Second Project: Parsing & AST

The second project requires you to implement a Parser for the MiniJava language and use it to build an abstract syntax tree (AST) as output.

Like the first project, you will also use the SLY library. Moreover, a series of classes representing all AST nodes will be provided.

❗ Advices

- *Read before doing*
  There are a lot of implementation details to be aware of. If you find them in your code without any context, debugging will be rather painfull.
- *Attention to comments in code blocks*
  The comments highlight the changes between code blocks.
- *Don't skip SLY's documentation*
  We'll discuss some aspects regarding SLY here, but it is by no means a replacement for the full-fledged documentation.
- *First, semantic rules, then, semantic actions*
  Don't write the semantic rules and build the AST simultaneously. First define the semantic rules and make sure they are valid. Then start building the AST by implementing the semantic actions.
- *Test Driven Development.*
  The unit tests are sorted by difficulty. When building the AST, do it test by test: instantiate only the nodes necessary for the first test to pass, then move onto the second, and so on.

📚 References

- BNF vs EBNF - A bit about BNF and its extended form.
- MiniJava EBNF grammar - The grammar you'll have to implement.
- SLY's parser documentation - Read it with care.
- MiniJava AST examples - Look at the in-out test files for each test.

Goal

Our goal in this project is essentially to feed a code snippet to our compiler and receive an AST as result.

Say we are given three global declarations. The first is a initialized integer, second is a uninitialized integer, and the third an integer matrix:

```
class Example1 {
  int a = 3 * 4 + 5, c;
  int [] b = new int[10];
}
```

Then, we wish to represent the given code as an AST like the following:

```
Program:
    ClassDecl: ID(name=Example1) @ 1:1
        VarDecl: ID(name=a) @ 2:5
            Type: int @ 2:1
            BinaryOp: + @ 2:9
                BinaryOp: * @ 2:9
                    Constant: int, 3 @ 2:9
                    Constant: int, 4 @ 2:13
                Constant: int, 5 @ 2:17
        VarDecl: ID(name=c) @ 2:20
            Type: int @ 2:1
        VarDecl: ID(name=b) @ 3:8
            Type: int[] @ 3:1
            NewArray: @ 3:12
                Type: int[]
                Constant: int, 10 @ 3:20
```

Parser

Let's take one step at a time.

First, we will see how we can parse a simple variable declaration into a tree-like representation.

Say we have the following input:

```
a = 3 * 4 + 5;
```

And our goal is to properly represent this computation using a tree-like notation:

```
('program', ('var_decl', 'a', ((3, '*', 4), '+', 5)))
```

Grammar

The first order of business it to define the grammar that will be able to parse our input code snippet.

Take a look at the grammar below:

```
program ::= var_decl

var_decl ::= ID EQUALS expr SEMI

expr ::= expr PLUS expr
       | expr TIMES expr
       | INT_LITERAL
```

We can easily implement this grammar using PLY.

Lexer

First, we need a lexer to tokenize our input. If we have as the input:

```
a = 3 * 4 + 5
```

The Lexer would return the following output

```
LexToken(ID,'a',1,0)
LexToken(ASSIGN,'=',1,2)
LexToken(INT_LITERAL,3,1,4)
LexToken(TIMES,'*',1,6)
LexToken(INT_LITERAL,4,1,8)
LexToken(PLUS,'+',1,10)
LexToken(INT_LITERAL,5,1,12)
```

