THEORY OF COMPUTATION AND COMPILERS

Unit - III

SEMANTIC ANALYSIS, INTERMEDIATE CODE GENERATOR & SYMBOL TABLE

SEMANTIC ANALYSIS

- Attributed Grammars
- Syntax Directed Translation

INTERMEDIATE CODE GENERATOR

- Intermediate Forms of Source Programs - Abstract Syntax Tree, Polish Notation and Three Address Codes
- Intermediate Code Forms
- Type Checker

SYMBOL TABLE

- Symbol Table Format
- Organization for Block Structures Languages
- Hashing

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

INTERMEDIATE-CODE GENERATION

Outline:

- Introduction to Intermediate-Code Generation
- Intermediate Forms of Source Programs
- Syntax Tree
- Directed Acyclic Graphs (DAG'S) for Expressions

Intermediate-Code Generation

- 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.
- Ideally, details of the source language are confined to the front end, and details of the target machine to the back end.
- With a suitably defined intermediate representation, a compiler for language ***i*** and machine ***j*** can then be built by combining the front end for language ***i*** with the back end for machine ***j***.
- This approach to creating suite of compilers can save a considerable amount of effort: ***m X n*** compilers can be built by writing just ***m*** front ends and ***n*** back ends.

Intermediate-Code Generation

Logical structure of a compiler front end is shown in below *fig*:

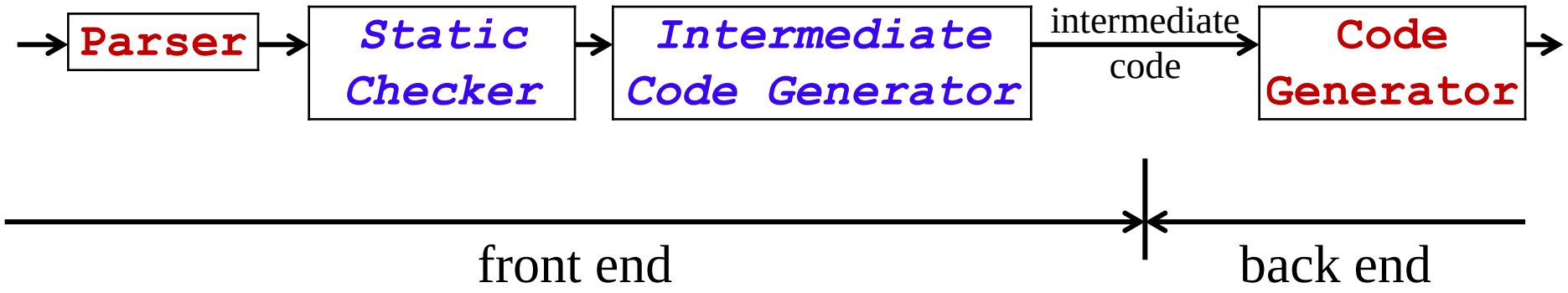

Fig: Logical structure of a compiler front end

Intermediate Forms of Source Programs

The following are commonly used intermediate code representations:

1. Postfix Notation (Also known as reverse Polish notation or suffix notation)
2. Syntax Tree
3. Three-Address Code

Intermediate-Code Generation

- 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-**, f**or-**, or **switch-statement**; an error is reported if such an enclosing statement does not exist. The approach in this chapter can be used for a wide range of intermediate representations, including syntax trees, Postfix Notation (or reverse Polish or suffix notation notation) and three-address code.

Intermediate-Code Generation

- The term "three-address code" comes from instructions of the general form ***x = y op z*** with three addresses: two for the **operands** ***y*** and ***z*** and one for the result ***x***.
- 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 shown in below *Fig*.

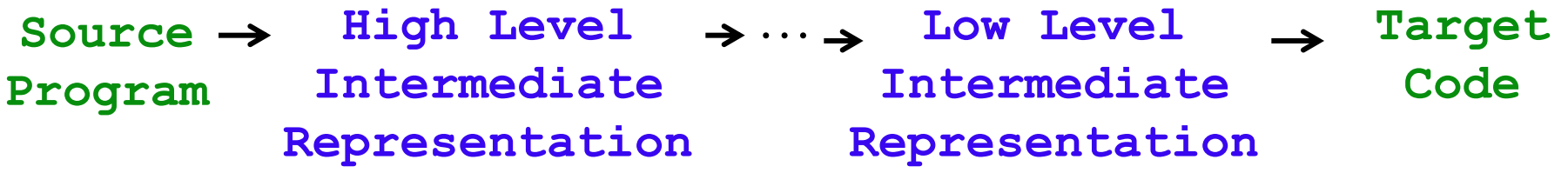

Fig: A compiler might use a sequence of intermediate representations

Intermediate-Code Generation

- **High-level representations** are close to the source language and **low-level representations** are close to the target machine.
- **Syntax trees** are **high level**; they depict the natural hierarchical structure of the source program and are well suited to tasks like **static type checking**.
- A **low-level representation** is suitable for machine-dependent tasks like **register allocation** and **instruction selection**.
- **Three-address code** can range from high- to low-level, depending on the choice of operators.

Intermediate-Code Generation

- For expressions, the differences between **syntax trees** and **three-address code** are superficial.
- For looping statements, for example, a **syntax tree** represents the components of a statement, whereas **three-address code** contains labels and jump instructions to represent the flow of control, as in **machine language**.
- The choice or design of an **intermediate representation** varies from compiler to compiler.
- An **intermediate representation** may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler.

Intermediate-Code Generation

- **C** is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available.
- The original **C++** compiler consisted of a front end that generated **C**, treating a **C** compiler as a back end.

