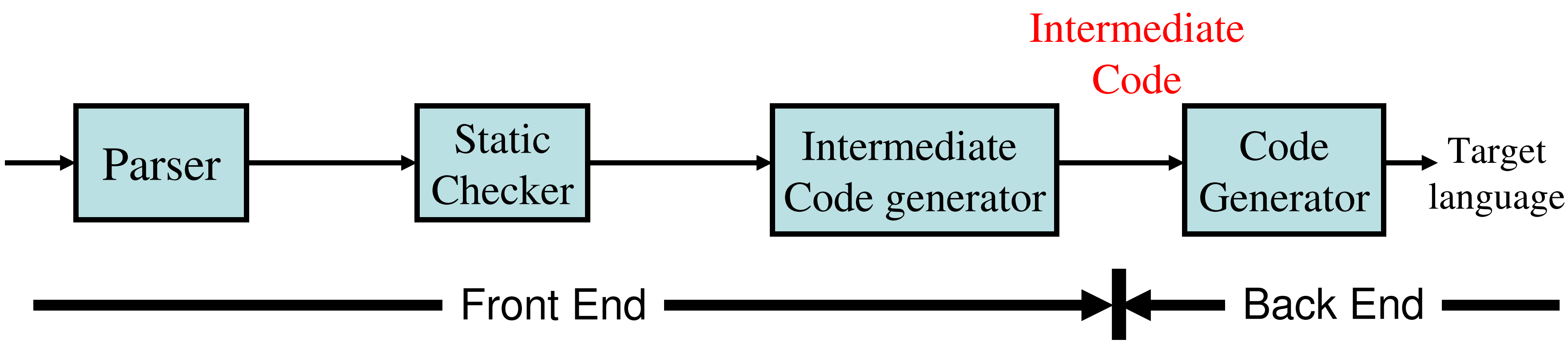
Intermediate Code Generation

Part I

Compiler Architecture



- m $\times$ n compliers can be built by writing m front ends and n back ends – save considerable amount of effort
- We assume parsing, static checking and IC generation is done sequentially
  - These can be combined and done during parsing
- Static checking
  - Operator operand compatibility
  - Proper placement of break/continue keywords etc.

Intermediate Code (IC)

- The given program in a source language is converted to an equivalent program in an intermediate language by the IC generator.
- Ties the front and back ends together
- Language and Machine neutral
- Many forms
- Level depends on how being processed
- More than one intermediate language may be used by a compiler

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Intermediate language levels

- **High**

T1 ← a[i,j+2]

- **Medium**

t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t 4
t7 ← *t6

- **Low**

r1 ←[fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3*20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp – 216
f1 ← [r7+r6]

Intermediate Languages Types

- Graphical IRs:
  - Abstract Syntax trees
  - Directed Acyclic Graphs (DAGs)
  - Control Flow Graphs
- Linear IRs:
  - Stack based (postfix)
  - Three address code (quadruples)

Graphical IRs

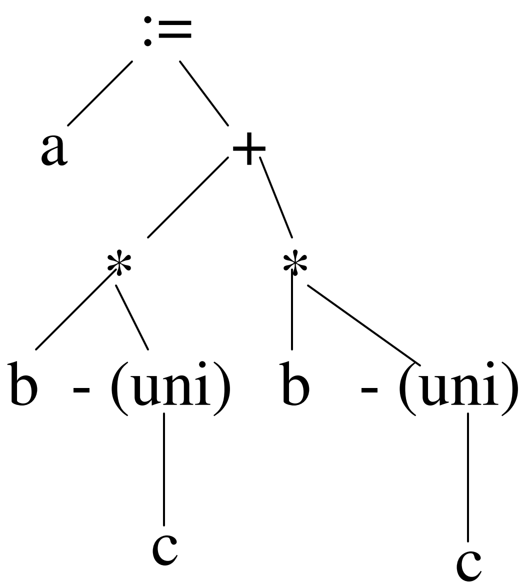
- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.

