Intermediate Representations/ Intermediate Code Generation

Intermediate Representations

- A compiler transforms the source program to an intermediate form that is mostly independent of the source language and the machine architecture.

- This approach isolates the **front-end** and the **back-end**

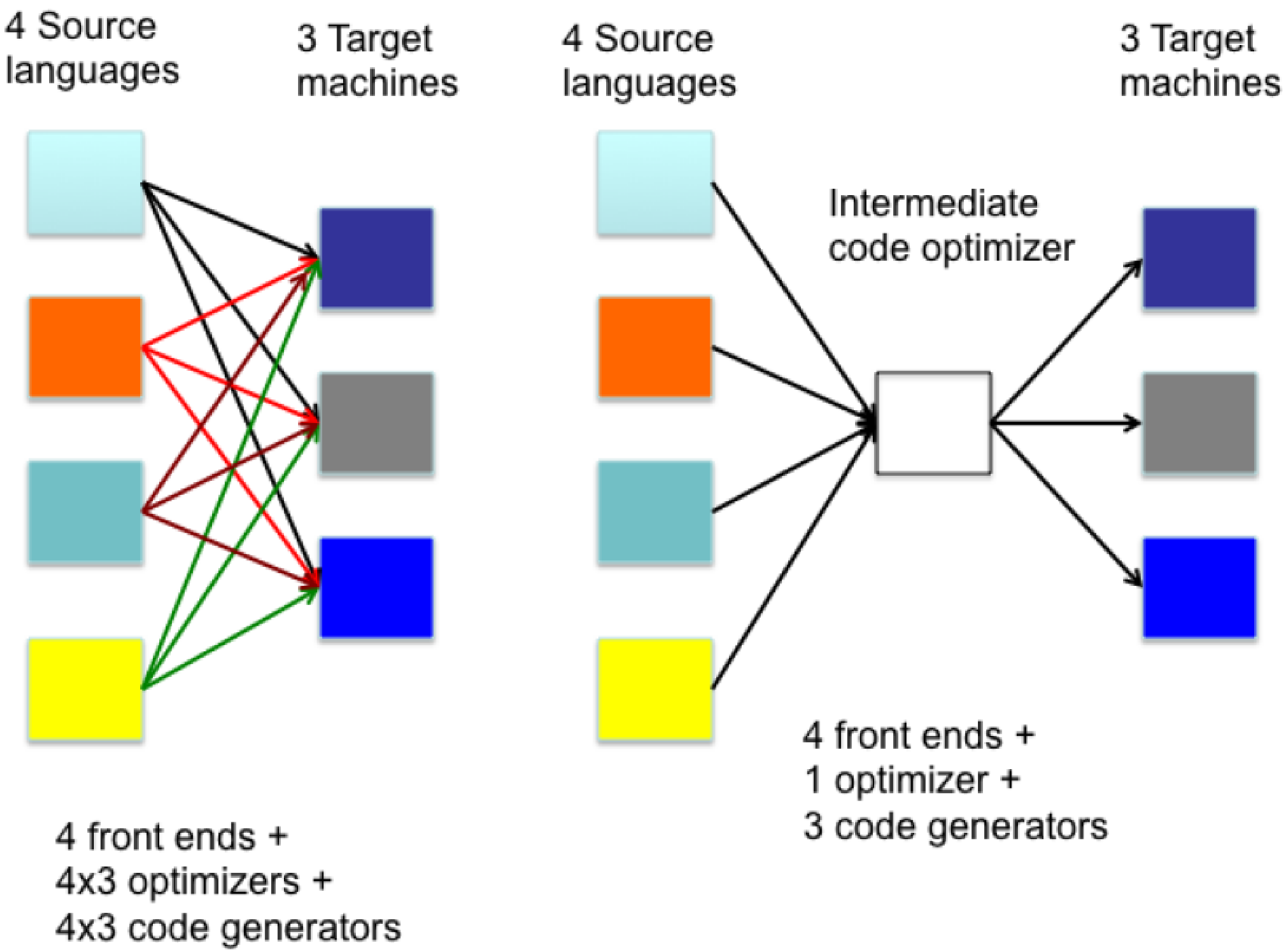
- Every source language has its front-end and every target language has its back-end

Why use an intermediate representation?

- break the compiler into manageable pieces
  - good software engineering technique
- simplifies retargeting to new host
  - isolates back-end from front-end
- simplifies handling of "poly-architecture" problem
  - **m** lang's, **n** targets ⇒ **m+n** components
- enables machine-independent optimization
  - general techniques, multiple passes

An intermediate representation is a ***compile-time*** data structure

Why use an intermediate representation?

