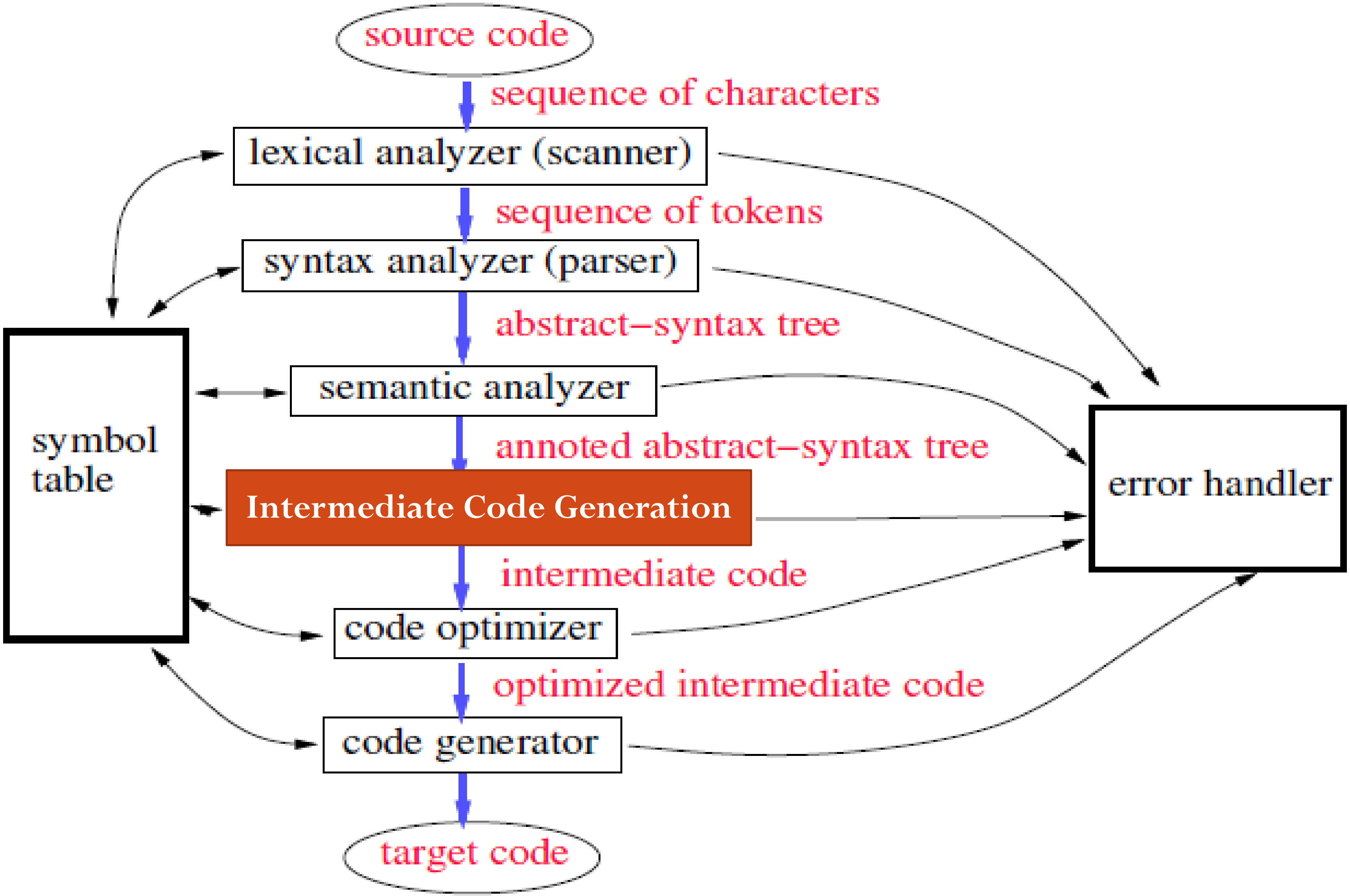
Intermediate Code Generation
&
Code Optimization

By Birku L.

