

CSE 302: Compilers | Lab 2

A Basic Frontend

Out: 2023-09-14

1 INTRODUCTION

The purpose of this lab is to build a frontend for (a slight extensions of) the BX fragment that was used in lab 1. You will write a lexer, a parser, and a syntax checker for this BX fragment using off-the-shelf lexer and parser generators. For the specific case of Python, this document explains how to use the PLY library. For other languages you will need to do some research on your own for equivalent libraries.

2 THE BX FRAGMENT WE WILL USE

Like lab 1, the source language for this lab remains a simplified fragment of BX. There still are no control structures such as loops, conditionals, or functions, and you can still only output individual numbers by means of the `print` statement. You will also continue to use 64-bit signed integers as the sole value type. The main extensions from lab 1 are as follows;

- Variable declarations are now allowed to occur anywhere in the body of a function, not just at the start of the function.
- Variable initializers are allowed to be any expression—not just `0`—that can be built using the variables that have been declared earlier.

The complete grammar of BX is shown in figure 1. Note that every occurrence of text in quotes—things like `"this"`—will represent a single token that is generated by the lexer. The grammar is also presented in extended BNF as described in the lecture 2.

The table in figure 2 lists the operators, their arities, and their precedence values. Note that operators with higher precedence bind tighter, so `1 + 2 * 3` would implicitly stand for `1 + (2 * 3)` and not `(1 + 2) * 3`.

3 TASK: LEXER

To turn a BX source file into a sequence of tokens, you must write a lexical scanner. If you are using Python/PLY, this is achieved using the `ply.lex` module. As explained in the lecture, this amounts to performing the following:

1. Define a tuple called `tokens` that contains all the token categories (PLY calls them *types*). Each token category is just an ordinary Python string that, by convention, is in UPPERCASE.
2. For every token that doesn't have a payload—i.e., everything but identifiers and numbers—write a Python variable definition whose name is the name of the token category, prefixed with `t_`, and whose value is a regular expression string that matches the token.
3. For every token with a payload, write a Python function definition with a name prefixed with `t_`. The regular expression that matches the token will be given in the documentation string of the function, which is the first string that occurs in the body of the function. This function will have a

```

<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.

operator	description	arity	associativity	precedence
	bitwise or	binary	left	10
^	bitwise xor	binary	left	20
&	bitwise and	binary	left	30
<<, >>	bitwise shifts	binary	left	40
+, -	addition, subtraction	binary	left	50
*, /, %	multiplication, division, modulus	binary	left	60
-	integer negation	unary	-	70
~	bitwise complement	unary	-	80

Figure 2: BX operator properties. Operators with higher precedence value bind tighter.

single parameter that represents the token object. You can add a payload to the token object by modifying its `.value` attribute, and even change the token category by modifying its `.type` attribute. After suitably modifying the token object, you should then **return** it; not returning anything from the token function is equivalent to ignoring the token.

4. Any characters that must be filtered out of the input can be specified with the special variable definition `t_ignore`, which is an ordinary Python string. Any character that occurs in it is ignored.
5. The special token function `t_error(t)` can be used to handle illegal characters in the input.

The minimum information per token that is needed to create useful error messages is the character offset, which you can obtain with the `.lexpos` attribute of the token object. This is automatically managed for you by `PLY`. It is up to you whether you want to track line number information as well in the lexer. An example of how to do this is in the `PLY` documentation.

4 TASK: PARSER

To build the parser you should use the `ply.yacc` module of `PLY` as explained in the lecture. This requires the following steps.

1. Make sure that the tokens list, `tokens`, is visible at the top level of the Python file where you write the Parser. You may need a suitable `import` statement.
2. The parser is defined using functions that begin with `p_`. Each such parser function corresponds to one (or more) productions of the `BX` grammar. The parser function also has 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.

Each parser function must finish by *defining* `p[0]` in terms of `p[1]`, `p[2]`, It is pointless to **return** anything from parser functions as all such returned values are ignored by `PLY`.

3. The precedence table for operators is specified using the precedence tuple, which lists the associativity and precedence groups of operator tokens in the order of increasing precedence. An example was given in lecture 2.
4. The special error parser function `p_error` can be used to detect parse errors in the token stream.

As usual, you should read the detailed documentation about `PLY` parsers.

EXTENDED BNF The grammar of `BX` in figure 1 uses the extended BNF notation $\langle \text{stmt} \rangle^*$ to denote a sequence of zero or more $\langle \text{stmt} \rangle$ s. This operator is not present as is in `PLY` and needs to be encoded specially. In the general case, suppose you want to parse a production such as the following:

$\langle a \rangle ::= \langle b \rangle^*$

In `PLY`, this can be done succinctly by using the isomorphism $\langle b \rangle^* \equiv \epsilon \mid \langle b \rangle^* \langle b \rangle$.

