بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Pick an intermediate representation

An intermediate representation is typically some combination of a graphical notation and three-address code. As in syntax trees, a node in a graphical notation represents a construct; the children of a node represent its subconstructs. Three address code takes its name from instructions of the form $x = y \text{ op } z$*, with at most one operator per instruction. There are additional instructions for control flow.*

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 in Fig. 6.2. 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.

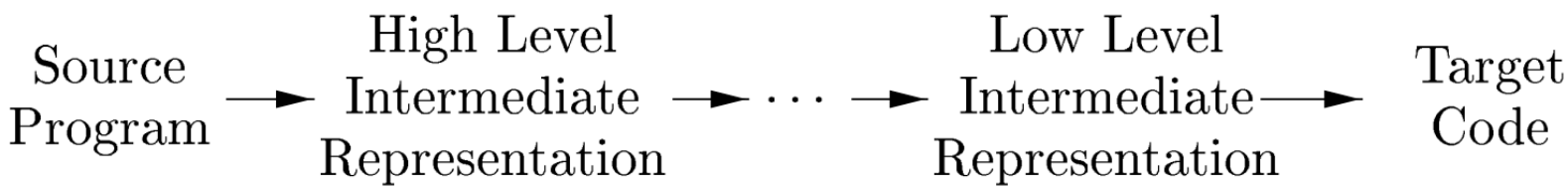


Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. 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. 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.

DAGs

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 succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1 : Figure 6.3 shows the DAG for the expression

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

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`. □

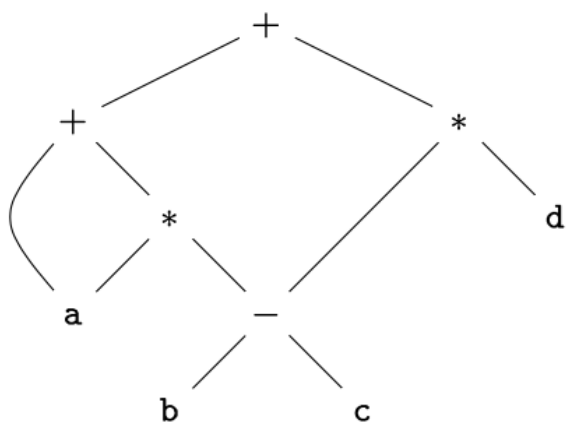
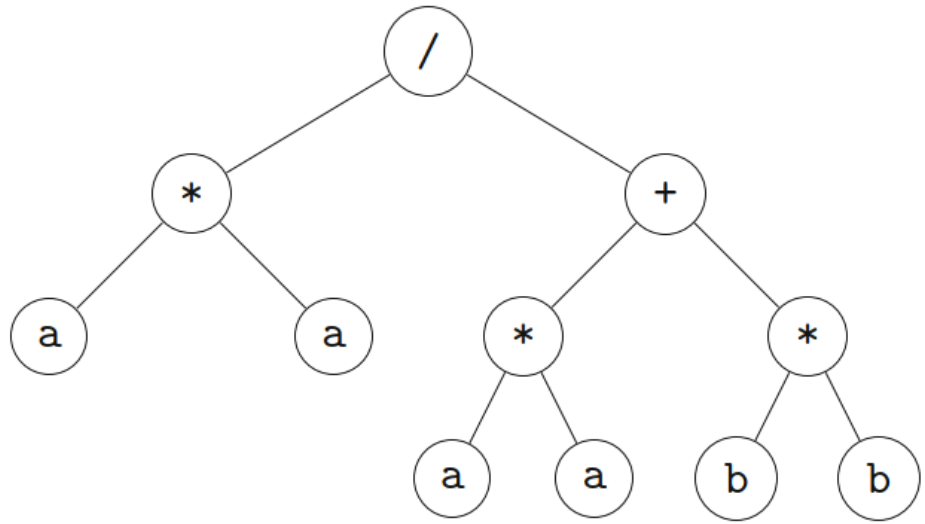
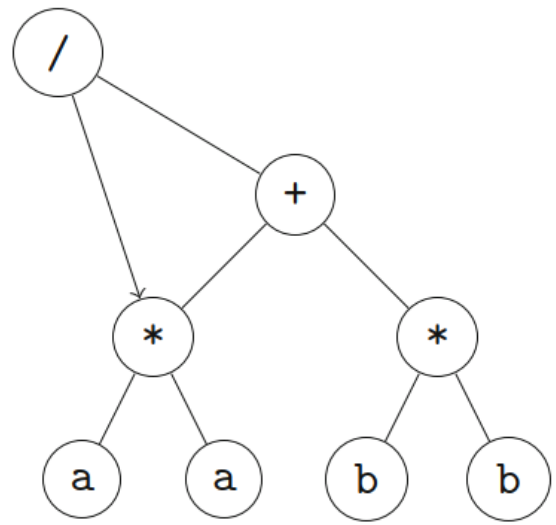