Semantic Rules

Next step is to implement each semantic rule of our grammar using SLY's parsing tools and the lexer we've just created:

```
from sly import Parser

class MJParser(Parser):
     """I am a parser for the MiniJava language."""
    tokens = MJLexer.tokens
    start = "program"

    def __init__(self, debug=True):
        """I create a new MJParser."""
        self.debug = debug
        self.mjlex = MJLexer(self._lexer_error)

        # Keeps track of the last token given to yacc (the lookahead token)
        self._last_yielded_token = None

    def parse(self, text, debuglevel=0):
        """I parse a input code snippet."""
        self._last_yielded_token = None
        return super().parse(self.mjlex.tokenize(text))

    def _lexer_error(self, msg, line, column):
        # use stdout to match with the output in the .out test files
        print("LexerError: %s at %d:%d" % (msg, line, column), file=sys.stdout)
        sys.exit(1)

    def _parser_error(self, msg, coord=None):
        # use stdout to match with the output in the .out test files
        if coord is None:
            print("ParserError: %s" % (msg), file=sys.stdout)
        else:
            print("ParserError: %s %s" % (msg, coord), file=sys.stdout)
        sys.exit(1)


    # The decorator represents the semantic rule
    @_("var_decl")
    def program(self, p):
        # The body is the semantic action for the given rule.
        return ('program')

    # tokens are used in the same way as defined in the lexer
    @_("ID EQUALS expr SEMI")
    def var_decl(self, p):
      pass

    #  When there are several possible rules for the same non-terminal,
    # you can either separate them into different methods, or list the
    # rules in the same method
    @_(
      "expr PLUS expr", # 1º Semantic rule to expr
      "expr TIMES expr" # 2º Semantic rule to expr
    )
    def expr(self, p):
      pass
    @_("INT_LITERAL") # 3º Semantic rule to expr
    def expr(self, p):
      pass


def main():
    # create argument parser

    parser = argparse.ArgumentParser()
    parser.add_argument("input_file", help="Path to file to be parsed", type=str
    args = parser.parse_args()

    # get input path
    input_file = args.input_file
    input_path = pathlib.Path(input_file)

    # check if file exists
    if not input_path.exists():
        print("ERROR: Input", input_path, "not found", file=sys.stderr)
        sys.exit(1)

    parser = MJParser(debug=False)

    # open file and print ast
    with open(input_path) as f:
        ast = parser.parse(f.read())
        # print(parser.log.text)
        ast.show(buf=sys.stdout, showcoord=True)

if __name__ == "__main__":
    main()
```

Once you have written your semantic actions and semantic rules correctly, you will be able to sucessfully generate an LALR parser for our grammar.

Now we should be able to parse semantically valid inputs while also raising errors for semantically **invalid** inputs:

```
# checking for errors on invalid input
parser.parse("a = ;")
```

The output:

```
Error near symbol ";"
```

When parsing an invalid input, our error funcion p_error was correctly called, and when parsing a valid input, no error was raised:

```
# checking if valid input passes
parser.parse("a = 3 * 4 + 5;")
```

So far, so good.

Semantic Actions

Although we can correctly parse the code, It's of no use if there is no output from it.

Our next step then is to add semantic *actions* for each of the semantic *rules* in our parser so that it may return something useful to us.

A *semantic action* is essentially the action to be made once a semantic rule is identified/reduced.

In SLY's case, the semantic action for each rule is defined in the function's body.

Let's create a simple abstract sintax tree of our input code using semantic actions:

```
from sly import Parser

class MJParser(Parser):
     """I am a parser for the MiniJava language."""
    tokens = MJLexer.tokens
    start = "program"

    # Parser functions
    ...

    # Solve ambiguity
    precedence = ()

    # The decorator represents the semantic rule
    @_("var_decl")
    def program(self, p):
        # The body is the semantic action for the given rule.
        # Once an 'program' production is identified,
        # SLY will execute this function's body as the semantic action.
        return ('program', p.var_decl)

    # Lexer tokens are used in the semantic rule
    @_("ID EQUALS expr SEMI")
    def var_decl(self, p):
      # Once an 'var_decl' production is identified, SLY will execute
      # this function's body as the semantic action.
      return ('var_decl', p.ID, p.expr)

    #  When there are several possible rules for the same non-terminal you can e
    @_(
      "expr PLUS expr", # 1º Semantic rule to expr
      "expr TIMES expr" # 2º Semantic rule to expr
    )
    def expr(self, p):
      # Some functions, like this one, may handle multiple production rules.
      # This is useful if different rules should trigger the same semantic
      # action. In this case, regardless of the operator, we wish to generate
      # the same tuple, so the same semantic action is used.
      # Each element in the production is a attribute in p
      # expr    PLUS    expr
      # p[0]    p[1]    p[2]
      return (p[0], p[1], p[2])

    @_("INT_LITERAL") # 3º Semantic rule to expr
    def expr(self, p):
      return (p.INT_LITERAL)


...

if __name__ == "__main__":
    main()
```

Now, whenever a semantic rule is reduced in our parser, it will return whatever value the semantic action returns in the function.

Suppose the semantic action returns the following tuple T from the expression E:

```
E: "a = 3 * 4 + 5;"
T: ('program', ('var_decl', 'a', (3, '*', (4, '+', 5))))
```

The tuple T is a very simple abstract tree.

