Module 01

Pralay Mitra
Partha Pratim
Das

Objectives & Outline

Phases of a Compiler
C Compilation
Front-end
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generator
Code Optimization
Back-end
Code Optimization
Target Code Generation

Sample Translation
Infix ⟶ Postfix

Summary

Module 01: CS31003: Compilers:

Overview: Phases of a Compiler

Pralay Mitra
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

pralay@cse.iitkgp.ac.in
ppd@cse.iitkgp.ac.in

July 16 & 22, 2019

Course Outline

- Outline of Principles
- Outline of Implementation
- Books:
  - Compilers: Principles, Techniques, and Tools (2nd Edition) by A.V. Aho, Monica S Lam, R. Sethi, Jeffrey D. Ullman (Pearson / Addison-Wesley)
  - Flex and Bison by John Levine (O'Reilly)
  - Compiler Design in C by Allen Holub
  - Advanced Compiler Design and Implementation by Steven Muchnick

Module Objectives

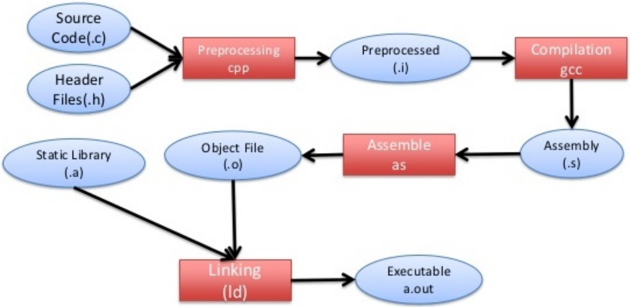
- Understand an outline of the course
- Understand the phases of a compiler

Module Outline

- Course Outline
- Phases of a Compiler
  - C Compilation Process
  - Compiler Front-End
    - Lexical Analysis
    - Syntax Analysis
    - Semantic Analysis
    - Intermediate Code Generator
    - Code Optimization
  - Compiler Back-End
    - Code Optimization
    - Target Code Generation
- Sample Translation
  - Infix → Postfix Translation

Compiling a C Program

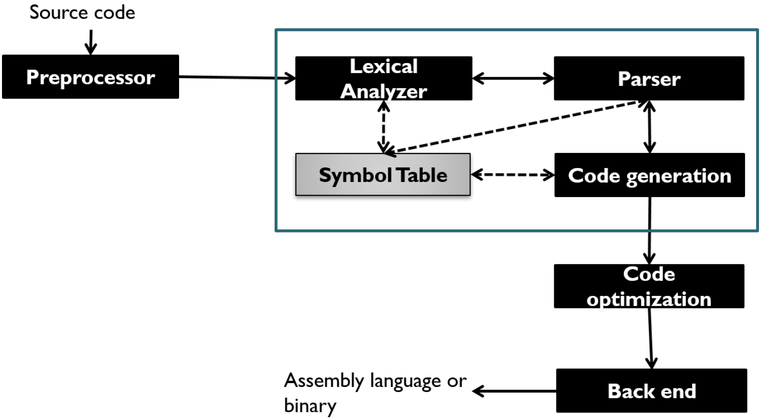
- C Pre-Processor (CPP)
- C Compiler
- Assembler
- Linker



Compilation Flow Diagrams for gcc

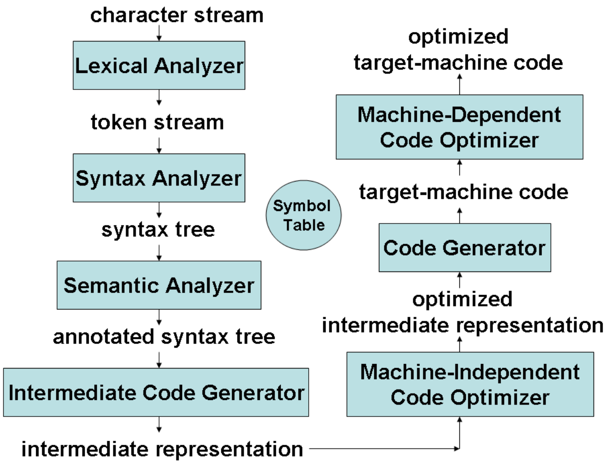
Source: *http://www.slideshare.net/Bletchley131/compilation-and-execution (slide #2)*

