CS3300 Compiler Design (Jul - Nov 2024)

Assignment 4

Due **Nov 10 [23:59]** on moodle

Instructor: PROF. RUPESH NASRE

Problem Statement:

In this assignment, your task is to develop a lexer and parser that translates **Three-Address Code (TAC)** into **optimized TAC**. You will implement functionality to remove redundant assignments by performing **live variable analysis**. Once the TAC is optimized, you will answer queries about which variables are live at specified lines in the code.

Objective:

The assignment requires you to:

1. **Optimize the TAC:** Eliminate redundant assignments. A redundant assignment occurs when a variable is defined but its value is never used in any subsequent instruction. Once you remove a redundant assignment, you must revisit earlier instructions to check whether the variables involved in the removed assignment become redundant and will also be eliminated.
2. **Answer Live Variable Queries:** Given specific queries, compute which variables are live at certain line numbers in the TAC. Live variables are those that hold values needed in future instructions before being overwritten.
3. The output should include the **live variables** at each queried line number.

Input Format:

The input will consist of two files:

1. `FILE_NAME.tac` contains input TAC.
2. `FILE_NAME.txt` contains the live variable queries. Each of the lines contain a positive integer **Q** representing the line number corresponding to a specific line in the TAC.

Output Format:

You must generate one output file:

1. `SAMPLE_NAME.txt` contains the **output of queries**. For each query, the output should list the variables that are live at the specified line number. The live variables should be printed in lexicographically sorted order, with the results for each query displayed on a new line.
2. Tester script will ignore any extra spaces or redundant newlines.
3. If line is removed after optimization print "**Line removed in optimized TAC**".

Optimization Rules:

1. A variable assignment (e.g., `x = y + z`) is **redundant** if the variable **x** is never used in subsequent instructions before being overwritten or the program terminates.
2. If an assignment is removed, check whether the variables used in that assignment become redundant and remove them if possible.
3. Only remove assignments that do not affect the correctness of the program.
4. To compute live variables at each line, follow the **IN(B)** and **OUT(B)** formulae:

- **IN(B)**: The IN set for a block B represents the variables that are live before the block. It is computed using the USE and DEF sets:

  $$\mathbf{IN(B) = USE(B) \cup (OUT(B) - DEF(B))}$$

  USE(B): The set of variables that are used in block B before being defined.
  DEF(B): The set of variables that are defined (or assigned a value) in block B**.**
- **OUT(B):** The OUT set for a block B represents the variables that are live after the block. It is computed as the union of the IN sets of all successor blocks: 3

  $$\mathbf{OUT(B) = \cup IN(S)}$$

  where **S** is any successor of block B.

Input Grammar for TAC:

1. `A?` indicates that non-terminal A is optional (it may or may not be present).
2. `A*` indicates that non-terminal A may appear zero or more times (it can repeat).
3. `<ID>` refers to a valid identifier, starting with a letter followed by alphanumeric characters.
4. `<INTEGER LITERAL>` refers to a constant integer.
5. `<LABEL>` refers to a unique label in the code.

6. **Program Structure**
   - Program ::= (TACLine)*
   - TACLine ::= AssignmentStatement
     | UnaryAssignmentStatement
     | ConditionalJump
     | UnconditionalJump
     | LabelDefinition
     | IOStatement

7. **Statements**

I. **Assignment Statement:**

AssignmentStatement ::= `<ID> "=" Expression`

Expression ::= `<ID> Operator <ID>`
`| <ID> Operator <INTEGER LITERAL>`
`| <INTEGER LITERAL> Operator <ID>`
`| <INTEGER LITERAL> Operator <INTEGER LITERAL>`
`| <ID>`
`| <INTEGER LITERAL>`

Operator ::= `"+" | "-" | "*" | "/"`

Constraint: Only binary operations or direct assignments are allowed (no function calls).

Ex: `x = y + z`

II. **Unary Assignment Statement:**

UnaryAssignmentStatement ::= `<ID> "=" UnaryOperator <ID>`

UnaryOperator ::= `"-" | "!"`

Constraint: Only unary negation (-) or logical NOT (!) operations are allowed.

Ex: `x = -y`

III. **Control Flow:**

a. **Conditional Jump**

ConditionalJump ::= `"if" <ID> ComparisonOperator <ID> "goto" <LABEL>`

ComparisonOperator ::= `"==" | "!=" | "<" | "<=" | ">" | ">="`

Constraint: Only comparison operations between two variables are allowed.

