

Three-Address Code (TAC)

In the Intermediate Code Generation phase, compilers convert high-level language constructs into a simpler, machine-independent form called Three-Address Code (TAC).

Each instruction in TAC involves at most three operands (hence the name three-address). It helps in optimization and target code generation

Three Address Codes are Used in Compiler Applications

- **Optimization:** TAC is used as an intermediate code by compilers to perform code optimization and improve performance.
- **Code Generation:** TAC is also the basis for generating efficient, correct machine code for different platforms.
- **Debugging:** TAC makes debugging easier by providing human-readable, low-level instructions that help trace execution and spot errors.
- **Language Translation:** TAC acts as a common intermediate form, which simplifies translating code between different programming languages.

Implementation of Three Address Code

There are 3 representations of three address code namely

- Quadruple
- Triples
- Indirect Triples

1. Quadruple: It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example - Consider expression $a = b * - c + b * - c$. The three address code is:

t1 = uminus c (Unary minus operation on c)

t2 = b * t1

$t3 = \text{uminus } c$ (Another unary minus operation on c)

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$ (Assignment of t5 to a)

| # | Op | Arg1 | Arg2 | Result |
|-----|--------|------|------|--------|
| (0) | uminus | c | | t1 |
| (1) | * | t1 | b | t2 |
| (2) | uminus | c | | t3 |
| (3) | * | t3 | b | t4 |
| (4) | + | t2 | t4 | t5 |
| (5) | = | t5 | | a |

Quadruple representation

2. Triples: This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example - Consider expression $a = b * - c + b * - c$

| # | Op | Arg1 | Arg2 |
|-----|--------|------|------|
| (0) | uminus | c | |
| (1) | * | (0) | b |
| (2) | uminus | c | |
| (3) | * | (2) | b |
| (4) | + | (1) | (3) |
| (5) | = | a | (4) |

Triples representation

3. Indirect Triples This representation makes use of pointer to the listing of all references to computations which is made separately and stored. It's similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example - Consider expression $a = b * - c + b * - c$

List of pointers to table

| # | Op | Arg1 | Arg2 |
|------|--------|------|------|
| (14) | uminus | c | |
| (15) | * | (14) | b |
| (16) | uminus | c | |
| (17) | * | (16) | b |
| (18) | + | (15) | (17) |
| (19) | = | a | (18) |

| # | Statement |
|-----|-----------|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

Indirect Triples representation

Implementation of Three Address Code

1. Source Code Parsing and Abstract Syntax Tree Generation.

- Lexical Analysis: source code is converted into tokens, keywords, identifiers, operators, literals, etc.
- Syntax Analysis: An Abstract Syntax Tree that would be the representation of the grammatical structure of the code.
- Semantic Analysis: Derivation of semantic errors type mismatches, undeclared variables, etc. and builds a symbol table.

2. TAC Instructions Generation

- Traversal of the Abstract Syntax Tree: visiting each of the nodes in the abstract syntax tree generates the proper TAC instructions.
- Use temporaries for intermediate computation.
- Three address form: Each instruction should have at most three operands, two source and one target.

3. Evaluation of Expressions

- Arithmetic Expressions: Divide complex expression into simpler expressions by using the temporaries. Example $a + b * c$ which should be encoded as follows:

- $t1 = b * c$
- $t2 := a + t1$

3. Logical Expressions:

- Translate the logical AND, OR and NOT operators into conditional jumps. Uses temporary registers.

4. Control Flow Constructs

- If Statements :Use conditional jumps.
- While Loops : Involve in unconditional jumps to the loop header and conditional jumps to exit.
- For Loops: Converted to equivalent while loops.
- Goto Statements: Unconditionally jump to the specified labels.

5. Procedure Calls

- Argument Passing : Arguments are pushed onto the stack
- Function Call: A jump is made to the function's entry point.
- Return Value: Stored in a temporary variable while popping the arguments off the stack.