CSC 3F002 EP: Compilers | Lab 02

Generating Three Address Code (TAC)

Out: `2025-09-13`

1 INTRODUCTION

The goal of this lab is to modify the parser of Lab 01 so that it generates an Abstract Syntax Tree (AST) and then to write a pair of *maximal munch* stages to transform the AST for straightline BX to TAC.

2 STRAIGHTLINE BX

The source language for this lab is the same as the one of Lab 01. There are no control structures such as loops, conditionals, or functions, nor is there any way to produce any output except by means of the `print` statement. This language works with data of a single type, signed 64 bit integers in 2's complement representation. This can represent all integers in the range $[-2^{63}, 2^{63})$.

The relevant grammar for this fragment of BX is shown in figure 1. Every BX program is produced by the ⟨program⟩ non-terminal, which in this case will be a single procedure called `main` that takes no arguments. Within the body of this function is a sequence of local variable declarations (given by the ⟨vardecl⟩ non-terminal) and of statements (given by the ⟨stmt⟩ non-terminal). Statements are of two kinds: assignment statements (⟨assign⟩) and print statements (⟨print⟩). Both of these make use of expressions (⟨expr⟩).

⟨program⟩ ::= `"def" "main" "(" ")" "{"` ⟨stmt⟩* `"}"`

⟨stmt⟩ ::= ⟨vardecl⟩ | ⟨assign⟩ | ⟨print⟩

⟨vardecl⟩ ::= `"var" IDENT "="` ⟨expr⟩ `":" "int" ";"`
⟨assign⟩ ::= `IDENT "="` ⟨expr⟩ `";"`
⟨print⟩ ::= `"print("` ⟨expr⟩ `")" ";"`

⟨expr⟩ ::= `IDENT` | `NUMBER`
| ⟨expr⟩ `"+"` ⟨expr⟩ | ⟨expr⟩ `"-"` ⟨expr⟩
| ⟨expr⟩ `"*"` ⟨expr⟩ | ⟨expr⟩ `"/"` ⟨expr⟩ | ⟨expr⟩ `"%"` ⟨expr⟩
| ⟨expr⟩ `"&"` ⟨expr⟩ | ⟨expr⟩ `"|"` ⟨expr⟩ | ⟨expr⟩ `"^"` ⟨expr⟩
| ⟨expr⟩ `"<<"` ⟨expr⟩ | ⟨expr⟩ `">>"` ⟨expr⟩
| `"-"` ⟨expr⟩ | `"~"` ⟨expr⟩
| `"("` ⟨expr⟩ `")"`

`IDENT` ::≈ `/[A-Za-z][A-Za-a0-9_]*/` (except reserved words)
`NUMBER` ::≈ `/0|[1-9][0-9]*/` (value must fit in 63 bits)

Figure 1: Grammar of the straightline fragment of BX. Whitespace and line comments are ignored.

The Abtract Syntax Tree In this lab, you will construct a class hierarchy that represents the various constructs of the BX programming language (programs, statements and expressions). We give below an example on how this hierarchy could be defined:

```
import abc
import dataclasses as dc

@dc.dataclass
class AST(abc.ABC):
  pass

@dc.dataclass
class Expression(AST):
  pass

@dc.dataclass
class Statement(AST):
  pass

@dc.dataclass
class VarExpression(Expression):
  name: str

@dc.dataclass
class NumberExpression(Expression):
  value: int

# etc ...
```

Then, you will modify the parser so that it constructs the AST. For example:

```
def p_expression_var(p):
    """expr : name"""
    p[0] = VarExpression(name = p[1])

def p_expression_int(p):
    """expr : NUMBER"""
    p[0] = NumberExpression(value = p[1])

# etc ...
```

We remind you that parser functions have a single argument, `p`, that represents an enumeration of the components of the production, with `p[0]` standing for the left hand side of the production and `p[1]`, `p[2]`, … standing for the right and sides.

Finally, you will have to implement extra checks, namely you will have to verify that a program

- only uses of previously declared variable, and
- do not do multiple declarations of the same variable (i.e. with the same name).