This tree is wrong though: the sum operation preceeds the multiplication. This would return `3 * (4 + 5)` instead of `(3 * 4) + 5`.

Why is it doing this you ask? Because we ignored the `shift/reduce` conflicts.

Shift/Reduce Conflicts

⭐ *Tip: see parser.debug file to better visualize this*

Shift/reduce conflicts essentially mean the grammar is ambiguos. In other words, there is more than one valid AST for the same code.

Let's dissect the cause of the conflict.

Once the parser reaches the `+` token, it encounters the following situation:

```
ID EQUALS expression TIMES expression . PLUS
```

The `.` indicates the top of our parser's stack and the `PLUS` token is the lookahead token. The parser has two options here:

- Shift the `PLUS` token into the stack.

  ```
  ID EQUALS expression TIMES expression PLUS .
  ```

- Reduce the top of the stack using the semantic rule `expression ::= expression TIMES expression`.

  ```
  ID EQUALS expression . PLUS
  ```

If you run a parser with this conflicts you will see this message:

> WARNING: 4 shift/reduce conflicts

❗ **Important**: This message does not appear in pytest, only when running the tests individually and only after the first time running a new grammar. To check if your grammar is free of conflicts, we advise you to check the end of the parser.debug file that has the list of conflicts. The parser.debug file will be generated automatically after the template runtime errors are resolved. In addition, if there is more than 1 shift/reduce conflict or any reduce/reduce conflict, 3 points will be deducted from the total lab grade.

The parser is not sure if it should shift the lookahead or reduce the top of the stack.

It also says there are 4 conflicts. This is because the same conflict happens in 4 different scenarios:

```
ID EQUALS expression TIMES expression . PLUS
ID EQUALS expression PLUS expression . TIMES
ID EQUALS expression TIMES expression . TIMES
ID EQUALS expression PLUS expression . PLUS
```

So, why did the parser not crash when it found this conflicts?

Well, not all shift/reduce conflicts are bad as long as we know how to resolve them.

However, SLY always resolves shift/reduce conflicts by shifting instead of reducing.

In this case, shifting is the wrong action, because it causes `(4, '+', 5)` to be generated before `(3, '*', 4)`.

To fix this, we must explicitly tell SLY how to resolve these conflicts.

Operator Precedence

⭐ *Tip: there are some good examples of this in SLY's doc*

SLY allows use to easily resolve these common shift/reduce conflicts between operators by using a special `precedence` variable.

Let's see how we can use the `precedence` variable to force SLY to prioritize `TIMES` tokens over `PLUS` tokens:

```
from sly import Parser


class MJParser(Parser):
    tokens = MJLexer.tokens
    start = "program"

    # Parser functions
    ...

    # Solve ambiguity
    precedence = (
      ('left', 'PLUS'),
      ('left', 'TIMES'),
    )


    # Parser rules
    ....
```

After generating the new parser, there is already an indication of success: the shift/reduce warnings are gone.

The simple abstract tree produced is:

```
('program', ('var_decl', 'a', ((3, '*', 4), '+', 5)))
```

Another very basic syntax tree. But a correct one this time.

Position Tracking

A last desirable feature for our parser is to track positions.

By tracking positions, we are able to know which part of the code generate which tree element.

To track this positions we'll use special `Cood` class and a helper function `_token_cood`:

```
from sly import Parser

class Coord:
    """Coordinates of a syntactic element. Consists of:
    - Line number
    - (optional) column number, for the Lexer
    """

    __slots__ = ("line", "column")

    def __init__(self, line, column=None):
        self.line = line
        self.column = column

    def __str__(self):
        if self.line and self.column is not None:
            coord_str = "@ %s:%s" % (self.line, self.column)
        elif self.line:
            coord_str = "@ %s" % (self.line)
        else:
            coord_str = ""
        return coord_str

class MJParser(Parser):
    tokens = MJLexer.tokens
    start = "program"

    # Other Parser functions
    ...


    def _token_coord(self, p):
        last_cr = self.mjlex.text.rfind("\n", 0, p.index)
        if last_cr < 0:
            last_cr = -1
        column = p.index - (last_cr)
        return Coord(p.lineno, column)

    # Semantic actions and rules
    ...

    @_("ID EQUALS expr SEMI")
    def var_decl(self, p):
      return ('var_decl', p.ID, p.expr, str(self._token_coord(p)))

    @_(
      "expr PLUS expr", # 1º Semantic rule to expr
      "expr TIMES expr" # 2º Semantic rule to expr
    )
    def expr(self, p):
      return (p[0], p[1], p[2], str(self._token_coord(p)))

    @_("INT_LITERAL") # 3º Semantic rule to expr
    def expr(self, p):
      return (p.INT_LITERAL, str(self._token_coord(p)))
```

