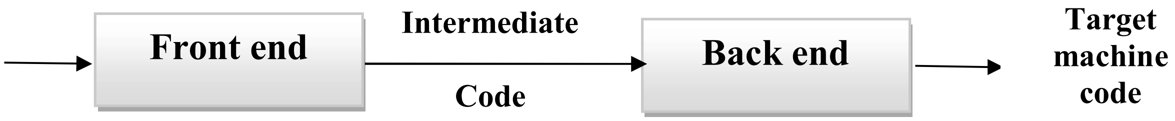
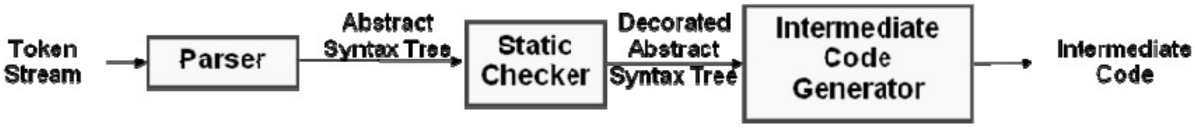
Intermediate Code Generation (IR)



In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates retargeting: enables attaching a back end for the new machine to an existing front end.

Logical Structure of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- Flow–of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks
  - Labels in case statements
- Name-related checks

Intermediate Representations

We could translate the source program directly into the target language. However, **there are <u>benefits</u> to having an intermediate, machine-independent representation**.

- A clear distinction between the **<u>machine-independent</u>** and **<u>machine-dependent</u>** parts of the compiler
- **Retargeting is facilitated**; the implementation of language processors for new machines will require replacing only the **<u>back-end</u>**
- We could apply **machine independent code optimization techniques**

Intermediate representations span the gap between the source and target languages.

- **High Level Representations**
  - Closer to the source language
  - Easy to generate from an input program
  - Code optimizations may not be straightforward

- **Low Level Representations**
  - Closer to the target machine
  - Suitable for register allocation and instruction selection
  - Easier for optimizations, final code generation

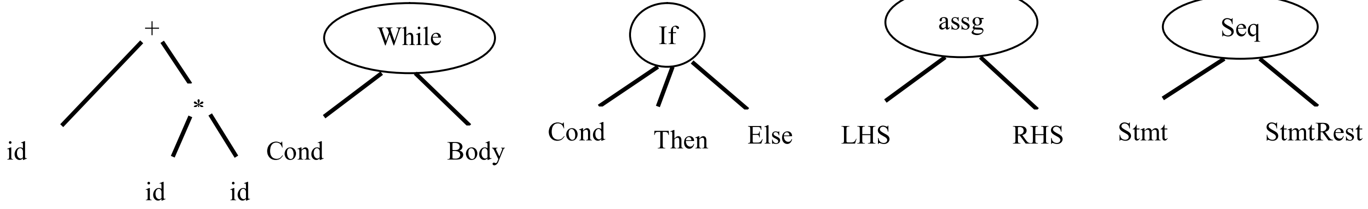
There are **several options for intermediate code**. They can be either

- Specific to the language being implemented
  - ➢ P-code for Pascal
  - ➢ Bytecode for Java
- Language independent:
  - ➢ 3-address code

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. **C** used as **intermediate language** as it is flexible, compiles into efficient machine code and its compilers are widely available.

In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single or small number of machine instructions. Machine code can then be generated (access might be required to symbol tables etc).

Syntax Trees

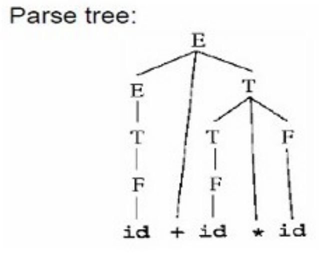
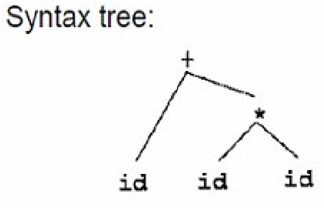
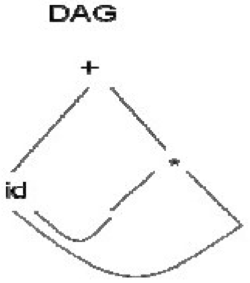


Syntax trees are high level IR. They depict the natural hierarchical structure of the source program. **Nodes represent constructs in source program and the children of a node represent meaningful components of the construct**. Syntax trees are suited for **static type checking**.