3 THREE ADDRESS CODE (TAC)

The main goal of this lib is to transform an AST to a TAC program. 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 that are placed in the procedure named `@main`. (Note the change from `main` in BX to `@main` in TAC– this will be explained later when we discuss global vs. local symbols.)

```
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/

⟨program⟩ ::= "proc @main:" (⟨instr⟩ ";") *

⟨instr⟩ ::= TEMP "=" "const" NUM64
          | TEMP "=" "copy" TEMP
          | TEMP "=" UNOP TEMP
          | TEMP "=" BINOP TEMP "," TEMP
          | "print" TEMP
          | "nop"                                      (does nothing)
```

Figure 2: Tokens and grammar of the TAC language (not directly relevant for this lab)

While TAC has a grammar, for this lab you will work instead with a JSON representation of TAC programs. The TAC programs you generate in this lab will have the following overall structure.

```
[ { "proc": "@main", "body": [ ⟨instr⟩, ⟨instr⟩, … ] } ]
```

Each instruction is represented by a JSON object with three fields, `opcode`, `args`, and `result`. The valid opcodes are all shown in figure 2. The `args` field is a tuple of temporaries or numbers, with temporaries represented as JSON strings (i.e., `"%42"` etc.), while numbers are represented as JSON ints. The `result` field is either a temporary or `null`. Here are some examples of instructions in JSON form:

```
{"opcode": "const", "args": [42],         "result": "%0"}
{"opcode": "copy",  "args": ["%0"],       "result": "%1"}
{"opcode": "mul",   "args": ["%2", "%1"], "result": "%3"}
{"opcode": "neg",   "args": ["%8"],       "result": "%9"}
{"opcode": "print", "args": ["%15"],      "result": null}
{"opcode": "nop",   "args": [],           "result": null}
```

An example of a complete TAC file in JSON form is shown in figure 3.

On the Moodle, you are provided a compiler pass from TAC in JSON form to assembly and executable, using GCC as the assembler and linker. This compiler pass is called `tac2x64.py` for the x64 architecture (resp. `tac2arm.py` for the MacOS M1/2 architecture) and can be used as follows:

```
[ { "proc": "@main",
    "body": [
        {"opcode": "const", "args": [10], "result": "%0"},
        {"opcode": "copy", "args": ["%0"], "result": "%1"},
        {"opcode": "const", "args": [2], "result": "%2"},
        {"opcode": "mul", "args": ["%2", "%1"], "result": "%3"},
        {"opcode": "copy", "args": ["%3"], "result": "%4"},
        {"opcode": "mul", "args": ["%4", "%4"], "result": "%5"},
        {"opcode": "const", "args": [2], "result": "%6"},
        {"opcode": "div", "args": ["%5", "%6"], "result": "%7"},
        {"opcode": "copy", "args": ["%7"], "result": "%1"},
        {"opcode": "const", "args": [9], "result": "%8"},
        {"opcode": "mul", "args": ["%8", "%1"], "result": "%9"},
        {"opcode": "mul", "args": ["%9", "%1"], "result": "%10"},
        {"opcode": "const", "args": [3], "result": "%11"},
        {"opcode": "mul", "args": ["%11", "%1"], "result": "%12"},
        {"opcode": "add", "args": ["%10", "%12"], "result": "%13"},
        {"opcode": "const", "args": [8], "result": "%14"},
        {"opcode": "sub", "args": ["%13", "%14" ], "result": "%15"},
        {"opcode": "print", "args": ["%15"], "result": null}
    ]
  }
]
```

Figure 3: A complete TAC program in JSON form

```
$ python3 tac2x64.py source.tac.json
$ gcc -o source.exe source.s
$ ./source.exe
…
```

4 TASKS

Your task is to extend your compiler so that: 1) it builds an AST from the parser, 2) it checks that the program is correct (all variables are declared, no multiple declarations of the same variable), and 3) it transforms that AST to TAC.

Remember that your compiler needs to be *correct*, meaning that any TAC program you produce must have exactly the same behavior as the source BX program.

GIVEN On the Moodle you can find `starter.zip` that contains a collection of valid and invalid BX programs.

DELIVERABLES You must build a program called `bxc.py`. This program will be given a single BX program in the command line, and it should produce the corresponding TAC file (in JSON format).

```
$ python3 bxc.py source.bx     # produces source.tac.json
```

If you implement *both* maximal munches, you program should allow the user to pick between them using the `--tmm` or `--bmm` flags.

```
$ python3 bxc.py --tmm source.bx     # top down
$ python3 bxc.py --bmm source.bx     # bottom up
```

Some Hints

- Create a mechanism to get a *fresh* temporary that has definitely not been used anywhere before. There are many ways to do this, such as with a global counter.
- In your maximal munch implementations, you will need to maintain a mapping from the local variables in your `main()` function to anonymous temporaries. You can use the collection of ⟨vardecl⟩s of the body of `main()` to seed this mapping.
- Give some thought to your class hierarchies for expressions and statements. Both categories will expand significantly in the coming weeks.