The positions are in our tree now:

```
E: "a = 3 * 4 + 5;"

# position format: @ line:column
T:  ('program',
    ('assignment',
      'a',
        ((3, '*', 4, '@ 1:7'), '+', 5, '@ 1:11'),
      '@ 1:1'))
```

Neat. Now its a basic syntax tree with coordinates.

With this info we know, for example, that the operation `(3, '*', 4, '1:7')` came from line 1 and column 7 of our code.

EBNF to BNF

🛑 **SLY does not support EBNF, only BNF.**

That is a bit of a catch that we haven't discussed yet.

What this means is the kleene start, kleene cross, and question mark operators will not work with SLY grammar rules.

So any rule in the grammar that uses the `+`, `*`, or `?` operators must be converted to BNF first.

Take a look below to se how EBNF rules may be converted to BNF.

EBNF:

```
animal ::= dog+
         | cat*
         | mouse?
```

BNF:

```
animal ::= dog_kcross
         | cat_kstar
         | mouse_opt

dog_kcross ::= dog_kcross dog
             | dog

cat_kstar ::= cat_kstar cat
             | empty

mouse_opt ::= mouse
            | empty

empty ::=
```

So, for the previous example, we write the SLY function as:

```
    @_("dog_kcross", "cat_kstar", "mouse_opt")
    def animal(self, p): # The production is the function name
      # Semantic action here
      pass

    @_("dog_kcross dog", "dog_kcross")
    def dog_kcross(self, p):
      pass

    @_("cat_kstar cat", "empty")
    def cat_kstar(self, p):
      pass

    @_("mouse", "empty")
    def mouse_opt(self, p):
      pass

    @_("")
    def empty(self, p):
      pass
```

You'll need to manually apply these conversions to use the MiniJava grammar in SLY.

Upgrading to MiniJava Grammar

The simplified example was useful to explain the parsing basics.

Now let's go back to our main goal:

```
class Example1 {
  int a = 3 * 4 + 5, c;
  int[] b = new int[10];
}
```

The old grammar is not enough to parse this. It has no concept of arrays nor typed declarations.

Let's increment the grammar by simplifying some of the MiniJava grammar rules:

```
<program> ::= {<class_declaration>}+

<class_declaration> ::= "class" <identifier> "{" {<compound_declaration>}* "}"

<compound_declaration> ::= <type_specifier> <init_declarator_list> ";"

<init_declarator_list> ::= <init_declarator>
                         | <init_declarator_list> "," <init_declarator>

<init_declarator> ::= <identifier>
                    | <identifier> "=" <assignment_expression>

<type_specifier> ::= "int"
                   | "int" "[" "]"


<assignment_expression> ::= <binary_expression>
                          | "new" "int" "[" <INT_LITERAL> "]"

<binary_expression> ::= <INT_LITERAL>
```

```
                        | <binary_expression> "*" <binary_expression>
                        | <binary_expression> "+" <binary_expression>
```

Next step: update our lexer and parser with the new grammar.

Shift/Reduce Conflict for ID

The Semantic Rules section shows how to create lists that accept zero or more elements (in the example, the list is called cat_kstar). In other words, rules that are followed by * in the grammar. However, in the grammar rule `{<statement>}*` that is inside `<compound_statement>`, creating the list as shown results in a shift/reduce conflict.

Note that the compound statement rule allows two consecutive lists * (kstar lists):

```
<compound_statement> ::= "{" {<compound_declaration>}* {<statement>}* "}"
```

If you follow the rules in the first kstar list of the compound declaration, you can see that it can expect an identifier as the first rule:

```
<compound_declaration> ::= <type_specifier> <init_declarator_list> ";"


<type_specifier> ::= <identifier>
                  | ...
```

The other rules are omitted using "..." since they are not important for analysis.

