COMPILERS: SYNTHESIS PHASE

Module - 6

St. Francis Institute of Technology, Ms. Ankita Karia

TOPICS TO COVER

Intermediate Code Generation:

1. Types of Intermediate codes: Syntax tree, Postfix notation,
2. Three address codes: Triples and Quadruples, indirect triple.

Code Optimization:

1. Need and sources of optimization
2. Code optimization techniques: Machine Dependent and Machine Independent.

Code Generation: Issues in the design of code generator, code generation algorithm. Basic block and flow graph.

INTERMEDIATE CODE GENERATION

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

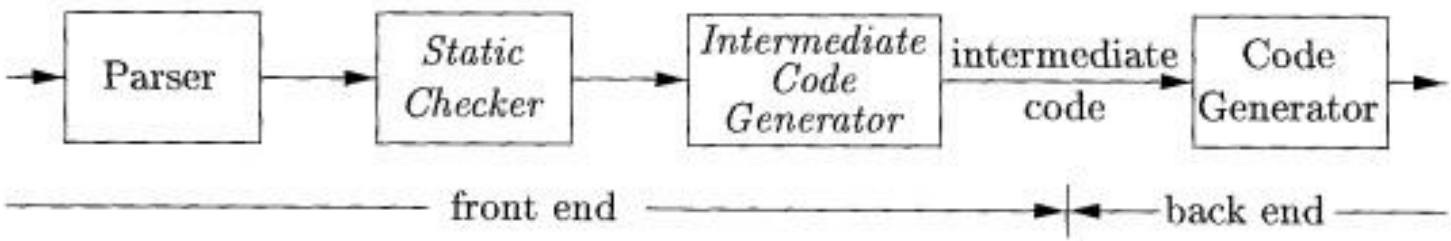
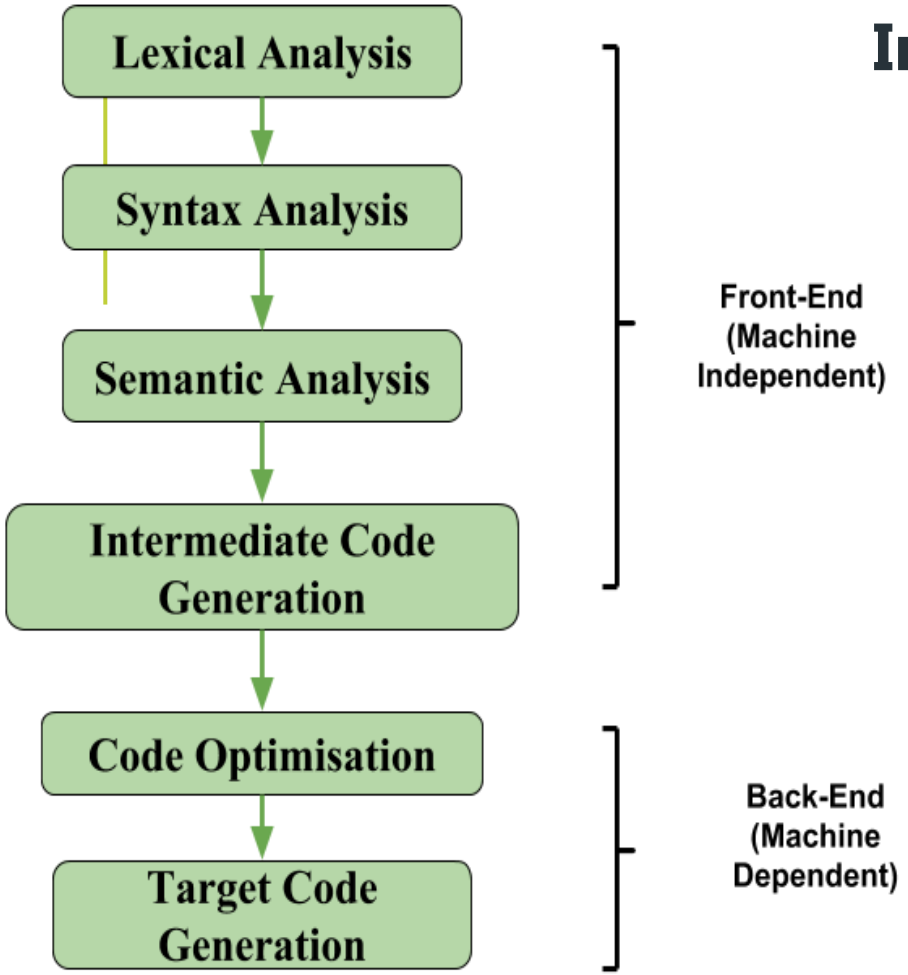


Figure 6.1: Logical structure of a compiler front end

Assume that the front end a compiler is organized as shown above;

Where Parsing, Static checking, and Intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing.

Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing. **For example,** static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.



Benefits of using Machine-Independent Intermediate code are:

1. Because of the machine-independent intermediate code, portability will be enhanced.
2. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
3. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.

NEED FOR INTERMEDIATE CODE

1.Suppose we have x no of the source language and y no of the target language:

1. **Without ICG** – we have to change each source language into target language directly, So, for each source-target pair we will need a compiler .Hence we need (x*y) compilers, which can be a very large number and which is literally impossible.

2. **With ICG –** we will need only x number of compiler to convert each source language into intermediate code. We will also need y compiler to convert the intermediate code into y target languages. so, we will need only (x+y) no of the compiler with ICG which is way lesser than x*y no of the compiler.

2.Re-targeting is facilitated : a compiler for a different machine can be created by attaching a back-end(which generate target code) for the new machine to an existing front-end(which generate intermediate code).

NEED FOR INTERMEDIATE CODE

3. **Machine independent:** A Machine independent code-optimizer can be applied to the intermediate code. So this can be run on any machine.

4. **Simplicity:** Intermediate code is simple enough to be easily converted to any target code. So ICG reduces the overhead of target code generation.

5. **Complexity :** Intermediate code is Complex enough to represent all complex structure of high-level languages.

6. **Modification:** We can easily modify our code to get better performance and throughput by applying optimization technique to the Intermediate code.

INTERMEDIATE REPRESENTATION

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as

1. **High-Level IR:** High-level intermediate codes closely resemble the source language, making them simple to build from source code. This also makes it easier to update the code to enhance execution. **E.g. Syntax Tree**
2. **Low-Level IR** – A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.

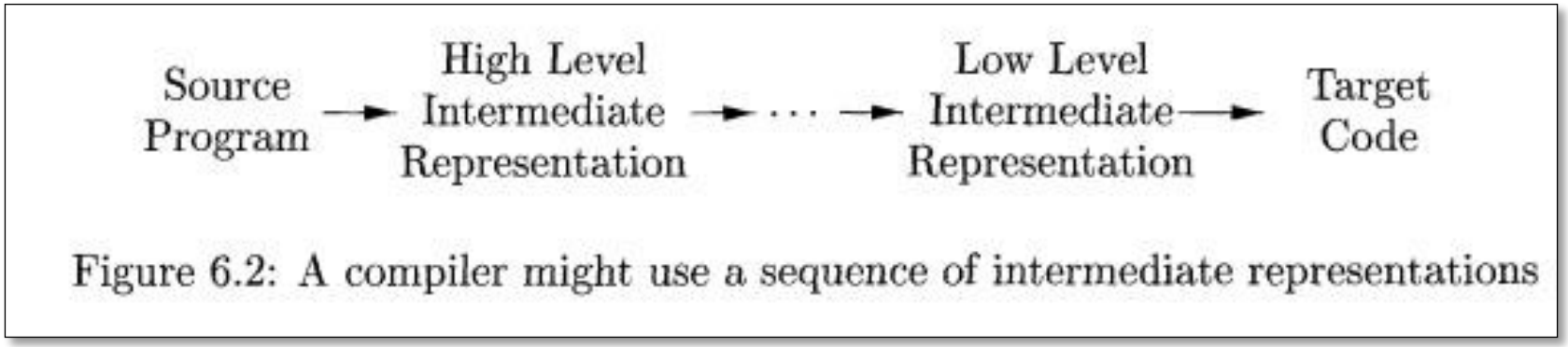


Figure 6.2: A compiler might use a sequence of intermediate representations

The following are commonly used intermediate code representations:

POSTFIX NOTATION, SYNTAX TREES, THREE ADDRESS CODE

POSTFIX NOTATION

- Compilers find it difficult to distinguish between the operators and parentheses. So, for this, we have postfix notation in compiler design.
- Postfix notation is also known as **Reverse Polish Notation**.
- In this notation, the operators are written after the operands, not like the infix in which the operator is in-between the operands.
- For example, the infix notation (5+6)*7 will be written as 56+7* in postfix notation.
- The postfix representation of the expression

(a – b) * (c + d) + (a – b)

ab – cd + *ab -+

ADVANTAGES OF POSTFIX NOTATION

1. **No need for parentheses:** In polish notation, there is no need for parentheses while writing the arithmetic expressions as the operators come before the operands.

