Intermediate Code Generation

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Why do we need Intermediate Code?

- Two phases compiler
  - Analysis or Front-end
    - Confined to details of source language
    - Lexical, Syntax, Semantic and Intermediate Code Generation
  - Synthesis or Back-end
    - Confined to details of target machine
    - Code Generation
- Assume we have m languages (e.g., C, Java, etc.) and n target machines (e.g., Windows, Linux, iOS, etc)
  - How many compilers do you need for all pair of language-machine?
    - $m \text{ x } n$
  - If intermediate representation is well-defined
    - m front-ends and n back-ends
    - $m + n$

Static Checking

- Type-Checking: Ensures the operators are applied to compatible operands
    - Can be done during parsing (using SDTs)

- Also, includes syntactic checking that remains after parsing
    - A *break* statement should come with-in a loop or a switch-case

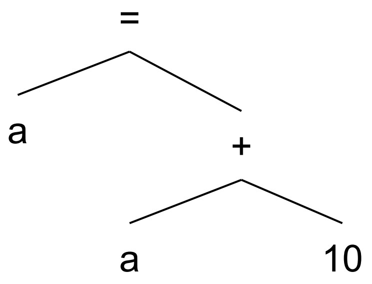
Intermediate Representation

- Two common representations
  - Syntax Tree
  - Three-Address Code (TAC)
    - $z = x \, \langle\text{operator}\rangle \, y$
- There can be a sequence of intermediate representation
  - High-level intermediate representation …………→ Low-level intermediate representation
  - High-level intermediate representation
    - Close to source language
      - Syntax Tree
  - Low-level intermediate representation
    - Close to target machine
      - Instruction selection etc.
  - Three-Address Code ranges from high-level to low-level depending upon choice of operators

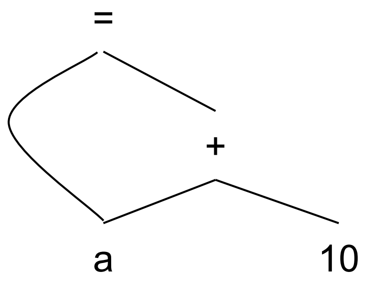
Syntax Trees

- A variant of syntax tree is directed acyclic graph (DAG)
  - Identifies a common subexpression and shares it
  - For example,

    a = a + 10



Syntax Tree



DAG

Three-Address Code

- Each instruction may have at most
  - Three addresses (variables)
  - One operator at right side of an instruction
- For, $a = x + y * z$
  - Three-Address Code: $t = y * z$
    $a = x + t$

    where t is a compiler-generated temporary name
- For, $a + a * (b - c) + (b - c) * d$
  - Three-Address Code:

$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

Three-Address Code

- An address can be one of the following
  - Name, e.g., a, b, sum, avg, etc.
  - Constant, e.g., 10, “c”
  - Compiler-generated temporary name, e.g., t_1, t_2, …, etc.
- Instructions can have symbolic labels
  - These labels alter the flow of control.
  - It represents the index of a three-address instructions

Three-Address Code

- Possible instruction forms
  - $a = x \text{ op } t$
  - $a = \text{op } t$
  - $a = t$
  - Jump
    - Unconditional: `goto` L
    - Conditional: `if` x `goto` L OR `if` $x \text{ relop } y$ `goto` L
  - Procedure calls and return
    - `call` p, n OR $x =$ `call` p, n
  - Indexed copy
    - $a = t[i]$ OR $a[i] = t$
  - Address and pointer assignment
    - $a = \&t$ OR $a = *t$ OR $*a = t$

Three-Address Code: Condition

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

Symbolic labels

	$t_0 = x < y;$
	`IfZ` t_0 `goto` L0;
	$z = x;$
	`goto` L1;
L0:	$z = y;$
L1:	$z = z * z;$

100:	$t_0 = x < y;$
101:	IfZ t_0 goto 104;
102:	$z = x;$
103:	goto 105;
104:	$z = y;$
105:	$z = z * z;$

Positional reference

Three-Address Code: Loop

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

$$
\begin{array}{ll}
\mathrm{L:} & t_1 = i + 1 \\
 & i = t_1 \\
 & t_2 = i * 4 \\
 & t_3 = a + t_2 \\
 & t_4 = *(t_3) \\
 & t_5 = t_4 < v \\
 & \texttt{if } t_5 == 1 \texttt{ goto } \mathrm{L}
\end{array}
$$

Three-Address Code: Arrays

```
arr[1] = arr[0] * 2;
```

$t_0 = 1;$
$t_1 = 4;$
$t_2 = t_1 * t_0;$
$t_3 = \text{arr} + t_2;$
$t_4 = 0;$
$t_5 = 4;$
$t_6 = t_5 * t_4;$
$t_7 = \text{arr} + t_6;$
$t_8 = *(t_7);$
$t_9 = 2;$
$t_{10} = t_8 * t_9;$
$*(t_3) = t_{10};$

Three-Address Code: Functions

- A label identifying the start of the function.
- A BeginFunc *N*; instruction reserving N bytes of space for locals and temporaries.
- The body of the function.
- An EndFunc; instruction marking the end of the function.

```
int foo(int a, int b)
{
    return a + b;
}
void main()
{
    int c, d;
    foo(c, d);
}
```

foo:

    BeginFunc 4;
    $t_0 = a + b$;
    Return t_0;
    EndFunc;

main:

    BeginFunc 12;
    PushParam d;
    PushParam c;
    t_1 = Call foo;
    PopParams 8;
    EndFunc;

Three-Address Code: Objects-1

```
class A
{
    void fn(int x)
    {
        int y;
        y = x;
    }
}

int main()
{
    A a;
    a.fn(137);
}
```

```
A.fn:
    BeginFunc 4;
    y = x;
    EndFunc;
main:
    BeginFunc 12;
    t0 = 137;
    PushParam t0;
    PushParam a;
    call A.fn;
    PopParams 8;
    EndFunc;
```

Three-Address Code: Objects-2

```
class A
{
    int y; int z;
    void fn(int x)
    {
        y = x;
        x = z;
    }
}
int main()
{
    A a;
    a.fn(137);
}
```

```
A.fn:
    BeginFunc 4;
    *(this + 4) = x;
    x = *(this + 8);
    EndFunc;
main:
    BeginFunc 12;
    t0 = 137;
    PushParam t0;
    PushParam a;
    call A.fn;
    PopParams 8;
    EndFunc;
```

Data Structure Representation for the IC

- Quadruple (or Quad)
  - Four fields: *operation*, arg_1, arg_2, and *results*
  - Instruction with unary operators do not use arg_2
  - Jump instructions put the target label in *results*
  - $b * -c + b * -c$

$$
\begin{aligned}
t_1 &= -\ c \\
t_2 &= b * t_1 \\
t_3 &= -\ c \\
t_4 &= b * t_3 \\
t_5 &= t_2 * t_4
\end{aligned}
$$

	Op	A_1	A_2	R
0	-	c		t_1
1	*	b	t_1	t_2
2	-	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5

- Triples
  - Three fields: *operation*, arg_1, and arg_2
  - Results field in Quad has temp variable
  - Points to the position instead of the temp variable

	Op	A_1	A_2
0	-	c	
1	*	b	(0)
2	-	c	
3	*	b	(2)
4	+	(1)	(3)