Compiling a C Program



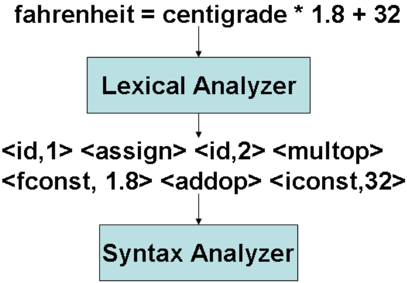
Four Pass Compiler

Phases


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
annotated syntax tree
Intermediate Code Generator
intermediate representation
Symbol Table
Machine-Independent Code Optimizer
optimized intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
optimized target-machine code

Source: Y N Srikant (NPTEL)

Lexical Analysis Phase



fahrenheit = centigrade * 1.8 + 32

Lexical Analyzer

<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>

Syntax Analyzer

$$\begin{aligned} fahrenheit &= centigrade * 1.8 + 32 \\ totalAmount &= principalAmount * 10 + principalAmount \\ finalVelocity &= acceleration * time + initialVelocity \end{aligned}$$

Source: Y N Srikant (NPTEL)

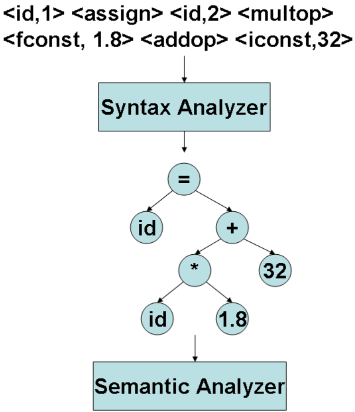
Lexical Analysis Phase

$$\begin{aligned} f &= c * 1.8 + 32 \\ b &= a * 10 + a \\ v &= a * t + u \end{aligned}$$

$$\begin{aligned} id &= id * num + num \\ id &= id * num + id \\ id &= id * id + id \end{aligned}$$

$$\begin{aligned} E &= E * E + E \\ (E &= ((E * E) + E)) \end{aligned}$$

Syntax Analysis Phase



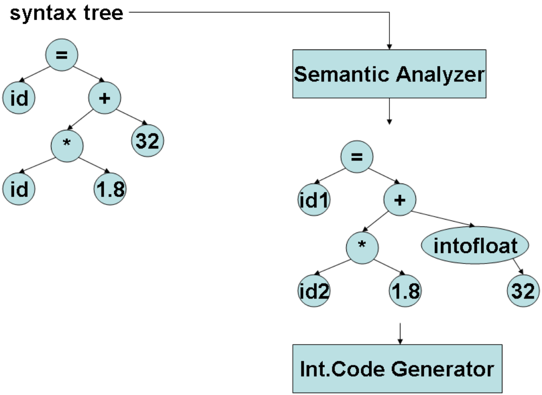
<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>

Syntax Analyzer

Semantic Analyzer

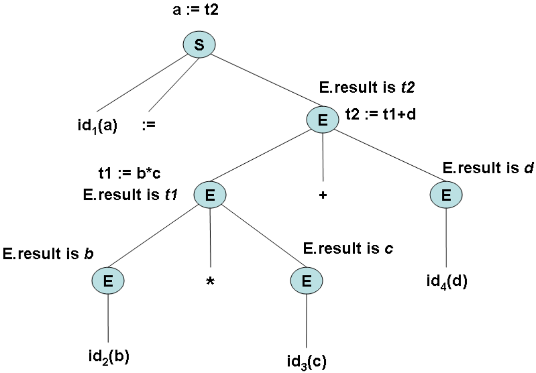
Source: Y N Srikant (NPTEL)

Semantic Analysis Phase



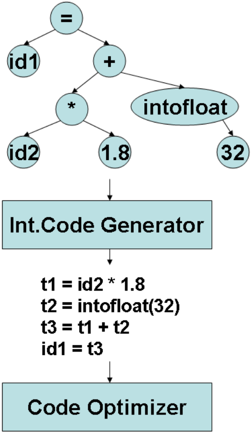
Source: Y N Srikant (NPTEL)

Expression Quads



Source: Y N Srikant (NPTEL)

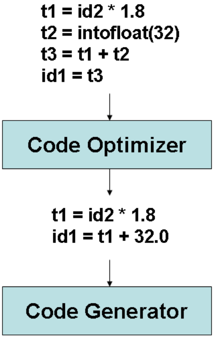
Intermediate Code Generator



t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3

Source: Y N Srikant (NPTEL)

Code Optimization

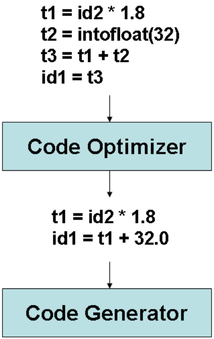


```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

Source: Y N Srikant (NPTEL)

Code Optimization



