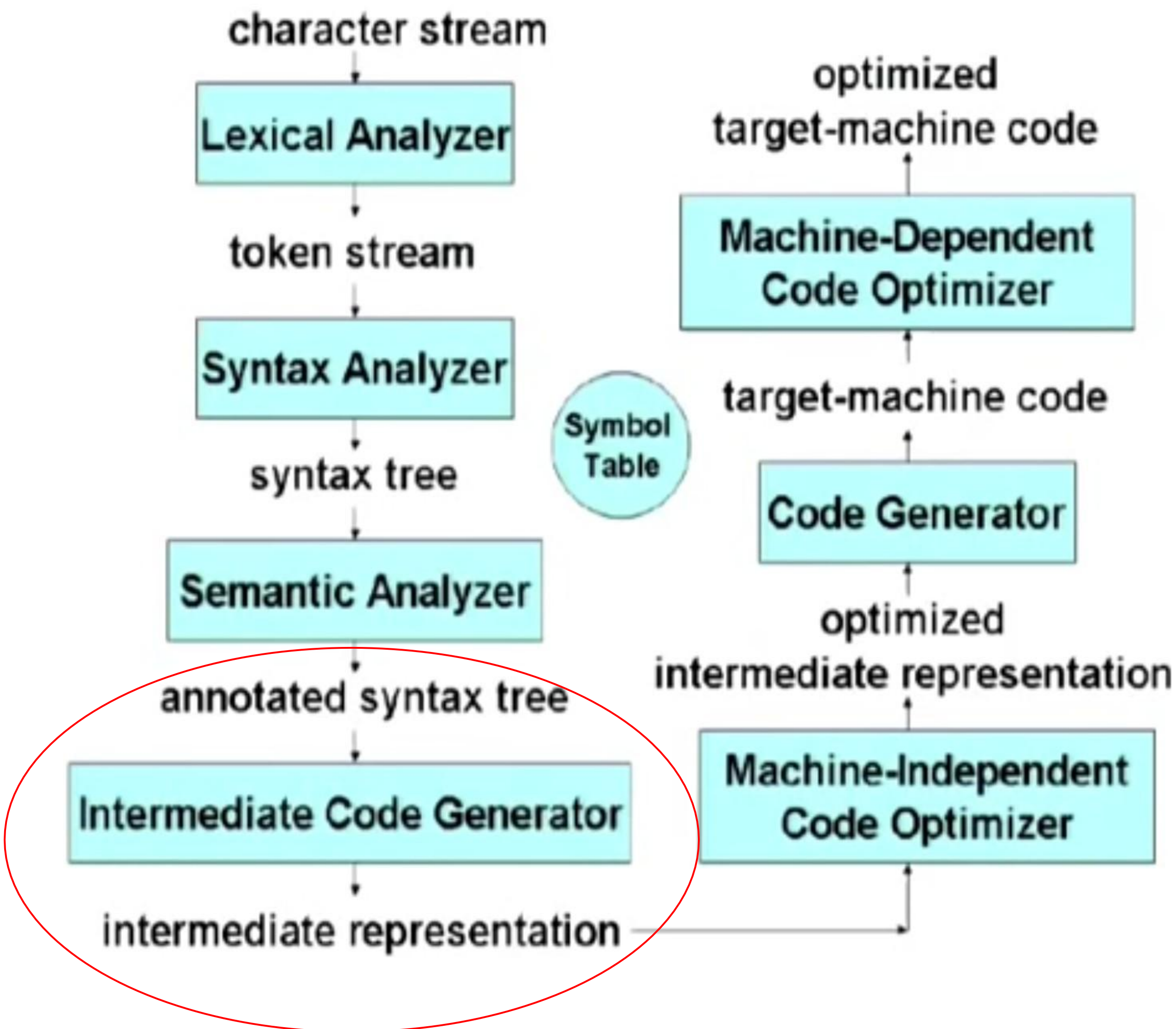
Intermediate Code Generation

CSE 415: Compiler Construction

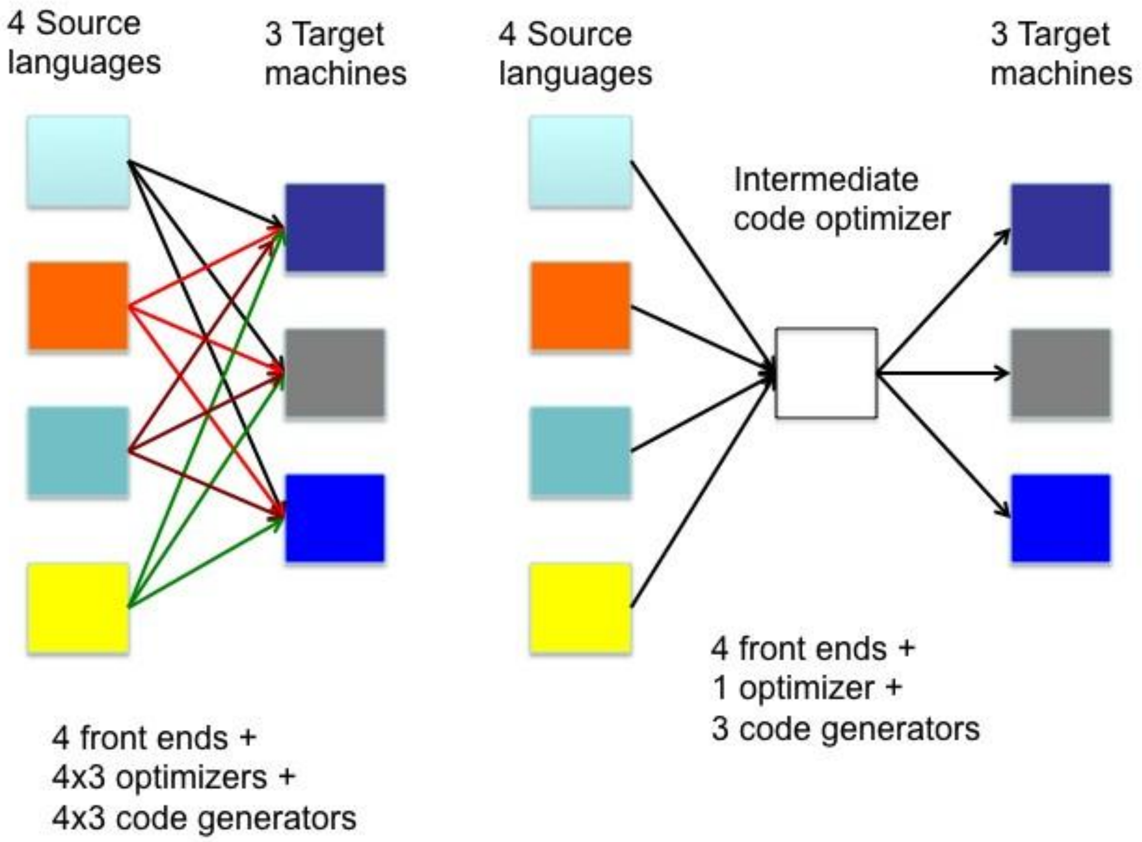
Phases of a Compiler



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 Codes



Why Intermediate Codes (Contd.)

- 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 Codes

- 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

Implementation of Three Address Code

3-address code

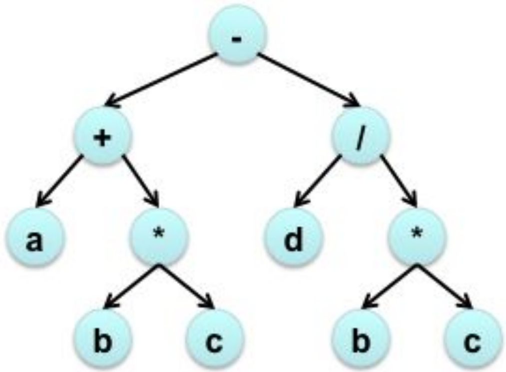
```
1  t1 = b*c
2  t2 = a+t1
3  t3 = b*c
4  t4 = d/t3
5  t5 = t2-t4
```

Quadruples

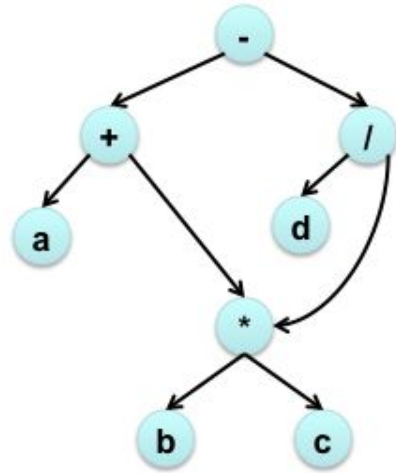
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

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 (~)
   - 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

- *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*)
- *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*)
- *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*)))

Three Address Code – Example 1

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:
```

Three Address 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
```

Three Address 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
```

Three Address Code – Example 4

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
```

Three Address Code –Example 5

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             |
```