3.5 APPLICATIONS OF SDTS: TYPE CHECKING, GENERATING THREE-ADDRESS CODE

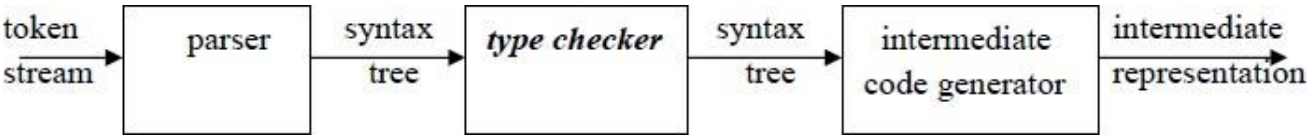
3.5.1 Type Checking

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called static checking, detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** –Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



A type checker verifies that the type of a construct matches that expected by its context.

For example : arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer.

Type information gathered by a type checker may be needed when code is generated.

3.5.2 Three address code in Compiler

An intermediate representation used in compiler design to simplify complex expressions by breaking them into a sequence of simple instructions. Each instruction contains at most three addresses: two operands and one result.

The result of each operation is typically stored in a temporary variable.

TAC makes the order of operations explicit, which helps in code optimization and simplifies the generation of machine code.

Applications of Three-Address Code in Compilers

Code Optimization: Three-address code is commonly used during the optimization phase of compilation. It allows the compiler to analyze the program structure and apply optimization techniques to improve performance

Code generation: TAC serves as an intermediate representation during the code generation phase, allowing the compiler to produce efficient machine code for the target platform.

Debugging: Because TAC is simpler and closer to the original source code than machine code, it can help developers trace program execution and identify errors.

Language translation: TAC can be used as a common intermediate representation for translating programs from one programming language to another. **General Representation** a = b a = op b a = b op c

Where a, b, or c represents operands like names, constants or compiler-generated temporaries and op represents the operator

Example-1: Convert the expression a * - (b + c) into three address codes.

The given expression is:

a * − (b + c)

To convert it into three-address code, we break it into smaller operations using temporary variables:

1. First, compute the expression inside the parentheses: t1 = b + c

2. Apply unary minus to the result: t2 = -t1

3. Multiply the result with a: t3 = a * t2

Implementation of Three Address Code

There are 3 representations of three address code namely

1. Quadruple:

A quadruple is an intermediate code representation consisting of four fields: op, arg1, arg2, result.

op represents the operator.

arg1 and arg2 represent the operands. result stores the outcome of the operation.

This representation makes it easy to rearrange code, which helps during global optimization.

Temporary variable values can be quickly accessed using the symbol table, improving manageability.

However, it often introduces a large number of temporary variables.

Creating many temporaries can increase both time and space complexity of the compiler.

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 = 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 does not use extra temporary variables to store intermediate results.

Instead of temporaries, it uses pointers to refer to the result of another triple whenever needed.

It consists of three fields:

op, arg1, arg2.

Temporaries are implicit, which makes code rearrangement difficult.

Code optimization becomes harder because moving one triple requires updating all other triples that refer to it.

Using pointers allows direct access to symbol table entries, which helps in locating operands efficiently.

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 uses a pointer to a separate list that stores all references to computations.

The actual triples remain unchanged, while the pointer list is updated when code is rearranged.

It provides similar utility to quadruple representation but requires less storage space.

Temporary values are implicit, like in triple representation.

Code rearrangement becomes easier because only the pointer list needs to be modified, not the triples themselves.

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

Question - Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation - The three address code is:

(1) t1 = x + y

(2) t2 = y + z

(3) t3 = t1 * t2

(4) t4 = t1 + z

(5) t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

Process of Generating 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. **Expression Evaluation**

Arithmetic Expressions: Complex expressions are broken into simpler operations; for example, the expression **a + b * c** in three-address code is written as: **t1 = b * c; t2 = a + t1**

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..