Intermediate Code Generation - Part 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design
Outline of the Lecture

- Introduction
- Different types of intermediate code
- Intermediate code generation for various constructs
Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP
Why Intermediate Code? - 1

4 Source languages

3 Target machines

Intermediate code optimizer

4 front ends +
1 optimizer +
3 code generators

4 Source languages

3 Target machines

4 front ends +
4x3 optimizers +
4x3 code generators

Y.N. Srikant

Intermediate Code Generation
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, $m \times n$ optimizers, and $m \times n$ 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.
Different Types of Intermediate Code

- 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
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining
Three-Address Code

- Instructions are very simple
- **Examples:** \( a = b + c, x = -y, \text{if } a > b \text{ goto L1} \)
- 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 \( a + b* c - d / (b * c) \) is below

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

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>b</td>
<td>c</td>
<td>t1</td>
</tr>
<tr>
<td>+</td>
<td>a</td>
<td>t1</td>
<td>t2</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>c</td>
<td>t3</td>
</tr>
<tr>
<td>/</td>
<td>d</td>
<td>t3</td>
<td>t4</td>
</tr>
<tr>
<td>-</td>
<td>t2</td>
<td>t4</td>
<td>t5</td>
</tr>
</tbody>
</table>

### Triples

<table>
<thead>
<tr>
<th>op</th>
<th>arg₁</th>
<th>arg₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>+</td>
<td>a</td>
<td>(0)</td>
</tr>
<tr>
<td>*</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>/</td>
<td>d</td>
<td>(2)</td>
</tr>
<tr>
<td>-</td>
<td>(1)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

### Syntax tree

```
- 
  + 
  / 
  a  *  d  * 
  b  c  b  c
```

### DAG

```
- 
  + 
  / 
  a  *  d 
  b  c
```
Instructions in Three-Address Code - 1

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 (~)
   - Conversion operators are useful for converting integers to floating point numbers, etc.

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
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:
- `a = b[i]` (`a` is set to contents(contents(`b`)+contents(`i`)), where `b` is (usually) the base address of an array
- `a[i] = b` (i\textsuperscript{th} location of array `a` is set to `b`)

5 Pointer assignments:
- `a = &b` (`a` is set to the address of `b`, i.e., `a` points to `b`)
- `*a = b` (contents(contents(`a`)) is set to contents(`b`))
- `a = *b` (``a`` is set to contents(contents(`b`)))
C-Program

```c
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

```plaintext
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:
```
C-Program

```c
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

```
dot_prod = 0;
|   b1 = T6
a1 = &a
|   T7 = T3*T5
b1 = &b
|   T8 = dot_prod+T7
i = 0
|   dot_prod = T8
L1: if(i>=10)goto L2
|   T9 = i+1
T3 = *a1
|   i = T9
T4 = a1+1
|   goto L1
a1 = T4
T5 = *b1
T6 = b1+1
L2:
```
C-Program (function)

```c
int dot_prod(int x[], int y[])
{
    int d, i; d = 0;
    for (i=0; i<10; i++) d += x[i]*y[i];
    return d;
}
```

Intermediate code

```
func begin dot_prod | T6 = T4[T5]
d = 0; | T7 = T3*T6
i = 0; | T8 = d+T7
L1: if(i >= 10)goto L2 | d = T8
    T1 = addr(x) | T9 = i+1
    T2 = i*4 | i = T9
    T3 = T1[T2] | goto L1
    T4 = addr(y) | L2: return d
    T5 = i*4 | func end
```
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
C-Program (function)

```c
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      |
```
Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

If (E) S1 else S2
   code for E (result in T)
   if \( T \leq 0 \) 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 \( T \leq 0 \) goto L1 /* if T is false, jump to exit */
   code for S /* all exits from within S also jump to L1 */
L1: /* exit */
Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

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 */
Translations for *If-Then-Else* Statement

Let us see the 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 $E_1$ /* result in T1 */
10 if (T1 \( <= 0 \)), goto L1 (61)
   /* if T1 is false jump to else part */
11 code for $E_2$ /* result in T2 */
35 if (T2 \( <= 0 \)), goto L2 (43)
   /* if T2 is false jump to else part */
36 code for $A_1$
42 goto L3 (82)
43 L2: code for $A_2$
60 goto L3 (82)
61 L1: code for $A_3$
82 L3: code for $A_4$
Translating for \textit{while-do} Statement

\begin{itemize}
\item \textbf{Code fragment:}
\item \texttt{while (E_1) do \{if (E_2) then A_1; else A_2;\} A_3;}
\item \texttt{1 L1: code for E1 /* result in T1 */}
\item \texttt{15 if (T1 $\leq$ 0), goto L2 (79)
/* if T1 is false jump to loop exit */}
\item \texttt{16 code for E2 /* result in T2 */}
\item \texttt{30 if (T2 $\leq$ 0), goto L3 (55)
/* if T2 is false jump to else part */}
\item \texttt{31 code for A1}
\item \texttt{54 goto L1 (1)/* loop back */}
\item \texttt{55 L3: code for A2}
\item \texttt{78 goto L1 (1)/* loop back */}
\item \texttt{79 L2: code for A3}
\end{itemize}
SATG - Attributes

- **S.next, N.next**: list of quads indicating where to jump; target of jump is still undefined
- **IFEXP.falselist**: quad indicating where to jump if the expression is false; target of jump is still undefined
- **E.result**: pointer to symbol table entry
  - All temporaries generated during intermediate code generation are inserted into the symbol table
  - In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
  - However, textual examples will use names
**SATG - Auxiliary functions/variables**

- **nextquad**: global variable containing the number of the next quadruple to be generated
- **backpatch(list, quad_number)**: patches target of all ‘goto’ quads on the ‘list’ to ‘quad_number’
- **merge(list-1, list-2,...,list-n)**: merges all the lists supplied as parameters
- **gen(‘quadruple’)**: generates ‘quadruple’ at position ‘nextquad’ and increments ‘nextquad’
  - In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
  - However, textual examples will use names
- **newtemp(temp-type)**: generates a temporary name of type *temp-type*, inserts it into the symbol table, and returns the pointer to that entry in the symbol table
SATG for *If-Then-Else Statement*

- **IFEXP** $\rightarrow$ *if* *E*
  
  \[
  \begin{array}{l}
  \text{IFEXP.falselist} := \text{makelist}(\text{nextquad}); \\
  \text{gen}('\text{if E.result} \leq 0 \text{ goto } \_\_');
  \end{array}
  \]

- **S** $\rightarrow$ **IFEXP** *S*$_1$; *N* *else* *M* *S*$_2$
  
  \[
  \begin{array}{l}
  \text{backpatch(IFEXP.falselist, M.quad);} \\
  \text{S.next} := \text{merge}(S_1.next, S_2.next, N.next);
  \end{array}
  \]

- **S** $\rightarrow$ **IFEXP** *S*$_1$;
  
  \[
  \begin{array}{l}
  \text{S.next} := \text{merge}(S_1.next, \text{IFEXP.falselist});
  \end{array}
  \]

- **N** $\rightarrow$ $\epsilon$
  
  \[
  \begin{array}{l}
  \text{N.next} := \text{makelist}(\text{nextquad}); \\
  \text{gen}('\text{goto } \_\_');
  \end{array}
  \]

- **M** $\rightarrow$ $\epsilon$
  
  \[
  \begin{array}{l}
  \text{M.quad} := \text{nextquad};
  \end{array}
  \]