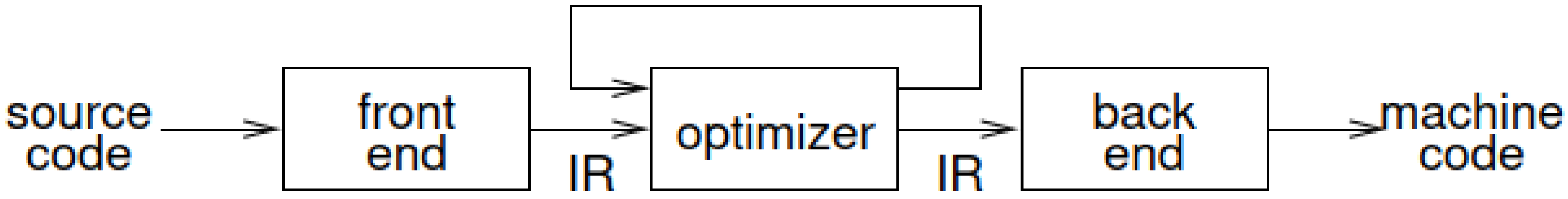


Why use an intermediate representation?

- While generating machine code directly from source code is possible, it entails two problems
  - With m languages and n target machines, we need to write m front ends, m×n optimizers, and m×n code generators
  - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused

- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- This means just **m** front ends, **n** code generators and **1** optimizer

Intermediate Representations



- front end produces ***IR***
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms ***IR*** into native code for the target machine

Different types of Intermediate representations

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of ***universal*** assembly language
  - Should not contain any machine-specific parameters

    (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for *machine-independent* **optimizations** and machine **code generation**
- Program Dependence Graph (PDG) is useful in automatic **parallelization**, instruction **scheduling**, and software pipelining

Intermediate Representations

- Some intermediate representations in the literature are:
  - abstract syntax trees (**AST**)
  - directed acyclic graphs (**DAG**)
  - control flow graphs
  - program dependence graphs
  - static single assignment form
  - 3-address code
  - Quadruples, triples, indirect triples
  - hybrid combinations

Properties- Intermediate Representations

Important IR Properties

1. ease of generation
2. ease of manipulation
3. cost of manipulation
4. level of abstraction
5. freedom of expression
6. size of typical procedure

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.

Level of exposed detail is a crucial consideration.

IR Design Issues

- **More than one intermediate representation may be used** for different levels of code improvement
- Is the chosen **IR** appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the **IR** *level*: close to language/machine.
- A **high-level intermediate form** preserves source language structure (e.g., *code improvements on loop* can be done)
- A **low-level intermediate form** is closer to target architecture

Intermediate Representations

Broadly speaking, IRs fall into three categories:

Structural

- structural IRs are graphically oriented
- examples include **trees**, **DAGs**
- heavily used in ***source*** *to* ***source*** translators

Linear

- pseudo-code
- large variation in level of abstraction
- easier to rearrange

Hybrids

- combination of graphs and linear code
- e.g., control-flow graphs

Parse Trees

- Parse tree is a representation of complete derivation of the input
- It has intermediate nodes labelled with non-terminals of derivation
- This is used (implicitly) for parsing and attribute synthesis

