Compiler Assignment Instructions

Group No-20
Deepanshu 210310 (deepanshu21@iitk.ac.in)
Ujjwal Gautam 211122 (ujjwalg21@iitk.ac.in)
Nirmal Prajapati 210735 (nirmal21@iitk.ac.in)

April 20, 2024

Instructions for Running the Code

To Compile the Bison & Lexer files, follow steps: Navigate to the directory "cs335-project" and enter the following commands in your terminal:

```
cd src
make
```

Listing 1: Running the Code

Now to compile any python file ¡file.python¿ using our compiler, run

```
./final --input <file.python>
```

Listing 2: Running the Code

To view different Intermediate files, options supported are:

```
    --help to check all the possible options
    --AST <filename> : Output file name for AST
    --TAC <filename> : Output file name for TAC
    --symboltable <filename> : Output file name for Symbol Table
    --x86 <filename> : Output file name for x86 code
    --verbose: Display debug information
```

On Successful compilation it will generate file *output.s*. Which will contain the final x86 code genearted by our compiler.

Contents

```
\tests
    - test1.py
    - test2.py
    - test3.py
    - test4.py
    - test5.py
\outputs
\doc
    - readme.pdf
\src
    - lexel.l
    - parser.y
    - makefile
    - helperFuncs.s
```

The `tests` directory contains 5 test cases - `test1.py` to `test5.py`. The `outputs` directory contains all output files generated. An Example run can be (Including cloning)

```
git clone https://git.cse.iitk.ac.in/ujjwalg/cs335-project.git
cd cs335-project
git checkout milestone3
cd Milestone3
cd src
make
./final --input ../tests/test1.py --verbose     //compile with debug on
gcc ../outputs/output.s                          //make executable
./a.out                                          //to check the output
```

Listing 3: Example run

3AC IR Modifications/Implementations

Implementation details:

- For Implementing **Non-basic** data types like **lists, Strings**. We have used Heap for allocations and these objects are always **used and passed by reference**
- Required **Backpatching** is done to implement For, while loops and if-else functionalities. By making labels stack. Implementation of Break and continue is same by using labels stack.
- **Classes**:
  - Variable Accesses using the offsets from symbol table of classes.
  - Variables can be only initialised for classes in *def __init__(self):* Function, Which automatically adds to symbol table of Class.
- Functionality for **Break, Continue** Statements also added(Was pending in Milestone 2)
- We have implements One more variabele in the QUADs named "types", For easiness in conversion of 3AC to x86 code.
- temporaries are now named as #tx where x is the temporary number, # is used to differentiate from normal variable tx.

x86 Implementation

Handling Temporaries

During the conversion from 3AC to x86 assembly, temporaries are currently pushed onto the stack. This ensures that temporary values are stored and accessible when needed during the execution of x86 instructions. However, a stack-based approach may lead to inefficiencies in accessing temporaries repeatedly. Thus, optimizing register allocation becomes crucial for improving performance.

Register Allocation and Handling

Efficient register allocation is essential for optimizing the performance of generated x86 code. By mapping temporaries and frequently accessed variables to registers, we can minimize memory accesses and reduce instruction overhead. Various register allocation algorithms, such as graph coloring or linear scan, can be employed to assign registers dynamically during code generation. Additionally, spill code generation strategies can be implemented to handle cases where the number of available registers is insufficient to accommodate all variables.

Mapping 3AC Instructions to x86 Assembly Instructions

Each 3AC instruction is translated into one or more x86 assembly instructions to faithfully replicate the behavior of the original code. For example:

- Arithmetic operations (e.g., addition, subtraction) are translated to corresponding x86 instructions (e.g., `ADD`, `SUB`).
- Control flow constructs (e.g., conditional branches, loops) are translated using conditional and unconditional jump instructions (`JMP`, `JZ`, `JNZ`, etc.).
- Memory operations (e.g., load, store) involve accessing memory locations using instructions like `MOV` and `LEA`.
- Function calls and returns are translated to `CALL` and `RET` instructions, respectively.

Each 3AC operation is carefully mapped to the most appropriate x86 instruction, considering factors such as efficiency, code size, and compatibility with the target architecture.

Language Features Supported

1. **Primitive Data types : int,bool,str**
2. **Basic operations**
   - Arithmetic operators: `+, -, *, /, //, %, **`
   - Relational operators: `==, !=, >, <, >=, <=`
   - Logical operators: `and, or, not`
   - Bitwise operators: `&, |, ^, ~, <<, >>`
   - Assignment operators: `=, +=, -=, *=, /=, %=, **=, &=, |=, ^=, <<=, >>=`
3. **Static 1D Lists**: 1D lists are implemented with len() functionality
4. **Classes, Constructors** class Methods, Class variable are supported, Variables in Class can be initialised in Constructor
5. **Loops, Breaks, If-else**: Proper Branching and Backpatching is done , For loops are implements as while loops, Iterable lists in for loops can only be as funccall() or some variable
6. **Functions, Class functions**: Functions can be called with any number of arguments.
7. **Recursion**: As We have implementaion through Stack Recursion can be Utilised easily.
8. **len() and Range()**: They can be used for Iterable Objects with only 1 argument.
9. **Print Strings, Numbers**: Print function Can be called with one argument for string bool or a Integer.
10. Type Checking is done for Function returns, Function Arguments, Assignments, Operations.

Not Supported

1. **Numeric types- Float, Double, Unsigned**: They have different intructions sets and require more type checking & converting implementations.(This wasn't asked too)
2. **Class Inheritance**: After trying a lot to implement this, We were unable to resolve the errors like, memory allocations and symbol table copying required etc

Effort sheet

The Project has been done in collaborative manner in presence of all the members, while Each member handled specific area for writing code.

Table 1: Contribution Sheet

Deepanshu	**Ujjwal**	**Nirmal**
33.33	33.33	33.33