Contents

- Introduction
- Why IR?
- Intermediate Representations
  - **Graphical IRs** (Abstract Syntax Trees, Directed Acyclic Graphs, Control Flow Graphs)
  - **Linear LRs** (Postfix notation, Static Single Assignment Form, Stack Machine Code, Three Address Code)
- Code Optimization
- Optimization Techniques

Introduction



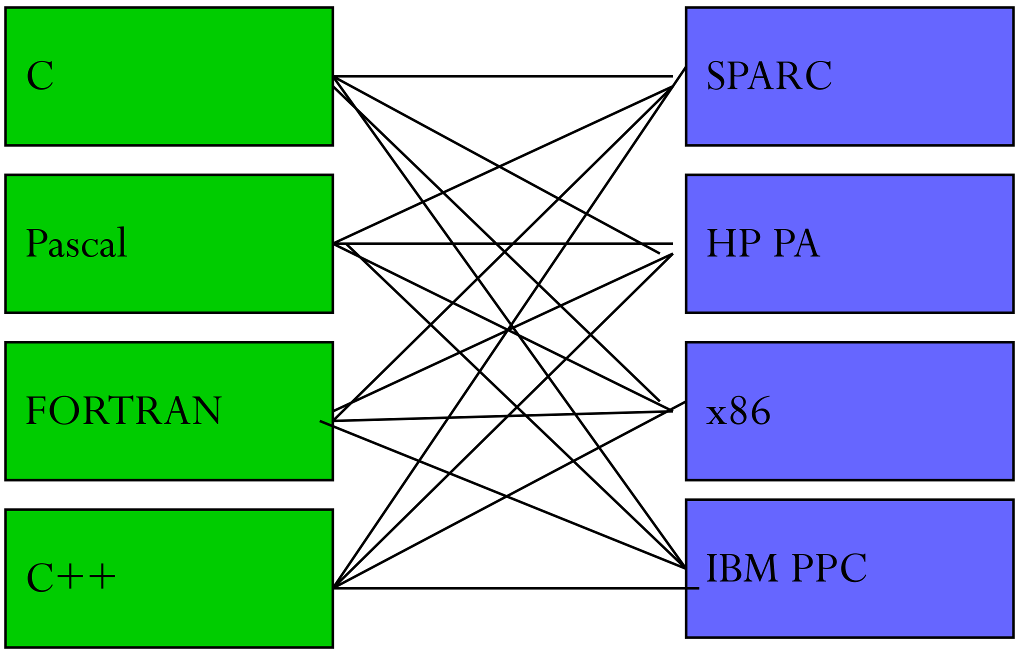
Introduction

- Intermediate code which is also called *Intermediate representation, intermediate language* is
  - a kind of abstract machine language that can express the target machine operations without committing too much machine details.

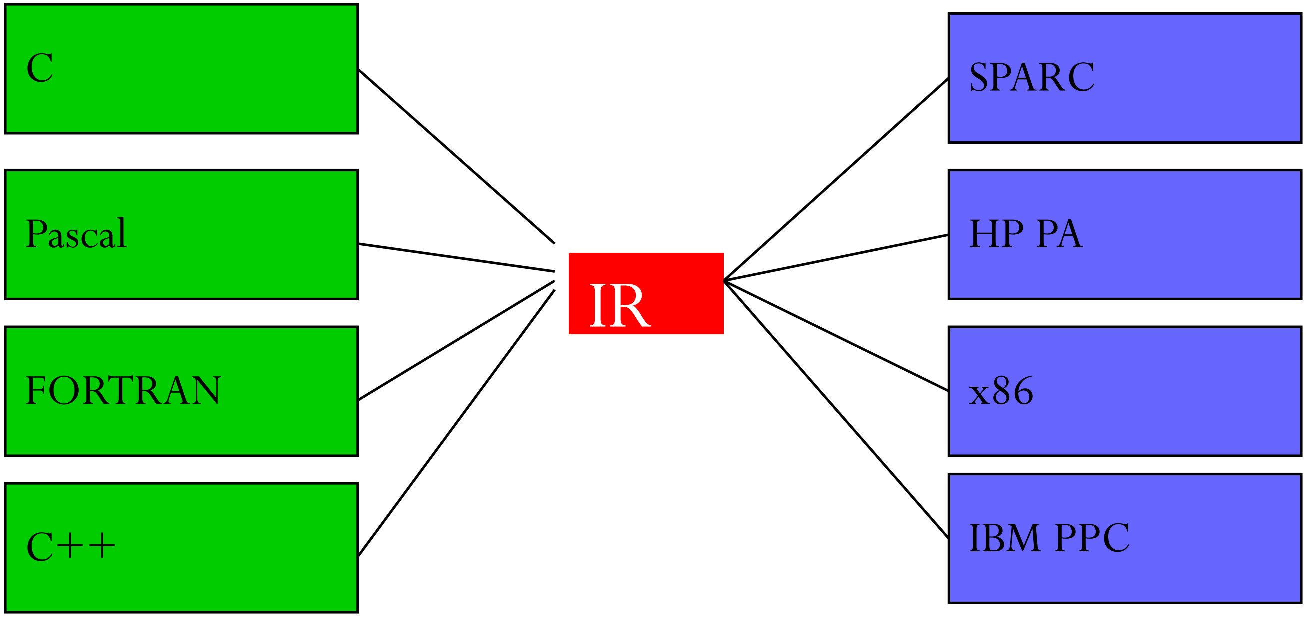
- *Intermediate representation*
  - It ties the front and back ends together
  - Language and Machine neutral
    - no limit on register and memory, no machine-specific instructions.
  - Many forms
    - Syntax trees, three-address code, quadruples.
- Intermediate code generation can affect the performance of the back end

