南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Chapter 5: Intermediate-Code Generation

Yepang Liu

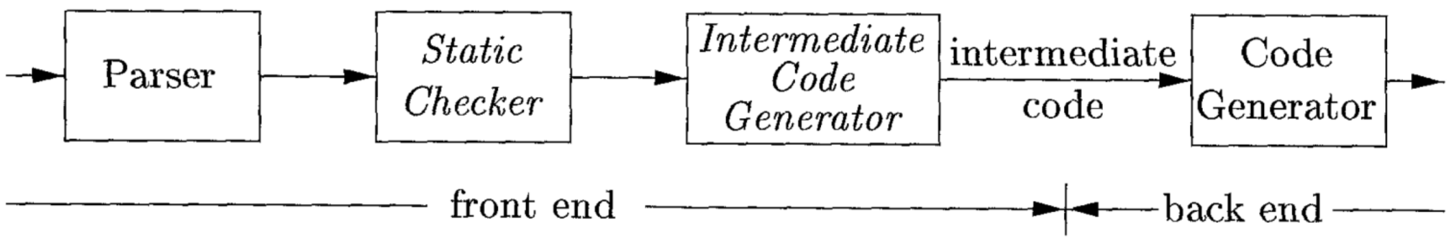
liuyp1@sustech.edu.cn

Outline

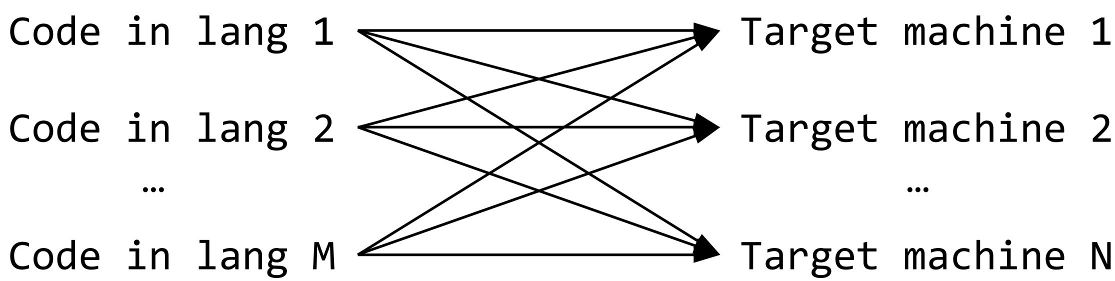
- Intermediate Representation
- Type and Declarations
- Translation of Expressions
- Type Checking
- Control Flow
- Backpatching

Compiler Front End

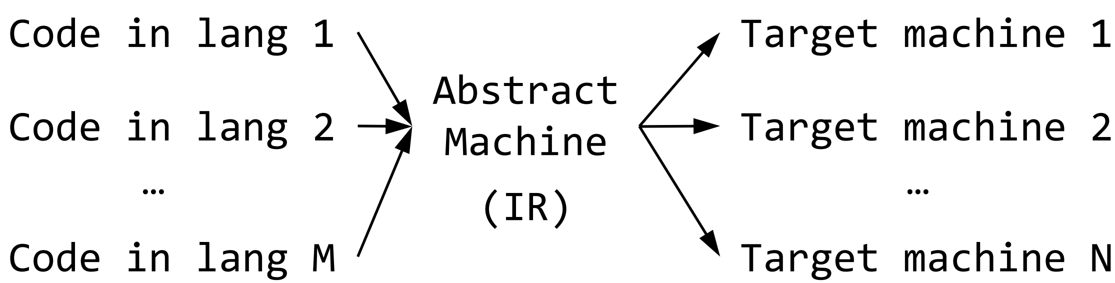
- The front end of a compiler analyzes a source program and creates an intermediate representation (IR, 中间表示), from which the back end generates target code
  - Details of the source language are confined to the front end, and details of the target machine to the back end