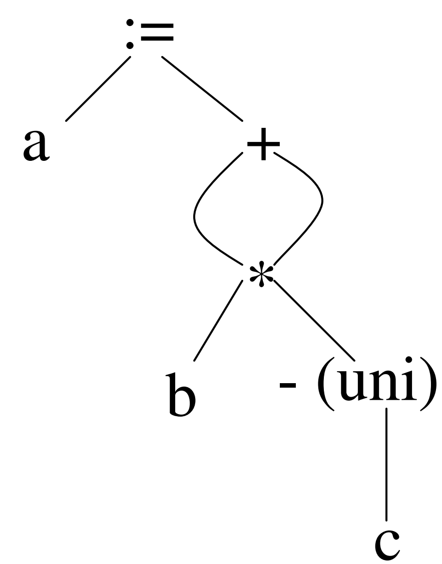
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well

- Control flow graphs (CFG) – explicitly model control flow

ASTs and DAGs:

a := b *-c + b*-c



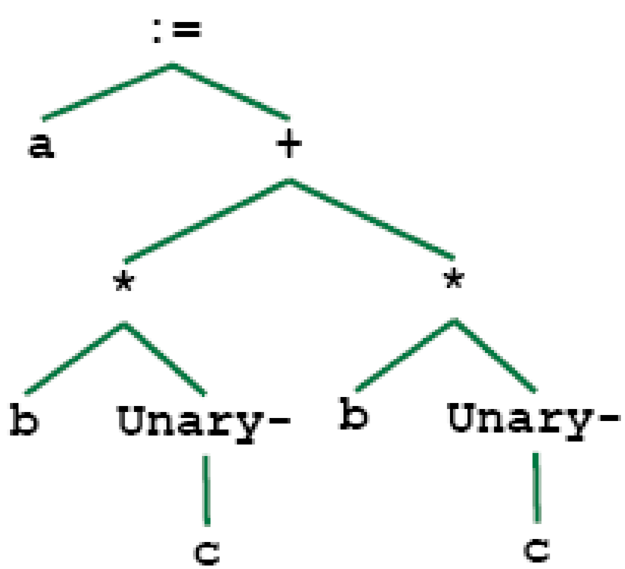


AST

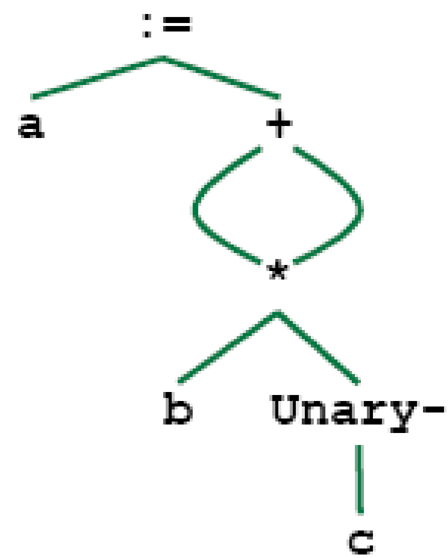
DAG

Implementation of DAG/AST: Value Number Method

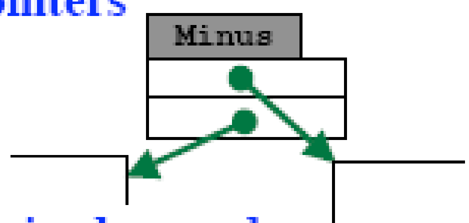
Tree:



DAG:



Structures and Pointers



An array of fixed-sized records

Q: How to build DAG's (i.e., trees with shared, common parts)?

A: When you are about to allocate a new node; look to see if you already have one with the same info.

0	id	b	-
1	id	c	-
2	unary-	1	-
3	mult	0	2
4	id	b	-
5	id	c	-
6	unary-	5	-
7	mult	4	6
8	add	3	7
9	id	a	-
10	assign	9	8

Three-Address Code

- A three-address code is:

  ```
  x := y op z
  ```

  where x, y and z are names, constants or compiler-generated temporaries; **`op`** is any operator.

- But we may also the following notation for three-address code (it looks like a machine code instruction)

  ```
  op  y,z,x
  ```

  apply operator `op` to `y` and `z`, and store the result in `x`.

- We use the term “three-address code” because each statement usually contains three addresses (two for operands, one for the result).

Linearized Representation of DAG/AST

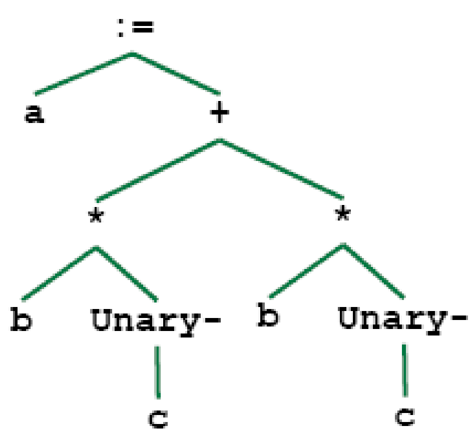
- Source Code
  - a = b * -c + b * -c
- Three address code

Each instruction has (up to) 3 operands.