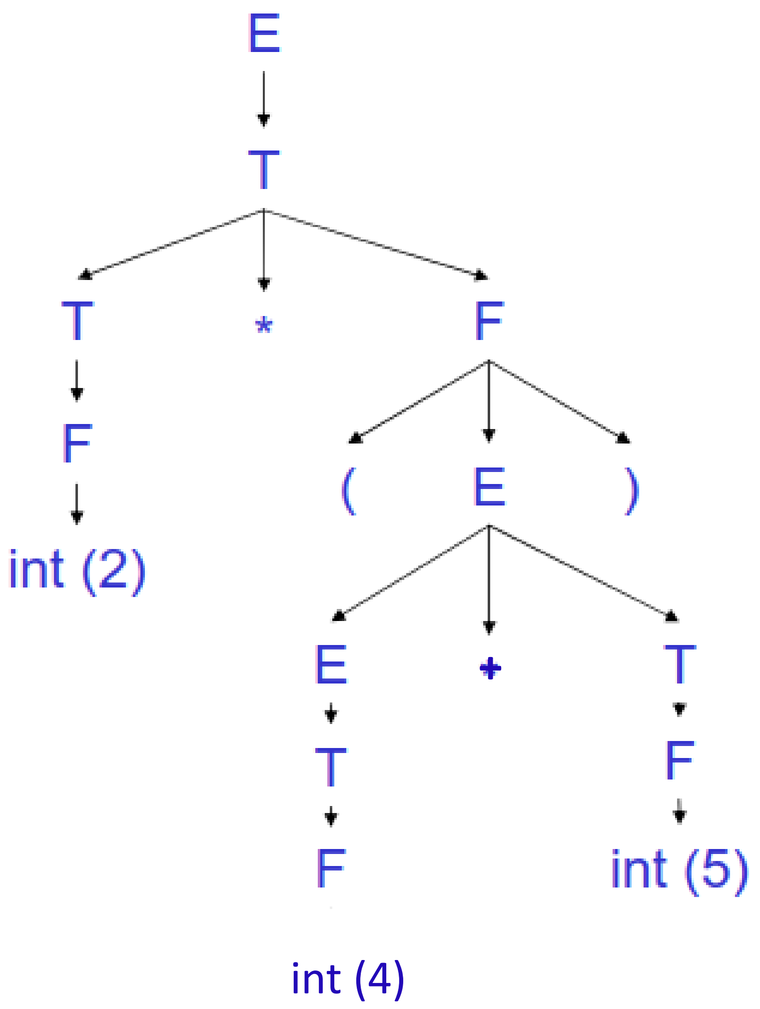
Syntax tree

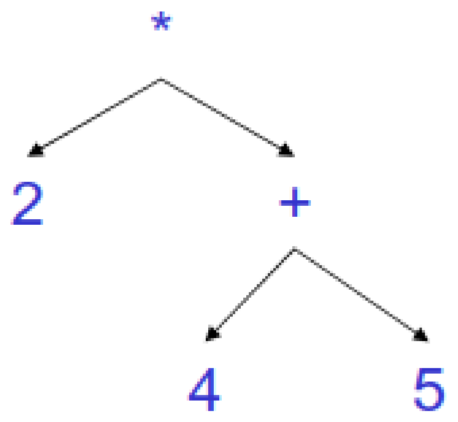
- An abstract syntax tree (**AST**) is very similar to parse tree where extraneous nodes are removed (nodes for most non-terminal symbols removed).

- Good intermediate representation that is close to the source language

- May be used in applications such as source-to-source translation

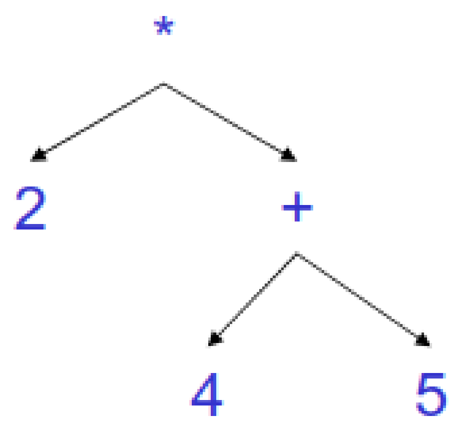
Example: parse tree v.s. AST- “ 2 * (4 + 5)”

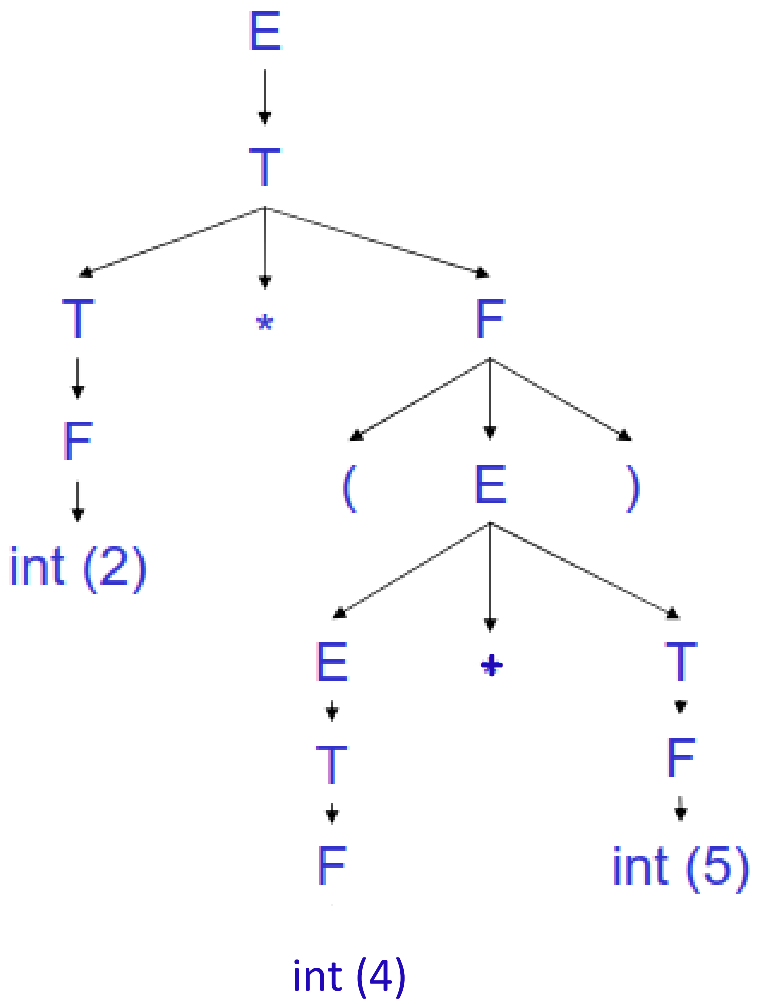




Syntax directed translation: Building AST

$E_1 \rightarrow E_2 + T$	E_1.trans = new PlusNode(E_2.trans, T.trans)
$E \rightarrow T$	E.trans = T.trans
$T_1 \rightarrow T_2 * F$	T_1.trans = new TimesNode(T_2.trans, F.trans)
$T \rightarrow F$	T.trans = F.trans
$F \rightarrow int$	F.trans = new IntLitNode(int.value)
$F \rightarrow (E)$	F.trans = E.trans