2. **Efficient Evaluation:** The evaluation of an expression is easier in polish notation because in polish notation stack can be used for evaluation.

3. **Easy parsing:** In polish notation, the parsing can easily be done as compared to the infix notation.

4. **Less scanning:** The compiler needs fewer scans as the parentheses are not used in the polish notations, and the compiler does not need to scan the operators and operands differently.

Why is stack used for RPN?

The use of a stack is advantageous for Reverse polish notation evaluation because it simplifies the process of keeping track of operands and operators. The stack provides a convenient way to store and retrieve the operands, and the last two operands can be easily retrieved when an operator is encountered.

THREE ADDRESS CODE

Three-address code is a linearized representation of a Syntax Tree or a DAG in which explicit names correspond to the interior nodes of the graph

1. Addresses and Instructions
2. Quadruples
3. Triples
4. Indirect Triples
5. Static Single-Assignment Form

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
Thus a source-language expression like **x+y*z** might be translated into the sequence of three-address instructions

$$t1 = y * z$$
$$t2 = x + t1$$

where t1 and t2 are compiler-generated temporary names. This solving of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

t3 = a + t2

EXAMPLE OF 3AC

a + a * (b – c) + (b – c) * d

t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

EXAMPLE OF 3AC & ITS CORRESPONDING DAG

a + a * (b – c) + (b – c) * d

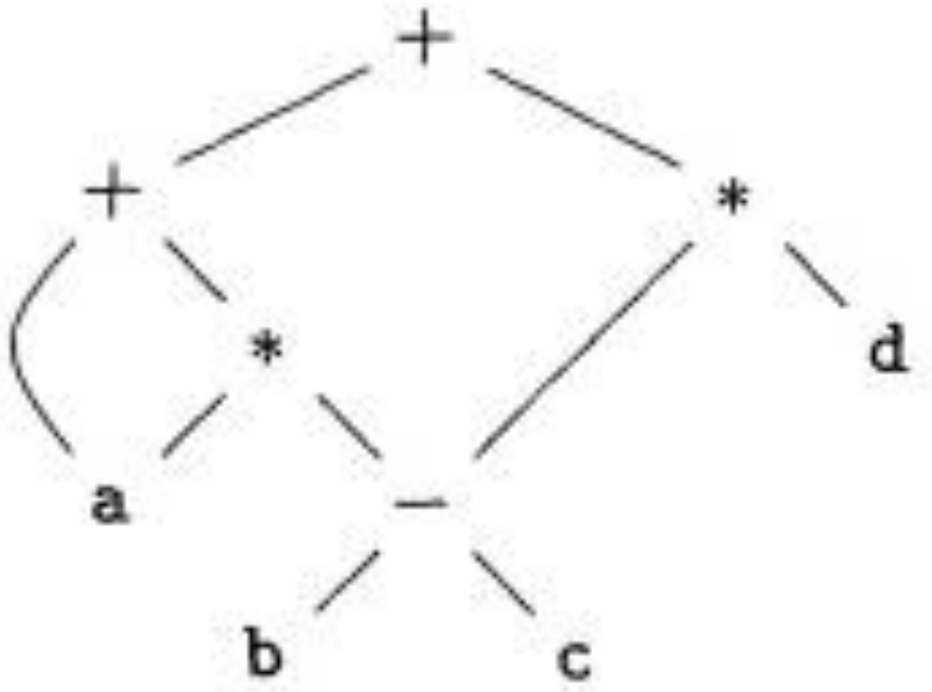
t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4



(a) DAG

QUADRUPLES

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure.

In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called “QUADRUPLES," “TRIPLES," and “INDIRECT TRIPLES.“

A quadruple (or just "quad!') has four fields, which we call op, arg1: arg2, and result.

For instance, the three-address instruction x = y + z is represented by

placing + in op,
y in arg1,
z in arg2, and
x in result. The following are some exceptions to this rule:

EXCEPTIONS TO THE RULES OF QUADRUPLES

1. Instructions with unary operators like x = minus y or x = y do not use arg2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

2. Operators like param use neither arg2 nor result.

3. Conditional and unconditional jumps put the target label in result.

Three-address code for the assignment a = b * - c + b * - c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

TRIPLES

- A ***TRIPLE*** has only three fields, which we call ***op, arg_x,* and *arg_2.***
- The ***result*** field in QUADRUPLES is used primarily for temporary names.
- Using triples, we refer to the result of an operation ***x*** *op* *y* by its position, rather than by an explicit temporary name.
- Thus, instead of the temporary t1 a triple representation would refer to position (0).
- Parenthesized numbers represent pointers into the triple structure itself.

