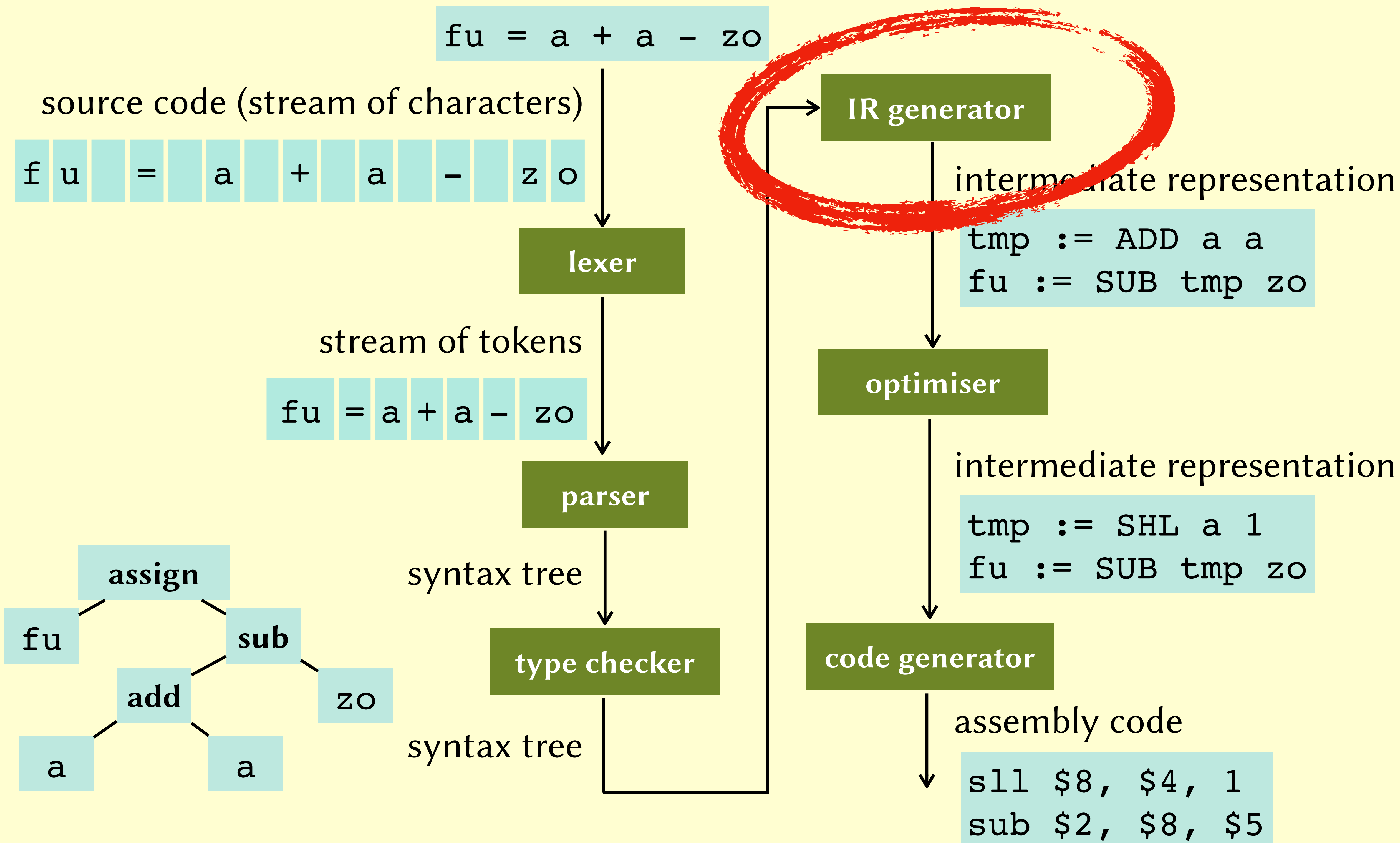
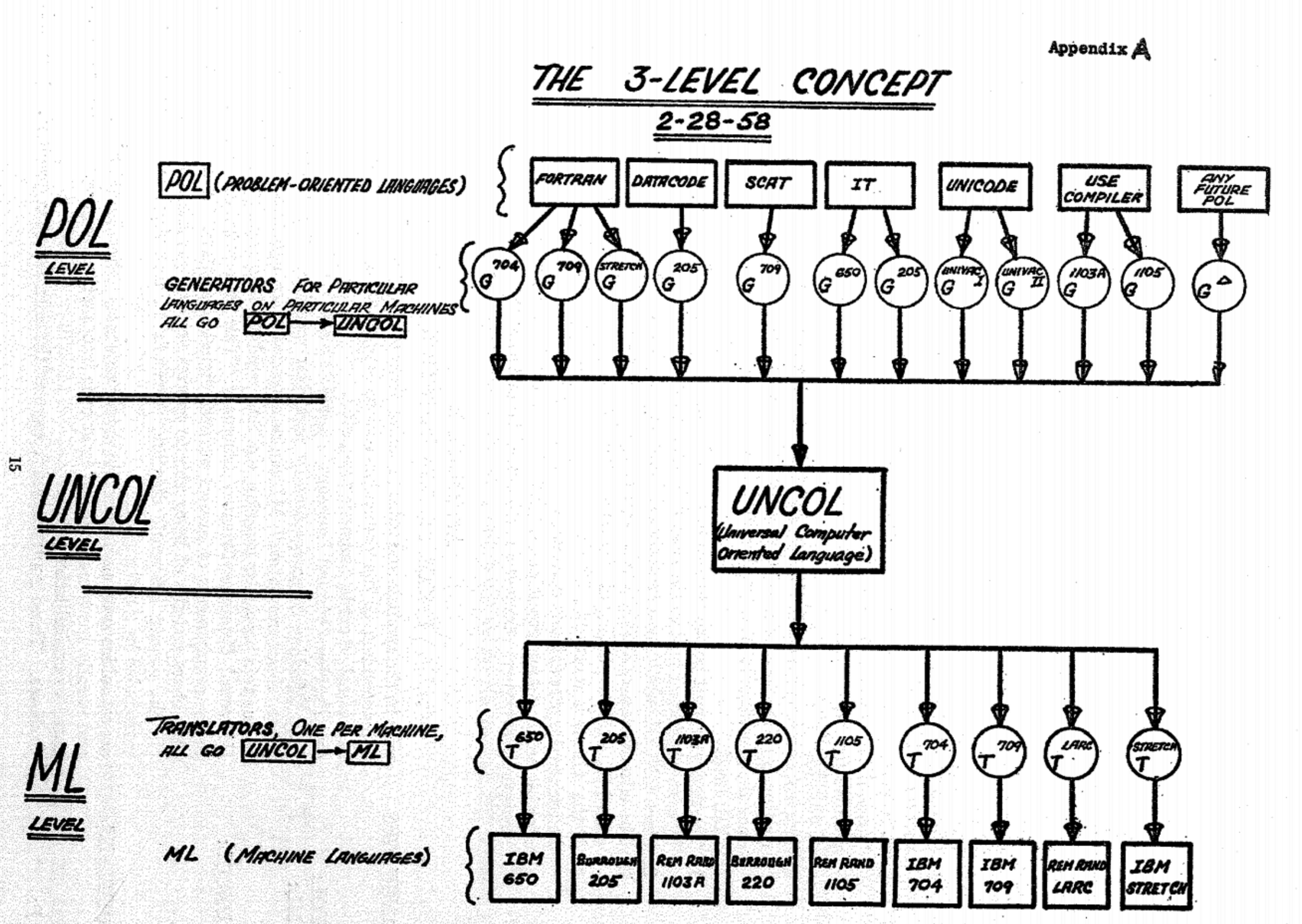
Lecture 6: Intermediate Representations