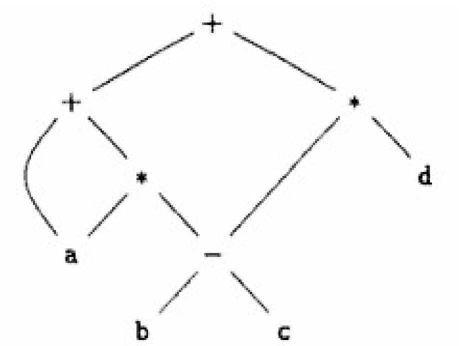
Variants of Syntax Trees (DAG)

A directed acyclic graph (**DAG**) for an **expression** identifies the common **sub expressions** (**sub expressions** that occur more than once) of the expression. **DAG**'s can be constructed by using the same techniques that construct syntax trees. A **DAG** has **leaves** corresponding to **atomic operands** and **interior nodes** corresponding to **operators**. A node **N** in a **DAG** **has more than one parent** if **N** represents a **common sub expression**, so a **DAG represents expressions concisely**. It gives clues to compiler about the generating **efficient code** to evaluate expressions.

Example 1: Given the grammar below, for the input string **id + id * id**, the **parse tree**, **syntax tree** and the **DAG** are as shown.

Grammar: E→E+T | T T→T*F | F F→ (E) | id	 Parse tree: E E T T T F F F id + id * id		

Example 2: **DAG** for the expression **(a + a * (b - c)) + ((b - c) * d)** is shown below.



Using the SDD (Syntax Direct Definition) to draw syntax tree or DAG for a given expression:-

- **Draw the parse tree**
- Perform a **post order** (left, right, and root) traversal of the parse tree
- Perform the **semantic actions** at every node during the traversal

– Creates a **syntax tree** if a **new node** is created each time functions **Leaf and Node** are called – Constructs a **DAG** if before creating a new node, these functions check whether an identical node already exists. If yes, the existing node is returned.

SDD to produce **Syntax trees** or **DAG** is shown below:

PRODUCTION	**SEMANTIC RULES**
E→ E_1+T	E.node= new Node('+',E_1.node,T.node)
E→ E_1-T	E.node= new Node('-',E_1.node,T.node)
E→ T	E.node=T.node
T→ (E)	T.node=E.node
T→ id	T.node = new Leaf(id,id.entry)
T→ num	T.node = new Leaf(num,num.val)

For the expression **(a + a * (b - c)) + ((b - c) * d)**, steps for constructing the **DAG** is as below.

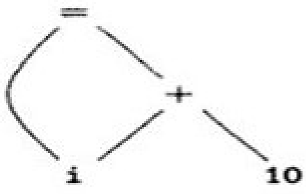
1) P_1= Leaf(id,entry-a)
2) P_2= Leaf(id,entry-a)= p_1
3) P_3= Leaf(id,entry-b)
4) P_4= Leaf(id,entry-c)
5) P_5= Node('-',p_3,p_4)
6) P_6= Node('*',p_1,p_5)
7) P_7= Node('+',p_1,p_6)
8) P_8= Leaf(id,entry-b)= p_3
9) P_9= Leaf(id,entry-c)= p_4
10) P_{10}= Node('-',p_3,p_4)= p_5
11) P_{11}= Leaf(id,entry-d)
12) P_{12}= Node('*',p_5,p_{11})
13) P_{13}= Node('+',p_7,p_{12})

Value-Number Method for Constructing DAGs

Nodes of a **syntax tree** or **DAG** are stored in an **array of records**. The **integer index** of the record for a node in the array is **known as the value number** of that node.

The **signature** of a node is a **triple < op, l, r>** where **op** is the **label**, **l** the value number of its **left child**, and **r** the value number of its **right child**. The value-number method for constructing the nodes of a **DAG** uses the **signature** of a node **to check if a node with the same signature already exists in the array**. If yes, returns the **value number**. **Otherwise**, **<u>creates a new node</u>** with the given signature.