Postfix Notation

- Also known as reverse Polish notation or suffix notation.
- In the infix notation, the operator is placed between operands, e.g., `a + b`
- Postfix notation positions the operator at the right end, as in `ab+`
- For any postfix expressions e_1 and e_2 with a binary operator (+) , applying the operator yields $e_1 e_2 +$.
- Postfix notation eliminates the need for parentheses, as the operator's position and arity allow unambiguous expression decoding.
- In postfix notation, the operator *consistently follows* the operand.

Postfix Notation

Example 1:

The postfix representation of the expression `(a + b) * c` is : `ab + c *`

Example 2:

The postfix representation of the expression `(a – b) * (c + d) + (a – b)` is :

`ab – cd + *ab –+`

Variants of Syntax Trees

- ***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.
- As we shall see in this section, **DAG**'s can be constructed by using the same techniques that construct **syntax trees**.

Directed Acyclic Graphs for Expressions

- Like the ***syntax tree*** for an expression, a **DAG** has leaves corresponding to atomic operands and interior nodes corresponding to operators.
- The difference is that a node ***N*** in a **DAG** has more than one parent if ***N*** represents a common subexpression; in a ***syntax tree***, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.
- Thus, a **DAG** not only represents expressions more clearly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Directed Acyclic Graphs for Expressions

Example-1: The following *Figure*. shows the **DAG** for the expression **a + a * (b - c) + (b - c) * d**

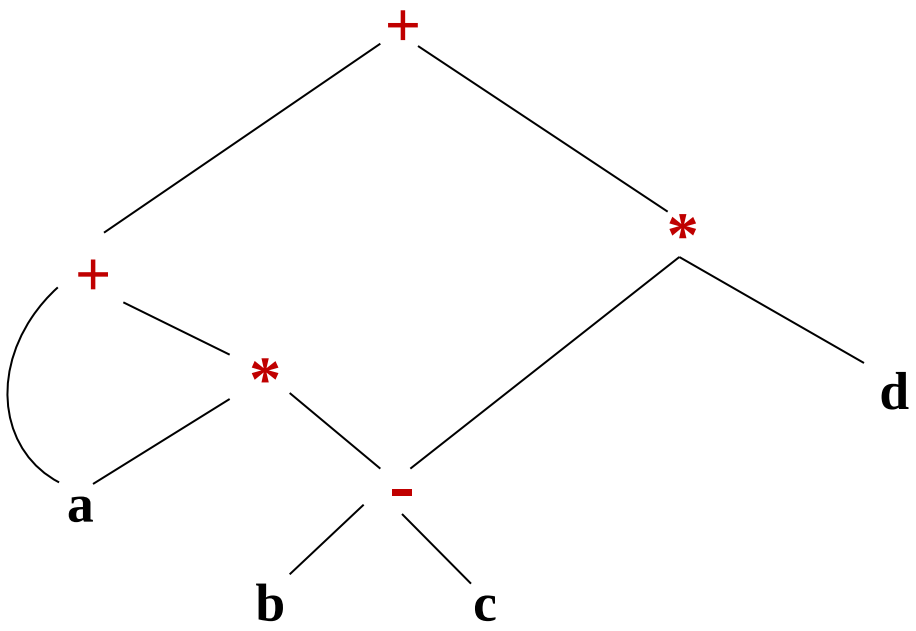

Fig: **DAG** for the expression **a + a *(b-c)+(b-c)*d**

Directed Acyclic Graphs for Expressions

Example-1:

- The leaf for **a** has two parents, because **a** appears twice in the expression.
- More interestingly, the two occurrences of the common subexpression **b-c** are represented by one node, the node labeled **-**. That node has two parents, representing its two uses in the subexpressions **a*(b-c)** and **(b-c)*d**.
- Even though **b** and **c** appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression **b-c**.

Directed Acyclic Graphs for Expressions

Example-2: Syntax-Directed Definition (SDD) to produce syntax trees or DAG's

Productions	Semantic Rules
$E \rightarrow E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	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)

Directed Acyclic Graphs for Expressions

Example-2: Steps for constructing the DAG of the above SDD

The sequence of steps shown below constructs the **DAG** as shown in the above *Fig.*, provided ***Node*** and ***Leaf*** return an existing node, if possible.

1) p_1 = **new** ***Leaf*** **(id,** ***entry-a*****)**;

2) p_2 = **new** ***Leaf*** **(id,** ***entry-a*****)** = p_1;

3) p_3 = **new** ***Leaf*** **(id,** ***entry-b*****)**;

4) p_4 = **new** ***Leaf*** **(id,** ***entry-c*****)**;

5) p_5 = **new** ***Node*****('-',** p_3**,** p_4**)**;

6) p_6 = **new** ***Node*****('*',** p_1**,** p_5**)**;

Directed Acyclic Graphs for Expressions

Example-2: Steps for constructing the DAG of the above SDD

7) p_7 = new Node('+', p_1, p_6);

8) p_8 = new Leaf (id, entry-b) = p_3;

9) p_9 = new Leaf (id, entry-c) = p_4;

10) p_{10} = new Node('-', p_3, p_4) = p_5;

11) p_{11} = new Leaf (id, entry-d);

12) p_{12} = new Node('*', p_5, p_{11});

13) p_{13} = new Node('+', p_7, p_{12});

Summary

Intermediate-Code Generation

- Introduction to Intermediate-Code Generation
- Intermediate Forms of Source Programs
- Syntax Tree
- Directed Acyclic Graphs (DAG'S) for Expressions

Reading: Aho2, Section 6.1: 6.1.1 & 6.1.2

Next Lecture: Three-Address Code