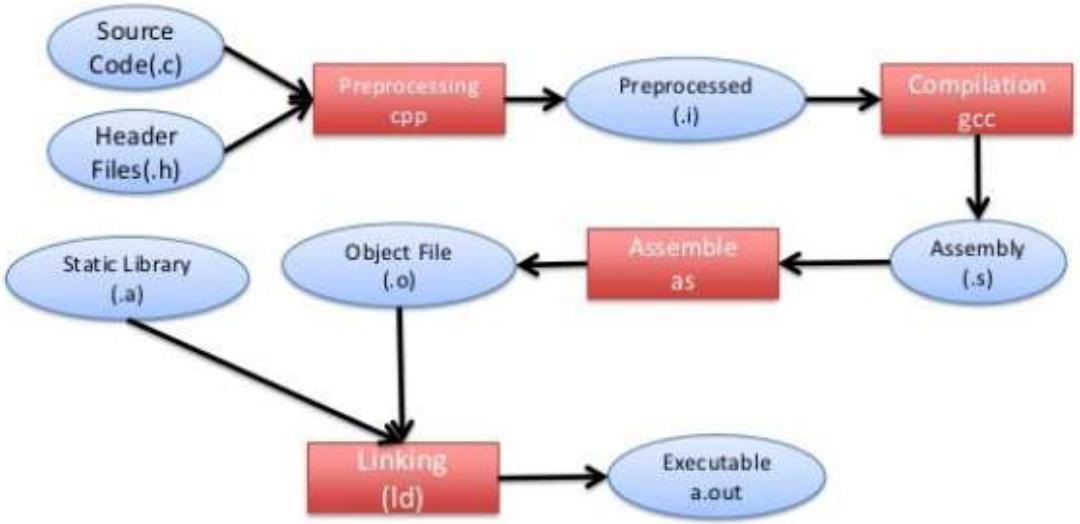
Compilers (CS31003)

Pralay Mitra

Compiling a C program

C Pre-Processor (CPP)
C Compiler
Assembler
Linker



Compilation Flow Diagrams for gcc

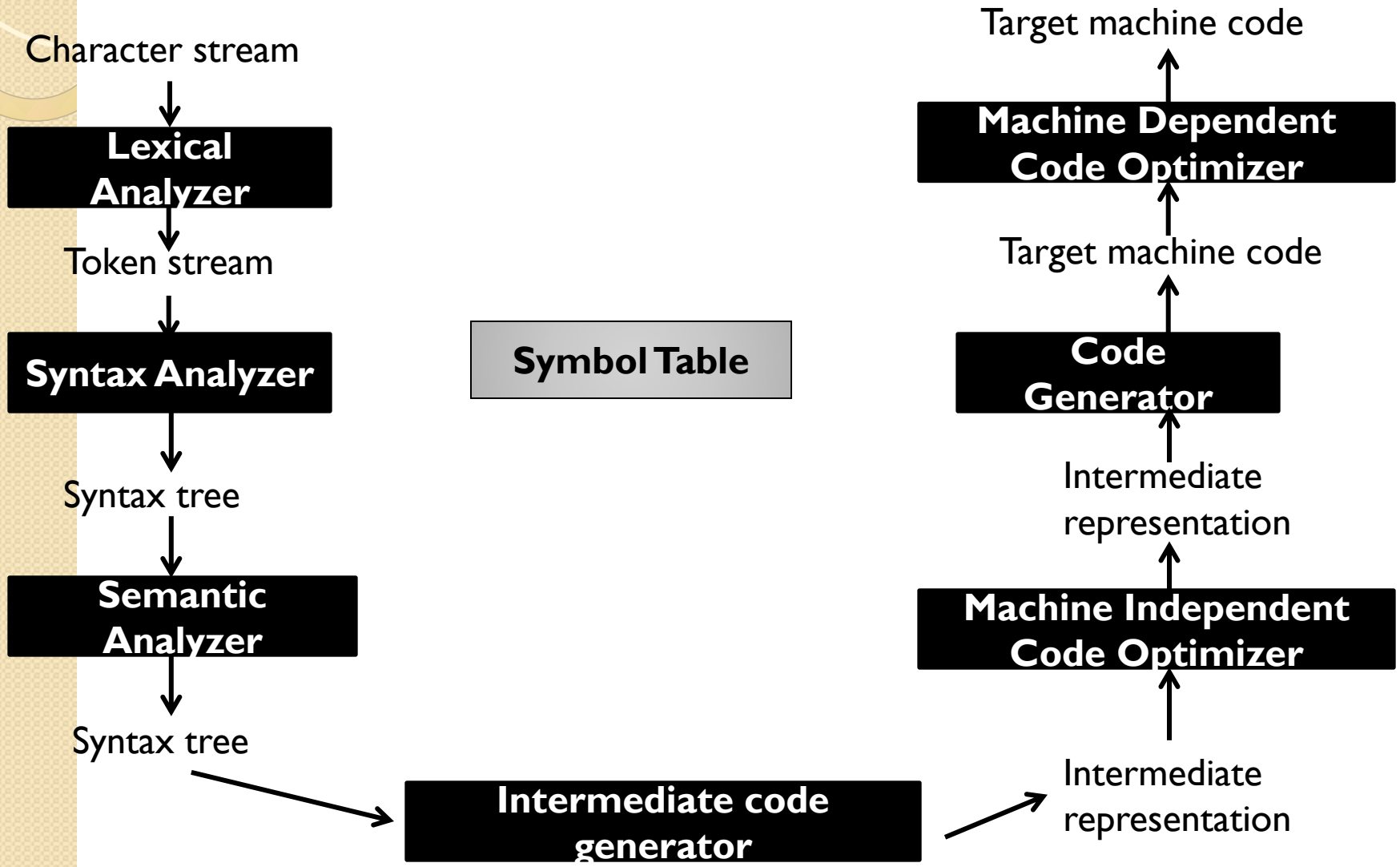
Source: *http://www.slideshare.net/Bletchley131/compilation-and-execution(slide#2)*

