# Intermediate Code Generation

Lecture 18

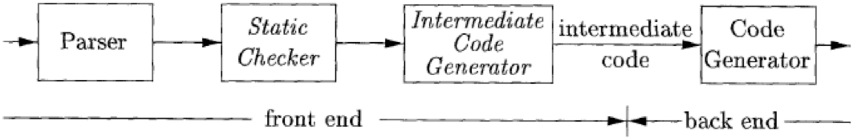
Chapter 6

Rokan U. Faruqui

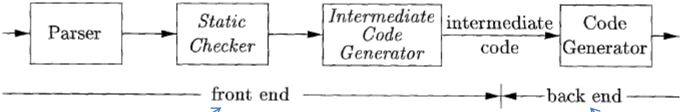
Associate Professor,
Dept of Computer Science and Engineering,
University of Chittaong
Email: *rokan@cu.ac.bd*

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation

print “Hello”;

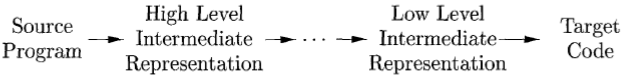



c++ —>[compiler]—> .exe/asm/bytecode



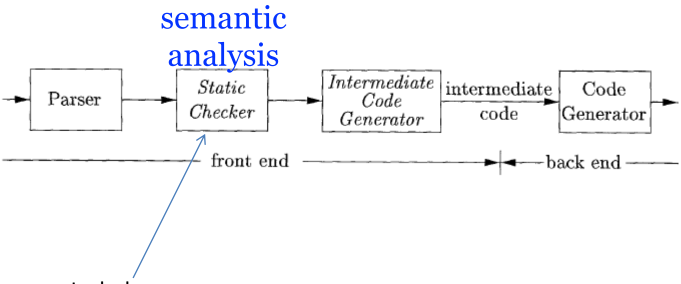

Close to source language (e.g. syntax tree)

Close to machine langua




m x n compilers can be built by writing just m front ends and n back ends

# Introduction




Includes:

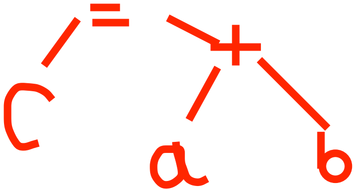
- Type checking
- Any syntactic checks that remain after parsing (e.g. ensure *break* statement is enclosed within while-, for-, or switch statements).

# Outline

1. Variants of Syntax Trees

2. Three Address Code

# Variants of Syntax Trees



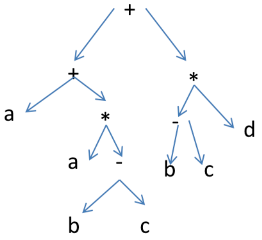

- It is sometimes beneficial to crate a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation

# Syntax Tree

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $E \rightarrow E_1 + T$ | $E.node = \textbf{new}\ Node('+', E_1.node, T.node)$ |
| 2) | $E \rightarrow E_1 - T$ | $E.node = \textbf{new}\ Node('-', E_1.node, T.node)$ |
| 3) | $E \rightarrow T$ | $E.node = T.node$ |
| 4) | $T \rightarrow (\ E\ )$ | $T.node = E.node$ |
| 5) | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6) | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$ |

**E.val = E1.val + T.val**

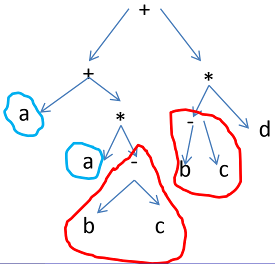
`a + a * (b - c) + (b - c) * d`



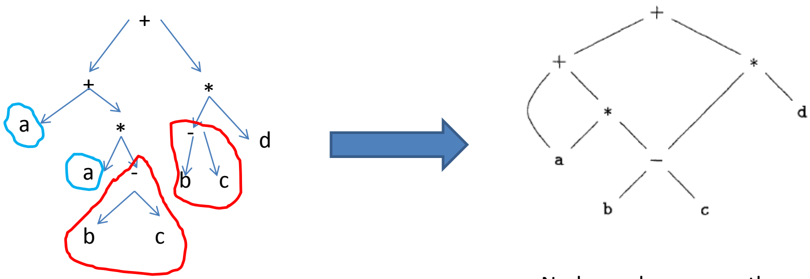
# Syntax Tree

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $E \to E_1 + T$ | $E.node = \textbf{new}\ Node('+', E_1.node, T.node)$ |
| 2) | $E \to E_1 - T$ | $E.node = \textbf{new}\ Node('-', E_1.node, T.node)$ |
| 3) | $E \to T$ | $E.node = T.node$ |
| 4) | $T \to (\ E\ )$ | $T.node = E.node$ |
| 5) | $T \to \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6) | $T \to \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$ |

```
a + a * (b - c) + (b - c) * d
```



# DAG




Node can have more than one parent

Directed Acyclic Graph (DAG):