Why IR?

Portability *- Suppose We have n-source languages and m-Target languages. Without Intermediate code we will change each source language into target language directly. So, for each source-target pair we will need a compiler. Hence we will require (n*m) Compilers, one for each pair. If we Use Intermediate code We will require n-Compilers to convert each source language into Intermediate code and m-Compilers to convert Intermediate code into m-target languages. Thus We require only (n+m) Compilers.*



Why IR?...



- ***Retargeting*** - Build a compiler for a new machine by attaching a new code generator to an existing front-end.
- ***Optimization*** - reuse intermediate code optimizers in compilers for different languages and different machines.
- ***Program understanding*** - Intermediate code is simple enough to be easily converted to any target code but complex enough to represent all the complex structure of high level language.

Intermediate Representations

- Intermediate Representations can be expressed using

Graphical IRs:
  - Abstract Syntax Trees (AST)
  - Directed Acyclic Graphs (DAG)
  - Control Flow Graphs (CFG)

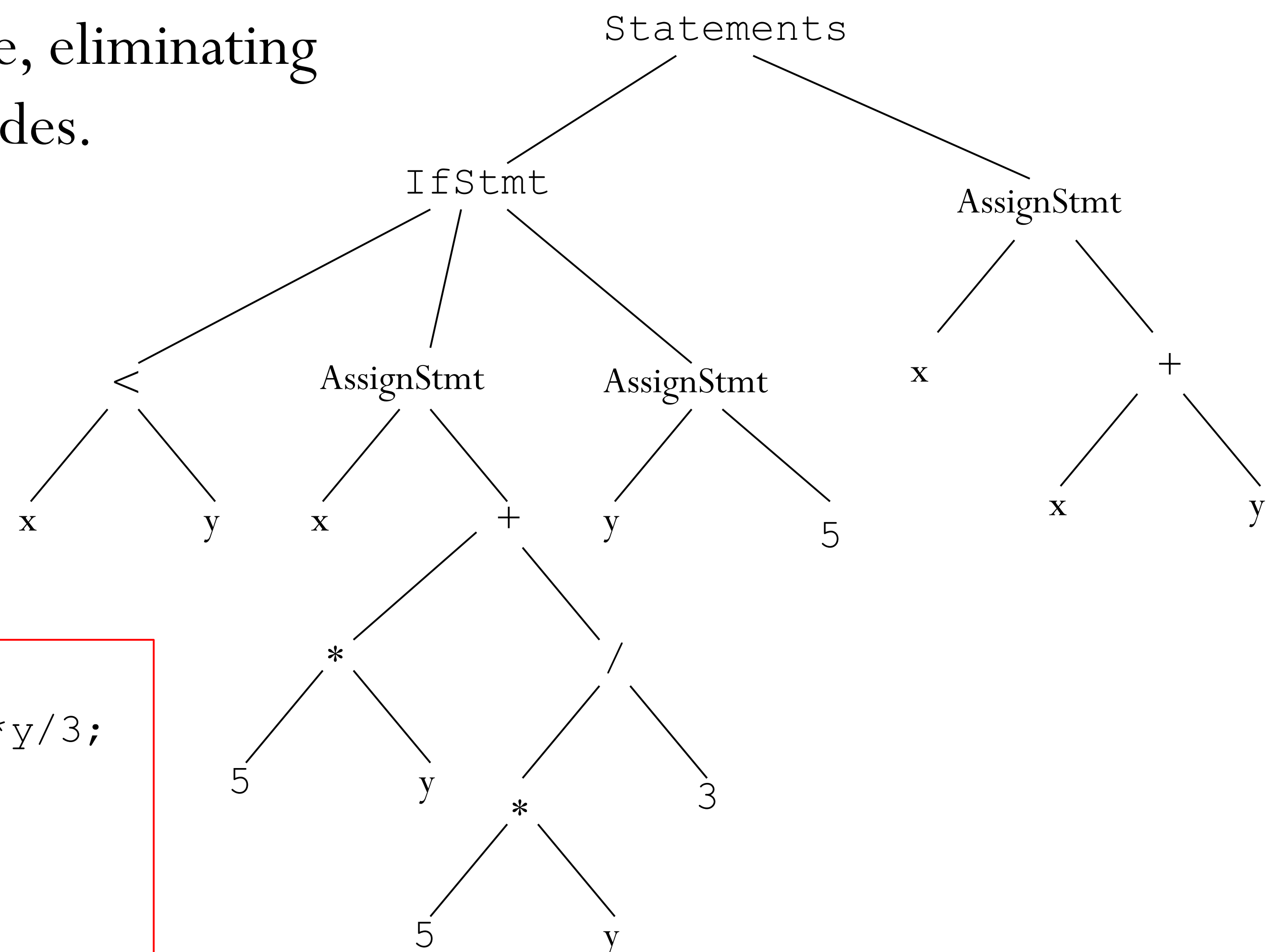
Linear IRs:
  - Postfix notation
  - Three Address Code
  - Static Single Assignment Form (SSA)
  - Stack Machine Code