Phases of a compiler

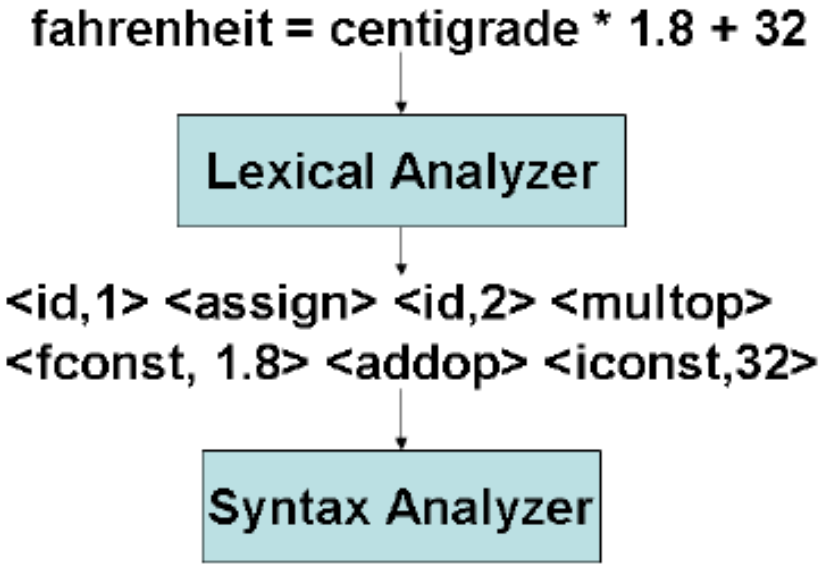
- **Compiler Front-end**
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - Code Optimization