Directed acyclic graph

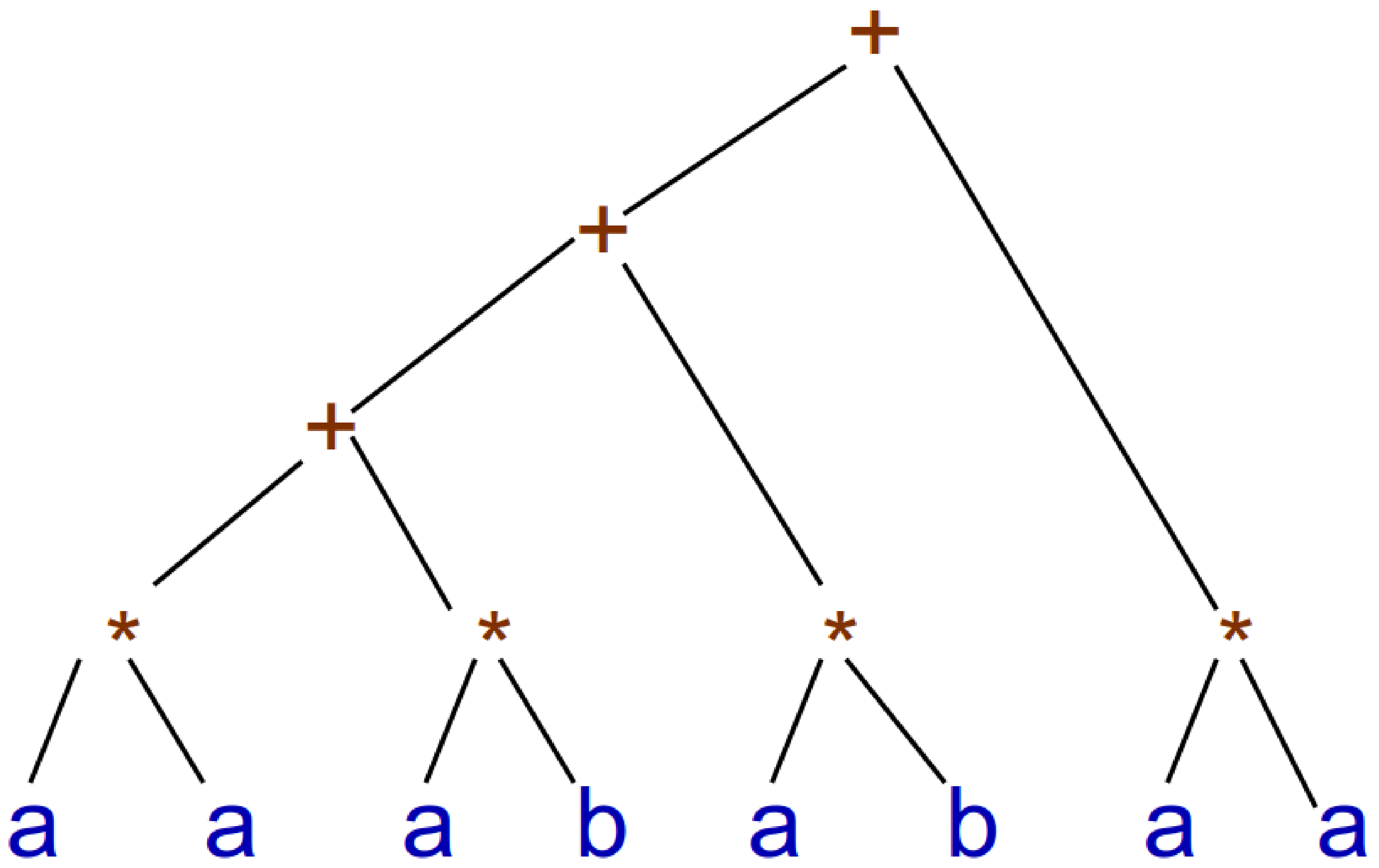
- A directed acyclic graph (**DAG**) is an **AST** with a **unique node for each value**.

- A DAG is an improvement over a syntax tree where duplications of subtrees such as common subexpressions are identified and shared

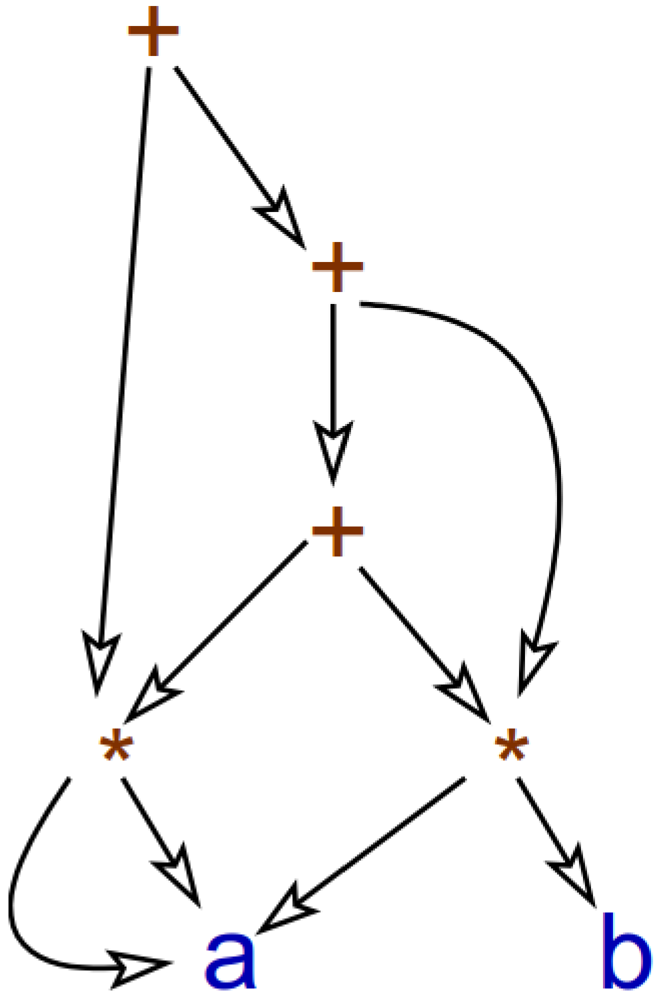
- Cost of evaluation can be reduced (by identifying ***common sub-expressions***)

