

Exercise walk-through

11.6

Semantic analysis and code generation

University Computer Science

Semantic analysis: decorate, then emit

- 1 **Program** $\text{count} = a1 + b2 * 3$ with all int: $t1 = b2 * 3$; $t2 = a1 + t1$; $\text{count} = t2$ - three instructions, one temp.
- 2 **Error** $b2$: str rejects $b2 * 3$ with the lexer position - positions are the pipeline error currency.
- 3 **Fold** Literal-only: $2 * 3$ becomes 6; $b2 * 3$ stays - folding is a proof about literals.

Symbol table and IR

decorate with meaning, then emit IR

```
count = a1 + b2 * 3
```

symbol table

a1	int	global
b2	int	global
count	int	global



IR (three-address)

```
t1 = b2 * 3
t2 = a1 + t1
count = t2
```

type walker: $b2 * 3$ (int x literal) then $a1 + \dots$ (int + int) — all int, accepted

error case: b2: str rejects $b2 * 3$ with the lexer position

why IR: m languages x n targets = $m + n$, not $m \times n$; folding is literal-only ($2 * 3 \rightarrow 6$)

The **symbol table** binds a1, b2, count as int; the IR is three instructions.

Check yourself

- The course DFA on aba ends at...? (q1 — rejected.)
- $\text{id} + \text{id} * \text{id}$: leftmost **derivation** steps...? (Six.)
- ifx lexes as...? (One IDENT.)
- $\text{count} = \text{a1} + \text{b2} * 3$ emits...? (t1, t2, count = t2.)

Exercises 2.1-2.3

2.1 What does the symbol table map, and when is it built? [3]

2.2 IR for $x = 2 * 3$ after folding – how many instructions? [3]

2.3 Why an IR at all (the arithmetic argument)? [3]

Exercises 2.1-2.3

2.1 What does the symbol table map, and when is it built? [3]

2.1

- Identifiers to (type, scope); built during the semantic walk as declarations are seen ✓.
-

2.2 IR for $x = 2 * 3$ after folding – how many instructions? [3]

2.2

- One: $x = 6$ – the literal product folds ✓.
-

2.3 Why an IR at all (the arithmetic argument)? [3]

2.3

- $m \times n$ direct compilers become $m + n$ components: each frontend targets the one IR, each backend starts from it ✓.

Exercise 3.1

3.1 Program: $r = a + b * c - d / e$ with all variables int.

Emit the three-address IR (temps numbered from t1). Give the instruction count and temp count. Give the type-check sequence (bottom-up, one line per operator). State which instruction changes if d were type str, and the error's reported position. [8]

Exercise 3.1

3.1 Program: $r = a + b * c - d / e$ with all variables int.

Emit the three-address IR (temps numbered from t_1). Give the instruction count and temp count. Give the type-check sequence (bottom-up, one line per operator). State which instruction changes if d were type str, and the error's reported position. [8]

3.1

- IR: $t_1 = b * c$; $t_2 = a + t_1$; $t_3 = d / e$; $t_4 = t_2 - t_3$; $r = t_4$ ✓.
- Five instructions, four temps ✓.
- Checks: $b \times c$ (int $\times$ int), $a + t_1$ (int + int), d/e (int / int), $t_2 - t_3$ (int - int) – all pass ✓.
- With d : str: $t_3 = d / e$ fails – reported at d 's lexed position ✓.

Exercise 3.2

3.2 Two programs:

$$(1) \ y = 2 * 3 + x \quad (2) \ y = x * 3 + 2$$

For each: the folded-or-not IR and the reason. State the folding rule in one sentence. Then argue why the IR makes m languages compile to n targets with $m + n$ components – name the alternative's cost ✓.

[7]

Exercise 3.2

3.2 Two programs:

$$(1) y = 2 * 3 + x \quad (2) y = x * 3 + 2$$

For each: the folded-or-not IR and the reason. State the folding rule in one sentence. Then argue why the IR makes m languages compile to n targets with $m + n$ components – name the alternative's cost ✓. [7]

3.2

- (1) $y = 6 + x$ – the literal product folds at IR build: one instruction ✓.
- (2) $t1 = x * 3; y = t1 + 2 - x * 3$ is not literal: no fold ✓.
- Rule: fold only when every operand of the node is a literal ✓.
- The IR argument: without it every (language, target) pair needs its own full compiler – mn of them; the IR makes each side write once against a fixed middle ✓.