- **Compiler Back-end**
  - Target Code Generation
  - Code Optimization

Phases of a compiler



Lexical Analysis Phase



$$f = c * 1.8 + 32$$
$$b = a * 10 + a$$
$$v = a * t + u$$

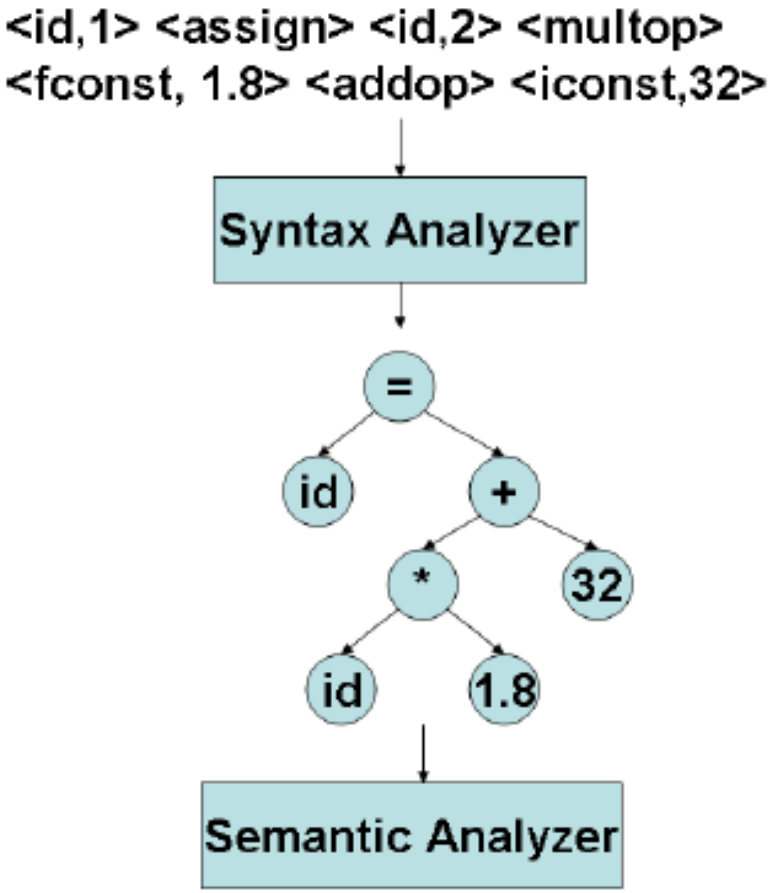
$$id = id * num + num$$
$$id = id * num + id$$
$$id = id * id + id$$

$$fahrenheit = centigrade * 1.8 + 32$$
$$totalAmount = principalAmount * 10 + principalAmount$$
$$finalVelocity = acceleration * time + initialVelocity$$

