Intermediate Code Generation - Part 1

Y.N. Srikant

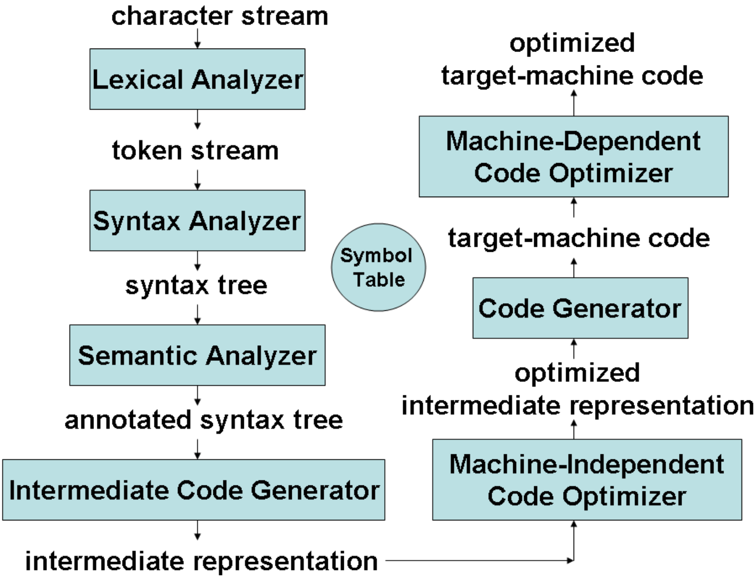
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

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

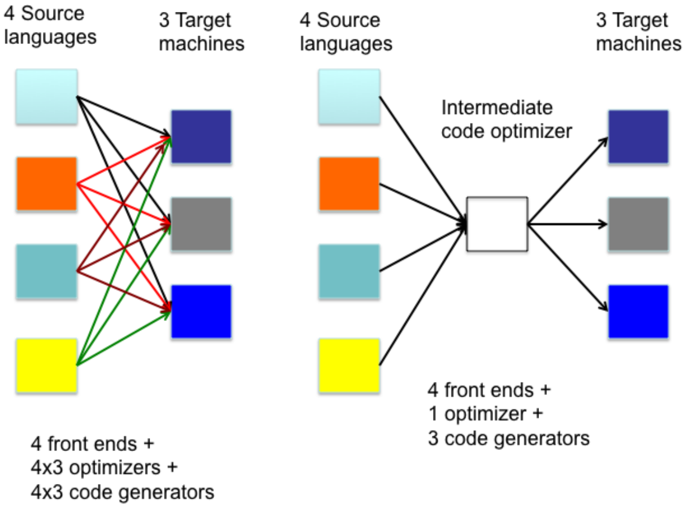
Compiler Overview



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



Why Intermediate Code? - 2

- 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`, `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)` 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

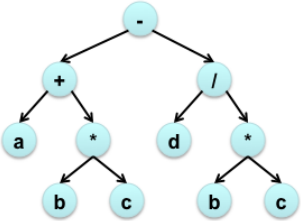
1 t1 = b*c
2 t2 = a+t1
3 t3 = b*c
4 t4 = d/t3
5 t5 = t2-t4

Quadruples

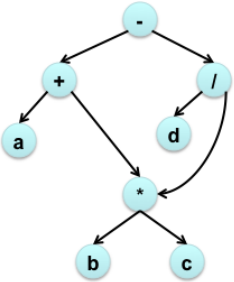
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

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



Syntax tree



DAG

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 ($\sim$)
   - 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

Instructions in Three-Address Code - 2

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^{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

```
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 2

C-Program

```
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                 |L2:
    T5 = *b1
    T6 = b1+1
```

Intermediate Code - Example 3

C-Program (function)

```
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
```

Intermediate Code - Example 3 (contd.)

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 4

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 for *If-Then-Else* Statement

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≤ 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 */
```

Code Template for *While-do* Statement

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 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
```

Translations for *while-do* 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
```

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

- $IFEXP \rightarrow if \;\; E$
  { IFEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __'); }
- $S \rightarrow IFEXP \;\; S_1; \;\; N \;\; else \;\; M \;\; S_2$
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- $S \rightarrow IFEXP \;\; S_1;$
  { S.next := merge(S_1.next, IFEXP.falselist); }
- $N \rightarrow \epsilon$
  { N.next := makelist(nextquad);
  gen('goto __'); }
- $M \rightarrow \epsilon$
  { M.quad := nextquad; }