John Wickerson

Compilers

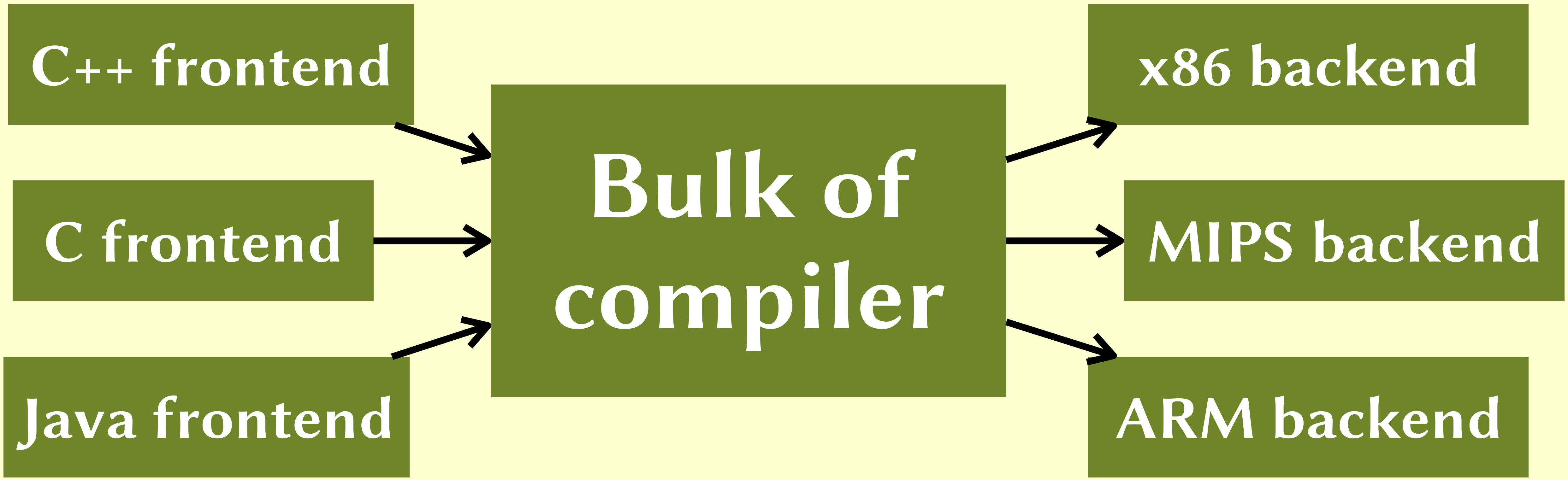
Anatomy of a compiler




Appendix A
THE 3-LEVEL CONCEPT
2-28-58
POL LEVEL
POL (PROBLEM-ORIENTED LANGUAGES)
FORTRAN
DATACODE
SCAT
IT
UNICODE
USE COMPILER
ANY FUTURE POL
GENERATORS FOR PARTICULAR LANGUAGES ON PARTICULAR MACHINES ALL GO POL → UNCOL
G 704
G 709
G STRETCH
G 205
G 709
G 650
G 205
G UNIVAC I
G UNIVAC II
G 1103A
G 1105
G Δ
UNCOL LEVEL
UNCOL (Universal Computer Oriented Language)
ML LEVEL
TRANSLATORS, ONE PER MACHINE, ALL GO UNCOL → ML
T 650
T 205
T 1103A
T 220
T 1105
T 704
T 709
T LARC
T STRETCH
ML (MACHINE LANGUAGES)
IBM 650
BURROUGHS 205
REM RAND 1103 A
BURROUGHS 220
REM RAND 1105
IBM 704
IBM 709
REM RAND LARC
IBM STRETCH
15

IR

- Allows modular compiler design



- Should be **independent** of the source language, but also able to express the source program without too much **information loss**.

IR

- A typical IR:
  - uses three-address code,
  - performs one operation per instruction,
  - uses unstructured control flow,
  - assigns to each variable at most once, and
  - assumes an unlimited number of registers.

