

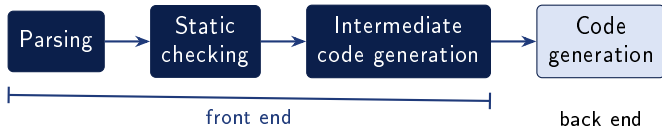
Intermediate Code Generation

Compiler Construction, Lecture 13

K.R. Chowdhary, Professor of CS

August 19, 2019 | Fall 2019

Why an Intermediate Representation?



- Source details stay in the front end, target-machine details in the back end
- With a common IR, m front ends and n back ends give $m \times n$ compilers; for $n = 10$, **20** pieces of code give **100** compilers
- **Static checking:** type checking, plus syntactic checks left after parsing, e.g. `break` must be inside a loop or switch

Levels of IR

High level (syntax trees): close to the source, suited to static type checking.

Low level (three-address code): close to the machine, suited to register allocation and instruction selection.

C is often used as an IR: the original C++ compiler generated C and treated a C compiler as its back end.

Constructing Syntax Trees

Each syntax-tree node is an object with an *op* field.

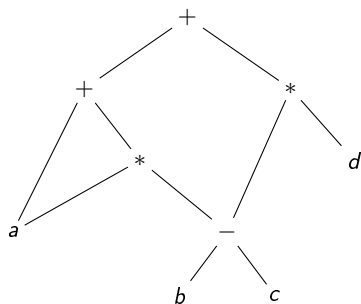
- **Leaf**(*op*, *val*) creates a leaf holding the lexical value
- **Node**(*op*, *c*₁, ..., *c*_{*k*}) creates an interior node with *k* children

The same SDD builds syntax trees or DAGs. An S-attributed definition suits bottom-up parsing; an L-attributed one suits top-down parsing.

A node for $E_1 + E_2$ has label $+$ and two children, the subexpressions.

	Production	Semantic rule
1	$E \rightarrow E_1 + T$	$E.node = new\ Node(+, E_1.node, T.node)$
2	$E \rightarrow E_1 - T$	$E.node = new\ Node(-, E_1.node, T.node)$
3	$E \rightarrow T$	$E.node = T.node$
4	$T \rightarrow (E)$	$T.node = E.node$
5	$T \rightarrow id$	$T.node = new\ Leaf(id, id.entry)$
6	$T \rightarrow num$	$T.node = new\ Leaf(num, num.val)$

DAGs: Sharing Common Subexpressions



DAG for $a + a * (b - c) + (b - c) * d$: a and $b - c$ each have two parents.

Before creating a node, *Leaf* and *Node* check for an identical existing node and return it. Steps:

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

Three-Address Code

At most one operator on the right:
 $x = y \text{ op } z$. Addresses are names,
constants or compiler-generated
temporaries.

$x + y * z$

$t1 = y * z$
 $t2 = x + t1$

DAG above, linearized

$t1 = b - c$
 $t2 = a * t1$
 $t3 = a + t2$
 $t4 = t1 * d$
 $t5 = t3 + t4$

	Form	Meaning
1	$x = y \text{ op } z$	binary arithmetic or logical
2	$x = \text{op } y$	unary: minus, not, shift, conversion
3	$x = y$	copy
4	$\text{goto } L$	unconditional jump
5	$\text{if } x \text{ goto } L, \text{ ifFalse } x \text{ goto } L$	conditional jump
6	$\text{if } x \text{ relop } y \text{ goto } L$	relational jump
7	$\text{param } x, \text{ call } p, n, y = \text{call } p, n, \text{ return } y$	procedure calls
8	$x = y[i], \quad x[i] = y$	indexed copy
9	$x = \&y, \quad x = *y, \quad *x = y$	address and pointer

Labels, Quadruples and Triples

do-while loop

```
do i = i + 1; while (a[i] < v);  
L: t1 = i + 1  
  i = t1  
  t2 = i * 4  
  t3 = a[t2]  
  if t3 < v goto L
```

With positions 100–104 the jump is 104 if $t3 < v$ goto 100; 4 is the integer size in bytes.

Expression statement

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

Quadruples (op, arg₁, arg₂, result)

	op	arg ₁	arg ₂	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

Triples (op, arg₁, arg₂): results are referred to by position

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

Static Single-Assignment Form and Review

SSA assigns each variable name exactly once, and uses ϕ to merge definitions from different control-flow paths.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

Merging with ϕ

```
if (flag) x1 = -1;
else x2 = 1;
x3 =  $\phi$ (x1, x2);
```

$\phi(x_1, x_2)$ returns the argument for the control-flow path taken.

Review

1. How do n front ends and n back ends give n^2 compilers?
2. High- vs. low-level IR: examples?
3. Can C serve as an IR?

Exercises

1. DAG for $((x + y) - (x + y)) + ((x + y) * (x - y))$; value numbers for $a + b + (a + b)$
2. $a + -(b + c)$ as tree, quads, triples, indirect triples
3. TAC for $a = b[i] + c[j]$ and $x = *p + \&y$