The Benefits of A Common IR



M * N compilers
without a common IR



M + N compilers
with a common IR

Different Levels of IRs

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

- A compiler may construct a sequence of IR's
  - High-level IR's like syntax trees are close to the source language
    - They are suitable for machine-independent tasks like static type checking
  - Low-level IR's are close to the target machines
    - They are suitable for machine-dependent tasks like <u>register allocation</u> and <u>instruction selection</u>

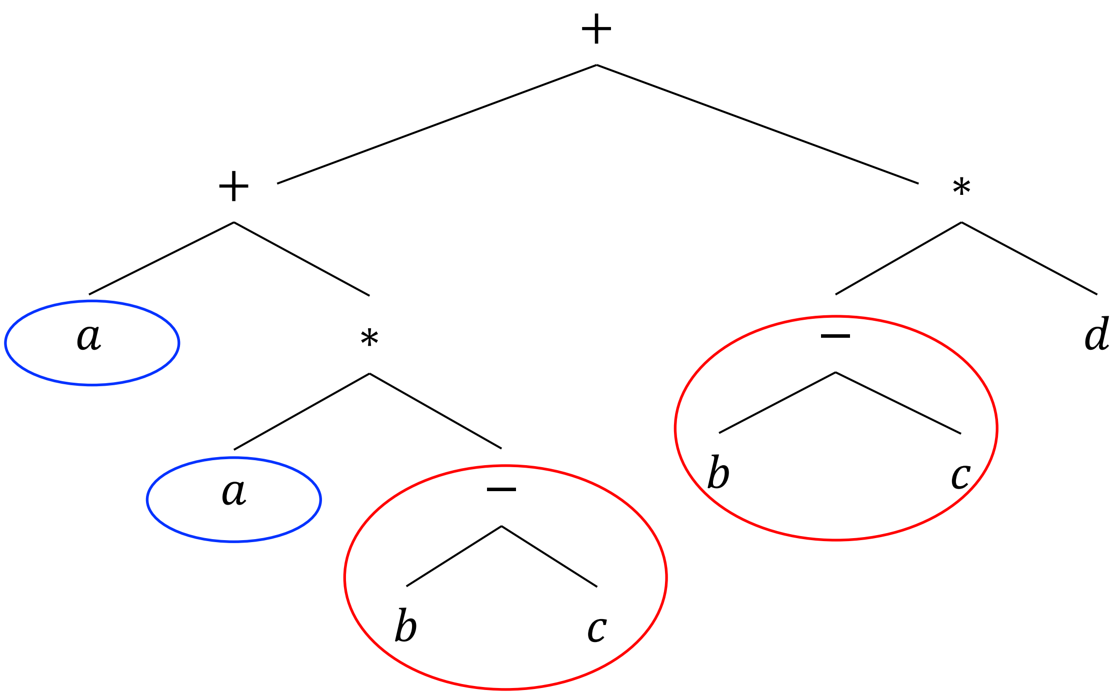
- Interesting fact: C is often used as an intermediate form. The first C++ compiler has a front end that generates C and a C compiler as a backend

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Translation of Expressions
- Type Checking
- Control Flow
- Backpatching