Outline

- How to represent three-address code
- How to convert high-level instructions into IR
- Static single assignment

Three-address code

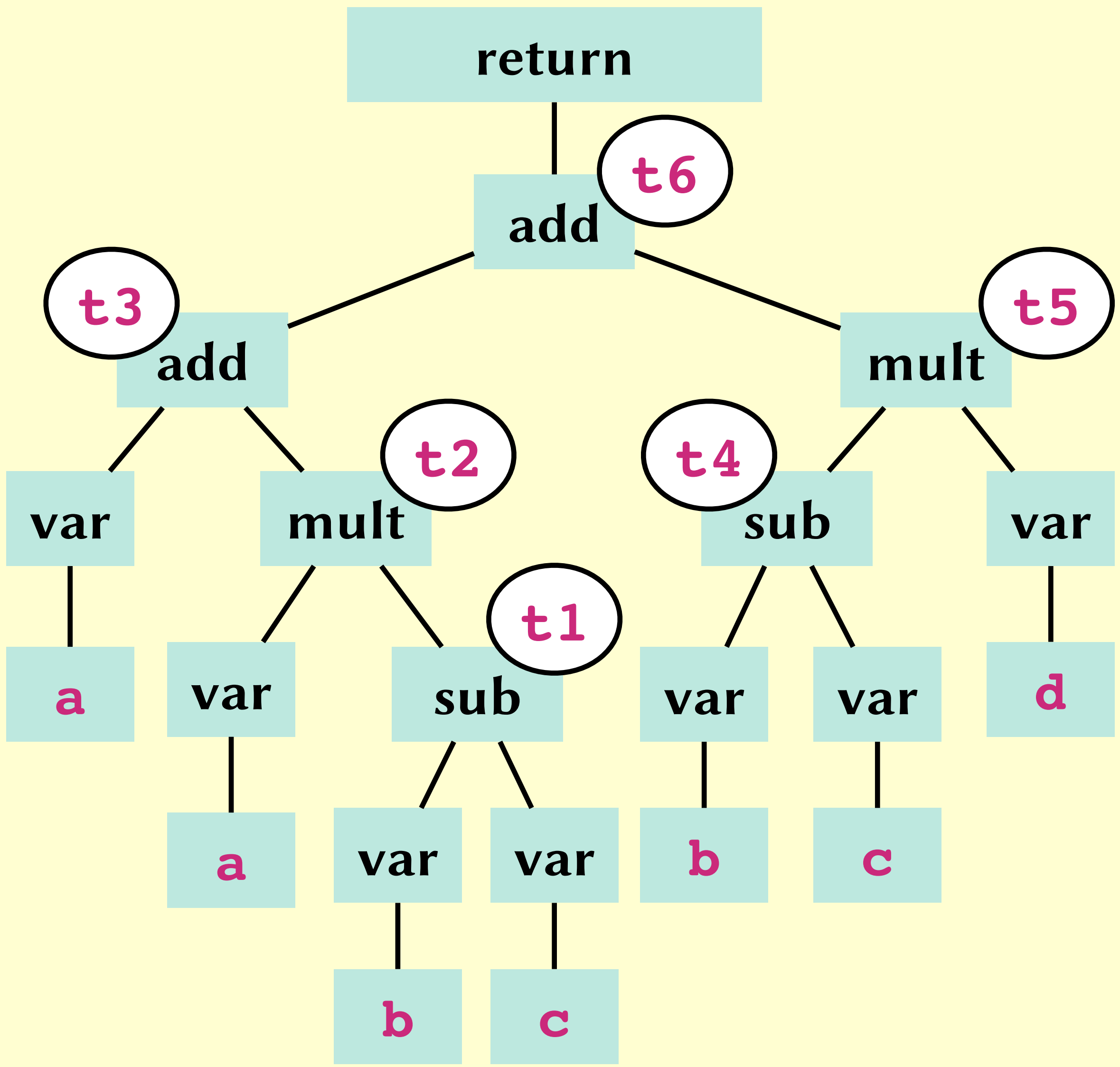
- In C:
```
return a + a * (b - c) + (b - c) * d;
```

- In 3AC:
```
t1 = b - c;
t2 = a * t1;
t3 = a + t2;
t4 = t1 * d;
t5 = t3 + t4;
return t5;
```

Three-address code

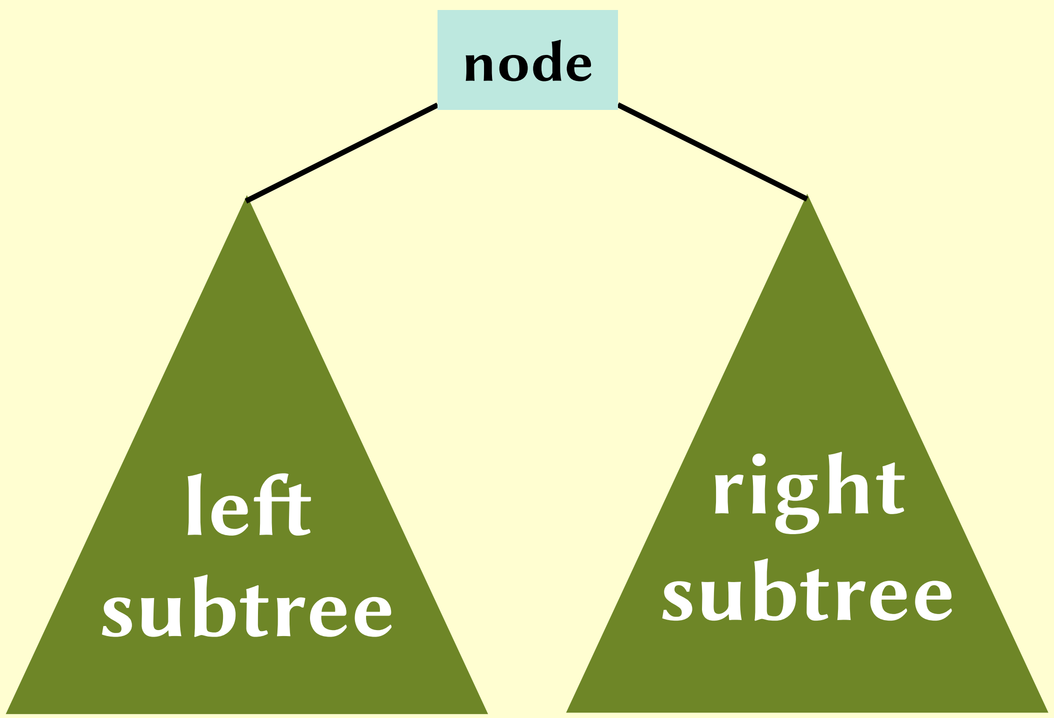
- In C:

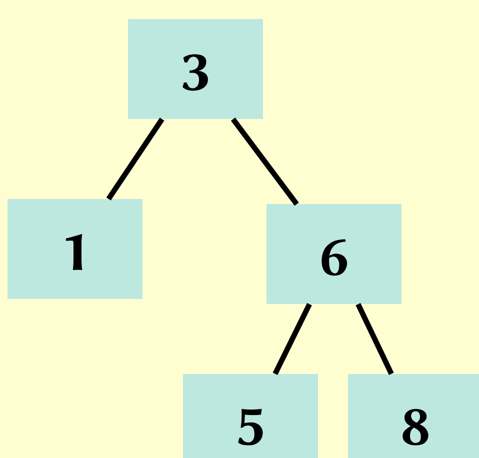
```
return a + a * (b - c) + (b - c) * d;
```




```
t1 = b - c;
t2 = a * t1;
t3 = a + t2;
t4 = b - c;
t5 = t4 * d;
t6 = t3 + t5;
return t6;
```

Tree traversal





Pre-order traversal:

1. node
2. left subtree
3. right subtree

3	1	6	5	8

Example use: copying a tree.