$$E = E * E + E$$
$$(E = ((E * E) + E))$$

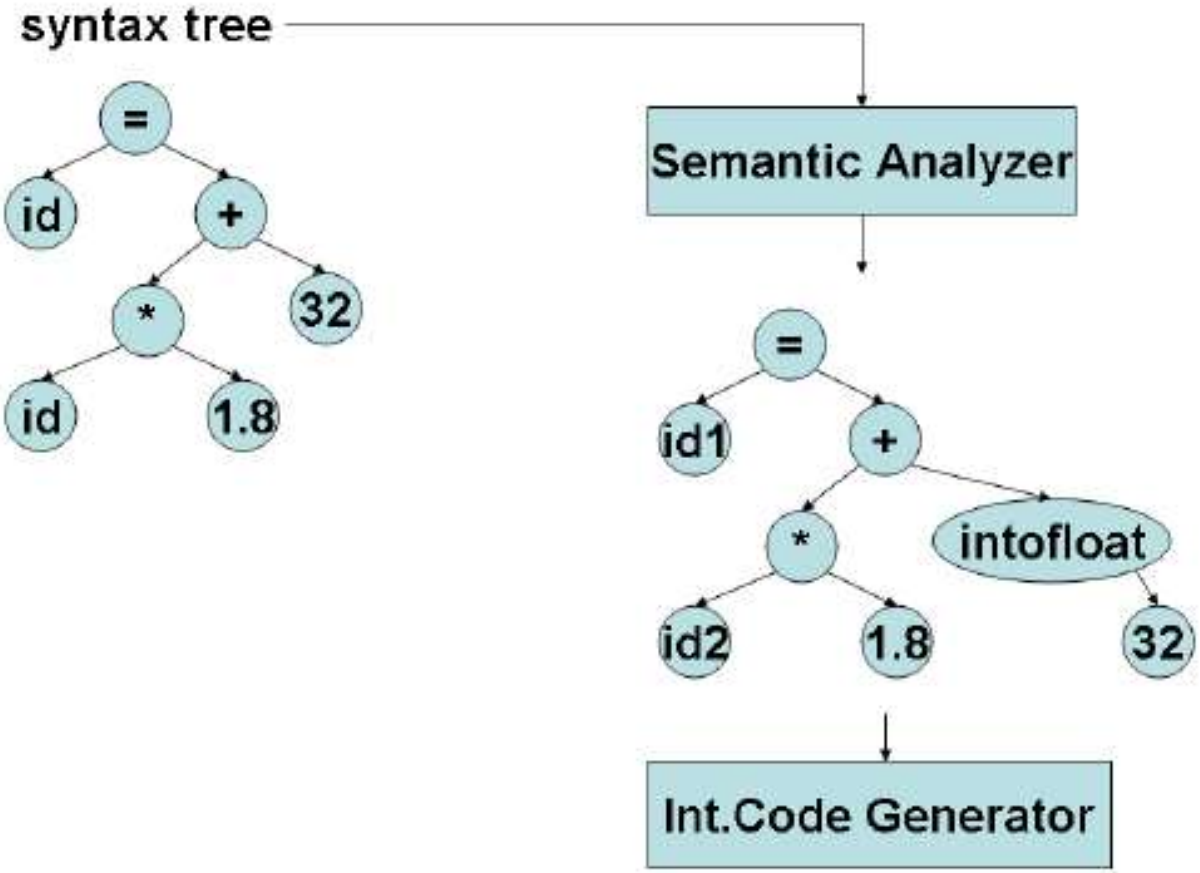
Source: Y N Srikant (NPTEL)

Syntax Analysis Phase



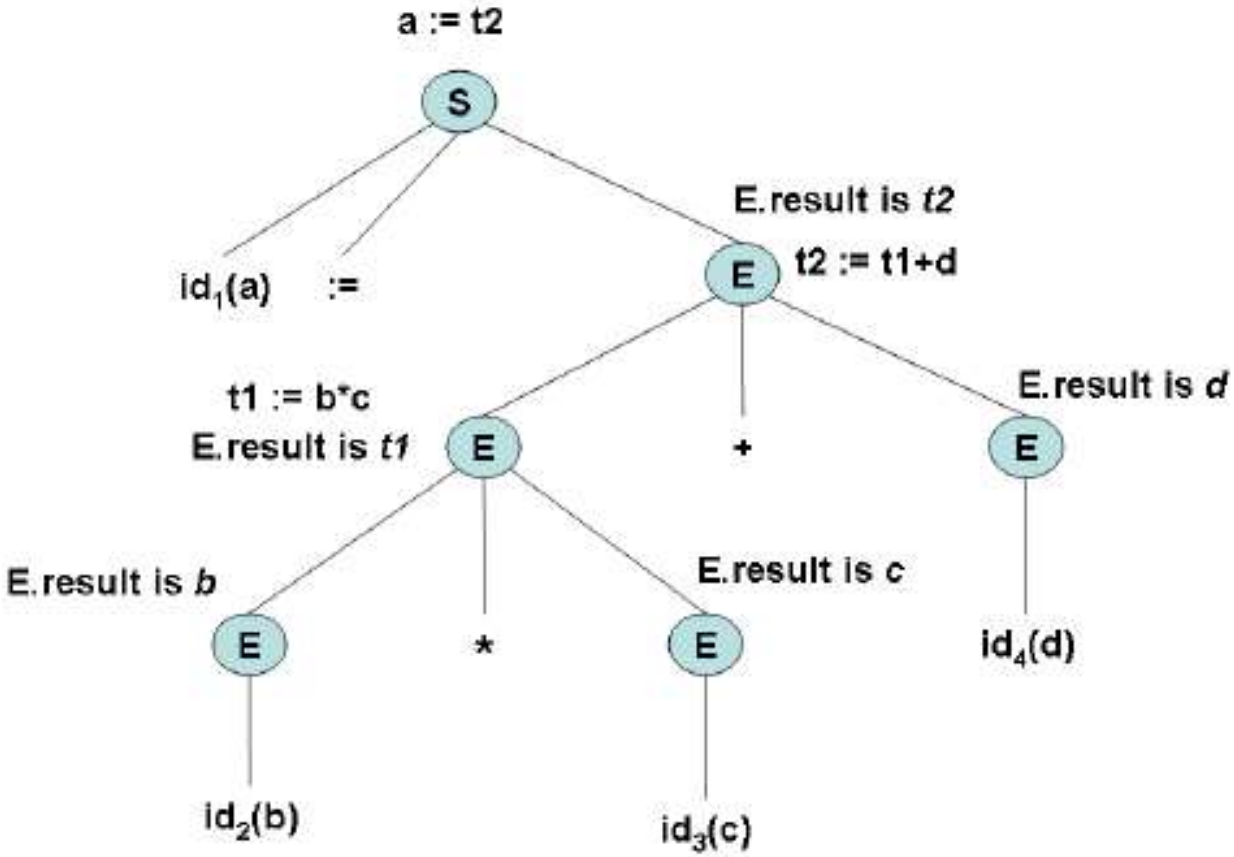
Source: Y N Srikant (NPTEL)