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

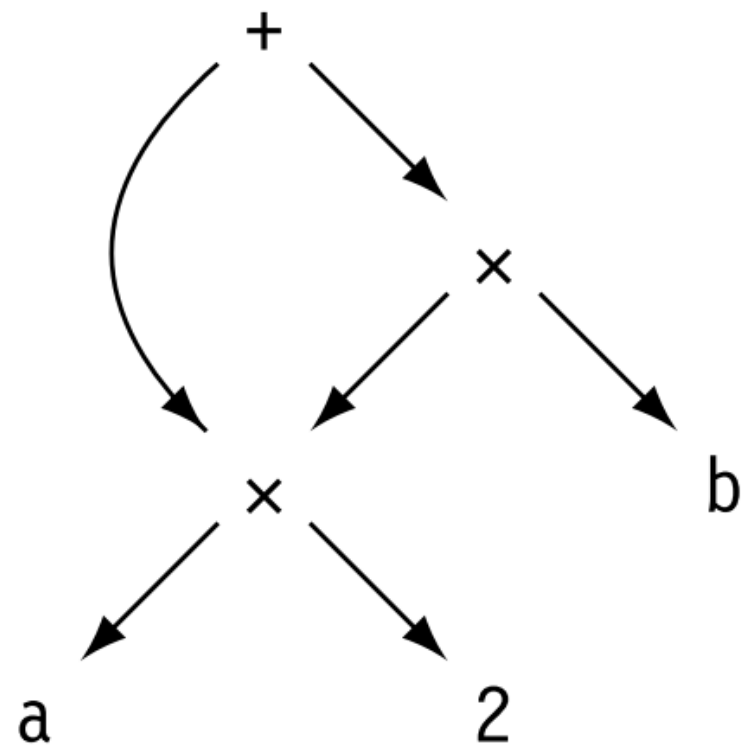
Tree for a ∗ a/(a ∗ a + b ∗ b)



Common subexpression



A DAG is an AST that represents each unique subtree once. DAGs are often called ASTs with sharing.



DAG for $a \times 2 + a \times 2 \times b$

Three-address code

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.

Three-address code is a sequence of elementary program steps, such as the addition of two values. Unlike the tree, there is no hierarchical structure. As we shall see in Chapter 9, we need this representation if we are to do any significant optimization of code. In that case, we break the long sequence of three-address statements that form a program into "basic blocks," which are sequences of statements that are always executed one-after-the-other, with no branching.

There are several points worth noting about three-address instructions.

☞ *First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done.*

☞ *Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction.*

☞ *Third, some "three-address instructions" like the first and last in the sequence*

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

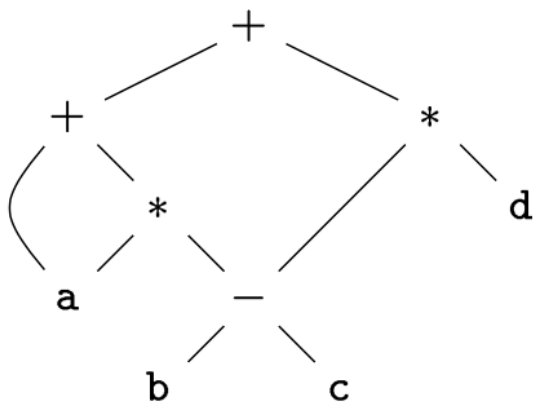
have fewer than three operands.

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like $\mathtt{x} + \mathtt{y} * \mathtt{z}$ *might be translated into the sequence of three-address instructions*

$$\mathtt{t}_1 = \mathtt{y} * \mathtt{z}$$
$$\mathtt{t}_2 = \mathtt{x} + \mathtt{t}_1$$

where $\mathtt{t}_1$ *and* $\mathtt{t}_2$ *are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization, as discussed in Chapters 8 and 9. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.*

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence. □



(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

Figure 6.8: A DAG and its corresponding three-address code

A list of the common three-address instruction forms:

1. Assignment instructions of the form $x = y \ op \ z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \ y$, where op is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

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

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$. The instruction $x = \&y$ sets the r-value of x to be the location (l-value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction $x = *y$, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, $*x = y$ sets the r-value of the object pointed to by x to the r-value of y.

Example 6.5 : Consider the statement

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

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9(a) uses a symbolic label L, attached to the first instruction. The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space. □

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

(a) Symbolic labels.

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

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

6.6.2 Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21 : The statement

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

might be translated into the code of Fig. 6.34. In this translation, the boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to L_1, skipping L_2 and the assignment `x = 0`. □

```
      if x < 100 goto L2
      ifFalse x > 200 goto L1
      ifFalse x != y goto L1
L2:   x = 0
L1:
```

Figure 6.34: Jumping code

برای ساخت *3AC* نظیر یک عبارت، می‌توان از *SDD*ها بهره گرفت

Example 6.22 : Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;          (6.13)
```

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        if x < 100 goto L2
        goto L3
L3 :    if x > 200 goto L4
        goto L1
L4 :    if x != y goto L2
        goto L1
L2 :    x = 0
L1 :
```

Figure 6.38: Control-flow translation of a simple if-statement

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.