Since searching an unordered array is slow, there are many better data structures to use. Hash tables are a good choice.



(a) DAG

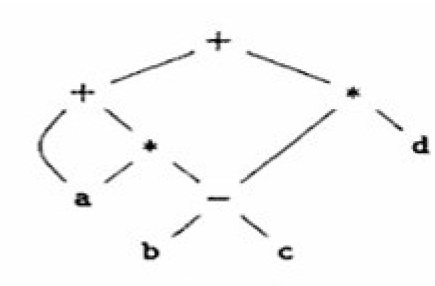
1	id	→ to entry for i	
2	num	10	
3	+	1	2
4	=	1	3
5	….		

Nodes of DAG for i=i+10 allocated in an array

<u>Three Address Code (TAC)</u>

TAC can range from **high- to low-level**, depending on the **choice of operators**. In general, it is a **statement containing at most 3 addresses or operands**.

The general form is **x: = y op z**, where **"op"** is an **operator**, **x** is the result, and **y** and **z** are **operands**. **x, y, z are variables**, **constants**, or "**temporaries**". A **three address instruction** consists of at most **3 addresses** for each statement. It is a linearized representation of a **binary syntax tree**. **<u>Explicit</u>** names correspond to **<u>interior nodes</u>** of the graph. E.g., for a **looping statement**, syntax tree represents components of the statement, whereas three-address code contains **labels and jump** instructions to represent the **flow-of-control** as in machine language.



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

A **TAC** instruction has at most **one operator** on the **RHS** of an instruction; no built up arithmetic expressions are permitted.

e.g. **x + y * z** can be translated as

t1 = y * z

t2 = x + t1

where **t1 & t2** are **compiler–generated temporary names**.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

Addresses and Instructions

• **TAC** consists of a **sequence of instructions**; each instruction may have up to **three addresses**, prototypically **t1 = t2 op t3**

• **Addresses** may be one of:

– A **name**. Each name is a **symbol table index**. For convenience, we write the **names as the identifier**.

– A **constant**.

– A **compiler-generated temporary**. Each time a temporary address is needed, the compiler generates another name from the stream t1, t2, t3, etc.

• Temporary names allow for code optimization to easily move instructions

• At target-code generation time, these names will be allocated to registers or to memory.

• TAC Instructions

– **Symbolic labels** will be used by instructions that alter the flow of control.

The instruction addresses of labels will be filled in later.

L: t1 = t2 op t3

– **Assignment instructions: x = y op z**

• Includes binary arithmetic and logical operations

– **Unary assignments**: **x = op y**

• Includes unary arithmetic op (-) and logical op (!) and type conversion

– **Copy instructions: x = y**

– **Unconditional jump: goto L**

• **L** is a symbolic label of an instruction

– **Conditional jumps**:

if x goto L If x is true, execute instruction L next

if False x goto L If x is false, execute instruction L next

– **Conditional jumps**:

if x relop y goto L

– **Procedure calls**. For a procedure call p(x1, …, xn)

```
param x1
…
param xn
call p, n
```

– **Function calls** : y= p(x1, …, xn) y = call p,n , return y

– **Indexed copy instructions**:   x = y[i]   and x[i] = y

- Left: sets x to the value in the location i memory units beyond y
- Right: sets the contents of the location i memory units beyond x to y

– **Address and pointer instructions**:

- x = &y sets the value of x to be the location (address) of y.
- x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.
- *x = y sets the value of the object pointed to by x to the value of y.

Example: Given the statement **do i = i+1; while (a[i] < v);** , the TAC can be written as below in **two ways**, using either **symbolic labels** or **position number** of instructions for labels.

L: t_1= i+1	100: t_1= i+1
i= t_1	101: i= t_1
t_2=i*8	102: t_2=i*8
t_3= a[t_2]	103: t_3= a[t_2]
If t_3 < v goto L	104: If t_3 < v goto 100
(a) Symbolic labels	**(b) Position numbers**

Data structures for representation of **<u>TAC can be objects or records</u>** with fields for operator and operands. Representations include **quadruples, triples and indirect triples**.