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

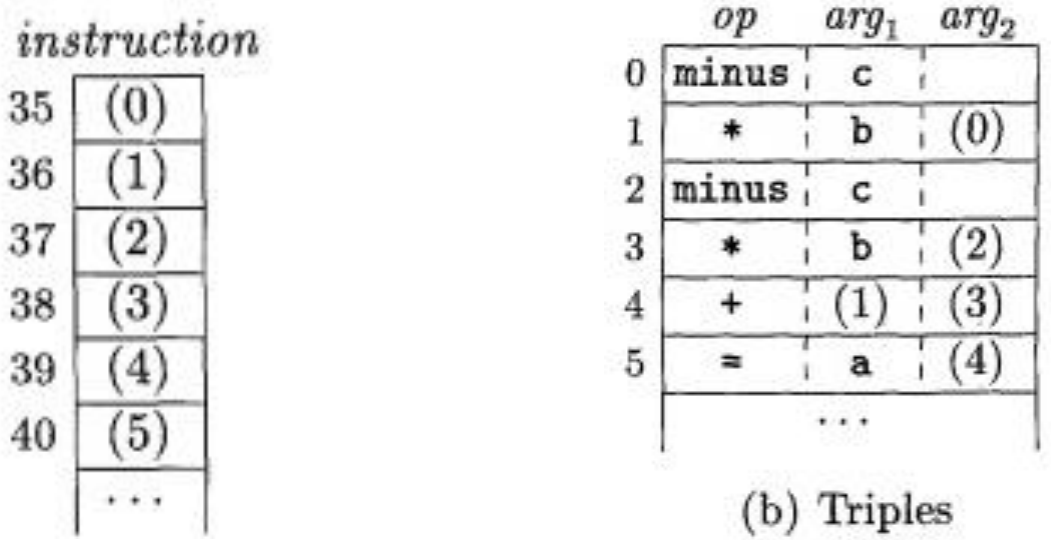
(b) Triples

INDIRECT TRIPLES

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.

For example, let us use an array *instruction* to list pointers to triples in the desired order.

Then, the triples might be represented as follows;



	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

STATIC SINGLE-ASSIGNMENT FORM (SSA)

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- All assignments in SSA are to variables with distinct names; hence the term *static single-assignment.*

SSA is a property of an intermediate representation, which requires that each variable is assigned exactly once , and every variable is defined before it is used.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( f l a g )
        x = -1;
 e l s e
        x =  1;
y = x * a;
```

Has two control-flow paths in which the variable x gets defined.

STATIC SINGLE-ASSIGNMENT FORM

- $\emptyset - function$ function to combine the two definitions of x:

```
if ( f l a g )
        x1 = -1;
 e l s e
        x2 = 1;
x3 = Ø(x1, x2);
```

- Here, ϕ(x1,x2) has the value x1 if the control flow passes through the true part of the conditional
- And the value x2 if the control flow passes through the false part.
- That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

SSA uses a notational convention called the $\emptyset - function$

DIRECTED ACYCLIC GRAPH

Nodes in a **syntax tree** represent constructs in the source program; the children of a node represent the meaningful components of a construct.

A **Directed Acyclic Graph (hereafter called a *DAG)*** for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression.

DIRECTED ACYCLIC GRAPHS FOR EXPRESSIONS

Directed Acyclic Graph (DAG) is a special kind of **Abstract Syntax Tree**.

Each node of it contains a unique value.

It does not contain any cycles in it, hence called **Acyclic**.

A DAG is constructed for optimizing the basic block.

A DAG is usually constructed using **Three Address Code**.

Transformations such as dead code elimination and common sub expression elimination are then applied.

Applications- **DAGs are used for the following purposes-**

1. To determine the expressions which have been computed more than once (called common sub-expressions).
2. To determine the names whose computation has been done outside the block but used inside the block.
3. To determine the statements of the block whose computed value can be made available outside the block.
4. To simplify the list of **Quadruples** by not executing the assignment instructions x:=y unless they are necessary and eliminating the common sub-expressions.