COMPILER CONSTRUCTION (Intermediate Code Generation)

Prof. K R Chowdhary
Email: kr.chowdhary@jietjodhpur.ac.in

Campus Director, JIET, Jodhpur

Thursday 18th October, 2018

Introduction

- In the analysis-synthesis model of a compiler, the *front end* analyzes a source program and creates an intermediate representation, from this onward the *back end* works and generates the target code.
- With a properly defined intermediate representation, a compiler for any language i and any machine j can then be built by combining the front end for language i with the back end for machine j.
- For simplicity, we assume that a compiler front end is organized as shown in Fig. 1, where parsing, *static checking*, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing.

Compiler front end

- Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks

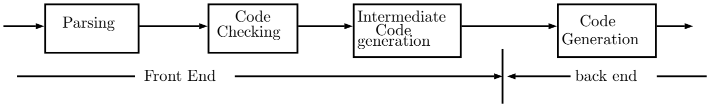

Figure 1: Logical structure of a compiler front end.

- There can be a wide range of intermediate representations, including syntax trees and three-address code. The term "three-address code" comes from instructions of the form $x = y\ op\ x$

A sequence of intermediate representations

- 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

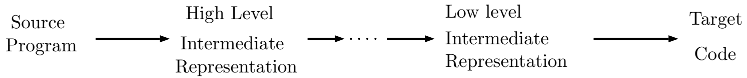
Source Program
High Level Intermediate Representation
· · · ·
Low level Intermediate Representation
Target Code

Figure 2: A compiler might use a sequence of intermediate representations.

- A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high-level to low-level,

Assembly code vs. intermediate representation

- For example the following sequence of statements are close to the assembly language.

$$\begin{aligned} label1:\ & x_i = y_i + z_i \\ & i = i + 1 \\ & if\ \ x < 100\ goto\ \ label1 \\ & next\ \ statement \end{aligned}$$

- 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.

Applications of Syntax-Directed Translation

- The main application of SDD is in construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree.
- We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, which is suitable for use during bottom-up parsing. The second, L-attributed,
- *Construction of Syntax Trees.* A syntax-tree node representing an expression "$E_1 + E_2$" has label $+$ and two children representing the subexpressions E_1 and E_2.
- We can implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field

Variant of Syntax Trees: DAG

- A directed acyclic graph (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.
- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression;

Construction of DAG

Example

Construct DAG for the expression:

$$a + a * (b - c) + (b - c) * d.$$

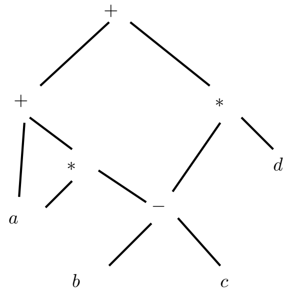

Figure 3: Dag for the expression $a + a * (b - c) + (b - c) * d$.

Construction of DAG...

- Figure 3 shows the DAG for above expression. The leaf for a has two parents, because a appears twice in the expression. The two occurrences of the common subexpression $b - c$ are represented by one node, the node labeled $-$.
- The SDD of Table 1 can be used to construct either syntax trees or a DAG's. The functions *Leaf* and *Node*, create a fresh node each time they are called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists.

Construction of DAG...

Table 1: Syntax-directed definition to produce syntax trees or DAG's

S.No.	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = new\ Node('+', E.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = new\ Node('-', E.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow id$	$T.node = new\ Leaf(id, id.entry)$
6)	$T \rightarrow num$	$T.node = new\ Leaf(num, num.val)$

If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, $Node(op, left, right)$ we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, Node returns the existing node; otherwise, it creates a new node.

Construction of DAG...

Table 2: Steps for constructing the DAG of Fig. 3 for $a + a * (b - c) + (b - c) * d$

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

Constructing a DAG

Example

Construct a Dag (i.e., Fig. 3) for the expression $a + a * (b - c) + (b - c) * d$.

Solution. The sequence of steps shown in Table 2 constructs the DAG of Fig. 3, provided *Node* and *Leaf* return an existing node, if possible. We assume that *entry-a* points to the symbol-table entry for *a*, and similarly for the other identifiers.
When the call to *Leaf(id, entry-a)* is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_8 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_9$. □