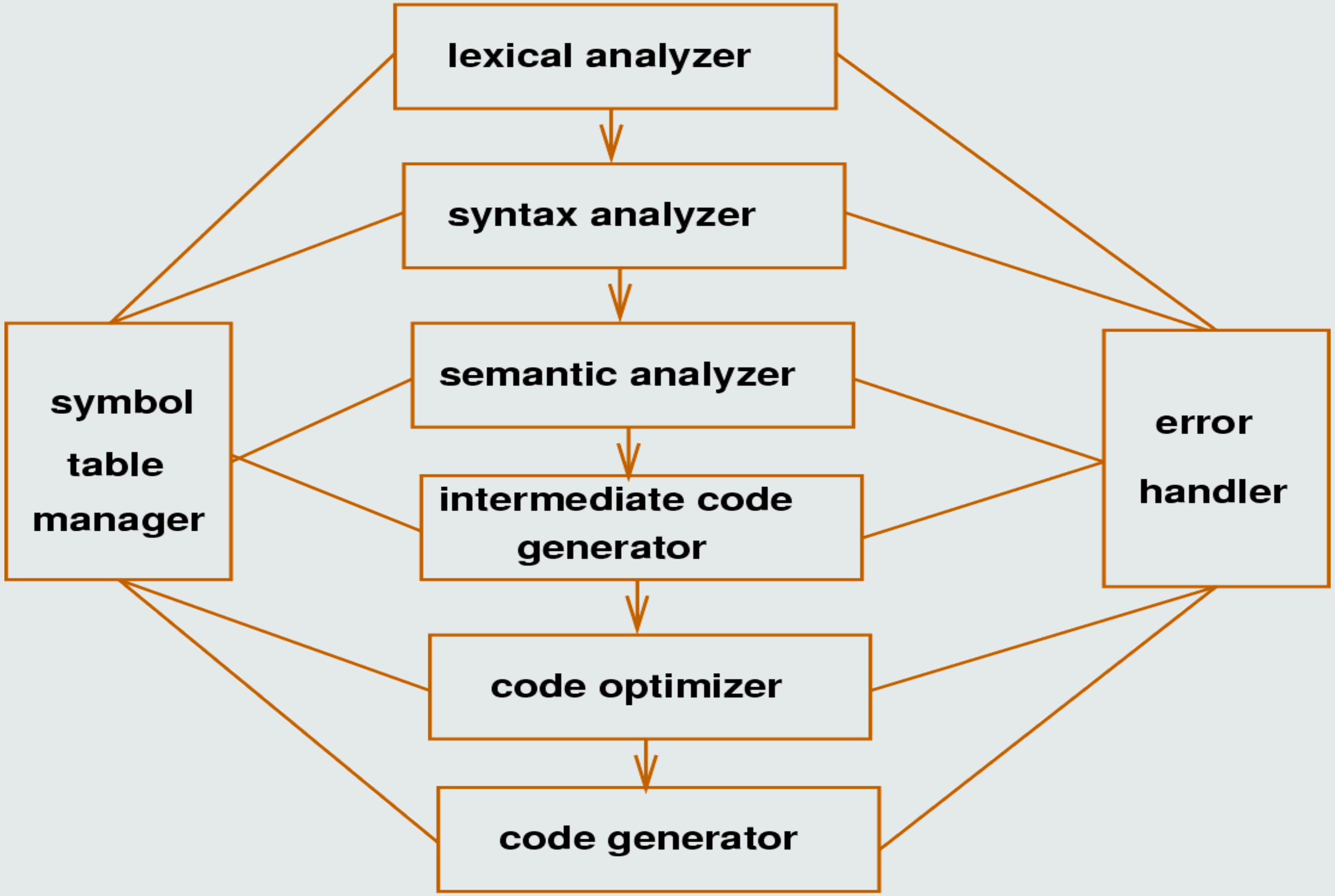
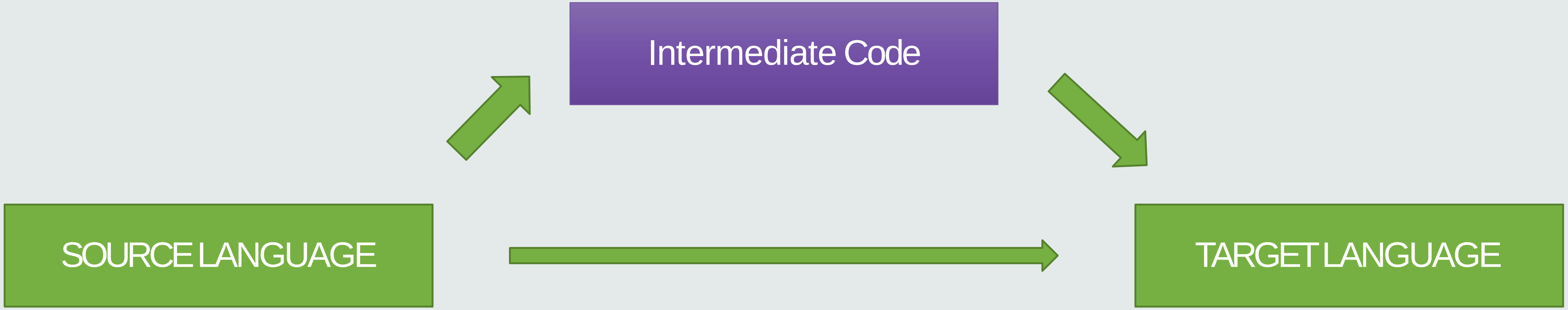
Three Address Code Generation

