Intermediate Code Generation

Zakia Zinat Choudhury
Lecturer
Department of Computer Science & Engineering
University of Rajshahi

1

Introduction

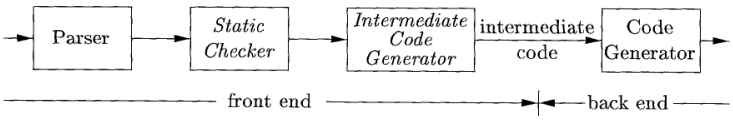
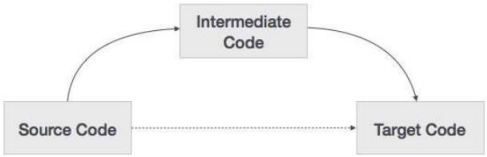


Figure: Logical structure of a compiler front end

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

2

Why Intermediate Code Generator is Used to Translate Target Code?



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

3

Directed Acyclic Graph

A directed acyclic graph (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression.

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks.

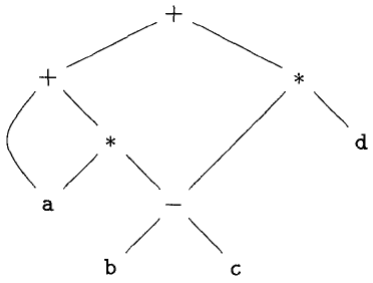
DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

4

Example of Directed Acyclic Graph

The constructed DAG for the Expression `a + a * (b - c) + (b - c) * d`



5

Three-Address Code

A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. 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 **x+y*z** might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

where t1 and t2 are compiler-generated temporary names.

A three-address code has at most three address locations to calculate the expression. Three-address code is built from **two concepts: addresses and instructions**. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses.

6

Addresses

An address can be one of the following:

- **A name:** For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- **A constant:** In practice, a compiler must deal with many different types of constants and variables.
- **A compiler-generated temporary:** It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

7

Instructions

Here is 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, shift operators, 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.

8

Instructions

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; callp, n and y = callp, 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(xl,x2,. . . ,x,). The integer n, indicating the number of actual parameters in "call p, n," is not redundant because calls can be nested.

9

Instructions

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 y2. Presumably, y is a name, perhaps a temporary, that denotes an expression with an expression with an l-vale 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.

10

Format of Three-Address Code

A three-address code can be represented in two forms :
quadruples and triples.

11

Quadruples

A quadruple (or just "quad') has four fields, which we call op, arg1, arg2, and result. The op field contains an internal code for the operator.
For instance, the three-address instruction x = y +z is represented by placing + in op, y in arg1, z in arg2, and x in result.

The following are some exceptions to this rule:

1. Instructions with unary operators like x = minus y or x = y do not use arg2.
2. Operators like param use neither arg2 nor result.
3. Conditional and unconditional jumps put the target label in result.

12

Example of Quadruples

Three-address code for the assignment `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		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
		...		

(b) Quadruples

13

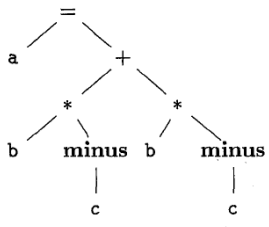
Triples

A triple has only three fields, which we call op, arg1, and arg2. The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions Note that the result field in is used primarily for temporary names. Using triples, we refer to the result of an operation x op y by its position, rather than by an explicit temporary name.

14

Example of Triples

Three-address code for the assignment `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

15

Indirect Triples

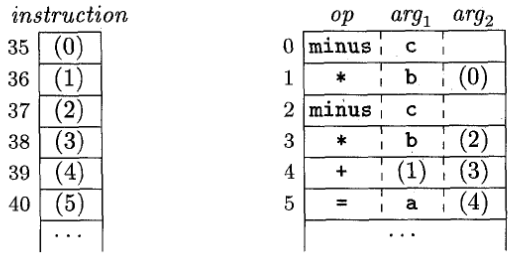
This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

16

Example of Indirect Triples

Three-address code for the assignment `a = b*-c+b*-c;`



	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)
		...	