Semantic Analysis Phase



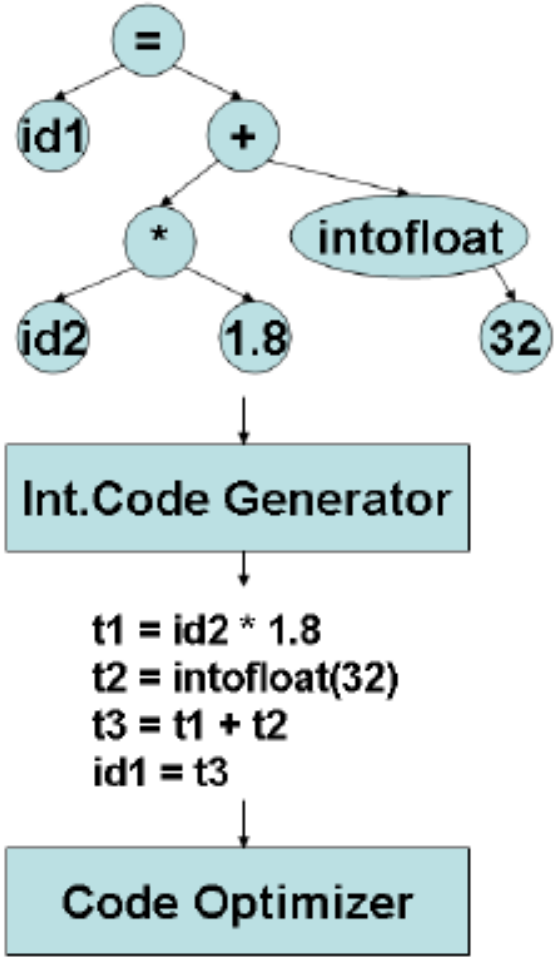
Source: Y N Srikant (NPTEL)

Expression Quads



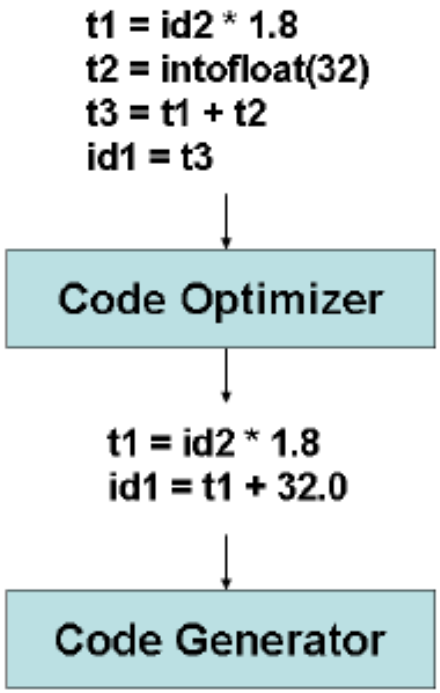
Source: Y N Srikant (NPTEL)