AST Vs DAG

a*a+a*b+a*b+a*a



Abstract Syntax tree



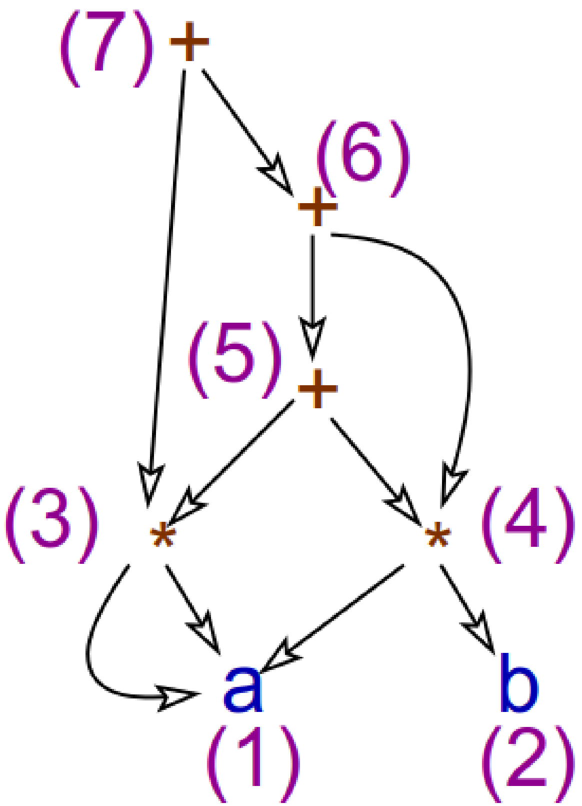
DAG

Exercise

Consider the expression grammar.

Provide Attribute translation grammar/Syntax directed translation for obtaining DAG.

DAG and its nodes



1	ID(a)		
2	ID(b)		
3	*	1	1
4	*	1	2
5	+	3	4
6	+	5	4
7	+	3	6

SymTab

Graph Representations

- Different type of graph representations used to represent and analyse properties of a program
- Control-flow graph: Models flow of control between basic blocks
- Data-dependency graph: Captures the definition/creation of new data and its usage
- Call-graph: used for inter-procdurial analysis of code (edge from each instance of call to the procedure)

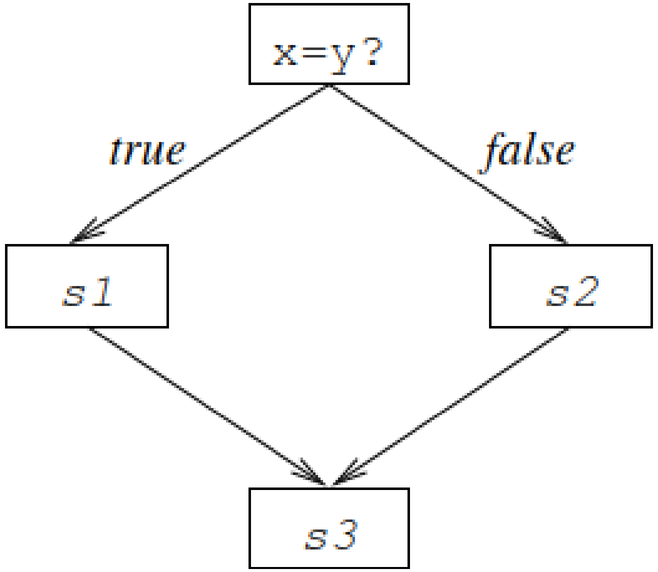
Control-flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are basic blocks
- edges in the graph represent control flow

loops, if-then-else, case, goto

```
if (x=y) then
   s1
else
   s2
s3
```



Linear Intermediate Representations

- Both high-level source code and target assembly code are linear in their text.

- The intermediate representation may also be linear sequence of codes (with ***conditional branches*** and ***jumps*** to control the flow of computation)

- A linear intermediate code may have **one** operand address, **two-address**, or **three-address** like RISC architectures.

- We only talk about three-address codes.

3-address code

- At most one operator on the right side of an instruction.
- 3-address code can mean a variety of representations.

- In general, it allows statements of the form:

  x ← y op z

  with a single operator and, at most, three names.
- Simpler form of expression:

  x - 2 * y

  becomes

  t1 ← 2 * y

  t2 ← x - t1

NOTE: *names for intermediate values*

3-address code: Addresses

- Three-address code is built from two concepts: *addresses* and *instructions*.
- An ***address*** can be
  - A **name**: source variable program name or pointer to the Symbol Table name.
  - A **constant**: Constants in the program.
  - Compiler generated **temporary**.