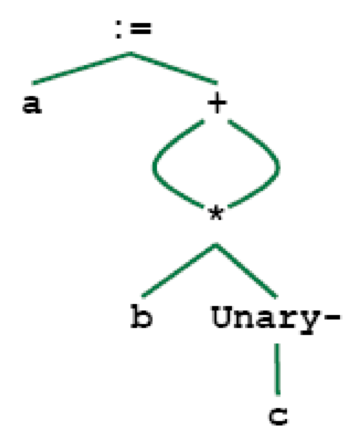
```
t1  := -c                 neg    c      ⇒ t1
t2  := b * t1             mult   b,t1   ⇒ t2
t3  := -c                 neg    c      ⇒ t3
t4  := b * t3             mult   b,t3   ⇒ t4
t5  := t2 + t4            add    t2,t4  ⇒ t5
a  := t5                  move   t5     ⇒ a
```

- Tree Representation

Tree:



DAG:



Three-Address Statements

Binary Operator: `op y,z,result` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:
```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

Unary Operator: `op y,,result` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:
```
uminus    a,,c
```

Three-Address Code

- Two concepts
  - Address
  - Instruction
- Address
  - Name: source-program names to appear as addresses
  - Constant: Different types of constants
  - Compiler Generated temporary:

Three-Address Instruction

Assignment Type 1: `x := y op z`

op is a binary arithmetic or logical operation

x, *y* and *z* are addresses

Assignment Type 2: `x := op z`

op is a unary arithmetic or logical operation

x and *z* are addresses

Copy Instruction: `x := y`

x and *z* are addresses and x is assigned the value of y

Three-Address Instructions

Unconditional Jump: `goto L`

We will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex: `goto L1` // jump to L1

`jmp 7` // jump to the statement 7

Conditional Jump 1: `if x goto L` and `ifFalse x goto L`

We will jump to the three-address code with the label `L` if `x` is TRUE and FALSE, respectively. Otherwise, the following three-address instruction in sequence is executed next.

Conditional Jump 2: `if x` ***`relop`*** `y goto L`

We will jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Three-Address Statements (cont.)

Procedure Parameters: `param x`

Procedure Calls: `call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters.

Ex:

```
param x1
param x2
......
param xn
call  p,n,
```

➔ $p(x_1, \ldots, x_n)$

n is necessary because call can be nested

```
f(x+1,y) ➔    add   x,1,t1
              param t1,,
              param y,,
              call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

```
x := y[i]
```

sets x to the value in location i memory units beyond location y

```
y[i] := x
```

sets contents of the location i memory units beyond location y to the value of x

Address and Pointer Assignments:

```
x := &y
```

sets the r-value of x to l-value of y

`x := *y` where y is a pointer whose r-value is a location

sets the r-value of x equal to the contents of that location

```
*x := y
```

sets the r-value of the object pointed by x to the r-value of y

Three address code example

do i=i+1; while (a[i] < v)

L: t_1=i+1
$\quad$ i=t_1
$\quad$ t_2=i*8
$\quad$ t_3=a[t_2]
$\quad$ 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[t2]
104: if t_3 < v goto 100

(B) Position Numbers

Representing 3-Address Statements

- Quadruples ("Quads")
- Triples
- Indirect Triples

```
t0 := x + y
t1 := t0 * z
a := t1
```

op, arg1, arg2, result

- `x = minus y`
  - Does not use arg2
- `x = y`
  - Op is =
- `param a1`
  - Uses neither arg2 nor result
- Conditional/Unconditional jumps
  - Put the target label in result

