP | TEMP EQ BINOP TEMP COMMA TEMP
| PRINT TEMP | NOP

Figure 2: Tokens and grammar of the TAC language..

```
from tac import Instr, execute
```

The names of temporaries are just represented as strings. For example, the name of the temporary `%42` in Python is `'%42'`.

The main class of note is the `Instr` class that has the following definition:

```
class Instr:
    def __init__(self, dest, opcode, arg1, arg2):
        # dest is None or a temporary
        self.dest = dest
        # opcode is a valid opcode (string)
        self.opcode = opcode
        # arg1 and arg2 are None or temporaries
        #   (except for const, where arg1 can be an int)
        self.arg1, self.arg2 = arg1, arg2

    def __repr__(self):
        # return an S-expression representation of the instruction;
        # used by the Python interpreter to print Instr instances
        ...

    def __str__(self):
        # turn the instruction into a string; used by print()
        # and f-strings
        ...
```

A TAC program is just a sequence (think: a list) of `Instr` instances. Given such an instruction sequence called `prog`, say, you can print it out to a file named `'test.tac'` as follows:

```
with open('test.tac', 'w') as tac_file:
    print(*prog, sep='\n', file=tac_file)
```

A TAC program can also be executed by using the `execute()` function of the `tac` library:

```
def execute(prog, show_instr=False, only_decimal=True):
    """
    Execute the instructions in `prog' one by one starting from the
    first instruction.

    If show_instr == True, also print the instruction being executed.
    This is intended as a debugging aid.

    If only_decimal == False, the print() statement will print the
    value simultaneously in decimal, hexadecimal, and binary.
    Otherwise it will only print it in decimal. Note that
    the hexadecimal and binary versions will print the underlying
    two's complement representation. This too is intended as a
    debugging aid.
    """
    ...
```

Here is an example:

```
>>> from tac import Instr, execute
>>> execute([Instr('%1', 'const',  42,  None),
...          Instr('%2', 'const', -130, None),
...          Instr('%3', 'mul',   '%1', '%2'),
...          Instr(None, 'print', '%3', None)],
...         show_instr=True,
...         only_decimal=False)
...
// %1 = const 42;
// %2 = const -130;
// %3 = mul %1, %2;
// print %3;
-5460 0xffffffffffffeaac 0b1111111111111111111111111111111111111111111111111101010101010100
>>>
```

3 DELIVERABLES

Start from your solution to lab 1, which contains a parser for BX0 programs. Your task in this lab is to write a compiler pass that goes from the parse tree produced by your parser (i.e., the `Node` tree/forest) to TAC programs. Remember that your compiler needs to be *correct*, meaning that any TAC program you produce must have exactly the same behavior as the source BX0 program.

3.1 *Structure of the Project*

You are given an archive `cse302_lab2.zip` containing `tac.py` as described above (and the PLY library). Your compiler will be run from a file called `bx0tac.py` (if you are using Python) or an executable called `bx0tac.exe` if using a different programming language. This program should accept a single BX0 file as its command line argument, whose name will have the extension `.bx`. On a successful compilation, it should output the equivalent TAC code to a file that changes the extension to `.tac`. For example:

```
$ python3 bx0tac.py test.bx          # should produce test.tac
$ python3 tac.py -vv test.tac        # execute the generated .tac
```

Describe any other features of your compiler in the `README.md` file. You may choose to add other command line options, which you can easily manage using the `getopt` standard library in Python.

3.2 *Your Tasks*

Your compiler should perform the following steps:

- [80% of score] Use one of the maximal munch algorithms to transform `BX0` to TAC. These algorithms are described in significant detail in lecture 2 (delivered on 14/09). However, note that these slides used poor software engineering techniques such as global variables (e.g., `last_temp`) and stateful functions (e.g., `get_temp_for()`). You should improve the design by suitably encapsulating this global state, either in the form of extra arguments to the munching functions, or in the form of a class with suitable methods and instance variables. Try to avoid global variables as much as possible so that the complexity of your compiler remains manageable.
- [20% of score] Perform both the *simple copy-propagation* and *dead copy elimination* passes that are described in sec. 3.3 below.

Remember to test your compiler thoroughly. Any test cases can be placed in the `tests/` subdirectory. Get into the habit of writing tests at the same time as features and running the tests on every modification to catch regressions. A little effort now will save you a lot of headache later.

3.3 *Optimizations*

Both maximal munch algorithms presented in lecture produce some redundant `copy` instructions. Many of them can be removed by the two simple optimization algorithms presented at the end of lecture 2. Indeed, with the exception of highly pathological examples, you will find that these optimizations get rid of all `copy` instructions entirely from the result of the maximal munches. (You can, of course, hand-write TAC that will have uneliminable `copy` instructions.)

Simple Copy-Propagation Consider the instruction: `Td = copy Ts;` In all subsequent instructions, everywhere the temporary `Td` is used as an argument temporary, one might as well use `Ts` instead without any change to the behavior of the program. Of course, this can only be done as long as the equivalence between `Ts` and `Td` persists! (Recall the corner cases from the lecture.)

You can perform this copy propagation *in place* by modifying the `.arg1` and `.arg2` fields of instructions, or create a completely new instruction list with modified instructions. The choice is yours, but the latter may be easier to write regression tests for since you can compare the before and after variants.

Dead Copy Elimination After copy-propagation, it may be the case that there are no remaining instructions that use the value stored in `Td` by the `copy` instruction. In this case the `copy` instructions is said to be *dead* and it can safely be removed. In order to facilitate this removal, you may temporarily replace these dead copies with the `nop` instruction, and then in a subsequent pass get rid of all the `nop`s at once (using, e.g., `functools.filter()`).

Stand-Alone Optimizer If you want, you can write your optimization passes as a stand-alone program that reads a `.tac` file and outputs an optimized TAC program. To do this, you may find the function `tac.load_tac()` useful. If you do this, mention it in `README.md`.