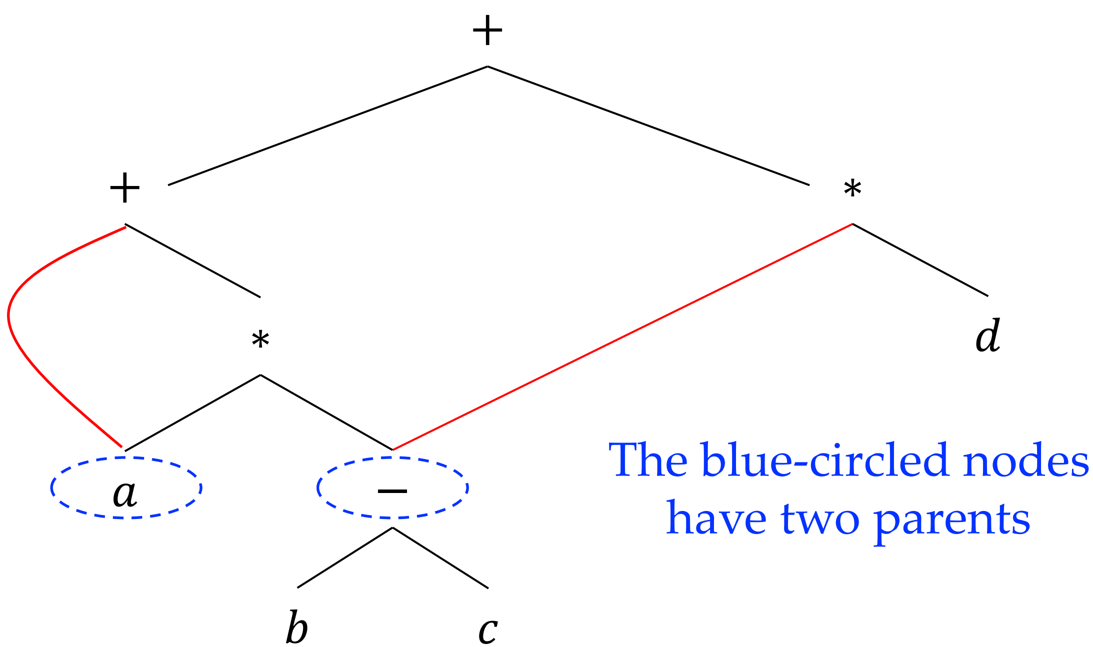
DAG's for Expressions

- In a syntax tree, the tree for a common subexpression would be replicated as many times as the subexpression appears
  - Example: $a + a * (b - c) + (b - c) * d$



DAG's for Expressions Cont.

- A *directed acyclic graph* (DAG, 有向无环图) identifies the common subexpressions and represents expressions succinctly
  - Example: $a + a * (b - c) + (b - c) * d$



Constructing DAG's

- DAG's can be constructed by the same SDD that constructs syntax trees
- **The difference:** When constructing DAG's, a new node is created if and only if there is no existing identical node

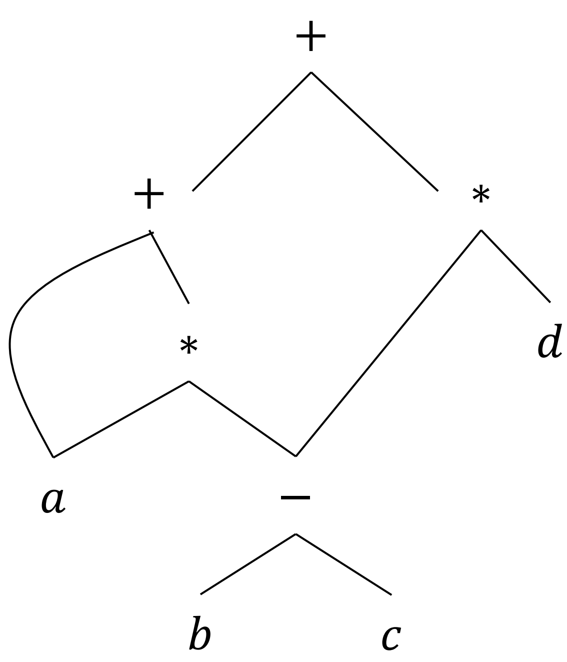
	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \boxed{\textbf{new}}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \boxed{\textbf{new}}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \boxed{\textbf{new}}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \boxed{\textbf{new}}\ Leaf(\textbf{num}, \textbf{num}.val)$

Special "new":

Reuse existing nodes when possible

Constructing DAG's Cont.

- The construction steps



1) $p_1 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}a)$
2) $\boxed{p_2 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}a) = p_1}$
3) $p_3 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}b)$
4) $p_4 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}c)$
5) $p_5 = \mathit{Node}('-', p_3, p_4)$
6) $p_6 = \mathit{Node}('*', p_1, p_5)$
7) $p_7 = \mathit{Node}('+', p_1, p_6)$
8) $p_8 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}b) = p_3$
9) $p_9 = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}c) = p_4$
10) $p_{10} = \mathit{Node}('-', p_3, p_4) = p_5$
11) $p_{11} = \mathit{Leaf}(\mathbf{id}, \mathit{entry}\text{-}d)$
12) $p_{12} = \mathit{Node}('*', p_5, p_{11})$
13) $p_{13} = \mathit{Node}('+', p_7, p_{12})$

