# CS335: COMPILER DESIGN

# Milestone 3

# HDS PYTHON COMPILER

Harsh Bihany (210406)
Danish Mehmood (210297)
Siddhant Suresh Jakhotiya (211030)

April 18, 2024

# 1 Introduction

**HDS** is a compiler that converts `statically-typed Python` language to `x86_64` assembly. It uses `Flex` for lexical analysis and `Bison` for parsing the `Python3.12.3` grammar. It then uses `graphviz` to create a `DOT` script to produce the `Abstract Syntax Tree` of a given program. The `global symbol table`, which is parent to the symbol tables of all the classes of a program, in turn parents to the symbol tables of their member functions, is created while the AST is produced from parsing the program.

The Abstract Syntax Tree and Symbol Table are used to generate the three-access code for the input program. Finally, the 3AC representation is translated to x86_64 bit assembly code.

The compiler is written in C++ and can be used by running the commands as given in the following section.

# 2 Running the files

The implementation of Parser and Lexer is in Flex+Bison, and is written in C++. Further, DOT is used to create a graphical representation of the AST and exporting it to a PDF file. The symbol table and the 3AC code have been output in `.csv` and `.txt` files respectivily and the assembly code for `x86_64` is written to the file `x86_code.s`. User should install all the above (if not already installed) by running the following commands:

```
$ sudo apt-get update
$ sudo apt-get install flex
$ sudo apt-get install bison
$ sudo apt-get install graphviz
```

A script file `pyrun.sh` has been made to help run the program against a test case. The command line options provided by our program are as follows:

- `-i` or `-input`: Filename from which the compiler reads the Python program.
- `-o` or `-output`: Filename to which the compiler outputs the `DOT` file.
- `-t` or `-tac`: Filename to which 3AC code will be output.
- `-a` or `-asm`: Filename to which `x86_64` assembly code will be output.
- `-s` or `-sym`: Filename to which symbol table will be output.
- `-v` or `-verbose`: Prints additional checkpoints to show progress.
- `-h` or `-help`: Shows the usage for the script.

Each of the commands mentioned above is optional and can be utilized in any sequence. The script takes `../tests/test1.py` as default input and `graph.pdf`, `symbol_table.csv`, `three_address_code.txt` as the default output for the graph, symbol_table and 3AC respectivily , does not print additional messages and does not perform `make clean` after execution.

An example of a successful default execution is as follows:

```
$ make
```

Doing this will create `pycompiler.o`
Now to run this file do:

```
$ ./pycompiler.o -input ../tests/test3.py -o graph1.dot -verbose
  ↪ -a asm.s
```

This example command line code will create a `.asm` file. To execute this file do:

```
$ ./pyrun asm.s graph1.dot
```

The above execution will also output the AST graph as `graph1.pdf`.

# 3 Codebase

The codebase can be found at our GitLab ID `hds-cs335`, inside the repository `python-compiler-2024`. The whole codebase is inside the directory `milestone3` which is on the `main` branch of the repository mentioned before. The file structure of the directory `milestone3` is as follows:

```
milestone3/
|-- src/
|   |-- include
|       |-- _3AC.hpp
|       |-- node.hpp
|       |-- symbol_table.hpp
|       |-- x86.hpp
|   |-- Makefile
|   |-- main.cpp
|   |-- _3AC.cpp
|   |-- symbol_table.cpp
|   |-- node.cpp
|   |-- pylex.l
|   |-- pyparse.y
|   |-- pyrun.sh
|   `-- x86.cpp
|-- tests/
|   |-- test1.py
|   |-- test2.py
|   |-- test3.py
|   |-- test4.py
|   `-- test5.py
`-- doc/
    `-- hds_milestone3_report.pdf
```

The same file structure has also been following in the zip file uploaded to **Canvas**.

Description for different files is as follows:

- **pylex.l** : This `Flex` file houses the code for the lexical scanner. It scans the input file and passes the `tokens` to the `parser`. It also keeps track of the line number for error reporting.
- **pyparse.y** : This `Bison` file contains the code for the parser. It drives the lexer and generates the `Abstract Syntax Tree`.
- **pyrun.sh** : This file is the `script` that helps run the compiler against a testcase. It also implements the command line options `-input`, `-output`, `-verbose` and `-help`. It concludes with a `make` command that executes the `Makefile`, which then runs the program with the supplied arguments to generate the output PDF.
- **main.cpp** : This `C++` file has the code to drive the `parser` and generate the `dot` script.
- **_3AC.cpp/_3AC.hpp** : These files contain the definition of the `struct 3AC` and also include the global **IR** vector. Generated **3AC** codes are added to this vector, which is then used to produce the final IR output file.

- **symbol_table.cpp/symbol_table.hpp** : These files contain the definitions for the `struct symbol_table`, symbol table entries, and other necessary fields for the symbol_table. This setup aids in creating the global symbol table.
- **node.cpp / node.hpp** : These files house the function definitions of `struct node` methods and functions for `AST` generation.
- `x86.hpp/x86.cpp`: This file houses the code to generate x86 assembly code from 3AC code.
- Additionally, the files `test1.py`, `test2.py`, `test3.py`, `test4.py`, `test5.py` are the test cases provided by us (these were not explicitly asked for in the problem PDF).