3-address code

- Instructions are very simple
- Examples: a = b + c

  x = -y

  if a > b goto L1

- **LHS** is the target and the **RHS** has at most two sources and one operator
- **RHS** sources can be either variables or constants

- **Three-address** code is a *generic form* and **can be implemented as** quadruples, triples, indirect triples, tree or DAG

Example

- The three-address code for **a+b*c-d/(b*c)**

1. t1 = b*c
2. t2 = a+t1
3. t3 = b*c
4. t4 = d/t3
5. t5 = t2-t4

3-address codes (typical instructions types)

- assignments $x \leftarrow y \text{ op } z$
- assignments $x \leftarrow \text{op } y$
- assignments $x \leftarrow y[i]$
- assignments $x \leftarrow y$
- **Branches** (unconditional jump) goto L
- **conditional branches** if x goto L
- **procedure calls**
- address and pointer assignments

Instructions in 3-address code (assignment instructions)

1. *Assignment instructions*:
   `a = b biop c`, `a = uop b`, and `a = b` (copy), where
   - *biop* is any binary arithmetic, logical, or relational operator
   - *uop* is any unary arithmetic (-, shift, conversion) or logical operator ($\sim$)
   - Conversion operators are useful for converting integers to floating point numbers, etc.

Instructions in 3-address code (Jump instructions)

2. *Jump instructions*:
   `goto L` (unconditional jump to L),
   `if t goto L` (it *t* is *true* then jump to L),
   `if a relop b goto L` (jump to L if *a* relop *b* is *true*),
   where
   - *L* is the label of the next three-address instruction to be executed
   - *t* is a boolean variable
   - *a* and *b* are either variables or constants

Instructions in 3-address code (functions)

3. *Functions*:
   `func begin <name>` (beginning of the function),
   `func end` (end of a function),
   `param p` (place a value parameter *p* on stack),
   `refparam p` (place a reference parameter *p* on stack),
   `call f, n` (call a function *f* with *n* parameters),
   `return` (return from a function),
   `return a` (return from a function with a value *a*)

4. *Indexed copy instructions*:

5. *Pointer assignments*:

Example: 3-address code

```
int x;
int y;

int x2 = x * x;
int y2 = y * y;
int r2 = x2 + y2;
```

```
x2 = x * x;
y2 = y * y;
r2 = x2 + y2;
```

Example: 3-address code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

Example: 3-address code

```
int a;
int b;
int c;
int d;

a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

Example: 3-address code (control-flow statements)

```
int x;
int y;
int z;

if (x < y)
   z = x;
else
   z = y;

z = z * z;
```

Example: 3-address code (control-flow statements)

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Labels

- **TAC** allows for named labels indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
- Goto **label**;
- If **value** Goto **label**;
- Note that If is always paired with Goto.

Example: 3-address code (complete function)

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

Example: 3-address code (complete function)

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

Implementations of 3-address code

3-address code

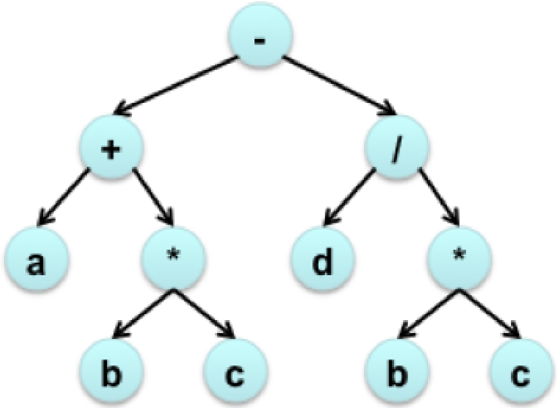
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

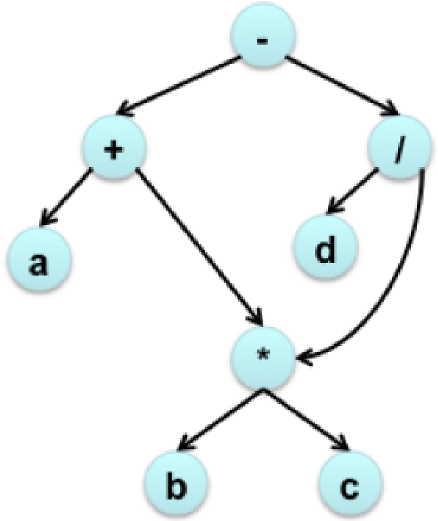
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

3-address code implementation- Quadruples

- Has four fields: op, arg1, arg2 and result.
- Some instructions (e.g. unary minus) do not use arg2.
- Instructions like param don't use neither arg2 nor result.
- **Jumps** put the target label in result.

x - 2 * y

	op	result	arg1	arg2
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

- simple record structure with four fields
- **easy to reorder**
- **explicit names**

3-address code implementation- Triples

Triples

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

- use **table index** as “**implicit name**”
- require only three fields in record
- **harder to reorder**

3-address code implementation- Indirect Triples

Indirect Triples

`x - 2 * y`

	exec-order	stmt	op	arg1	arg2
(1)	(100)	(100)	load	y	
(2)	(101)	(101)	loadi	2	
(3)	(102)	(102)	mult	(100)	(101)
(4)	(103)	(103)	load	x	
(5)	(104)	(104)	sub	(103)	(102)

- simplifies moving statements (change the execution order)
- more space than triples
- implicit name space management

Indirect triples- advantage