```

def p_a(p):
    """a : bstar"""
    p[0] = p[1]

def p_bstar(p):
    """bstar :
    | bstar b"""
    if len(p) == 1:      # empty case
        p[0] = []
    else:                # nonempty case
        p[0] = p[1]
        p[0].append(p[2])

```

The other extended BNF operators can be handled in a similar way by making use of the equivalences $\langle b \rangle^+ \equiv \langle b \rangle \mid \langle b \rangle^+ \langle b \rangle$ and $\langle b \rangle^? \equiv \epsilon \mid \langle b \rangle$. In all these cases since the choices on the right have different lengths, you can use `len(p)` to find out which choice was matched by the parser.

Note that it would also have been OK to use the isomorphism $\langle b \rangle^* \equiv \epsilon \mid \langle b \rangle \langle b \rangle^*$ to define `p_bstar`, but in this case we would have to use prepending to a list to update `p[0]`. Prepending in Python is an $O(n)$ operation, while the `.append()` function takes $O(1)$ time (amortized).

5 TASK: SYNTAX CHECKING

The following checks should be performed on the output of the parser to make sure that the syntactic constraints of BX are respected.

1. There must be a single function named `main` with no arguments and no return type.
2. Every statement must be syntactically correct, meaning:
 - For assignments, the LHS should be a variable declared earlier, and the RHS should be a syntactically correct expression.
 - For print statements, the argument should be a syntactically correct expression.
 - For variable declarations, the variable must not already be declared, and the initial value should be a syntactically correct expression (without the variable currently being defined in scope).
3. For expressions:
 - Every variable already be declared with an earlier variable declaration.
 - Every number must fit in 63 bits, i.e., its value needs to be in $[0, 2^{63})$.

6 DELIVERABLES

You are given the file `starter.zip` consisting of:

- A collection of *correct* examples in the `examples/`
- A *regression suite* in `regression/` which consists of files that have lexing, parsing, or syntactic correctness errors. All of these files should be rejected. (Note: you should not rely on these tests to catch all your bugs. Write your own tests as well!)
- the PLY library (in the `py/` subdirectory).

You must deliver a frontend pass as a program called `bx2tac.py` that will read the BX source file specified in the command line and will output a TAC program in JSON format, just like in lab 1.

```
$ python3 bx2tac.py program.bx      # produces program.tac.json
```

You should build on the `ast2tac` program you wrote in lab 1. You should be able to reuse most of the code with very trivial modifications. In particular, there should be only minimal changes needed in the class hierarchy for representing the AST.

If the input program has errors, your program should output an error message such as:

```
$ python3 bx2tac.py regression/duplicate_vardecl.bx
regression/duplicate_vardecl.bx:line 6:Error:Duplicate declaration of variable "x"
regression/duplicate_vardecl.bx:line 3:Info:Earlier declaration of "x"
```

You have considerable freedom in the format of these error messages. You may choose to mimic the error message format of your favorite compiler. The example messages above are inspired by those of GCC. In case a BX program cannot be processed by your frontend, it should exit with an exit code of 1 (i.e., using `sys.exit(1)`) – after printing any error messages **to the standard output**.