high-level → low-level

- Hybrid approaches mix graphical and linear representations
  - SGI and SUN compilers use three address code but provide ASTs for loops if-statements and array references
  - Use three-address code in basic blocks in control flow graphs

Abstract Syntax Trees (ASTs)

- retain essential structure of the parse tree, eliminating unneeded nodes.



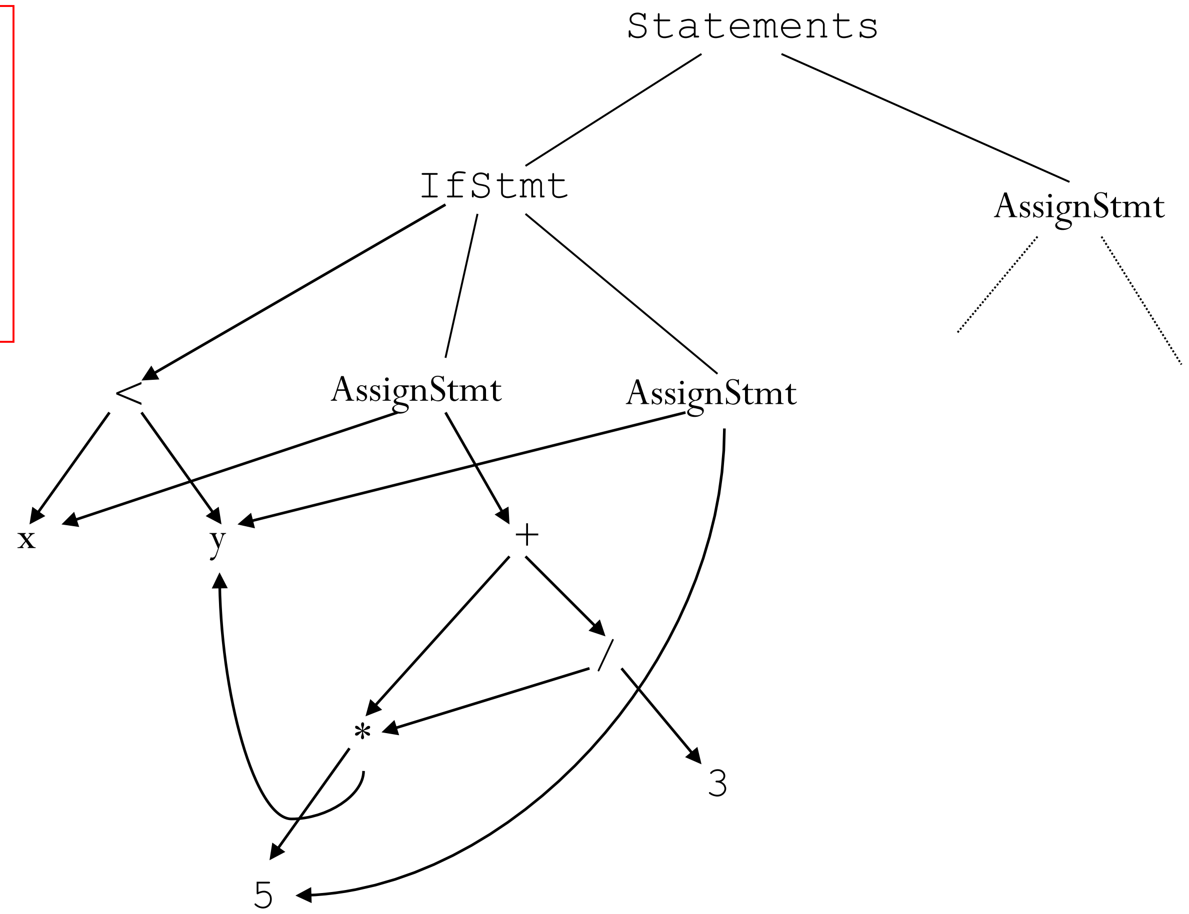

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```

Directed Acyclic Graphs (DAGs)