```
for i:=1 to 10 do
begin
 a=b*c
 d=i*3
end
```

Optimized version

```
a=b*c
for i:=1 to 10 do
begin
 d=i*3
end
```

```
(1)  := 1 i
(2)  nop
(3)  * b c
(4)  := (3) a
(5)  * 3 i
(6)  := (5) d
(7)  + 1 i
(8)  := (7) i
(9)  LE i 10
(10)  IFT goto (2)
```

Execution Order (a) : 1 2 3 4 5 6 7 8 9 10

Execution Order (b) : 3 4 1 2 5 6 7 8 9 10

Intermediate Representations

- Many kinds of ***IR*** are used in practice.
- A compiler may need several different ***IRs***
- Choose ***IR*** with right level of detail
- Keep manipulation costs in mind

Gap between HLL and IR

- Gap between HLL and IR
  - High level languages may allow complexities that are not allowed in IR (*such as **expressions with multiple operators***).
  - High level languages have **many syntactic constructs**, not present in the IR (*such as **if-then-else or loops***)
- Challenges in translation:
  - We need a systematic approach to IR generation.
- Goal:
  - A HLL to IR translator.
  - Input: A program in **HLL**.
  - Output: A program in **IR** (may be an AST or program text)

Intermediate code- Example

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
     dot_prod = 0;          |     T6 = T4[T5]
     i = 0;                 |     T7 = T3*T6
L1:  if(i >= 10)goto L2     |     T8 = dot_prod+T7
     T1 = addr(a)           |     dot_prod = T8
     T2 = i*4               |     T9 = i+1
     T3 = T1[T2]            |     i = T9
     T4 = addr(b)           |     goto L1
     T5 = i*4               |L2:
```

Intermediate code- Example (cont.)

C-Program (main)

```
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Intermediate code

```
    func begin main
    refparam a
    refparam b
    refparam result
    call dot_prod, 3
    p = result
    func end
```

Intermediate Code- Example (rec.)

C-Program (function)

```
int fact(int n){
   if (n==0) return 1;
   else return (n*fact(n-1));
}
```

Intermediate code

```
func begin fact     |       T3 = n*result
if (n==0) goto L1   |       return T3
T1 = n-1            |   L1: return 1
param T1            |       func end
refparam result     |
call fact, 2        |
```

Code Templates (*If-then-Else statement*)

```
If  (E)  S1  else  S2
        code for E (result in T)
        if  T≤ 0  goto  L1 /* if T is false, jump to else part */
        code for S1 /* all exits from within S1 also jump to L2 */
        goto L2 /* jump to exit */
L1:     code for S2 /* all exits from within S2 also jump to L2 */
L2:     /* exit */
```

```
If  (E)  S
        code for E (result in T)
        if  T≤ 0  goto  L1 /* if T is false, jump to exit */
        code for S /* all exits from within S also jump to L1 */
L1:     /* exit */
```

Example: Translation- *if-then-else statement*

- Code generated for the following code fragment:

  A_i are all assignments, and E_i are all expressions

  if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

```
1           code for E1 /* result in T1 */
10          if (T1 <= 0), goto L1 (61)
            /* if T1 is false jump to else part */
11          code for E2 /* result in T2 */
35          if (T2 <= 0), goto L2 (43)
            /* if T2 is false jump to else part */
36          code for A1
42          goto L3 (82)
43    L2:   code for A2
60          goto L3 (82)
61    L1:   code for A3
82    L3:   code for A4
```

Code templates (*While statement*)

```
while  (E)  do  S
L1:      code for E (result in T)
         if  T≤ 0  goto  L2 /* if T is false, jump to exit */
         code for S /* all exits from within S also jump to L1 */
         goto L1 /* loop back */
L2:      /* exit */
```

Example: Translation- *While statement*

Code fragment:

while (E_1) do {if (E_2) then A_1; else A_2;} A_3;