Intermediate Code Generator



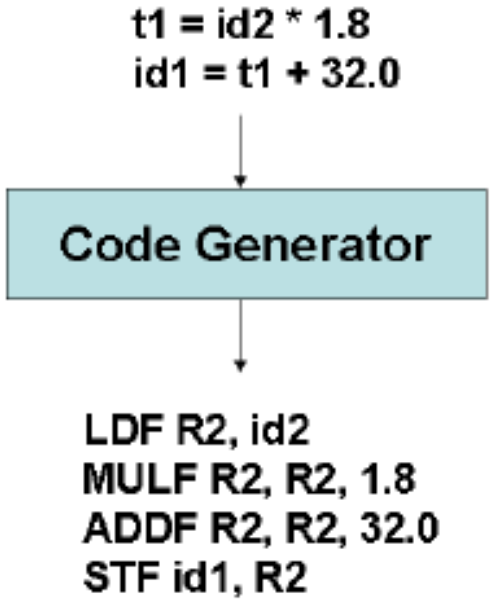
Source: Y N Srikant (NPTEL)

Code Optimization



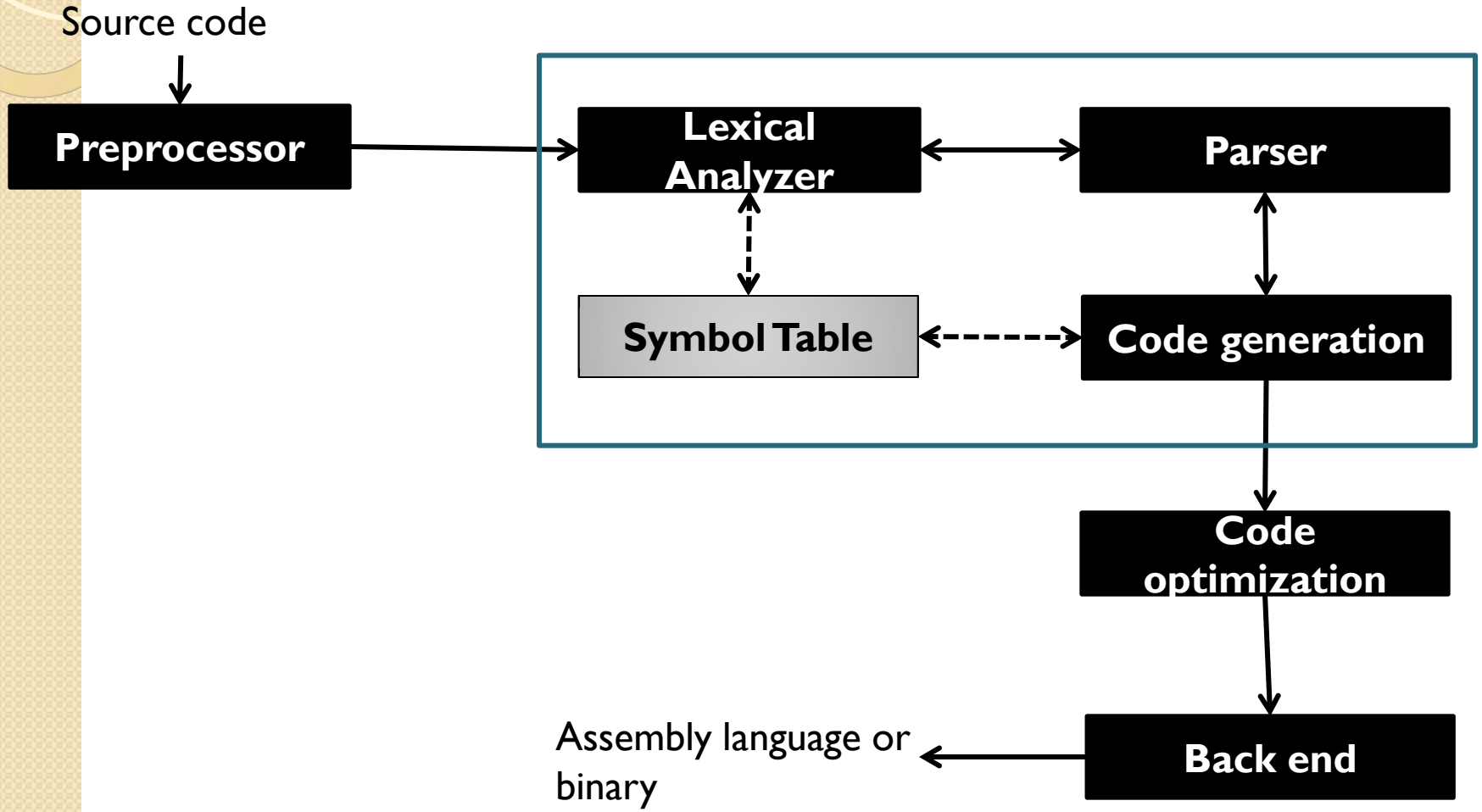
Source: Y N Srikant (NPTEL)