On the other hand, if you follow the rules in the second kstar list of the statement, you can see that it can **also** expect an identifier as the first rule:

```
<statement> ::= <expresssion_statement>
             | ...

<expression_statement> ::= <expression> ";"

<expression> ::= <assignment_expression>
              | ...

<assignment_expression> ::= <unary_expression> "=" <assignment_expression>
                         | ...

<unary_expression> ::= <postfix_expression>
                    | ...
```

```
<postfix_expression> ::= <primary_expression>
                       | ...

<primary_expression> ::= <identifier>
                       | ...
```

Because both rules start with identifier there is a shift /reduce conflict, and, since the parser resolves using shift, the statement starts to be interpreted as a compound declaration resulting in an error in the parser in the tests that cover this case.

To resolve this conflict, simply change the statement rule to one of these two options:

1. Split the kstar rule into two, making a rule that accepts empty and kcross (lists with +):

```
@_("statement_kcross", "empty")
def statement_kstar(self, p):
    pass
@_("statement_kcross statement", "statement")
def statement_kcross(self, p):
    pass
```

2. Use recursion on the right side:

```
@_("statement statement_kstar", "empty")
def statement_kstar(self, p):
   pass
```

Abstract Syntax Tree

> Abstract syntax trees are data structures that better represent the structure of the program code than the parse tree. An AST can be edited and enhanced with information such as properties and annotations for each element it contains.

We've shown how to use SLY to parse a grammar and generate a tree-like structure from it.

Tuples, however, are a bit limited. Let's upgrade our AST to classes.

Each node in our tree will be represented by a class. **Keep in mind that these classes will be given to you.**

Your job is to use the parser to connect these classes/nodes using the SLY parser to effectively buid the AST.

Building Declarations

Our goal here is to leverage the parse tree to build the AST.

Whenever a reduction occurs in the parser, a semantic action is executed.

With this knowledge we can return nodes that represent the reduction in question instead of simple tuples.

Let's build the AST for the code below:

```
class Example1 {
  int[] b = new int[10];
}
```

Which should look like:

```
Program:
    ClassDecl: ID(name=Example1) @ 1:1
        VarDecl: ID(name=b) @ 3:8
            Type: int[] @ 3:1
            NewArray: @ 3:12
                Type: int[]
                Constant: int, 10 @ 3:20
```

There are 7 explicit nodes in the AST above: `Program`, `ClassDecl`, `VarDecl`, `Type`, `NewArray`, `ID`, and `Constant`.

Also, some of these nodes inherit from special helper classes.

Before continuing, we must defined these nodes and its helpers.

Helpers

Below is a list of the helper methods/classes:

- `class Node` - Every nodes inherts this class. It holds some base functionality for all nodes.
- `def represent_node` - Used to dump each node as a string.

These helpers don't deserve much atention since you won't be using then directly.

Let's just move on.

AST Nodes

With the helpers defined, we can proceed to define the AST nodes.