In-order traversal:

1. left subtree
2. node
3. right subtree

1	3	5	6	8

Example use: reading out values.

Post-order traversal:

1. left subtree
2. right subtree
3. node

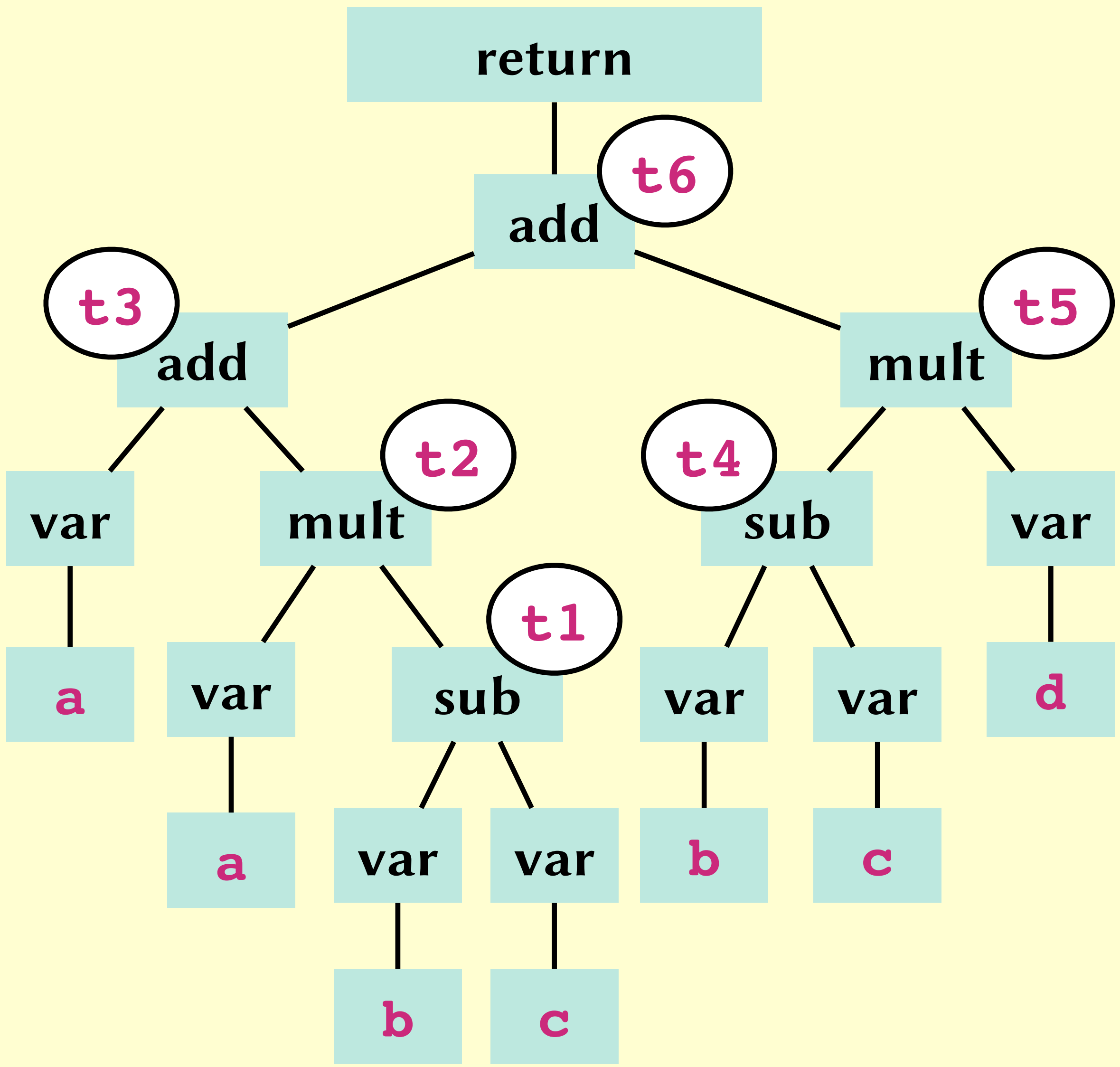
1	5	8	6	3

Example use: deleting a tree.

Three-address code

- In C:

