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

⟨vardecl⟩ ::= `"var"` `IDENT` `"= 0 : int;"`

⟨stmt⟩ ::= ⟨assign⟩ | ⟨print⟩
⟨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

CSE 302: Compilers | Lab 1

Generating Three Address Code (TAC)

Out: `2023-09-07`

---

1 INTRODUCTION

This lab is primarily intended to get you set up with your development libraries and associated tools. The goal of this lab is to write a pair of *maximal munch* stages to transform the AST for straightline BX to TAC.

2 STRAIGHTLINE BX

SOURCE LANGUAGE The source language for this lab is a drastically simplified fragment of BX. 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, followed by a sequence of statements given by the ⟨stmt⟩ non-terminal. Every variable declaration declares an integer variable initialized to 0. Statements are of two kinds: assignment statements (⟨assign⟩) and print statements (⟨print⟩). Both of these make use of expressions (⟨expr⟩).

JSON Representation For this lab, you will be not need to worry about reading BX source directly. Instead, you will start from a representation of the BX source file in the form of a JSON object, which makes it trivial to load in almost any programming language. The JSON representation of a BX file always has the following external structure:

```
{
  "provenance": "/path/to/sourcefile.bx", // the source file that was parsed
  "source": "...",                        // the original text of the source file
  "ast": ...                              // the AST (described here)
}
```

The `provenance` and `source` fields may safely be ignored. They can be used to create fancy error messages (not required!) as described below. We will primarily concern ourselves with the `ast` field, which contains a compact representation of the BX abstract syntax tree (AST).

Most AST subtrees are encoded as JSON lists of length 2 with the following shape:

```
[ <constructor>, { "position": position, payload... } ]
```

where `<constructor>` & `payload` describes the actual AST structure, and `position` is a dictionary of the form:

```
{ "start": [start_line, start_col], "end": [end_line, end_col] }
```

In this lab you can completely ignore the `position` tuple if you want. However, here is is what it means: the AST node is located in the original source between character located at line `start_line` & column `start_col` and character located at line `end_line` & column `end_col`.

The `ast` field of the overall JSON object will be a list of *toplevel declarations* of BX. For this lab, there is only one toplevel declaration, which is the declaration of the *main* function. This is encoded in the JSON object as demonstrated below, where we use ☒ to indicate the `location` components (which, recall, you can ignore for this lab). Most of the complexity of this definition comes from supporting the full BX language. However, in this lab you can take the above as a fixed structure and focus on the contents of the body of the procedure, indicated by $\cdots$ above. (Pay close attention to the brackets and nesting!)

```
[ [ "<decl:proc>",
    { "position": ☒,
      "name": [ "<name>", { "position": ☒, "value": "main" } ],
      "arguments": [],
      "returntype": null,
      "body": [ ··· ] } ] ]
```

The body of the procedure, shown by $\cdots$ above, consists of a sequence of local variable declarations followed by a sequence of statements.

- Local variables: all such declarations have the following shape:

```
[ "<statement:vardecl>",
  { "position": ⊠,
    "name": [ "<name>", { "position": ⊠, "value": "x" } ],
    "type": [ "<type:int>", { "position": ⊠ } ],
    "init": [ "<expression:int>", { "position": ⊠, "value": 0 } ] } ]
```

The only compoenent that will vary in this lab is the name of the variable, which is `x` in the above shape. The initial value and type are fixed at `0` and **`int`** respectively.

- Assignment statement (⟨assign⟩): these statements have the following shape:

```
[ "<statement:assign>",
  { "position": ⊠,
    "lvalue": [
      "<lvalue:var>",
      { "position": ⊠,
        "name": [ "<name>", { "position": ⊠, "value": "x" } ] }
    ],
    "rvalue": ⟨expr⟩ } ]
```

The left hand side of the assignment, here shown with the variable `x`, will change from one assignment statement to the next. The right hand side of the assignment is shown with ⟨expr⟩ above, whose JSON shapes are specified below.

- Print statements (⟨print⟩) have the following shape:

```
[ "<statement:eval>",
  { "position": ⊠,
    "expression": [
      "<expression:call>",
      { "position": ⊠,
        "target": [ "<name>", { "position": ⊠, "value": "print" } ],
        "arguments": [ ⟨expr⟩ ] } ] } ]
```

The argument to the `print` statement is shown with ⟨expr⟩ above. The JSON for `print` statements is more complex than strictly necessary, but it anticipates the extension of to the full BX to come, when `print()` will become a part of the BX *standard library* of functions.

Expressions (⟨expr⟩) are represented by any of the following JSON object shapes.