- Directed acyclic graphs (DAGs)
  - Like compressed trees
    - Leaves are distinct: variables, constants available on entry
    - internal nodes: operators
  - Can generate efficient code – since it encode common expressions
  - But difficult to transform
- Check whether an operand is already present
  - if not, create a leaf for it
- Check whether there is a parent of the operand that represents the same operation
  - if not create one, then label the node representing the result with the name of the destination variable, and remove that label from all other nodes in the DAG

Directed Acyclic Graphs (DAGs)…

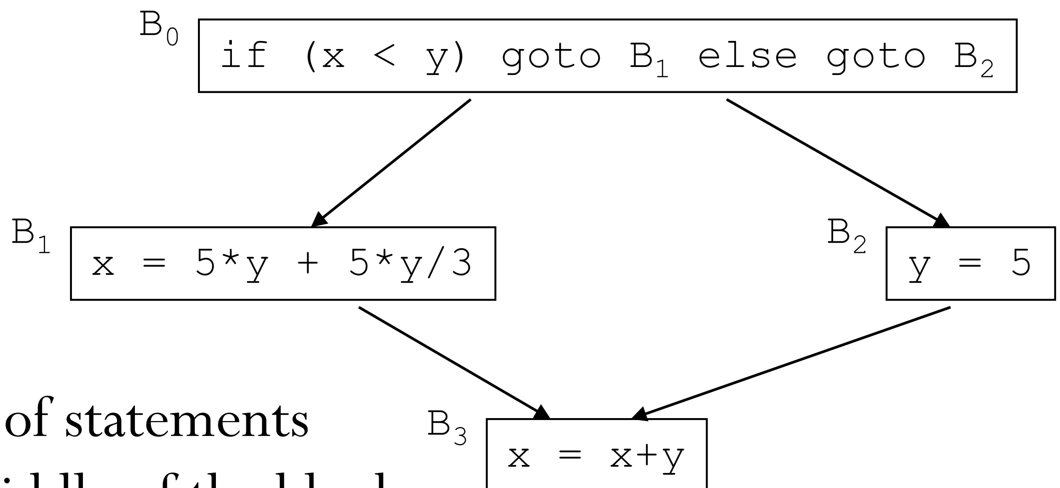
```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```



Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A *basic block* is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