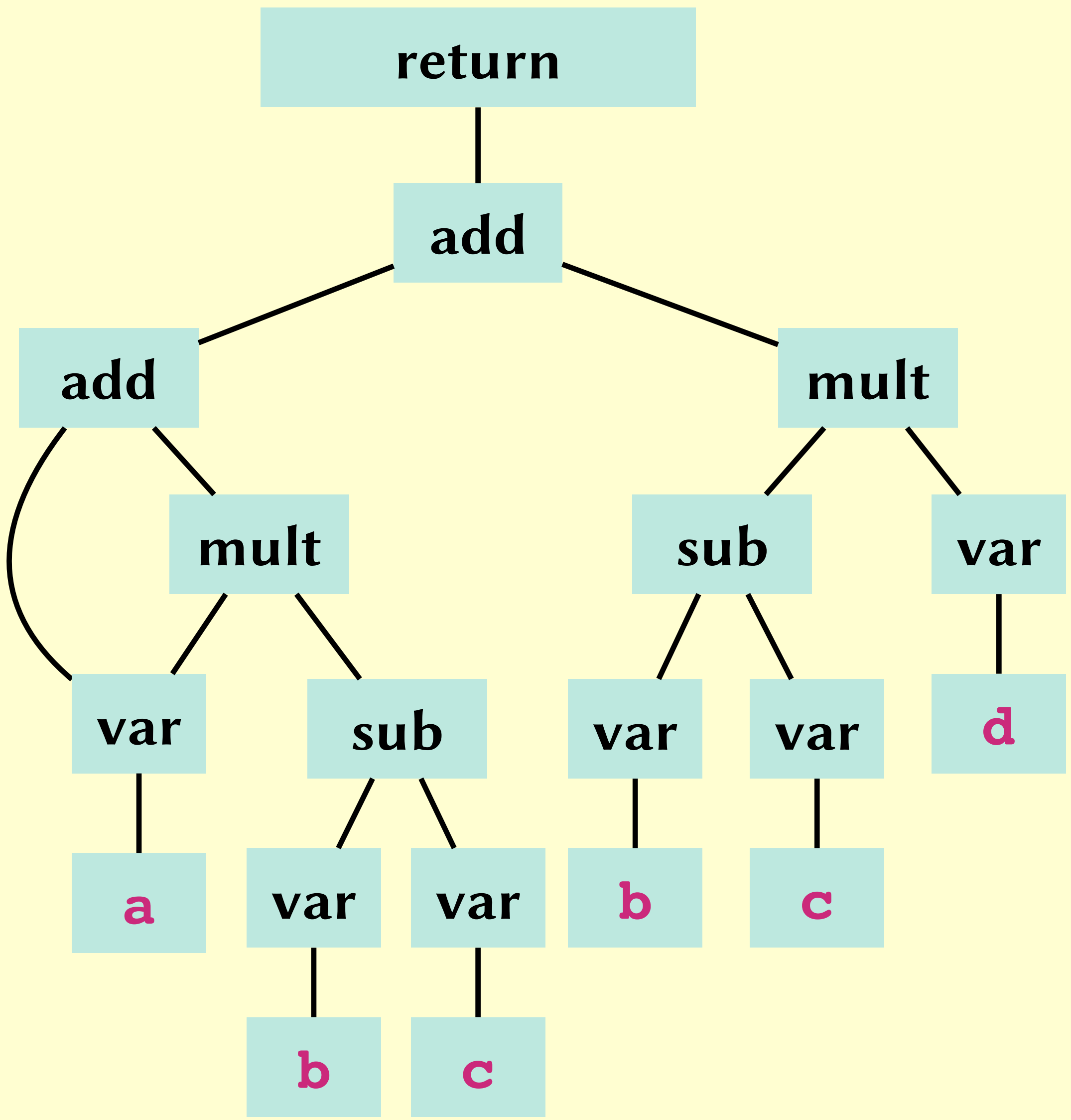
```
return a + a * (b - c) + (b - c) * d;
```




```
t1 = b - c;
t2 = a * t1;
t3 = a + t2;
t4 = b - c;
t5 = t4 * d;
t6 = t3 + t5;
return t6;
```