We must defined `Program`, `ClassDecl`, `VarDecl`, `Type`, `NewArray`, `ID`, and `Constant` nodes:

```
class Program(Node):
    """Node that represent the MiniJava Program"""

    attr_names = ()

    def __init__(self, class_decls: list[ClassDecl], coord: Coord = None):
        """
        :param class_decls: program's class declarations.
        :param coord: code position.
        """
        self.class_decls = class_decls
        self.coord = coord

    def children(self):
        return tuple(
            (
                (f"class_decls[{idx}]", class_decl)
                for idx, class_decl in enumerate(self.class_decls or [])
            )
        )

class ClassDecl(Node):
    """Node representing a Class Declaration"""

    attr_names = ("name",)

    def __init__(
        self,
        name: ID,
        extends: ID,
        var_decls: list[VarDecl],
        method_decls: list[MethodDecl],
        coord: Coord = None,
    ):
        """
        :param name: Class name.
        :param extends: Name of the extended class.
        :param var_decls: List of var declarations.
        :param method_decls: List of method declarations.
        """
        self.name = name
        self.extends = extends
        self.var_decls = var_decls
        self.method_decls = method_decls
        self.coord = coord

    def children(self):
        decls = (
            ([self.extends] if self.extends is not None else [])
            + self.var_decls
            + self.method_decls
        )
        return tuple(((f"decls[{idx}]", decl) for idx, decl in enumerate(decls o

class VarDecl(Node):
    """Node that represents a variable declaration"""

    attr_names = ("name",)

    def __init__(self, type: Type, name: ID, init, coord: Coord = None):
        """
        :param type: variable primitive type.
        :param name: variable name.
        :param init: initialization value.
        :param coord: code position.
        """
        self.type = type
        self.name = name
        self.init = init
        self.coord = coord

    def children(self):
        nodelist = []
        if self.type is not None:
            nodelist.append(("type", self.type))
        if self.init is not None:
            nodelist.append(("init", self.init))
        return tuple(nodelist)

class Type(Node):
    """Node representing a type specifier"""

    attr_names = ("name",)

    def __init__(self, name: str, coord: Coord = None):
        """
        :param name: type name (int, char, ...).
        :param coord: code position.
        """
        self.name = name
        self.coord = coord

    def children(self):
        return ()

class NewArray(Expr):
    """Expression representing a New Array allocation."""

    attr_names = ()

    def __init__(self, type: Type, size: Expr, coord: Coord = None):
        """
        :param type: Array type (char[] or int[]).
        :param size: Array size.
        :param coord: code position.
        """
        self.type = type
        self.size = size
        self.coord = coord

    def children(self):
        nodelist = []
        if self.type is not None:
            nodelist.append(("type", self.type))
        if self.size is not None:
            nodelist.append(("size", self.size))
        return tuple(nodelist)

class ID(Node):
    """Node representing an identifier"""

    attr_names = ("name",)

    def __init__(self, name: str, coord: Coord = None):
        """
        :param name: ID unique name.
        :param coord: code position.
        """
        self.name = name
        self.coord = coord

    def children(self):
        return ()
```

```
class Constant(Expr):
    "Node representing a constant"

    attr_names = ("type", "value")

    def __init__(self, type: str, value, coord: Coord = None):
        """
        :param type: constant type.
        :param value: constant value.
        :param coord: code position.
        """
        self.type = type
        self.value = value
        self.coord = coord

    def children(self):
        return ()
```

Each node has a set of attributes that represent one of two things:

- A intrinsic property of the node (like code position).
- Child nodes (like the list of `ClassDecl` in the `Program` node)

Each node also has a `attr_names` class attribute. This is used to print the nodes.

Just a reminder that these nodes are already defined in the project's template.

MiniJava Declarations

A MiniJava type consists of a basic type declaration or a char/int array type.

In `int[] b`, for example is a int array. When initialized with `new int[10]` it produces a `NewArray` node in the AST.

In our AST, `VarDecl` represents the basic var declaration, `ClassDecl` the class declaration, `MethodDecl` a method declaration and `MainMethodDecl` is the method declaration of MiniJava main method.

> Note that in the MiniJava grammar var declarations must be written before any other rules in the method body.

```
/*
Valid code in MiniJava, var is declared
before the print statement.
*/
```

```
class Example1 {
  public static void main(String[] args) {
    int var = 2;
    print("Hello");
  }
}


/*
Invalid code in MiniJava, var is declared
after the print statement.
The compiler must emit a parser error.
*/
class Example2 {
  public static void main(String[] args) {
    print("Hello");
    int var = 2;
  }
}

/*
Valid code in MiniJava, var is declared
before the print statement.
In MiniJava, a value can be assigned to
a variable after its declaration.
*/
class Example3 {
  public static void main(String[] args) {
    int var;
    print("Hello");
    var = 2;
  }
}
```

Building the AST

Now we know how declarations are parsed and structured with AST nodes.

Next step is to join these two informations in order to build the AST.