```
if (x < y)
   x = 5*y + 5*y/3;
else
   y = 5;
x = x+y;
```



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Postfix Notation (PN)

- A mathematical notation wherein every operator follows all of its operands.
  - Example: The PN of expression 9* (5+2) is 952+*

- Rules:

1. If E is a variable/constant, the PN of E is E itself
2. If E is an expression of the form E_1 op E_2, the PN of E is $E_1' E_2'$op (E_1' and E_2' are the PN of E_1 and E_2, respectively.)
3. If E is a parenthesized expression of form (E_1), the PN of E is the same as the PN of E_1.

- ***Example:***
  - The PN of expression (a+b)/(c-d) ? is (ab+)(cd-)/

Three-Address Code

- A popular form of intermediate code used in optimizing compilers
- Each instruction can have at most three operands
- Types of three address code statements
  - Assignment statements: x := y op z and x := op y
  - Indexed assignments: x := y[i] and x[i] := y
  - Pointer assignments: x := & y, x := * y, * x := y
  - Unconditional jumps: goto L
  - Conditional jumps: if x relop y goto L
  - Function calls: param x, call p, n , and return y
  - p(x1,..., xn) => param x1 . . .

    Param xn

    call p, n

Three-Address Code

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```

Variables can be represented with their locations in the symbol table

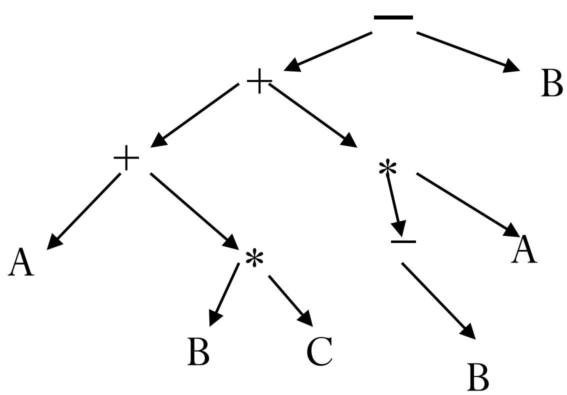
```
        if x < y goto L1
        goto L2
L1:     t1 := 5 * y
        t2 := 5 * y
        t3 := t2 / 3
        x := t1 + t3
        goto L3
L2:     y := 5
L3:     x := x + y
```

Temporaries: temporaries correspond to the internal nodes of the syntax tree

- Three address code instructions can be represented as an array of
  quadruples: operation, argument1, argument2, result
  triples: operation, argument1, argument2
  (each triple implicitly corresponds to a temporary)

Syntax tree vs. Three address code

- Expression: (A+B*C) + (-B*A) – B

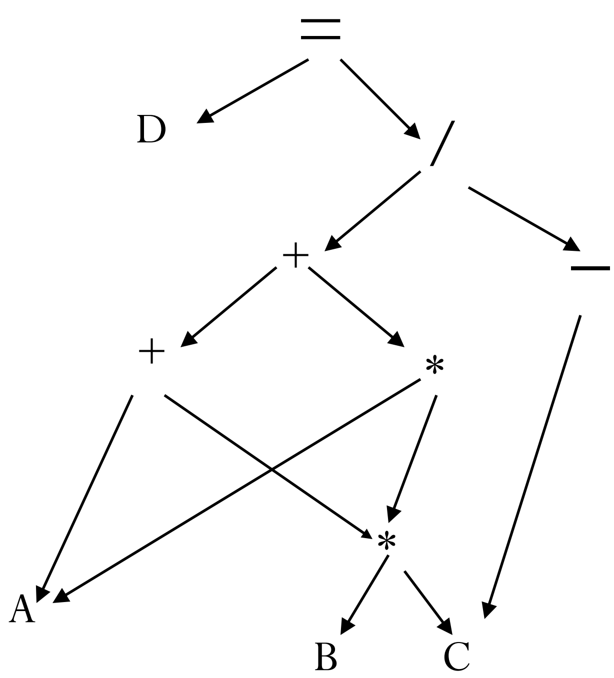


T1 := B * C
T2 = A + T1
T3 = - B
T4 = T3 * A
T5 = T2 + T4
T6 = T5 – B

- Three address code is a linearized representation of a syntax tree (or a DAG) in which explicit names (temporaries) correspond to the interior nodes of the graph.

DAG vs. Three address code

Expression: D = ((A+B*C) + (A*B*C))/ -C




```
T1 := A
T2 := C
T3 := B * T2
T4 := T1+T3
T5 := T1*T3
T6 := T4 + T5
T7 :=  –T2
T8 := T6 / T7
D   := T8
```

```
T1 := B * C
T2 := A+T1
T3 := A*T1
T4 := T2+T3
T5 :=  – C
T6 := T4 / T5
D   := T6
```

Question: Which IR code sequence is better?

Three Address code for Control flow

- While Statements

Production	Semantic Rule
$S \rightarrow$ **while** E **do** S_1	*S.begin* := *newlabel()* *S.after* := *newlabel()* *S.code* := *gen*(*S.begin* ':') || *E.code* || *gen* ('if' *E.place* '=' '0' 'goto' *S.after*) || *S1.code* || *gen* ('goto' *S.begin*) || *gen*(*S.after* ':')

Example

Source program fragment