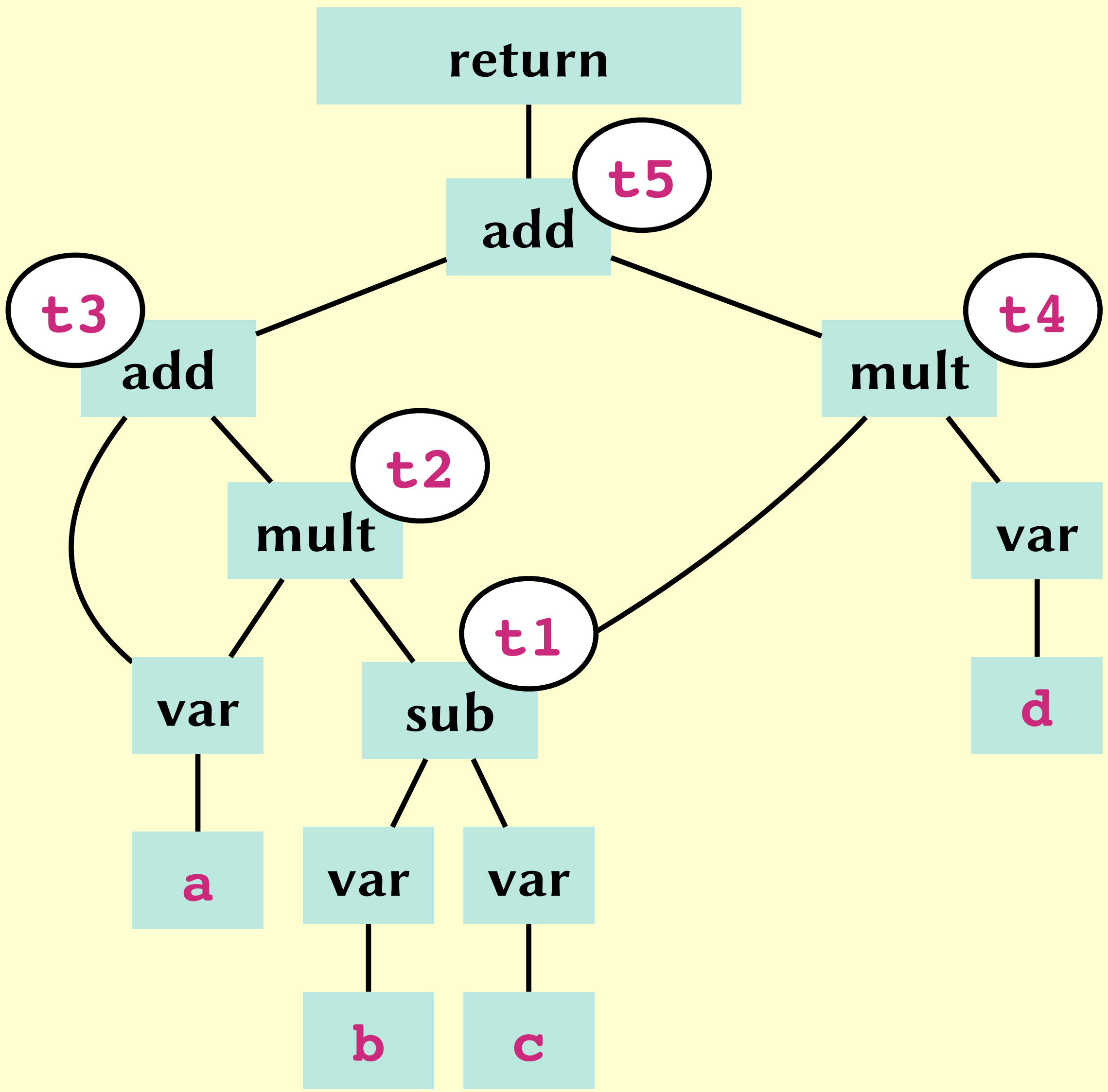
Three-address code

- In C:
  `return a + a * (b - c) + (b - c) * d;`



Three-address code

- In C:
  `return a + a * (b - c) + (b - c) * d;`




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

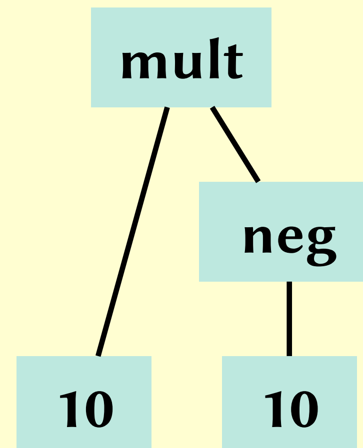
From trees to dags

```
struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
}
```

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



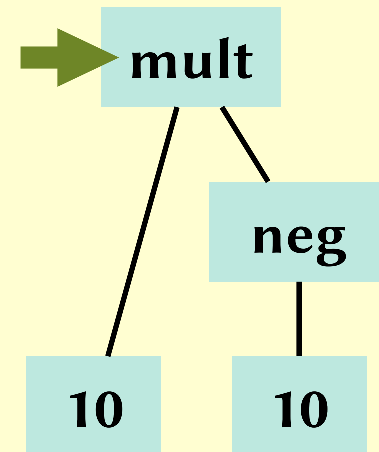
Key	Value

Addr	Data
80	mult
84	●
88	●
92	
96	10
100	●
104	●
108	
112	neg
116	●
120	●
124	
128	10
132	●
136	●
140	
144	

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



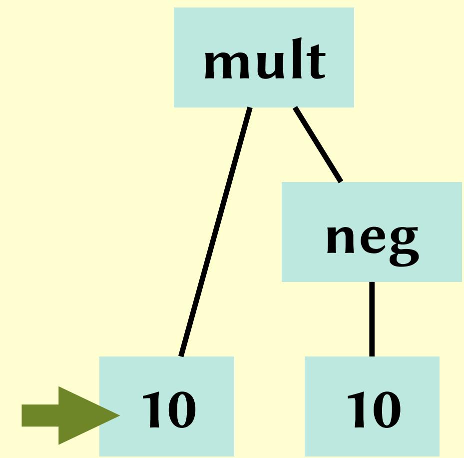
Addr	Data
80	mult
84	96
88	112
92	
96	10
100	0
104	0
108	
112	neg
116	128
120	0
124	
128	10
132	0
136	0
140	
144	

Key	Value

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



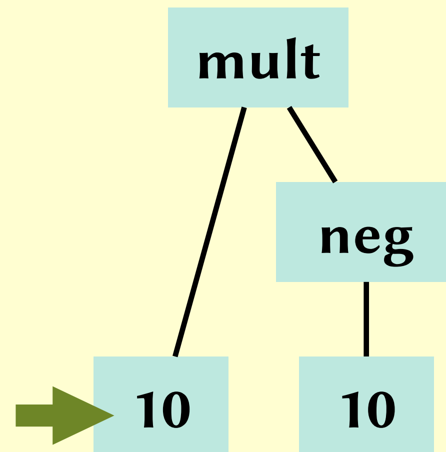
Key	Value

Addr	Data
80	**mult**
84	96
88	112
92	
96	**10**
100	0
104	0
108	
112	**neg**
116	128
120	0
124	
128	**10**
132	0
136	0
140	
144	

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



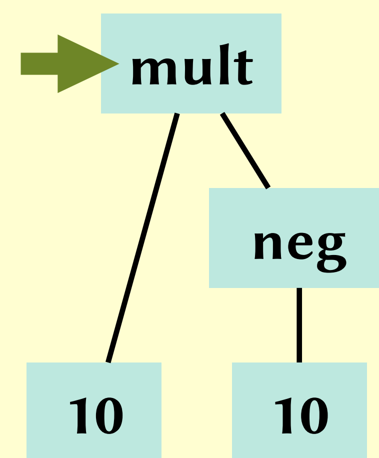
Addr	Data
80	mult
84	96
88	112
92	
96	10
100	0
104	0
108	
112	neg
116	128
120	0
124	
128	10
132	0
136	0
140	
144	

Key	Value
(10,0,0)	96

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



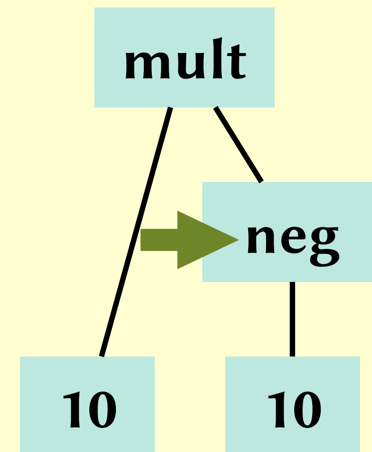
Addr	Data
80	mult
84	96
88	112
92	
96	10
100	0
104	0
108	
112	neg
116	128
120	0
124	
128	10
132	0
136	0
140	
144	

Key	Value
(10,0,0)	96

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



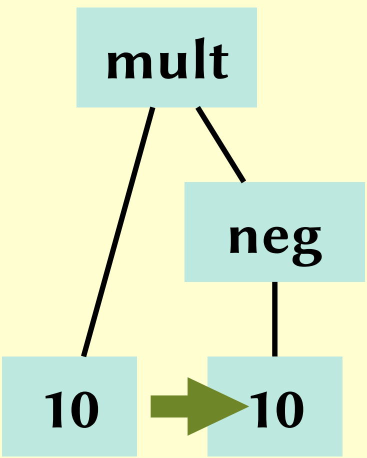
Key	Value
(10,0,0)	96

Addr	Data
80	mult
84	96
88	112
92	
96	10
100	0
104	0
108	
112	neg
116	128
120	0
124	
128	10
132	0
136	0
140	
144	