- Variables are represented by the following shape:

```
[ "<expression:var>",
  { "position": ⊠,
    "name": [ "<name>", { "position": ⊠, "value": "x" } ] } ],
```

(shown here with the example of the variable `x`).

- Numbers are represented by the following shape:

```
[ "<expression:int>", { "position": ⊠, "value": 42 } ]
```

(shown here with the example of 42).

- Unary operator applications are represented by the following shape:

```
[ "<expression:uniop>",
  { "position": ⊠,
    "operator": [ "<name>", { "position": ⊠, "value": ⟨op⟩ } ],
    "argument": [ ⟨expr⟩ ] } ]
```

  where ⟨op⟩ ∈ {"opposite", "bitwise-negation"}.

- Binary operator applications are represented by the following shape:

```
[ "<expression:binop>",
  { "position": ⊠,
    "operator": [ "<name>", { "position": ⊠, "value": ⟨op⟩ } ],
    "left": ⟨expr⟩,
    "right": ⟨expr⟩ } ]
```

  where ⟨op⟩ ∈ {"addition", "substraction", "multiplication", "division", "modulus", "bitwise-xor", "bitwise-or", "bitwise-and"}.

Loading an AST in JSON Format To load JSON from a file named `file.json` in Python, you can proceed as follows:

```
with open('file.json', 'r') as fp:
    js_obj = json.load(fp)
    # js_obj is a representation of the object using Python data structures
```

It is *strongly* recommended that you don't use the `js_obj` object directly, since their specific form is very closely tied to the JSON input that will evolve over the course of the labs. Instead, build a class hierarchy of expressions and statements such as:

```
class Expression:
    pass

class ExpressionVar(Expression):
    def __init__(self, name):
        self.name = name

class ExpressionInt(Expression):
    def __init__(self, value):
        self.value = value

class ExpressionUniOp(Expression):
    def __init__(self, operator, argument):
        self.operator = operator
        self.argument = argument

class ExpressionBinOp(Expression):
    # ... (similar)
```

Then, write a recursive JSON loader function that transforms the JSON object into the corresponding element of the class hierarchy. For example, for loading expressions from JSON objects into the hierarchy

```
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)

depicted earlier, you can write a function that has the following structure.

```
def json_to_name(js_obj):
    return js_obj[1]['value']

def json_to_expr(js_obj):
    if js_obj[0] == '<expression:var>':
        return ExpressionVar(json_to_name(js_obj[1]['name']))
    if js_obj[0] == '<expression:int>':
        return ExpressionInt(js_obj[1]['value'])
    if js_obj[0] == '<expression:uniop>':
        operator = json_to_name(js_obj[1]['operator'])
        argument = json_to_expr(js_obj[1]['argument'])    # recursive call
        return ExpressionUniOp(operator, argument)
    if js_obj[0] == '<expression:binop>':
        # ...
    # ...
    raise ValueError(f'Unrecognized <expression>: {js_obj[0]}')
```

3 THREE ADDRESS CODE (TAC)

The purpose of this lab is to compile a BX program 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.)

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

```
[ { "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

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.

You are provided a compiler pass from TAC in JSON form to AMD64 assembly and executable using GCC as the assembler and linker. This compiler pass is called `tac2asm.py` and can be used as follows:

```
$ python3 tac2asm.py tacfile.tac.json
tacfile.tac.json -> tacfile.s
tacfile.s -> tacfile.exe
$ ./tacfile.exe
…
```

4 TASK: MAXIMAL MUNCH

Your task in this lab is to write a compiler pass that goes from an AST represented in JSON format 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.

Setup Create a private Github repository with a name such as `cse302labs` and assign me (`@strub`) as a collaborator. Then, start working in a subdirectory such as `cse302labs/1`. Please *do not* create new repositories for each lab as that will drastically increase my grading workload.

Given On the Moodle you can find `starter.zip` that contains the TAC compiler phase (`tac2asm.py`) and a collection of example ASTs in JSON format in the `examples/` subdirectory. Start by unpacking this in your `cse302labs/1` directory.

Deliverables You must build a program called `ast2tac.py`.[1] This program will be given a single AST in JSON format in the command line, and it should produce the corresponding TAC file (also in JSON format).

```
$ python3 ast2tac.py astfile.json     # produces astfile.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 ast2tac.py --tmm astfile.json     # top down
$ python3 ast2tac.py --bmm astfile.json     # bottom up
```

Creating Fresh AST Files In order to test your compiler pass, you can directly write your test cases using the JSON format for ASTs. Altenatively, you can use the script `bx2ast.py` to convert BX source files to ASTs. Note that the script requires an working Internet connection.

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 at the top 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.

[1] If you're not using Python, make sure that your source compiles to `ast2tac.exe` or `ast2tac.sh`. Document this clearly in a `README.txt`.