```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

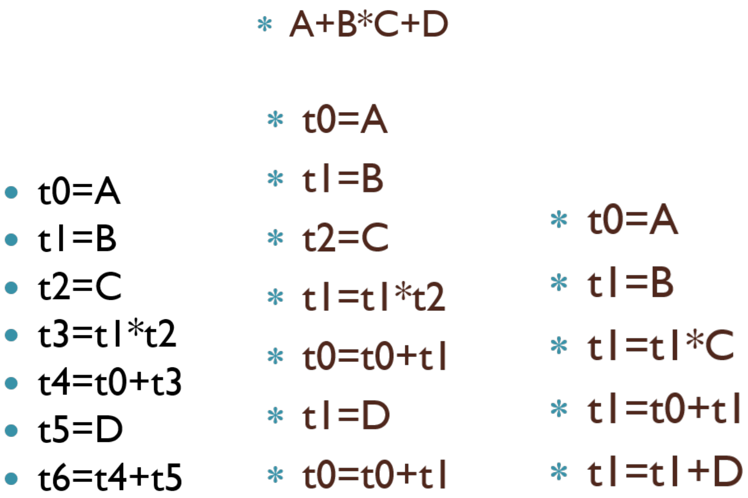
Code Optimizer

```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

Code Generator

Source: Y N Srikant (NPTEL)

Code Generation and Optimization: Practice Example



- A+B*C+D

- t0=A
- t1=B
- t2=C
- t3=t1*t2
- t4=t0+t3
- t5=D
- t6=t4+t5

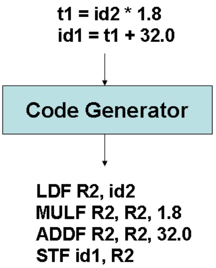
- t0=A
- t1=B
- t2=C
- t1=t1*t2
- t0=t0+t1
- t1=D
- t0=t0+t1

- t0=A
- t1=B
- t1=t1*C
- t1=t0+t1
- t1=t1+D

Target Code Generation

- Data Flow and Control Flow Analysis
- Registration Allocation and Assignment
- Code Generation

Target Code Generation



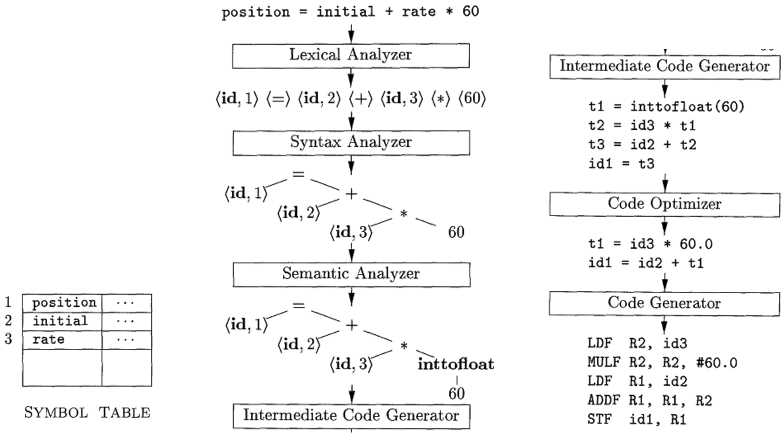
```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

Code Generator

```
LDF R2, id2
MULF R2, R2, 1.8
ADDF R2, R2, 32.0
STF id1, R2
```

Source: Y N Srikant (NPTEL)

Sample pass through Phases



```
position = initial + rate * 60
```

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Source: Dragon Book

Figure: Translation of an assignment statement

Sample Translation

```
{
    int i; int j;
    float a[100]; float v; float x;

    while (true) {
        do i=i+1; while(a[i]<v);
        do j=j-1; while(a[j]>v);
        if (i>=j) break;
        x=a[i]; a[i]=a[j]; a[j]=x;
    }
}
```

```
01: i = i + 1
02: t1 = a [ i ]
03: if t1 < v  goto 01
04: j = j - 1
05: t2 = a [ j ]
06: if t2 > v  goto 04
07: ifFalse i >= j goto 09
08: goto 14
09: x = a [ i ]
10: t3 = a [ j ]
11: a [ i ] = t3
12: a [ j ] = x
13: goto 01
14: .
```

Expression Translation – Resolving Ambiguity

```
9 + 5 * 2 =
    ((9 + 5) * 2) = 28
    (9 + (5 * 2)) = 19

9 - 5 + 2 =
    ((9 - 5) + 2) = 6
    (9 - (5 + 2)) = 2
```

