CPL to QUAD Compilator (20364) [OUI - 2021] by Alexander Yermakov

```
def boolfactor(self
    return
if __name__ == '__main__':
    io = CPQInput()
    lexer = CPQLexer()
    parser = CPQParser()
    optimizer = QUADOPtimizer()

    ou_file = io.read_ou(sys.argv)
    if ou_file:
        ast = parser.parse(lexer.tokenize(ou_file))
        code = ast.interpret()
```

Python CPQ

- Uses SLY for lexical & semantic analysis
- Builds custom AST for direct translations
- Impelements backpatching & code optimization

General information

Included files:

- **cpq.py** - Program entry
- **cpq_io_handler.py** - Deals with file handling
- **cpq_lexer.py** - SLY Lexer
- **cpq_parser.py** - SLY Parser
- **cpq_ast.py** - Abstract syntax tree
- **cpq_interpreter.py** - AST interpreter
- **cpq_scope.py** - Symbol table
- **err_handler.py** - Collects errors
- **quad_optimizer.py** - Optimizes QUAD code

Basic program flow:

1. Check command line arguments to find *.ou file
2. Tokenize & Parse it according to language rules
3. While parsing generate AST of node objects
4. Traverse AST using Interpreter
5. Generate Symbol Table & update it along the way
6. Convert each node to proper QUAD command
7. Keep track of labels & use backpatching
8. Load finalized translation to list of code lines
9. Optimize the resulting code & write it to *.qud file

Implementation details

- **Error Handling**
    - External class that is imported by the rest
    - Alternatie to exception mechanism, but can be adapted to work with it
    - Gathers all the errors, through all stages, storing them in chronological order
- **Lexer/Parser**
    - LARL parser, similar to bison
    - Uses SLY built in functionality for build abstract syntax trees
    - Tries different methods for recovery, for maximal error collection
- **Abstract syntax tree**
    - Uses custom objects for every node type
    - Each node implements interpret() function for traversal
    - Tree root is being returned upon completion of parsing algorithm
- **Interpreter**
    - Interprets abstract syntax tree nodes directly to QUAD
    - Up until the end, stores labels as indexes in virtual label dictionary
    - Generates extra code for IF, ELSE statements for further easy NOT backpatching
    - Reports, but tries to ignore errors as much as possible to produce debug code
- **Scope/Symbol Table**
    - Uses OrderedDict to keep track of used variables
    - Can generate temporary variables, reserves 'tr', 'ti', 'sw' prefixes for them
    - Tries to reuse previously allocated temporary variables as much as possible
    - Used to alanyze all factor types, be it numbers or variables
- **Code Optimization**
    - Optimizations are performed on final QUAD code (since we don't generate IR code)
    - Unaware of the rest of the program, theoretically can be used on any QUAD code
    - Able to delete unnecessary JUMP statements & redundant variable allocations
    - Suffers from some 're-parsing' overhead, since iterates over QUAD code to analyze it

How to run it

Standart way:

- Reqiures Python >= 3.8
- Use command line call with single argument, to run the program:

```
$ python cpq.py <file_name>.ou
```

- On success, it will:
  - Generate *.qud file, with compiled QUAD code
- Otherwise, it will:
  - Try to recover, by skipping problematic tokens
  - Output ecountered problems to std.error
  - If possible save partial parse to *.dump file

Alternative way:

- You can package *.py files to *.exe, making them portable for windows/mac eco-systems
- Easisest way would be to use PyInstaller & Auto PY to EXE package
- To install via PyPi, type:

```
$ pip install auto-py-to-exe
```

- To run it's web interface, use:

  ```
  $ auto-py-to-exe
  ```