Phases Of Compiler



Intermediate Code

- An language b/w source and target language
- Provides an intermediate level of abstraction
  - More details than the source
  - Fewer details than the target



Intermediate Code

SOURCE LANGUAGE

TARGET LANGUAGE

Benefits of intermediate code generation

- A compiler for different machines can be created by attaching different backend to the existing front ends of each machine
- A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source language to existing back end.
- A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation

Three Address Code

- Is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.
- Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator
- In TAC, there is at most one operator on the right side of an instruction. That is no built-up arithmetic expressions are permitted

Example : x + y * z

t1 = y * z

t2 = x + t1

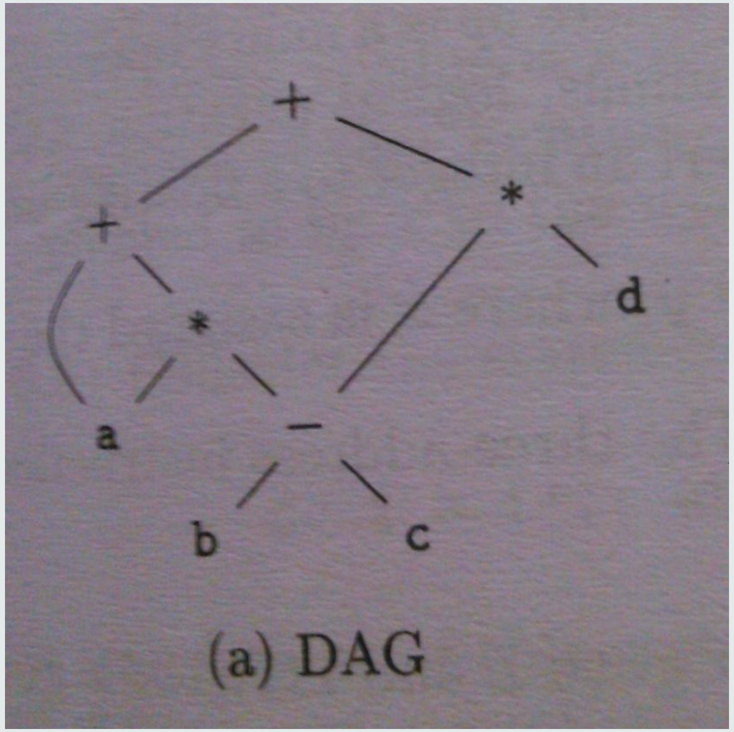
t1 and t2 are compiler-generated temporary names

- Statements in this language are of the form:

$$x := y \text{ op } z$$

- where x, y and z are names, constants or compiler-generated temporary variables, and ‘op’ stands for any operator

- Three Address Code is a linearized representation of a syntax trees or a DAG



(a) DAG

T1 = b – c

T2 = a * t1

T3 = a + t2

T4 = t1 * d

T5 = t3 +t4

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

- b * minus c + b * minus c

Three address code

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

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples

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

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Disadvantage Of quadruples

- Temporary names must be entered into the symbol table as they are created.
- This increases the time and size of the symbol table.

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Disadvantage Of TRIPLES

- Moving a statement that define a temporary value requires us to change all references to that statement in arg1 and arg2 arrays. This problem makes triple difficult to use in an optimizing compiler.

Types of Three-Address Code

Assignment statement	x :=y op z
Assignment statement	x :=op y
Copy statement	x :=y
Unconditional jump	goto L
Conditional jump	if x relop y goto L
Procedural call	param x call p return y

Assignment Statement

- Assignment statements can be in the following two forms

  1. x:=op y

  2. x:=y op z

First statement op is a unary operation. Essential unary operations are unary minus, logical negation, shift operators and conversion operators.

Second statement op is a binary arithmetic or logical operator.

Three-Address Statements

A popular form of intermediate code used in optimizing compilers is three-address statements.

Source statement:

$$x = a + b * c + d$$

Three address statements with temporaries t_1 and t_2:

$$t_1 = b * c$$
$$t_2 = a + t_1$$
$$x = t_2 + d$$

Jump Statements

source statement like if-then-else and while-do cause jump in the control flow through three address code so any statement in three address code can be given label to make it the target of a jump.

The statement

goto L

Cause an unconditional jump to the statement with label L. the statement

if x relop y goto L

Causes a jump to L condition if and only if

Boolean condition is true.

This instruction applies relational operator relop (>,=,<, etc.)

to x and y, and executes statement L next of x statement x relop y. If not, the three address statement following if x relop y goto L is executed next, as in the usual sequence.

Procedure Call/ Return

A procedure call like P(A1,A2, A3,……..…An) may have to many addresses for one statement in three-address code so it is shown as a sequence of n +1 statements'

Param A1

Param A2

M

Param An

Call p,n

Where P is the name of the procedure and and n is a integer indicating the number of actual parameters in the call.

This information is redundant, as n can be computed by counting the number of par am statements.

It is a convenience to have n available with the call statement.

Indexed Assignment

Indexed assignment of the form A:=B[I] and A[I]:=B.

the first statement sets A to the value in the location I memory units beyond location B.

In the later statement A [I]:=B, sets the location I units beyond A to the value of B.

In Both instructions ,A, B, and I are assumed to refer data objects and will represented by pointers to the symbol table.

Address and Pointer Assignment

Address and pointer assignment

```
x := &y
x :=*y
*x := y
```

First statement, sets the value of x to be the location of y.

In x := *y, here y is a pointer or temporary whose r-value is a location. The r-value of x is made equal to the contents of that location.

*x := y sets the r-value of the object pointed to by a to the r-value of y.

Summary

- Intermediate Code
- 3Address Code
- Data Structures Of 3Address Code
- Types of Three-Address Code