Expression Ambiguity Resolution: Infix → Postfix

$$
\begin{aligned}
9 + 5 * 2 = {} & (9 + (5 * 2)) = 9\ 5\ 2 * + \\
& ((9 + 5) * 2) = 9\ 5 + 2 *
\end{aligned}
$$

$$
\begin{aligned}
9 - 5 + 2 = {} & (9 - (5 + 2)) = 9\ 5\ 2 + - \\
& ((9 - 5) + 2) = 9\ 5 - 2 +
\end{aligned}
$$

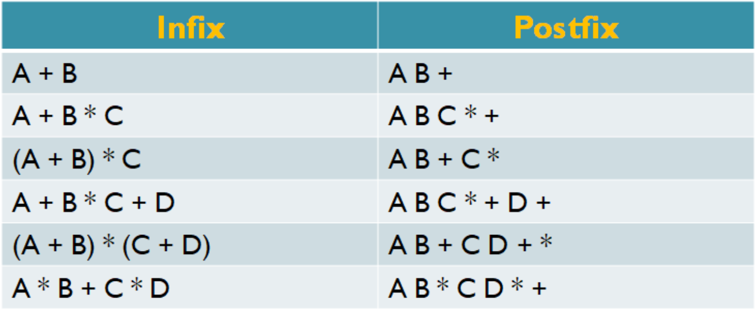
Postfix notation is also called Reverse Polish Notation (RPN)

Associativity and Precedence

Operators

- $*,\ /$ (left)
- $+,\ -$ (left)
- $<,\ \leq,\ >,\ \geq$ (left)
- $!=,\ ==$ (left)
- $=$ (right)

Infix → Postfix: Examples



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 → 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.
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.)

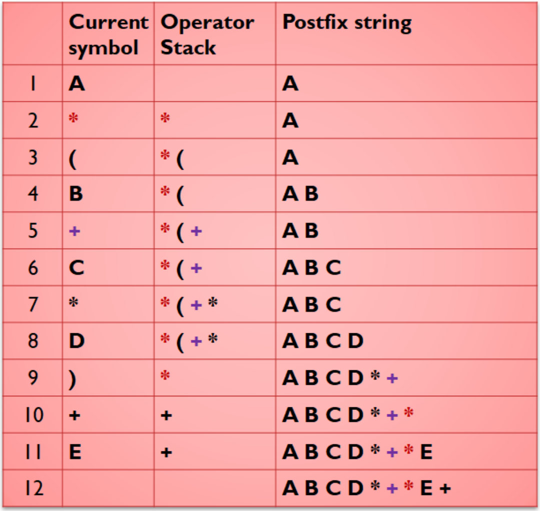
Operator Precedence Table

	Input						
	$	$+$	$-$	$*$	$/$	(	)
$		$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	
$+$	$\gg$	$\gg$	$\gg$	$\ll$	$\ll$	$\ll$	$\gg$
$-$	$\gg$	$\gg$	$\gg$	$\ll$	$\ll$	$\ll$	$\gg$
$*$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\ll$	$\gg$
$/$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\ll$	$\gg$
(	$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	$=$
)							

Infix → Postfix: Rules

- **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 → Postfix: Rules



	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 +

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Evaluating Postfix Expression

- **Create a stack to store operands (or values)**
- **Scan the given expression and do following for every scanned element**
  - If the element is a number, push it into the stack
  - If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
- **When the expression is ended, the number in the stack is the final answer**

A Typical Compiler Techniques

Promote high level languages by minimizing the execution overhead

Support HPC systems

Compiler

Support several source languages

Potential to translate correctly infinite set of programs written in the source language.

Support several target machines

Collection of compilers

Software engineering techniques

Generate optimal target code from source program ??

Languages by Translation Types

Language	Compilation	Typing	Framework
C	Static	Weak[1], Static	No
C++	Static	Strong[2], Static[3]	No[4]
Java	Static	Strong, Static[5]	Yes[6]
Python	Dynamic[7]	Strong, Dynamic	Yes[8]

[1] For example, `void*` breaking typing

[2] If typical C features are not used

[3] Dynamic w/ Polymorphism

[4] RTTI for `dynamic_cast`

[5] Dynamic w/ Polymorphism

[6] Java Virtual Machine – JVM

[7] Interpreter

[8] Python Virtual Machine – PVM

Module Summary

- Outline of Course and Material provided
- Recap on the outline of C Compilation Process
- Brief discussion on Phases of a Compiler to understand
    - Front-end flow: Language to TAC
    - Back-end flow: TAC to Machine
- Infix to Postfix Translation
- Outline of languages with translation types