- More compact representation
- Gives clues regarding generation of efficient code

# DAG

- Construct the dag for the following expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

# DAG from SDD

| | Production | Semantic Rules |
|---|---|---|
| 1) | $E \rightarrow E_1 + T$ | $E.node = \textbf{new } Node('+', E_1.node, T.node)$ |
| 2) | $E \rightarrow E_1 - T$ | $E.node = \textbf{new } Node('-', E_1.node, T.node)$ |
| 3) | $E \rightarrow T$ | $E.node = T.node$ |
| 4) | $T \rightarrow ( E )$ | $T.node = E.node$ |
| 5) | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$ **symbol table** |
| 6) | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$ |

All what is needed is that functions such as ***Node*** and ***Leaf*** above check whether a node already exists. If such a node exists, a pointer is returned to that node.

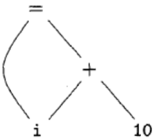
# DAG from SDD

| | Production | Semantic Rules |
|---|---|---|
| 1) | $E \rightarrow E_1 + T$ | $E.node = \textbf{new}\ Node('+', E_1.node, T.node)$ |
| 2) | $E \rightarrow E_1 - T$ | $E.node = \textbf{new}\ Node('-', E_1.node, T.node)$ |
| 3) | $E \rightarrow T$ | $E.node = T.node$ |
| 4) | $T \rightarrow ( E )$ | $T.node = E.node$ |
| 5) | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6) | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$ |

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
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) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

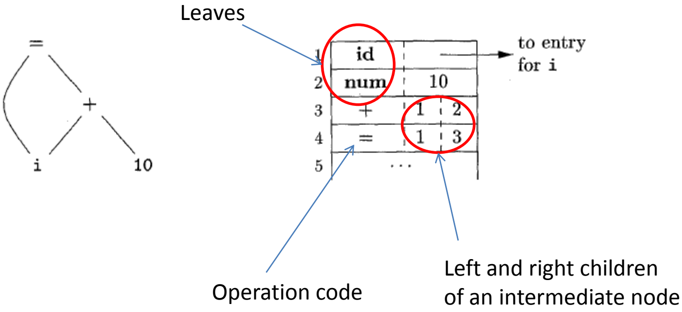
```
a + a * (b - c) + (b - c) * d
```

# Data Structure: Array




| | | | | |
|---|---|---|---|---|
| 1 | **id** | | | to entry for i |
| 2 | **num** | 10 | | |
| 3 | + | 1 | 2 | |
| 4 | = | 1 | 3 | |
| 5 | ... | | | |

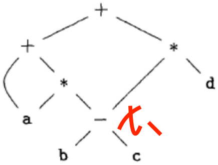
# Data Structure: Array



# Outline

# Three Address Code

- Another option for intermediate presentation
- Built from two concepts: addresses and instructions
- At most one operator




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

# Three Address Code

Address can be one of the following:

- A name: source program name
- A constant
- Compiler-generated temporary

# Instructions

Assignment instructions of the form $x = y \ op \ z$

Assignments of the form $x = op \ y$

*Copy instructions* of the form $x = y$

An unconditional jump `goto` $L$

Conditional jumps of the form `if` $x$ `goto` $L$ and `ifFalse` $x$ `goto` $L$

Conditional jumps such as `if` $x$ *relop* $y$ `goto` $L$ **if x relop y goto L**

Procedure call such as p(x1, x2, ..., xn) is implemented as:

```
param x1
param x2
...
param xn
call p, n
```

Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$

Address and pointer assignments of the form $x = \& y$, $x = * y$, and $* x = y$

# Example

```
do i = i+1; while (a[i] < v);
```

```
L:  t1 = i + 1
    i = t1
    t2 = i * 8
    t3 = a [ t2 ]
    if t3 < v goto L
```

OR

```
100:  t1 = i + 1
101:  i = t1
102:  t2 = i * 8
103:  t3 = a [ t2 ]
104:  if t3 < v goto 100
```

# Choice of Operator Set

- Rich enough to implement the operations of the source language
- Close enough to machine instructions to simplify code generation

# Data Structure

- How to present these instruction in a data structure
  - Quadruples
  - Triples
  - Indirect triples

# Quadruples

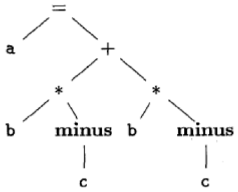
- Has four fields: op, arg1, arg2 and result

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

| | *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 |
| | | ... | | |

# Triples

- Temporaries are not used and instead references to instructions are made




(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) |
| | | . . . | |

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

(b) Triples

Representations of $a + a * (b - c) + (b - c) * d$

# Indirect Triples

- In addition to triples, a list of pointers to triples

| | *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) |
| | | . . . | |

List of pointers to triples

Optimizing complier can reorder instruction list, instead of affecting the triples themselves

# Example

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

| | *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 |
| | … | | | |