Let's try to buit the ast for the input code:

```
class Example1 {
  int[] b = new int[10];
}
```

```
from sly import Parser


class MJParser(Parser):
    tokens = MJLexer.tokens
    start = "program"

    # Parser functions
    ...

    # Solve ambiguity
    precedence = (
      ('left', 'PLUS'),
      ('left', 'TIMES'),
    )

    @_("class_declaration_list")
    def program(self, p):
        # The root node of AST
        return Program(class_decls=p.class_declaration_list, coord=None)

    # Class declaration list must return a python list
    @_("class_declaration")
    def class_declaration_list(self, p):
        return [p.class_declaration]

    @_("class_declaration_list class_declaration")
    def class_declaration_list(self, p):
        p.class_declaration_list.append(p.class_declaration)
        return p.class_declaration_list



    @_(
        "CLASS identifier LBRACE compound_declarations RBRACE"
    )
    def class_declaration(self, p):
        return ClassDecl(
            name=p.identifier,
            var_decls=p.compound_declarations,
            coord=self._token_coord(p),
        )

    @_("empty")
    def compound_declarations(self, p):
        return []
```

```
@_("compound_declarations compound_declaration")
def compound_declarations(self, p):
    compound_declarations = p.compound_declarations + p.compound_declaration
    return compound_declarations

@_("type_specifier init_declarator_list SEMI")
def compound_declaration(self, p):
    for var_decl in p.init_declarator_list:
        var_decl.type = p.type_specifier
    return p.init_declarator_list

@_("init_declarator")
def init_declarator_list(self, p):
    return [p.init_declarator]

@_("init_declarator_list COMMA init_declarator")
def init_declarator_list(self, p):
    p.init_declarator_list.append(p.init_declarator)
    return p.init_declarator_list

@_("identifier")
def init_declarator(self, p):
    return p[0]

@_("<identifier> ASSIGN <assignment_expression>")
def init_declarator(self, p):
    return VarDecl(
        type=None, name=p.identifier, init=p.assignment_expression, coord=se
    )

@_("ID")
def identifier(self, p):
  return ID(name=p.ID, coord=self._token_coord(p))

@_("INT")
def type_specifier(self, p):
    return Type(name=p[0], coord=self._token_coord(p))

@_("INT LBRACKET RBRACKET")
def type_specifier(self, p):
    return Type(name="int[]", coord=self._token_coord(p))

@_("binary_expression")
def assignment_expression(self, p):
    return p.assignment_expression
@_("NEW INT LBRACKET INT_LITERAL RBRACKET")
def assignment_expression(self, p):
    arr_size = Constant(type="int", value=p.INT_LITERAL, coord=self._token_c
    return NewArray(type="int", size=arr_size, coord=self._token_coord(p))

@_("INT_LITERAL")
def binary_expression(self, p):
    return Constant(type="int", value=p.INT_LITERAL, coord=self._token_coord

@_(
    "binary_expression TIMES binary_expression",
    "binary_expression PLUS binary_expression",
)
def binary_expression(self, p):
    return BinaryOp(op=p[1], left=p[0], right=p[2], coord=self._token_coord(

@_("")
def empty(self, p):
    return None
```

The AST result:

```
Program:
    ClassDecl: ID(name=Example1) @ 1:1
        VarDecl: ID(name=b) @ 3:8
            Type: int[] @ 3:1
            NewArray: @ 3:12
                Type: int[]
                Constant: int, 10 @ 3:20
```

Wow. It works.

Let's discuss some bits of the code.

Other Considerations

We're almost done here. There are just a few more considerations.

Single Expressions in Expression Lists

Expression lists with a single expression are handled differently.

To not "pollute" AST, single expressions are not wrapped in a `ExprList` node.

See the example below:

```
class Main {
    public static void main(String[] args) {
        String mc = "Susy";
        print("Hello", mc, ". Welcome to MC921");
    }
}
```

And its AST:

```
Program:
    ClassDecl: ID(name=Main) @ 1:1
        MainMethodDecl: @ 2:5
            ID: args @ 2:38
            Compound: @ 2:44
                VarDecl: ID(name=mc) @ 3:16
                    Type: String @ 3:9
                    Constant: String, "Susy" @ 3:21
                Print: @ 4:9
                    ExprList: @ 4:15
                        Constant: String, "Hello" @ 4:15
                        ID: mc @ 4:24
                        Constant: String, ". Welcome to MC921" @ 4:28
```

To do this, we just write the code in the parser's `expression` production:

```
@_("expression COMMA assignment_expression")
def expression(self, p):
    expr_list = p.expression
    if not isinstance(expr_list, ExprList):
        expr_list = ExprList(exprs=[p.expression], coord=p.expression.coord)
    expr_list.exprs.append(p.assignment_expression)
    return expr_list
```

› The If-Else Shift/Reduce Conflict

↳ 1 cell hidden

› Why does my Parser have 373 Shift/Reduce Conflicts?

↳ 1 cell hidden

I Have a Reduce/Reduce Conflict

↳ 1 cell hidden

Wrapping it Up

↳ 1 cell hidden