Intermediate Code Generation

Sudakshina Dutta

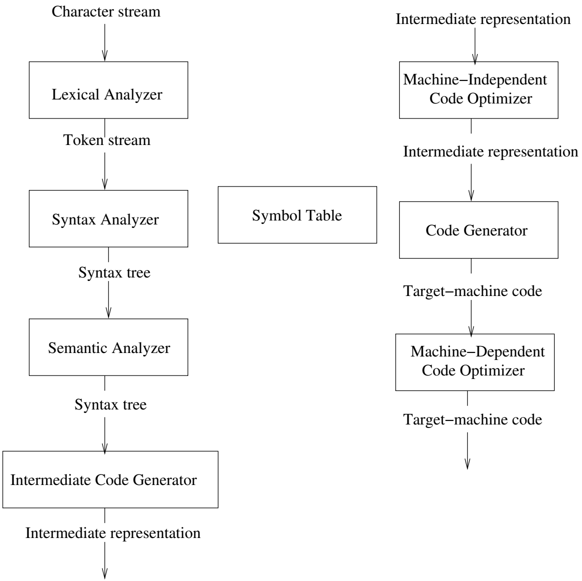
IIT Goa

8^{th} April, 2022

Intermediate Code Generation

- The front end analyzes the source program and creates an intermediate representation, from which the back end generates target code
- If the intermediate code is suitable, then the combination of any front end with any back end can be performed
- Considerable amount of effort is saved
  - $m \times n$ compilers can be built by writing just m front ends and n back ends

Phases of a compiler



Intermediate Code Generation

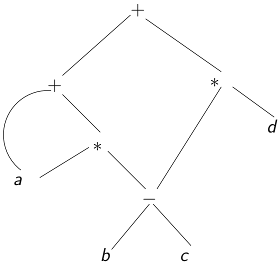
- Choice of intermediate representation varies from compiler to compiler
- Intermediate representation may contain data structures which are shared by phases of the compiler
- It is a sort of universal assembly language with no machine dependent constructs
- Example - Directed acyclic graph, three address code, quadruple, triple, static single assignment form

Directed Acyclic Graph

- A variant of syntax tree
- Consists of nodes and edges
- Leaves correspond to atomic operands and interior nodes correspond to operators
- A node with common sub-expression can be referred from different nodes

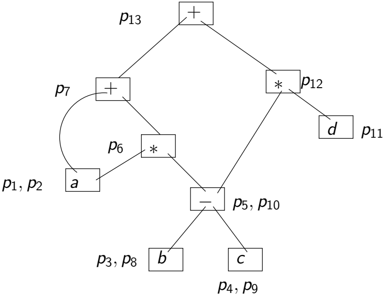
Directed Acyclic Graph

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



- Note the common sub-expression is $b - c$

Directed Acyclic Graph



1) $p_1 = Leaf(id, entry - a)$

2) $p_2 = Leaf(id, entry - a) = p_1$

3) $p_3 = Leaf(id, entry - b)$

4) $p_4 = Leaf(id, entry - c)$

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) $p_8 = Leaf(id, entry - b) = p_3$

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

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

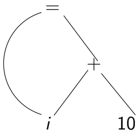
11) $p_{11} = Leaf(id, entry - d)$

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

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

- While creating nodes, the semantic rules check whether the identical node with the same operator and the left and the right children is present in the symbol table
- If it exists, then the existing node is returned

Value-Number Method for constructing DAG



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

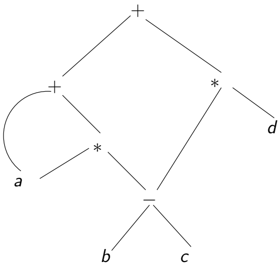
- Implemented by hash table
- Example — The node labeled + has value number
- The left and right children have value numbers 1 and 2, respectively

Three-Address Code

- There is at most one operator on the right hand side
- The source-language expression $x + y * z$ might be translated into the following sequence of three-address instructions
  - $t_1 = y * z$
  - $t_2 = x + t_1$
- 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

DAG versus three-address-code

- The expression is $a + a * (b - c) + (b - c) * d$



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

Three-address code

- An address can be of following types:
  - Source-program names can appear in three-address code. In the implementation, it is replaced by a pointer to the symbol-table entry
  - A constant
  - A compiler-generated temporary

Three-address code

- A three-address instruction can be of following types:
  - Assignment instructions of the form x = y op z
    — op is a binary (arithmetic or logical) operation and x, y and z are addresses
  - Assignments of the form x = op y , where op is a unary operation
    — An unary operation can be unary minus, logical negation, conversion operator
  - Copy instructions of the form x = y
  - An unconditional jump of the form goto L

Three-address code

- Conditional jump of the form if x goto L and ifFalse x goto L
- Conditional jumps such as if x relop y goto L, which apply a relational operator ($<$, $==$, $>=$, etc)
  — If the condition is not true, the instruction following this one is executed
- Procedure calls and returns
  - param x for parameters
  - call p, n for procedure call
  - y = call p, n for function call
  - Sequence of three-address instructions for function call p(x_1, x_2, $\cdots$, x_n)
    param x_1
    param x_2
    $\ldots$
    param x_n

Three-address code

- Indexed copy instruction of the form $x = y[i]$ and $x[i] = y$
  — The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y
- Address and pointer assignments of the form $x = y$, $x = *y$, and $*x = y$
  — The instruction $x = \&y$ sets r$-$value of x to be the l$-$value of y

Example of three-address code

Consider the code segment
for $(i = 1;\ i < n;\ i++)\ a[i] = i;$

$t_1 = 1$
$i = t_1$
if $i < n$ goto L
goto L'
$L:\ t2 = i * 8$
$\quad a[t2] = i$
$L':$

This is wrong right?