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



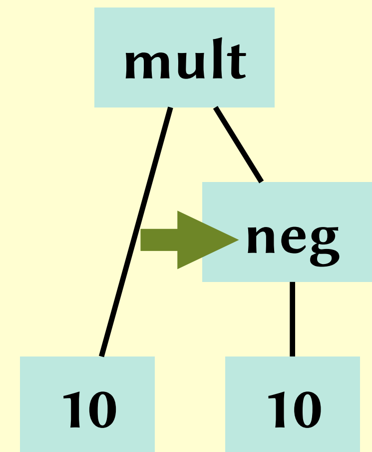
Key	Value
(10,0,0)	96

Addr	Data
80	**mult**
84	96
88	112
92	
96	**10**
100	0
104	0
108	
112	**neg**
116	128
120	0
124	
128	**10**
132	0
136	0
140	
144	

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



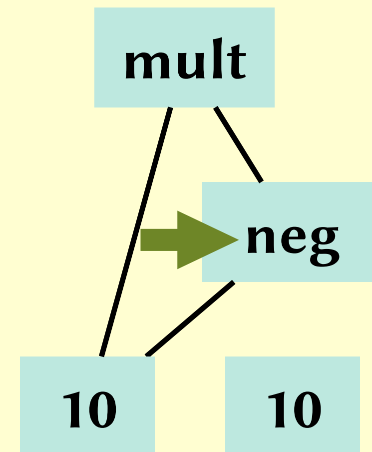
Key	Value
(10,0,0)	96

Addr	Data
80	**mult**
84	96
88	112
92	
96	**10**
100	0
104	0
108	
112	**neg**
116	128
120	0
124	
128	**10**
132	0
136	0
140	
144	

From trees to dags

```
typedef struct node_ {
  int contents;
  struct node_ *left;
  struct node_ *right;
} node;

node *dagify(node *n) {
  if (n == NULL) return n;
  n->left = dagify(n->left);
  n->right = dagify(n->right);
  node *result = lookup(*n);
  if (result == NULL) {
    put(*n, n);
    return n;
  }
  return result;
}
```



Key	Value
(10,0,0)	96

Addr	Data
80	mult
84	96
88	112
92	
96	10
100	0
104	0
108	
112	neg
116	96
120	0
124	
128	10
132	0
136	0
140	
144	

Outline

✔ How to represent three-address code

- How to convert high-level instructions into IR

- Static single assignment

Converting IF

- In C:

```
if(x < 42) {a = 16; b = 59;} else {c = 17;}
```

- In 3AC:

```
t1 = x < 42;
ifFalse t1 goto ELSE;
a = 16;
b = 59;
goto ENDIF;
ELSE:
c = 17;
ENDIF:
```

Note: you may need to generate fresh labels like ELSE1, ELSE2, etc.

Converting WHILE

- In C:

```
while(x > 49) {a = 16; b = 59;}
```

- In 3AC:

```
STARTLOOP:
t1 = x > 49;
ifFalse t1 goto ENDLOOP;
a = 16;
b = 59;
goto STARTLOOP;
ENDLOOP:
```

Converting WHILE

- In C:

```
while(x > 49) {if (x > 59) a = 1;}
```

- In 3AC:

```
STARTLOOP:
t1 = x > 49;
ifFalse t1 goto ENDLOOP;
t2 = x > 59;
ifFalse t2 goto ELSE;
a = 1;
ELSE:
ENDIF:
goto STARTLOOP;
ENDLOOP:
```

Converting BREAK

- In C:

```
while(x > 49) {if (x > 59) break;}
```

- In 3AC:

```
STARTLOOP:
t1 = x > 49;
ifFalse t1 goto ENDLOOP;
t2 = x > 59;
ifFalse t2 goto ELSE;
goto ENDLOOP;
ELSE:
ENDIF:
goto STARTLOOP;
ENDLOOP:
```

Converting CONTINUE

- In C:

```
while(x > 49) {if (x > 59) continue;}
```

- In 3AC:

```
STARTLOOP:
t1 = x > 49;
ifFalse t1 goto ENDLOOP;
t2 = x > 59;
ifFalse t2 goto ELSE;
goto STARTLOOP;
ELSE:
ENDIF:
goto STARTLOOP;
ENDLOOP:
```

Converting FOR

- In C:
  ```
  for(i = 0; i < 42; i++) { stuff(); }
  ```
- Also in C:
  ```
  i = 0; while(i < 42) { stuff(); i++; }
  ```

Converting && and ||

- In C:

```
t = (a && b);
```

- Also in C:

```
if (a) {
  if (b) {
    t = 1;
  } else {
    t = 0;
  }
} else {
  t = 0;
}
```

- In C:

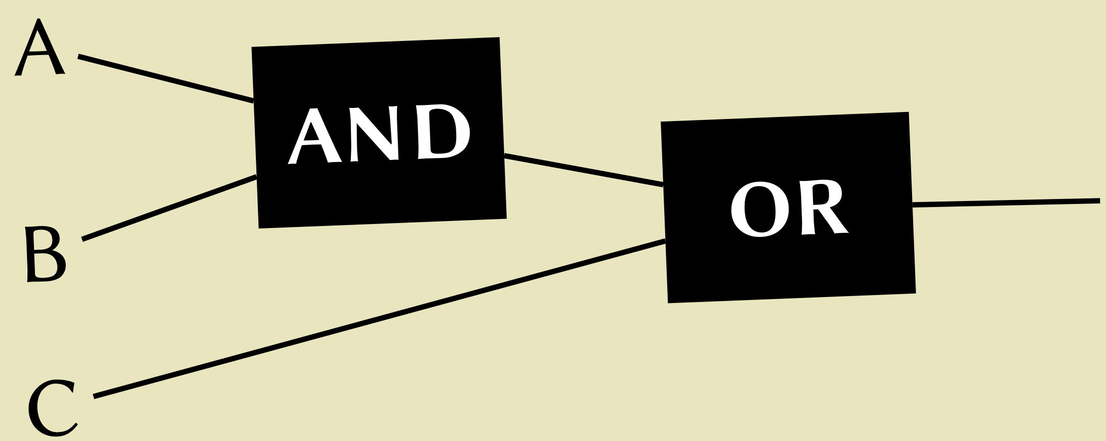
```
t = (a || b);
```

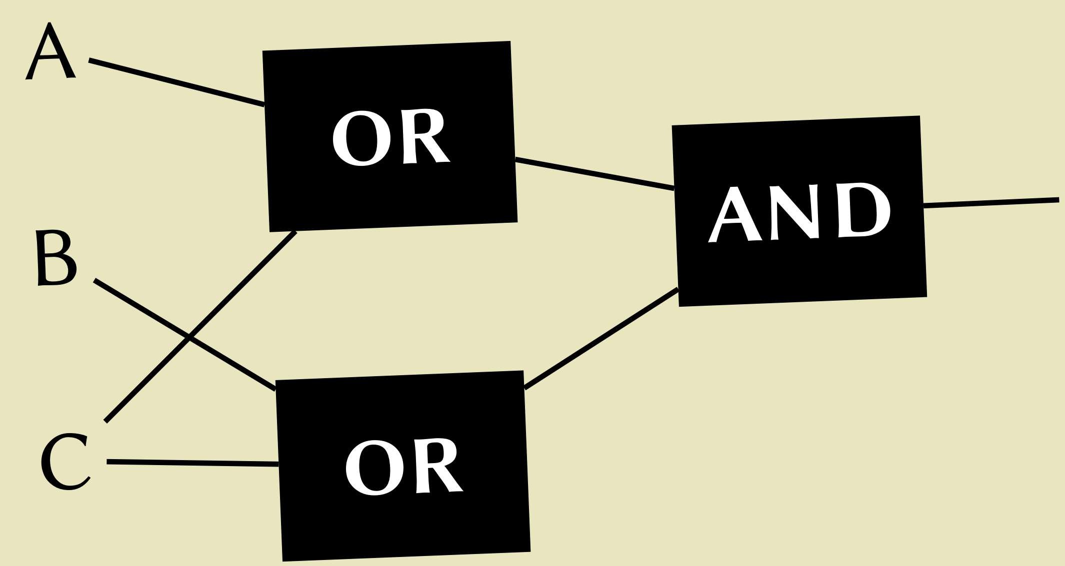
- Also in C:

```
if (a) {
  t = 1;
} else {
  if (b) {
    t = 1;
  } else {
    t = 0;
  }
}
```

Aside:

- In Boolean logic: $(A \wedge B) \vee C \;\equiv\; (A \vee C) \wedge (B \vee C)$





- But `(A&&B)||C` is not the same as `(A||C)&&(B||C)`!

Converting SWITCH

- In C:

```
switch(i % 3) {
  case 0: a = 1;
  case 1: b = 2;
  default: c = 3;
}
```

Doesn't give right semantics for `switch`

- In 3AC:

```
t = i % 3;
ifNeq t 0 goto L2;
a = 1;
goto ENDSWITCH;
L2:
ifNeq t 1 goto L3;
b = 2;
goto ENDSWITCH;
L3:
c = 3;
ENDSWITCH:
```

DEMO

Converting SWITCH

- In C:

```
switch(i % 3) {
  case 0: a = 1;
  case 1: b = 2;
  default: c = 3;
}
```

- In 3AC:

```
t = i % 3;
ifEq t 0 goto L1;
ifEq t 1 goto L2;
goto L3;
L1: a = 1;
L2: b = 2;
L3: c = 3;
```

Code generator can spot sequence of **ifEq** statements and generate an efficient N-way branch

Converting function calls

- In C:

```
r = foo(42 + x, 59, y);
```

- In 3AC:

```
t1 = 42 + x;
param t1;
param 59;
param y;
r = call foo 3;
```

Converting arrays

- In C:

```
int a[2];
int c, i;
return c + a[i];
```

- In 3AC:

```
t1 = i * 4;
t2 = load(a + t1);
t3 = c + t2;
```

`a : array(int, 2)`

Byte	Data
43	a[0]
44	
45	
46	
47	a[1]
48	
49	
50	
51	
52	
53	
54	
55	
56	
57	
58	
59	

Converting arrays

- In C:

```
int a[2][3];
int c, i, j;
return c + a[i][j];
```

- In 3AC:

```
t1 = i * 12;
t2 = j * 4;
t3 = t1 + t2;
t4 = load(a + t3);
t5 = c + t4;
```

`a : array(array(int, 3), 2)`

Byte	Data
43	a[0][0]
44	
45	
46	
47	a[0][1]
48	
49	
50	
51	a[0][2]
52	
53	
54	
55	a[1][0]
56	
57	
58	
59	

Outline

- ✔ How to represent three-address code
- ✔ How to convert high-level instructions into IR
- Static single assignment

Static Single Assignment

- Before:

```
a = 42;
b = a + 5;
a = b * 17;
return a - 1;
```

- After:

```
a = 42;
b = a + 5;
a1 = b * 17;
return a1 - 1;
```

- **Motto:** never need to assign to the same variable twice.

- SSA simplifies various code optimisations.

- What about this?

```
L: a = 42 + i; goto L;
```

Static Single Assignment

- Before:

```
ifFalse t goto ELSE;
b = 10;
goto ENDIF;
ELSE:
b = 5;
ENDIF:
c = b + 1;
```

- After:

```
ifFalse t goto ELSE;
b1 = 10;
goto ENDIF;
ELSE:
b2 = 5;
ENDIF:
b = Φ(b1,b2);
c = b + 1;
```

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate:

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue, for-loops

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue, for-loops, short-circuiting boolean operators

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue, for-loops, short-circuiting boolean operators, switch statements

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue, for-loops, short-circuiting boolean operators, switch statements, function calls

Summary

- Pick an **intermediate representation** (IR), such as **three-address code** (3AC).
- May help to store the program as a **dag** rather than a **tree**.
- Generating IR involves **traversing** this tree/dag.
- How to translate: if-statements, while-loops, break, continue, for-loops, short-circuiting boolean operators, switch statements, function calls, and array accesses.
- **Static single assignment** (SSA) makes optimisations easier.