Target Code Generation



Source: Y N Srikant (NPTEL)

Four pass compiler



Mathematical Calculations – A Challenge

The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.

A challenge when evaluating an expression.

Example: **A + B * C**

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → A + (B * C) → A (B * C) + → A B C * +

Infix to Postfix Rules

1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

3. If the incoming symbol is a left parenthesis, push it on the stack.

4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

Infix to Postfix Rules

6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

```
stack s
char  ch, element

while(tokens are available)   {
     ch = read(token);
     if(ch is operand)   {
               print ch ;
     } else  {
               while(priority(ch) <= priority(top most stack))  {
                         element = pop(s);
                         print(element);
               }
               push(s,ch);
       }
}
while(!empty(s))  {
               element = pop(s);
               print(element);
}
```

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Associativity and Precedence

- left associative: + - * / (different precedence)

expression → expression + term | expression – term | term

term → term * factor | term / factor | factor

factor → digit | (expression)

Syntax directed translation

- Postfix notation
  - 1. E=Postfix(E) if E is a variable/constant
  - 2. E_1, E_2, op = Postfix(E_1 op E_2)
  - 3. E_1=Postfix((E_1))

- Attributes and Semantic Rules
  - 9-5+2

Evaluating Postfix Expression

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.
    a) If the element is a number, push it into the stack
    b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

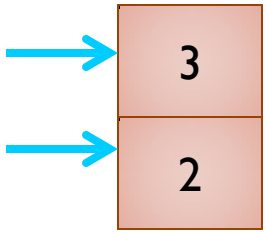
3) When the expression is ended, the number in the stack is the final answer

Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / - **Evaluate Expression**

2 3 * 4 5 / -

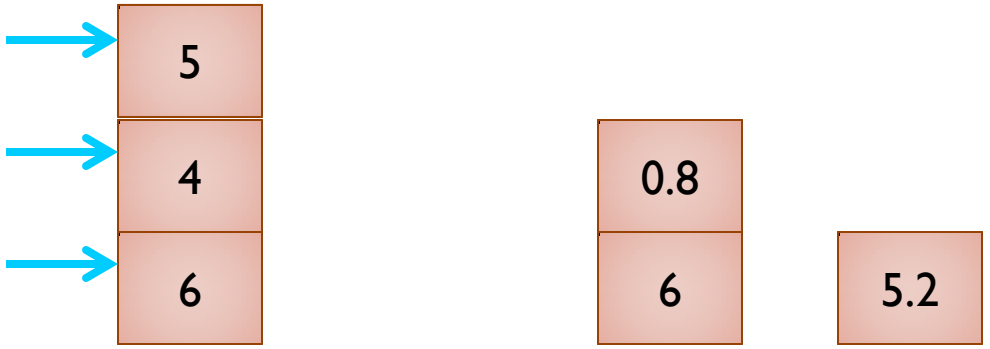


Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / -

Evaluate Expression



Evaluated Expression (Stack top element) = 5.2