Node reuse

Outline

- Intermediate Representation
  - DAG's for Expressions
  - Three-Address Code
- Type and Declarations
- Translation of Expressions
- Type Checking
- Control Flow
- Backpatching

Three-Address Code (三地址代码)

- In three-address code, there is at most one operator on the right side of an instruction
  - Instructions are often in the form $x = y\ op\ z$

- Operands (or addresses) can be:
  - Names in the source programs
  - Constants: a compiler must deal with many types of constants
  - Temporary names generated by a compiler

Instructions (1)

1. Assignment instructions:
   - $x = y\ op\ z$, where op is a binary arithmetic/logical operation
   - $x = op\ y$, where op is a unary operation
2. Copy instructions: $x = y$
3. Unconditional jump instructions: goto L, where L is a label of the jump target
4. Conditional jump instructions:
   - if x goto L
   - ifFlase x goto L
   - if $x\ relop\ y$ goto L

Instructions (2)

5. Procedural calls and returns
   - param x_1
   - …
   - param x_n
   - call p, n (procedure call)
   - y = call p, n (function call)
   - return y

6. Indexed copy instructions: $x = y[i] \quad x[i] = y$
   - Here, $y[i]$ means the value in the location i memory units beyond location y

Instructions (3)

7. Address and pointer assignment instructions:

   - $x = \&y$ (set the r-value of x to be the l-value of y)
   - $x =* y$ (set the r-value of x to be the content stored at the location pointed to by y; y is a pointer whose r-value is a location)
   - $* x = y$ (set the r-value of the object pointed to by x to the r-value of y)

A variable has l-value and r-value:

- **L-value (location)** refers to the memory location, which identifies an object.
- **R-value (content)** refers to data value stored at some address in memory.

Example

- Source code: `do i = i + 1; while (a[i] < v);`

L:	t_1 = i + 1
	i = t_1
	t_2 = i * 8
	t_3 = a [t_2]
	if t_3 < v goto L

(a) Symbolic labels.

100:	t_1 = i + 1
101:	i = t_1
102:	t_2 = i * 8
103:	t_3 = a [t_2]
104:	if t_3 < v goto 100

(b) Position numbers.

Assuming each array element takes 8 units of space

Representation of Instructions

- In a compiler, three-address instructions can be implemented as objects/records with fields for the operator and the operands

- Three typical representations:
  - Quadruples (四元式表示方法)
  - Triples (三元式表示方法)
  - Indirect triples (间接三元式表示方法)

Quadruples (四元式)

- A *quadruple* has four fields
  - General form: *op* arg_1 arg_2 *result*
  - *op* contains an internal code for the operator
  - arg_1, arg_2, *result* are addresses (operands)
  - Example: $x = y + z$ ➔ $+ \quad y \quad z \quad x$

- Some exceptions:
  - Unary operators like $x = minus\ y$ or $x = y$ do not use arg_2
  - *param* operators use neither arg_2 nor *result*
  - Conditional/unconditional jumps put the target label in *result*