Ex: `if x < y goto L1`

b. **Unconditional Jump**

UnconditionalJump ::= `"goto" <LABEL>`

Constraint: Direct, unconditional jumps are allowed.
Ex: `goto L2`

IV. **Label Definition:**
LabelDefinition ::= `<LABEL> ":"`
Constraint: Labels must be unique and followed by a colon `(:)`.
Ex: `L1:`

V. **Input/Output Statement:**
IOStatement ::= `"print" <ID> | "read" <ID>`
Constraint: Only single-variable input and output statements are allowed.
Ex: `print x`

8. **Identifiers, Literals, and Labels:**

```
<ID>               ::= [a-zA-Z][a-zA-Z0-9]*
<INTEGER LITERAL>  ::= [0-9]+
<LABEL>            ::= [a-zA-Z][a-zA-Z0-9]* ":"
```

Constraints:

- The number of **TAC lines** will not exceed **1000**.
- Queries will be on valid line numbers (within the range of the TAC).
- Variables are alphanumeric strings starting with a letter.
- Operations can include: `+, -, *, /, ==, !=, <, <=, >, >=.`
- Labels are unique within the TAC.

Sample Test Cases

1.

INPUT	OUTPUT
***(note that line numbers are not part of the input TAC)*:** 1: a = b + c 2: d = a + 1 3: if d < e goto L1 4: f = d + e 5: goto L2 6: L1: f = e + 1 7: L2: print f 2 4 6	a e d e e
Conclusion: ● No redundant assignments are detected in this example.	

2.

INPUT	OUTPUT
1: i = 0 2: sum = 0 3: x = 0 4: L1: if i >= x goto L2 5: sum = sum + i 6: i = i + 1 7: goto L1 8: L2: print sum 4 5	i x i sum
Conclusion: ● No redundant assignments are detected in this example.	

3.

INPUT	OUTPUT
1: x = y + 1 2: if x > 10 goto L1 3: z = x + 2 4: if z < 20 goto L2 5: z = y + 3 6: goto L3 7: L1: z = x - 1 8: goto L3 9: L2: w = z + 1 10: L3: print w 1 3 9	x y z
Conclusion: ● Line 5 (z = y + 3): **Redundant** (overwritten in line 7 and not used before line 7). ● Remove line 5	

4.

INPUT	OUTPUT
1: p = q + r 2: s = p + 1 3: if s < t goto L1 4: u = s + t 5: v = q + r 6: L1: w = u + t 7: print w 1 2 5	q r t p t Line removed in optimized TAC
Conclusion: ● Line 5 (v = q + r): **Redundant** (never used). ● Remove line 5.	

5.

INPUT	OUTPUT
1: a = 10 2: b = a + 5 3: c = b + 2 4: d = c + 4 5: b = d + 1 6: e = a * 2 7: print e 2 3 5 6	Line removed in optimized TAC Line removed in optimized TAC Line removed in optimized TAC e

Conclusion:

- Line 5 (`b = d + 1`) was the first to be **removed** because `b` was re-assigned here but never used before the program ended.
- Line 4 (`d = c + 4`) became **redundant** after line 5 was removed, as `d` had no further uses.
- Line 3 (`c = b + 2`) was **removed** since `c` became redundant with line 4 gone.
- Line 2 (`b = a + 5`) was **removed** as it was no longer needed for any computations.

Submission Guidelines:

1. Create a new directory and rename it to your roll number, in the format CSXXBXXX (e.g., CS12B345).
2. Ensure that (a4.l and a4.y) files related to your project are placed inside this CSXXBXXX directory and there is no other directory inside CSXXBXXX.
3. Write a Makefile and place it inside the CSXXBXXX directory. The executable must be a.out (we will follow this for all the future labs + assignments, so there is no confusion).
4. Zip the directory using the following command:
   ```
   zip CSXXBXXX.zip -r CSXXBXXX
   ```
5. Unzipping CSXXBXXX.zip must give a folder CSXXBXXX containing only source files (a4.l, a4.y, your C/C++ files and Makefile). It would help if you check this after submitting on moodle.
6. For any queries about the assignment, please post on moodle, so everyone can know about it.