Examples

Write Three-address code for the following:

1. x = y * z + q / r

2. x = a[i], **where the array has type int**

Solution

1.	**2.**
t1 = y * z	
t2 = q / r	t1 = i * 4
x = t1 + t2	x = a [t1]

Quadruples

- In the **quadruple representation**, there are **four fields** for each instruction: **op, arg1, arg2, result**

– Binary ops have the obvious representation

– Unary ops don't use arg2

– Operators like param don't use either arg2 or result

– Jumps put the target label into result

The Figure below implement the three-address code in **(a)** and quadruples in **(b)**, for the expression

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

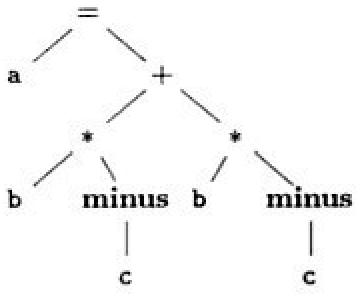
	op	arg_1	arg_2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

(b) Quadruples

Triples

• A **triple has only three fields** for each instruction: **op, arg1, arg2**
• The result of an operation x op y is referred to by its position.
• Triples are equivalent to signatures of nodes in DAG or syntax trees.
• Triples and DAGs are equivalent representations only for expressions; they are not equivalent for control flow.
• Ternary operations like x[i] = y requires two entries in the triple structure, similarly for x = y[i].
• Moving around an instruction during optimization is a problem

Example: Representations of a =b *– c + b * – c



(a) Syntax Tree

	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

Example: Translate the arithmetic expression **a+ - (b+c)** into **TAC, Quadruples, and Triples**

TAC	**Quadruples**				**Triples**			
t1= b+c	op	arg1	arg2	result		op	arg1	arg2
t2= - t1	+	b	c	t1	0	+	b	c
t3= a+t2	-	t1		t2	1	-	(0)	
	+	a	t2	t3	2	+	a	(1)

Indirect Triples

These **consist of a listing of pointers to triples**, rather than a listing of the triples themselves. An **optimizing compiler** can move an instruction by **reordering the instruction list**, **without affecting the triples themselves**.

	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)

Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code.

- All assignments in SSA are to variables with distinct names; hence static single-assignment.
- Φ-FUNCTION

Same variable may be defined in two different control-flow paths. For example,

if (flag) x = -1; else x = 1;
y = x * a;
using Φ-function it can be written as
if (flag) x_1 = -1; else x_2 = 1;
$x_3 = \Phi(x_1,x_2)$;
y = x_3 * a;

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.

Example: Convert the source program to SSA (Static single assignment form)
a = b + c
b = c + 1
d = b + c
a = a + 1
e = a + b

Solution

a1= b1 + c1
b2= c1 + 1
d1= b2 + c1
a2= a1 + 1
e1= a2 + b2

Types

A **type typically denotes a set of values and a set of operations** allowed on those values. **Applications of types** include **type checking** and **translation**.
Certain **operations are legal for each type**. For example, it doesn't make sense to add a function pointer and an integer in C. But it does make sense to **add two integers**. But both have the same assembly language implementation!
A **language's Type System specifies which operations** are **valid** for **which types**.
Type Checking is the process of **verifying fully typed programs**. Given an **operation and an operand of some type**, it **determines whether the operation is allowed**. The **goal of type checking** is to **ensure that operations are used with the correct types**. It uses logical rules to reason about the behavior of a program and enforces intended interpretation of values.
Type Inference is the process of filling in missing type information. Given the **type of operands, determine the meaning of the operation and the type of the operation; or, without variable declarations, infer type from the way the variable is used.**

Components of a Type System
- Built-in types
- Rules for constructing new types
- Rules for determining if two types are equivalent
- Rules for inferring the types of expressions

Type Expressions

Types have structure, represented using type expressions. **Type expressions** can be **either basic type** or **formed by applying type constructors**.

Example: an array type int[2][3] has the type expression is array(2, array(3, integer)) where **array** is the **operator** and takes **2 parameters**, a number and a type.