```
i := 2 * n + k
while i do
    i := i - k
```

Three-address code sequence

```
    t1 := 2
    t2 := t1 * n
    t3 := t2 + k
    i := t3
L1: if i = 0 goto L2
    t4 := i - k
    i := t4
    goto L1
L2:
```

Implementation of Three Address Code

- Quadruples
  - Four fields: op, arg1, arg2, result
    - Array of struct {op, *arg1, *arg2, *result}
  - x:=y op z is represented as op y, z, x
  - arg1, arg2 and result are usually pointers to symbol table entries.
  - May need to use many temporary names.
  - Many assembly instructions are like quadruple, but *arg1, arg2, and result are real registers.
  - Example:

a = b * - c + b * - c

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

(a) Three-address code

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

(b) Quadruples

Implementation of Three Address Code…

- Triples
  - Three fields: op, arg1, and arg2. Result become implicit.
  - arg1 and arg2 are either pointers to the symbol table or index/pointers to the triple structure.
  - No explicit temporary names used.
  - Need more than one entries for ternary operations such as x:=y[i], a=b+c, x[i]=y, … etc.

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

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

Three-address code

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

Triples

Pointer to symbol table

Static Single- Assignment Form

- Facilitâtes certain code optimisations
- Two distinctive aspects distinguish SSA from three-address code.
  - First, all assignments in SSA are to variables with distinct names; hence the term static single-assignment.

(a) Three-address code.	(b) Static single-assignment form.
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

(a) Three-address code. (b) Static single-assignment form.

  - Second, SSA uses a notational convention called the q-function to combine two or more definitions of a variable :
    - For example, the same variable may be defined in two different control-flow paths in a program

Static Single- Assignment Form

if (flag) x = -1; else x = 1;

y=x*a;

- If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment y = x * a;
- In this case q-function is used to combine the two definitions of x:

if (flag) x_1 = -1; else x_2 = 1;

x3 = q(x_1,x_2);

- the q-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the q-function.

Stack Machine Code

Assumes presence of operand stack

- Useful for stack architectures, JVM
- Operations typically pop operands and push results.
- Easy code generation and has compact form
- But difficult to reuse expressions and to rearrange

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```

```
    load x
    load y
    iflt L1
    goto L2
L1: push 5
    load y
    multiply
    push 5
    load y
    multiply
    push 3
    divide
    add
    store x
    goto L3
L2: push 5
    store y
L3: load x
    load y
    add
    store x
```

load x: pushes the value at the location x to the stack

iflt L1: pops the top two elements and compares them

multiply: pops the top two elements, multiplies them, and pushes the result back to the stack

store x: stores the value at the top of the stack to the location x

Code optimization

- It is a technique which tries to improve code by eliminating unnecessary code line and rearranging in such a sequence that speed up the program execution without wasting.

Its advantage

- Executes faster
- Efficient memory usage
- Yields better performance

Code Optimization

- An intermediate code can be optimized for two main reasons.
  - Optimizations for memory space.
    - How much code does it take to fill up the memory of a modern PC? A lot is the answer
  - Optimizations for speed
    - Speed is affected by Hard disk & File system, Network, Operating system kernel, Languages standard library, Memory, Processor
    - The compiler can affect only the last strongly. Sometimes it can improve memory bandwidth or latency, operating system and language libraries
- The behavior of the program should not be changed
- **Note that:** Optimization can be also done in other phases of compiler design

Cont….

- Intermediate code generation process introduce many inefficiencies

- Extra copy of variable
- Using variable instead of constant
- Repeat evaluation of express
- So code optimization reduce this inefficiencies and improve may be time ,space, and power consumption
- The factors that influencing the code optimization
- Machine
- Architecture of target CPU
- Machine Architecture

Cont…

- Themes behind optimization techniques
- ✓ Avoid redundancy
- ✓ Less code
- ✓ Straight line code, few jump
- ✓ Code locality
- ✓ Any optimization attempt by compiler must follow some conditions
- ✓ Semantic equivalence with the source program be maintained
- ✓ The algorithm should not be modified

Code Optimization…

- A very hard problem + non-decidable, i.e., an optimal program cannot be found in most general case.
- Many complex optimization techniques exist.
  - **Trade of:** Effort of implementing a technique + time taken during compilation vs. optimization achieved.
  - For instance, lexical/semantic/code generation phases require linear time in terms of size of programs, whereas certain optimization techniques may require quadratic or cubic order.
- In many cases simple techniques work well enough
- Code optimization is today the core problem of compiler design

Optimization Techniques

- A vast range of optimizations has been applied and studied. Some optimizations provided by a compiler includes:
  - Dead code elimination
  - Arithmetic simplification
  - Constant folding
  - Common sub-expression elimination
  - Inlining
  - Code Hoisting
  - Loop unrolling
  - Code motion
  - Loop fusion
  - Loop fission
  - Copy propagation
  - Peep-hole Optimization
- Some of these optimizations are done when the program is represented close to its source form, as for example tree, others are done later when it is in a low-level form

Optimization Techniques…

- **Dead code elimination**
  - Programmers occasionally write code that can't be reached by any possible path of execution.
  - rarely affects performance or code size
- **Arithmetic simplification**
  - Give algebraic expressions in their simplest form, but not always, simplification deals with this.
  - E.g. sqrt(exp(alog(x)/y)) = exp(alog(x)/2*y)
- **Constant Unfolding**
  - Find out at compile time that an expression is a constant
    - 2 * PI * x / 4 it will reduce to 1.570796337 * x.

Optimization Techniques…

- **Common sub-expression elimination (CSE)**
  - Programmers often repeat equations or parts of equations, e
  - Example
    - x = sqrt(M) * cos(θ);
    - y = sqrt(M) * sin(θ); // "sqrt(M)" the common subexpression
  - Store the result of the first sqrt(M) and reuse it instead of recalculating it
- **Inlining**
  - Repeatedly inserting the function code instead of calling it, saves the calling overhead and enable further optimizations.
  - Inlining large functions will make the executable too large.

Optimization Techniques…

- **Code hoisting**
  - Moving computations outside loops
  - Saves computing time
  - In the following example (2.0 * PI) is an invariant expression there is no reason to recompute it 100 times.