Quadruples Example

- Assignment statement: $a = b * -c + b * -c$

```
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

Temporaries

The result field is used primarily for temporary names
Temporary names waste space (symbol table entries)

Triples (三元式)

- A *triple* has only three fields: *op*, arg_1, arg_2
- We refer to the result of an operation *x op y* by its position without generating temporary names (an optimization over quadruples)

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

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
		…		

Quadruples

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		…	

Triples

Quadruples vs. Triples

- In optimizing compilers, instructions are often moved around

	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
		. . .		

Swap 1 and 2

	op	arg_1	arg_2	result
0	minus	c		t_1
1	minus	c		t_3
2	*	b	t_1	t_2
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		. . .		

Quadruples' advantage

The instructions that use t_1 and t_3 are not affected

Quadruples vs. Triples

- In optimizing compilers, instructions are often moved around

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

Swap 1 and 2

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	minus	c	
2	*	b	(0)
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		. . .	

Are they still correct after swapping?

Triples' problem

The instructions now refer to wrong results; The positions need to be updated.

Indirect Triples (间接三元式)

- *Indirect triples* consist of a list of pointers to triples

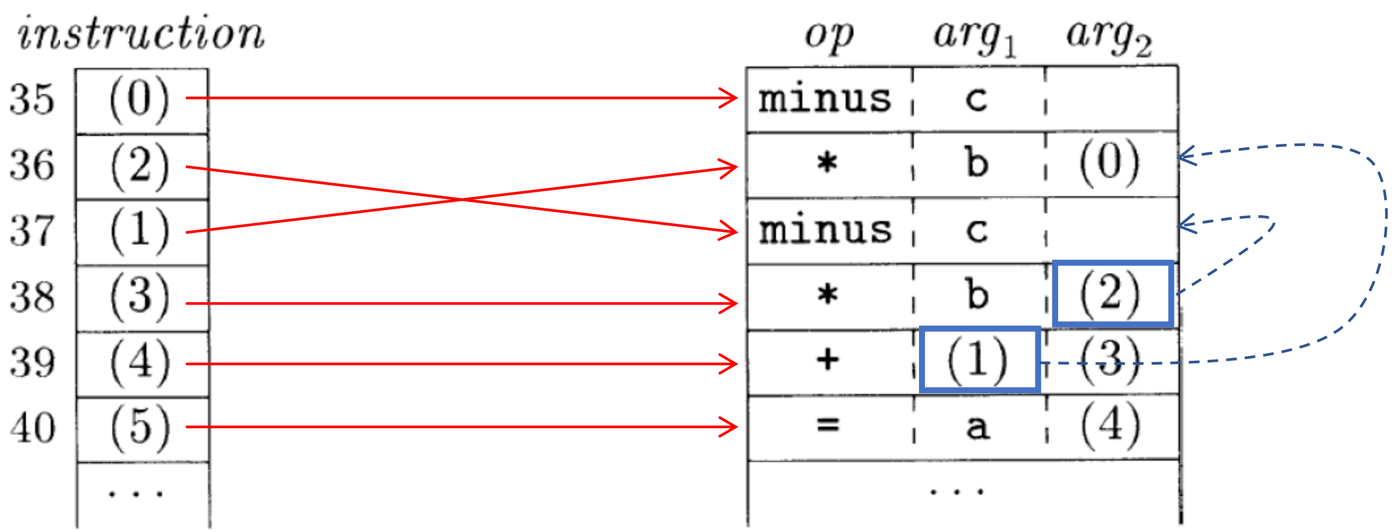
	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

Points to

op	arg_1	arg_2
minus	c	
*	b	(0)
minus	c	
*	b	(2)
+	(1)	(3)
=	a	(4)
...		

Indirect Triples (间接三元式)

- An optimization can move an instruction by reordering the *instruction* list



Swapping pointers!

The triples are not affected.

Static Single-Assignment Form

- Static single-assignment form (SSA, 静态单赋值形式) is an IR that facilitates certain code optimizations
- In SSA, each name receives a single assignment

(a) Three-address code.	(b) Static single-assignment form.
$\boxed{p}$ = a + b	$\boxed{p_1}$ = a + b
q = p - c	q_1 = p_1 - c
$\boxed{p}$ = q * d	$\boxed{p_2}$ = q_1 * d
p = e - p	p_3 = e - p_2
q = p + q	q_2 = p_3 + q_1

Static Single-Assignment Form

- The same variable may be defined in two control-flow paths

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

X_1 X_2

Which name should we use in y = x * a?

Static Single-Assignment Form

- The same variable may be defined in two control-flow paths

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

- SSA uses a notational convention called ϕ-function to combine the two definitions of x

$$\texttt{if (flag) } x_1 \texttt{ = -1; else } x_2 \texttt{ = 1;}$$
$$x_3 = \phi(x_1, x_2); \quad \text{// x1 if control flow passes through the true path; otherwise x2}$$
$$\texttt{y = } x_3 \texttt{ * a;}$$