CS 1622: Intermediate Representations & Control Flow

Jonathan Misurda
jmisurda@cs.pitt.edu

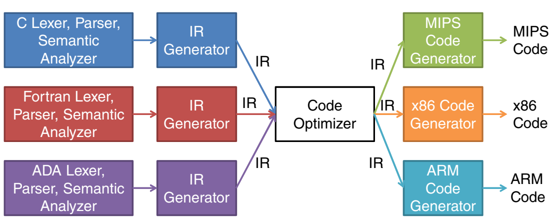
Intermediate Representation

To glue the front end of the compiler with the back end, we may choose to introduce an **Intermediate Representation** that abstracts the details of the AST away and moves us closer to the target code we wish to generate.

Thus, an IR does two things:

1. Abstracts details of the target and source languages
2. Abstracts details of the front and back ends of the compiler

Compiler Organization



Should We Use IR?

At the end of doing our semantic analysis phase, we can choose to omit IR code or not.

Reasons to use IR:

- IR is machine independent, and separates machine dependent/independent parts
- Front-end is retargetable
- Optimizations done at IR level is reusable

Reasons to forgo IR:

- Avoid the overhead of extra code generation passes
- Can exploit the high level hardware features, e.g., MMX

Types of IR

Postfix representation – used in earlier compilers

`a + b * c → c b * a +`

Tree-based IR

- Good for operations that do not alter control flow

Three address code

- Our choice

Static Single Assignment (SSA)

- Assist many code optimization in modern compilers

Three Address Code

Generic form is:

```
X := Y op Z
```

where X, Y, Z can be variables, constants, or compiler-generated temporaries.

Characteristics:

- Similar to assembly code, including statements of control flow
- It is machine independent
- Statements use **symbolic names** rather than **register names**
- Actual locations of labels are not yet determined

Example

An example:

```
x * y + z / w
```

is translated to:

```
t1 := x * y         ; t1, t2, t3 are temporary variables
t2 := z / w
t3 := t1 + t2
```

This yields a sequential representation of an AST.

Three-Address Statements

Assignment statement:

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

where op is an arithmetic or logical operation (binary operation)

Assignment statement:

```
x:= op y
```

where op is an unary operation such as unary minus, not, etc.

Copy statement:

```
x:= y
```

Unconditional jump statement:

```
goto L
```

where L is a label

Three-Address Statements

Conditional jump statement:

```
if (x relop y) goto L
```

where relop is a relational operator such as =, !=, >, <

Procedural call statement:

```
param x1, ..., param xn, call Fy, n
```

As an example, foo(x1, x2, x3) is translated to

```
param x1
param x2
param x3
call foo, 3
```

Procedure call return statement:

```
return y
```

where y is the return value (if applicable)

Three-Address Statements

Indexed assignment statement:

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

or

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

where x is a scalable variable and y is an array variable

Address and pointer operation statement:

```
x := & y
```

a pointer x is set to location of y

```
y := * x
```

y is set to the content of the address stored in pointer x

```
*y := x
```

object pointed to by x gets value y

Implementation

There are three possible ways to store the code:

- Quadruples
- Triples
- Indirect triples (we won't discuss)

Quadruples

Quadruples (4-tuples) store three address code as a set of four items:

op arg1, arg2, result

- There are four fields at maximum
- Arg1 and arg2 are optional
- Arg1, arg2, and result are usually pointers to the symbol table

Examples:

	(op,	arg1,	arg2,	result)
x:= a + b	(+,	a,	b,	x)
x:= - y	(-,	y,	,	x)
goto L	(goto,	,	,	L)

Triples

To avoid putting temporaries into the symbol table, we can refer to temporaries by the positions of the statements that compute them.

Example: a `:= b * (-c) + b * (-c)`

	Quadruples				Triples		
	op	arg1	arg2	result	op	arg1	arg2
(0)	-	c		t1	-	c	
(1)	*	b	t1	t2	*	b	(0)
(2)	-	c		t3	-	c	
(3)	*	b	t3	t4	*	b	(2)
(4)	+	t2	t4	t5	+	(1)	(3)
(5)	:=	t5		a	:=	a	(4)

Triples and Arrays

Triples for array statements have two operations in them:

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

We can translate this into:

```
(0) ( [], x, i )
(1) ( :=, y, (0) )
```

One statement is translated into two triples.

Control Flow

How do we construct the three address code version of loops and if statements?

Consider the code:

```
for(i = 0; i < 10; i++)
       a[i] = i;
```

In three-address code:

```
i := 0
a[i] := i
i := i + 1
if ( i < 10 ) goto ??
```

Control Flow

Symbolic labels:

```
       i := 0
L1:    a[i] := i
       i := i + 1
       if ( i < 10 ) goto L1
```

Numeric labels:

```
100:   i := 0
101:   a[i] := i
102:   i := i + 1
103:   if ( i < 10 ) goto 101
```

We like numeric labels when representing each IR instruction as an object in an array. Each array index is then automatically a label.

IRVisitor

```
class Quadruple {
    String operator;
    String argument1;
    String argument2;
    String result;

    public Quadruple(String op, String arg1, String arg2, String r){
        operator = op;
        argument1 = arg1;
        argument2 = arg2;
        result = r;
    }

    public String toString() {
        return result + " := " + argument1 + " " + operator +
                  " " + argument2;
    }
}
```

IRVisitor

```
public class IRVisitor implements Visitor {
    int temporaryNumber = 0;

    public ArrayList<Quadruple> IR = new ArrayList<Quadruple>();

    public void reset() {
        temporaryNumber = 0;
        IR = new ArrayList<Quadruple>();
    }
```

IRVisitor

```
public int visit(AddNode n) {
    Node lhs = n.children.get(0); Node rhs = n.children.get(1);
    int l = lhs.accept(this); int r = rhs.accept(this);
    String arg1; String arg2;

    if(lhs instanceof IntNode)
        arg1 = ""+l;
    else
        arg1 = "t" + l;

    if(rhs instanceof IntNode)
        arg2 = ""+r;
    else
        arg2 = "t" + r;

    IR.add(new Quadruple("+", arg1, arg2, "t"+(temporaryNumber++)));
    return temporaryNumber-1;
}
```

Calc

```
Visitor IRVisit = new IRVisitor();

System.out.println("Three Address Code:");
root.accept(IRVisit);
System.out.println(((IRVisitor)IRVisit).IR);
 ((IRVisitor)IRVisit).reset();
```

Output

```
$> java Calc test.txt
3 + 4 = 7
Visitor:
3 + 4 = 7
Three Address Code:
[t0 := 3 + 4]
-----------------------------------------
3 * 4 - 2 = 10
Visitor:
3 * 4 - 2 = 10
Three Address Code:
[t0 := 3 * 4, t1 := t0 - 2]
-----------------------------------------
( 3 + 2 ) * -2 = -10
Visitor:
( 3 + 2 ) * -2 = -10
Three Address Code:
[t0 := 3 + 2, t1 := t0 * -2]
-----------------------------------------
```