# 4 Features supported

The following language features are supported by **HDS** compiler:

- Primitive data types (e.g., `int`, `str`, `float`, and `bool`). Till **Milestone 2** `coercion` among the data types was also supported. For **Milestone 3**, all the data types are considered 8 bytes in length and `bool` are implicitly converted to `int`.
- Support has been added for 1D lists containing `int`, `str`, and `class` objects. However, as specified in **Milestone 1**, dictionaries, tuples, and sets are not included.
- Basic operators supported are as follows:
  - Arithmetic operators: $+$, $-$, $*$, $/$, $//$, $\%$, $**$
  - Relational operators: $==$, $!=$, $>$, $<$, $\geq$, $\leq$
  - Logical operators: `and`, `or`, `not`
  - Bitwise operators: $\&$, $|$, $\hat{}$, $\sim$, $<<$, $>>$
  - Assignment operators: $=$, $+=$, $-=$, $*=$, $/=$, $\%=$, $**=$, $\&=$, $|=$, $\hat{}=$, $<<=$, $>>=$

    All the operators have been supported for `int` data type. $=$ and **comparison ops** are also supported for `str` data type.
- Control flow via `if-elif-else`, `for`, `while`, `break`, and `continue` (`pass`, `do-while`, `switch` have been ignored as instructed in **Milestone 1**)
- Support for recursion
- Support for the library function `print()` for only printing the primitive Python types, one at a time.
- Support for classes and objects, including multilevel inheritance and constructors. Multiple inheritance (i.e., a class can have only one parent class) have been ignored.
- Methods and method calls for class objects have been supported

- Static polymorphism via method overloading have also been supported.
- In addition to basic **type checking**, our compiler also verifies the return type of functions. Specifically, if the actual return type of a function does not match the expected return type declared, the compiler will generate an error.

# 5 Additional Features

- Implemented the feature allowing class variables to be declared within any function (not only `__init__` function), enhancing flexibility in our compiler.
- Enhanced the compiler to support executing multiple function calls within a single line, streamlining code execution.
- Added functionality to the compiler to return lists and class objects from functions, broadening the range of return types.
- Our compiler is equipped to manage **block-level scopes** (e.g., `if-elif-else`, `while`, etc.). However, this feature has been disabled and the relevant code commented out as per instructions not to implement such handling.
- The compiler has been enhanced to support operations within function arguments. Specifically, it can now interpret and execute expression statements used as arguments. Again, `coercion` has been implemented here also but won't be needed as the test cases won't be having `float` data types
- Although it was specified that all elements of `list` would be homogeneous in the test cases, our compiler still performs **type checking** to ensure that the elements are of the same type.
- Our compiler supports multiple assignment operations for primitive data types such as `int` and `str`. Support for the `float` data type has been excluded as per the specifications of **Milestone 3**.
- In Python3, chained comparison operators are evaluated sequentially, unlike in C. For example, the expression `a == b > c` is interpreted as both `a == b` and `b > c`. Our compiler also supports this behavior accurately.

# 6 Changes in 3AC after Milestone 2

- We have added a few new quadruple types to improve and simplify the handling of the `str` data type, as well as assignment operations for `list` and `class objects` for easier `x86_64` assembly code generation. The following 3AC instructions have been introduced:
  - **Q_BINARY_STR**: Generates 3AC for binary comparison operations on the `str` data type.

  - **Q_PRINT_STR**: Produces 3AC for executing the `print` function specifically for `str` data type.
  - **Q_STORE**: Generates 3AC for storing the result of an assignment operation computation into a class member or list element.

- As mentioned before, for uniformity and simplification of register allocation, the size of each data type has been standardized to **8 bytes**. Consequently, when calculating offsets, the initial position is now multiplied by a multiple of 8 (rather than 4 which was the data type size earlier).

- To designate the function `if __name__ == "__main__"` as the entry point for assembly code generation, its 3AC handling has been updated.

# 7 x86_64 assembly code creation

We have written the assembly code using the AT&T syntax as we compiled it with `gcc`. The x86 code is made of subroutines obtained from the 3AC. It begins with the `.data` and `.globl` segments, followed by the `.text` segment.

# 8 Unsupported Features

All the necessary features (i.e., as asked in the **Milestone** PDFs and **Piazza** posts) have been successfully implemented. Infact, we have implemented some additional features (refer to Section 5.

# 9 Some notable limitations of HDS Compiler

- Using the function `len(array)`, where `array` is passed as an argument, may produce inaccurate results in this compiler because arrays are dynamically sized.

- An error is reported if a function, expected to return a non-null value, lacks a return statement. However, if return statements are placed within conditional blocks (such as if-elif-else), and there exists a possibility that control might reach the end of the function without encountering a return statement, the compiler does not generate an error. This approach avoids the complexity of managing numerous additional states.

# 10 Contribution

| # | Member Name | Member Roll No. | Member Email | Contribution |
|---|---|---|---|---|
| 1 | Harsh Bihany | 210406 | harshb21@iitk.ac.in | 36% |
| 2 | Danish Mehmood | 210297 | danishm21@iitk.ac.in | 28% |
| 3 | Siddhant Suresh Jakhotiya | 211030 | siddhantj21@iitk.ac.in | 36% |

# 11 References


- [DRAGv2] A. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools, 2nd edition pdf