```
DO I = 1, 100
    ARRAY(I) = 2.0 * PI * I
ENDDO
```

  - By introducing a temporary variable 't' it can be transformed to:

```
t = 2.0 * PI
DO I = 1, 100
    ARRAY(I) = t * I
END DO
```

Optimization Techniques…

- **Loop unrolling**
  - The loop exit checks cost CPU time.
  - Loop unrolling tries to get rid of the checks completely or to reduce the number of checks.
  - If you know a loop is only performed a certain number of times, or if you know the number of times it will be repeated is a multiple of a constant you can unroll this loop.
  - Example:

```
// old loop
for(int i=0; i<3; i++) {
        color_map[n+i] = i;
}
```

```
// unrolled version
    int i = 0;
    colormap[n+i] = i;
    i++;
    colormap[n+i] = i;
    i++;
    colormap[n+i] = i;
```

Optimization Techniques…

- **Loop fusion**
  - Replaces multiple loops with a single one

```
/* Before */
        for (i = 0; i < M; i = i + 1) a[i] = b[i] / c[i];
        for (i = 0; i < M; i = i + 1) d[i] = a[i] + c[i];
/* After */
        for (i = 0; i < M; i = i + 1) {
          a[i] = b[i] / c[i];
         d[i] = a[i] + c[i];
        }
```

loop Fusion

Optimization Techniques…

- **Code Motion**
  - Any code inside a loop that always computes the same value can be moved before the loop.
  - Example:

```
while (i <= limit-2)
do {loop code}
```

where the loop code doesn't change the limit variable. The subtraction, limit-2, will be inside the loop. Code motion would substitute:

```
t = limit-2;
while (i <= t)
do {loop code}
```

Optimization Techniques…

- **Copy propagation**
  - Deals with copies to temporary variables, a = b.
    - Compilers generate lots of copies themselves in intermediate form.
    - Copy propagation is the process of removing them and replacing them with references to the original. It often reveals dead-code.
- **Example**

Before

tmp0 = FP + offset A
temp1 = tmp0

After

tmp1 = FP + offset A

Optimization Techniques…

- **Peep-hole Optimization**
  - Look through small window at assembly code for common cases that can be improved
    1. Redundant load
    2. Redundant push/pop
    3. Replace a Jump to a jump
    4. Remove a Jump to next instruction
    5. Replace a Jump around jump
    6. Remove Useless operations
    7. Reduction in strength
  - Done after code generation - Makes small local changes to assembly

Optimization Techniques…

- ***Redundant Load***

Before

```
store  Rx, M
load   M, Rx
```

After

```
store  Rx, M
```

- ***Redundant Push / Pop***

Before

```
push  Rx
pop   Rx
```

After

… nothing …

- ***Replace a jump to a jump***

Before

```
goto  L1
…
L1:goto    L2
```

After

```
goto L2
L1:goto L2
```

Optimization Techniques…

- ***Remove a Jump to next Instruction***

  Before

  goto L1
  L1:…

  After

  L1:…

- ***Replace a jump around jump***

  Before

  if T0 = 0 goto L1
  else goto L2
  L1:…

  After

  if T0 != 0 goto L2
  L1:…

Optimization Techniques…

- ***Remove useless operations***

Before

add	T0, T0, 0
mul	T0, T0, 1

After

… nothing …

- ***Reduction in Strength***

Before

mul	T0, T0, 2
add	T0, T0, 1

After

shift-left	T0
inc	T0

Optimization Techniques…

- **Example: Optimize the code below**

```
load    Tx, M
add     Tx, 0
store   Tx, M
```

- After One Optimization:

```
load    Tx, M
store   Tx, M
```

- After Another Optimization:

```
load    Tx, M
```