17

Types and Declarations

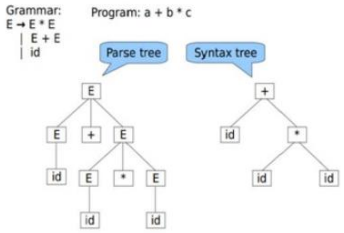
Read from the Textbook

Page 371-373

18

Difference between syntax tree and Parse tree

Parse Tree	Syntax Tree
Interior nodes are non-terminals, leaves are terminals.	Interior nodes are "operators", leaves are operands.
Rarely constructed as a data structure.	When representing a program in a tree structure usually use a syntax tree.
Represents the concrete syntax of a program.	Represents the abstract syntax of a program (the semantics).



19

Notation

The way to write arithmetic expression is known as a notation.

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.

These notations are:

- Infix Notation
- Polish (Prefix) Notation
- Reverse-Polish (Postfix) Notation

20

Infix Notation

The expression writes in infix notation where operators are used in-between operands.

It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Example:

Infix notation with parenthesis: (3 + 2) * (5 – 1)

21

Polish Notation

Polish notation is a notation form for expressing arithmetic, logic and algebraic equations. Its most basic distinguishing feature is that **operators are placed on the left or written ahead** of their **operands.** If the operator has a defined fixed number of operands, the syntax does not require brackets or parenthesis to lessen ambiguity.

Example:

Infix notation with parenthesis: (3 + 2) * (5 – 1)
Polish notation: * + 3 2 – 5 1

Polish notation is also known as **prefix notation**.

22

Reserved Polish Notation

Reserved Polish notation most basic distinguishing feature is that **operators are placed on the right or written after** of their **operands.**
The idea is simply to have a parenthesis-free notation that makes each equation shorter and easier to parse in terms of defining the evaluation priority of the operators.
Example:

Infix notation with parenthesis: (3 + 2) * (5 – 1)
Reserved Polish notation: 3 2+ 5 1-*

Reserved Polish notation is also known as **postfix notation**.

23

Boolean Expression

Boolean expressions **are composed of the Boolean operators** (**which we denote &&, ||, and !, using the C convention for the operators AND, OR, and NOT, respectively**) applied **to elements that are Boolean variables or relational expressions**. Relational expressions are of the form El re1 E2, where El and E2 are arithmetic expressions. In this section, we consider Boolean expressions generated by the following grammar:

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B \,) \mid E \;\mathbf{rel}\; E \mid \mathbf{true} \mid \mathbf{false}$$

We use the attribute rel.op to indicate which of the six comparison operators <, <=, =, !=, >, or >= is represented by rel. As is customary, we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

Given the expression B1 || B2, if we determine that B1 is true, then we can conclude that the entire expression is true without having to evaluate B2. Similarly, given B1&&B2, if B1 is false, then the entire expression is false.

24

Why Boolean Expression is Used?

The translation of statements such as if-else-statements and while-statements is tied to the translation of Boolean expressions. In programming languages, Boolean expressions are often used to

1. **Alter the flow of control:** Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such Boolean expressions is implicit in a position reached in a program. For example, in if (E) S, the expression E must be true if statement S is reached.

2. **Compute logical values:** A Boolean expression can represent true or false as values. Such Boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

25

Control-Flow Translation of Boolean Expression

Read from the Textbook
Page 403-405

26

Assignments

1. Construct the DAG for the expression:
   ((x + y)-((x + y)*(x - y)))+((x + y)*(x - y))
2. Translate the arithmetic expression a = b * c – b * d into:
   a) A syntax tree
   b) Quadruples
   c) Triples
   d) Indirect Triples

27

Thank You

28