Quadruples

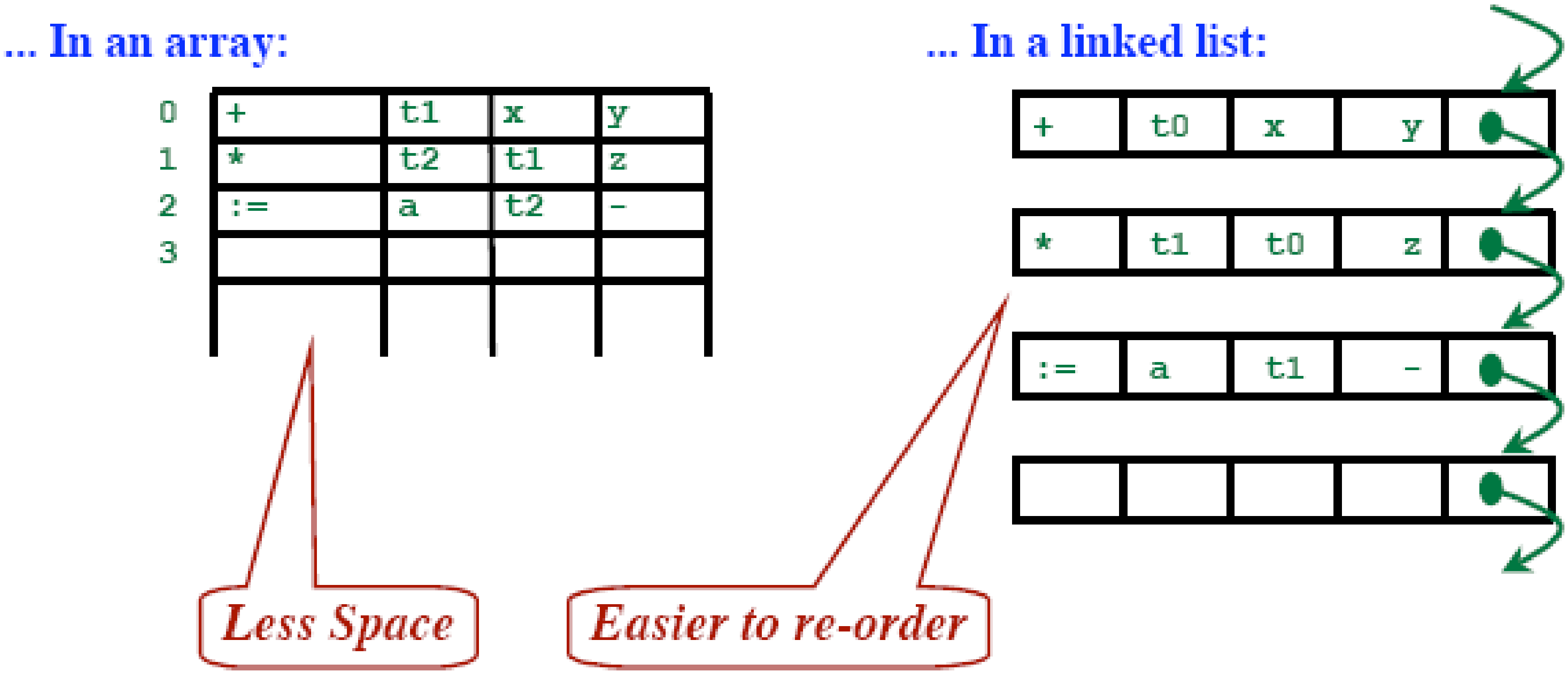
- a = b * -c + b * -c

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

Instr	Operation	Arg 1	Arg 2	Result
(0)	uminus	c		t_1
(1)	mult	b	t_1	t_2
(2)	add	a	t_2	t_3
(3)	move	t_3		a

Quadruples

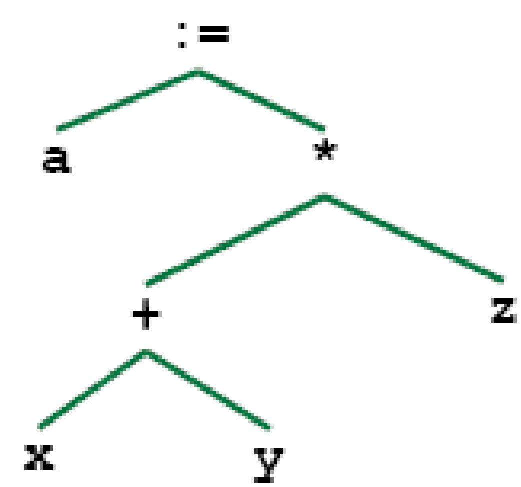
- Store each fields directly



Triples

Don't store the result directly.
Implicitly associate a temporary result with each triple.

```
t0 := x + y
t1 := t0 * z
a  := t1
```



Avoids creating the temporaries.
Saves storage.
Difficult to re-order instructions.

0	+	x	y
1	*	0	z
2	:=	a	1
3			

The following instruction is difficult

```
x[i] := y
```

It takes 2 triples.

	op	arg1	arg2
0	[ ]=	x	i
1	:=	0	y

Indirect Triples

Get around the re-ordering problem

... by introducing another data structure.

0:	100
1:	101
2:	102
3:	103

100:	+	x	y
101:	*	100	z
102:	:=	a	101
103:	:=	b	w

Quadruples

Less indirection, simpler

Easier to manipulate, reorder

Triples

Indirect Triples

About same amount of space as quadruples

May save space when lots of shared sub-expressions

More complex