Definition of Type Expressions

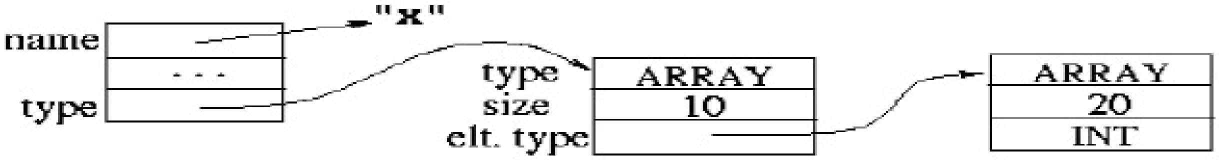
• A **basic type** is a **type expression**. Typical basic types for a language include Boolean, char, integer, float, and void.
• A **type name** is a **type expression**.
• A **type expression** can be formed by **applying the array type constructor** to a **number and a type expression**.
• A **record is a data structure** with **named fields**. A type expression can be formed by applying the **record type constructor** to the **field names and their types**.
• A type expression can be formed by using the **type constructor → for function types**. We write ***s***→ ***t*** for "function from type ***s*** to type ***t***.

If ***s*** and ***t*** are type expressions, then their **Cartesian product *s* × *t*** is a type expression. Products can be used to represent a list or tuple of types (e.g., for function parameters).
• Type expressions may contain variables whose values are type expressions.

Representing Type Expressions

– **Construct a DAG for type expression adapting the value-number method**.
– **Interior nodes represent type constructors**.
– **Leaves represent basic types, type names, and type variables**.

Type graphs: are graph-structured representations of type expressions:
– **Basic types are given predefined "internal values";**
– **Named types** can be represented via pointers into a hash table.
– A **composite type expression *f* (T_1,…,T_n)** is represented as a node identifying the constructor ***f*** and with pointers to the nodes for T_1, …,T_n.
E.g.: type graph for the type expression of int x[10][20] is shown below. (elt. =element)



<u>**Type Equivalence**</u>

The **two types of type equivalence** are **structural equivalence** and **name equivalence**.

<u>**Structural equivalence**</u>: When type expressions are represented by **graphs**, two types are **structurally equivalent** <u>**if and only if**</u> <u>**one**</u> of the following conditions is true:

1. They are the same basic type

2. They are formed by applying the same constructor to structurally equivalent types.

3. One is a type name that denotes the other

Name equivalence: If type names are treated as standing for themselves, **the first two conditions above lead to name equivalence of type expressions**.

Example 1: in the Pascal fragment

```
type p = ↑node;
     q = ↑node;
var x : p;
    y : q;
```

x and y are structurally equivalent, but not name-equivalent.

Example 2: Given the declarations

Type **t1** = Array [1..10] of integer;
Type **t2** = Array [1..10] of integer;

– **are they name equivalent?** No because they have different type names.

Example 3: Given the declarations

type vector = array [1..10] of real
type weight = array [1..10] of real
var x, y: vector; z: weight

Name Equivalence: When they have the same name.

– x, y have the same type; **z has a different type**.

Structural Equivalence: When they have the same structure.

– x, y, z have the same type.

Translation Applications of Types

From the **type of a name**, we can **determine the storage needed for the name at run time**, calculate the address denoted by an array reference, insert explicit type conversions, and choose the right version of an arithmetic operator.

Types and storage layout for names are declared within a procedure or class. **Actual storage is allocated at run time**. Type of a name can used to determine the amount of storage needed for the name at run time. **At compile time these amounts are used to assign each name a relative address**. Local declarations are examined and relative addresses are laid out with respect to an offset from the start of a data area. The **type and relative address are saved in the symbol-table entry for the name**. Pointer is assigned for data for varying length like strings and dynamic arrays.

Declarations

The grammar $D \rightarrow T\ id\ ;\ D \mid \varepsilon$
$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$
$B \rightarrow int \mid float$
$C \rightarrow \varepsilon \mid [\ num\ ]\ C$

declares just one name at a time;

D - generates a sequence of declarations,

T – generates basic, **array** or **record types**,

C – for 'component', generates **strings of zero or more integers** , each integer surrounded by brackets.