```
1     L1:   code for E1 /* result in T1 */
15          if (T1 <= 0), goto L2 (79)
            /* if T1 is false jump to loop exit */
16          code for E2 /* result in T2 */
30          if (T2 <= 0), goto L3 (55)
            /* if T2 is false jump to else part */
31          code for A1
54          goto L1 (1)/* loop back */
55    L3:   code for A2
78          goto L1 (1)/* loop back */
79    L2:   code for A3
```

Attributes grammars &
Attribute translation grammars

Translating expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

- Builds the three-address code for an ***assignment statement***.
- **Code**: synthesized attribute (denotes 3-address code)
- **addr**: a synthesized-attr of **E** (denotes the address holding the val of **E**.
- Constructs a three-address instruction and appends the instruction to the sequence of instructions.
- **top** is the current symbol table.

Example: 3-address code sequence generated for a = b+-c

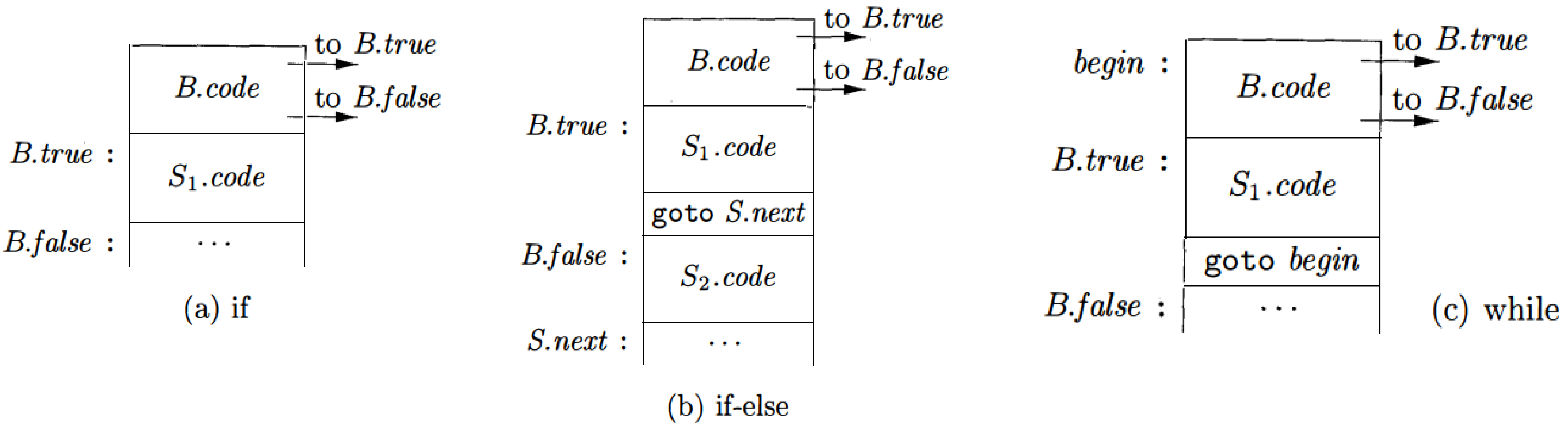
Translating expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \Vert$ $\quad gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \Vert\ E_2.code \ \Vert$ $\quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\quad \mid\ - E_1$	$E.addr = \mathbf{new}\ Temp()$ $E.code = E_1.code \ \Vert$ $\quad gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\quad \mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\quad \mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

Example: 3-address code sequence generated for a = b+-c

IR generation for flow-of-control statements

S -> ***if*** (B) S1
S -> ***if*** (B) S1 ***else*** S2
S -> ***while*** (B) S1



IR generation for flow-of-control statements

- **code** (***synthesized* attribute**) giving the code for that node.
- **S.next, B.true, B.false: Inherited attributes**
- **Assume**: ***gen*** only creates an instruction.
- **||** concatenates the code.

P -> S	**S.*next* =** newlabel() **p.*code*=** S.*code* || label (S.*next*)
S -> assign	**S.*code* =** assign.*code*
S -> if (B) S1	**B.*true* =** newlabel() **B.*false* =** S1.*next* = s.*next* **S.*code* =** B.*code* || label (B.*true*) || S1.*code*
S - > if (B) S1 else S2	**B.true =** newlabel() **B.false =** newlabel() **S1.next =** S2.next = S.next **S.code =** B.code || label(B.true) || S1.code || gen('goto' S.next) || label(B.false) || S2.code

IR generation for flow-of-control statements

S -> while (B) S1	**begin** = newlabel() **B.*true*** = newlabel() **B.*false*** = S.next **S1.next** = begin **S.*code*** = label (begin) || B.code || label(B.true) || S1.code || *gen* ('goto' begin)
S - > S1 S2	**S1.*next* =** newlabel() **S2.*next* =** S.*next* **S.*code* =** S1.*code* || label(s1.*next*) || S2.*code*

- **code** is an *synthesized* attribute: giving the code for that node.
- **Assume**: gen only creates an instruction.
- **||** concatenates the code.

Control-flow Translation of Boolean Expressions

B - > B1 || B2	**B1.true** = B.true **B1.false** = newlabel() **B2.true** = B.true **B2.false** = B.false **B.code** = B1.code || label(B1.false) || B2.code
B -> B1 && B2	**B1.true** = newlabel() **B1.false** = B.false **B2.true** = B.true **B2.false** = B.false **B.code** = B1.code || label(B1.true) || B2.code
B -> E1 rel E2	**B.Code** = E1.code || E2.code || gen ('if' E1.addr rel.op E2.addr 'goto' B.true) || gen ('goto' B.false)

Example: 3-address code using SDD’s defined

if (x < 100 || x > 200 && x != y) x = 0

Example: 3-address code using SDD's defined

if (x < 100 || x > 200 && x != y) x = 0

if x < 100 ***goto*** L2
goto L3

L3 : ***if*** x > 200 ***goto*** L4
goto L1

L4 : ***if*** x ! = y ***goto*** L2
goto L1

L2 : x = 0
L1 :