NPTEL ONLINE CERTIFICATION COURSES

Compiler Design

Introduction

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- What is a Compiler
- Compiler Applications
- Phases of a Compiler
- Challenges in Compiler Design
- Compilation Process – An Example
- Conclusion

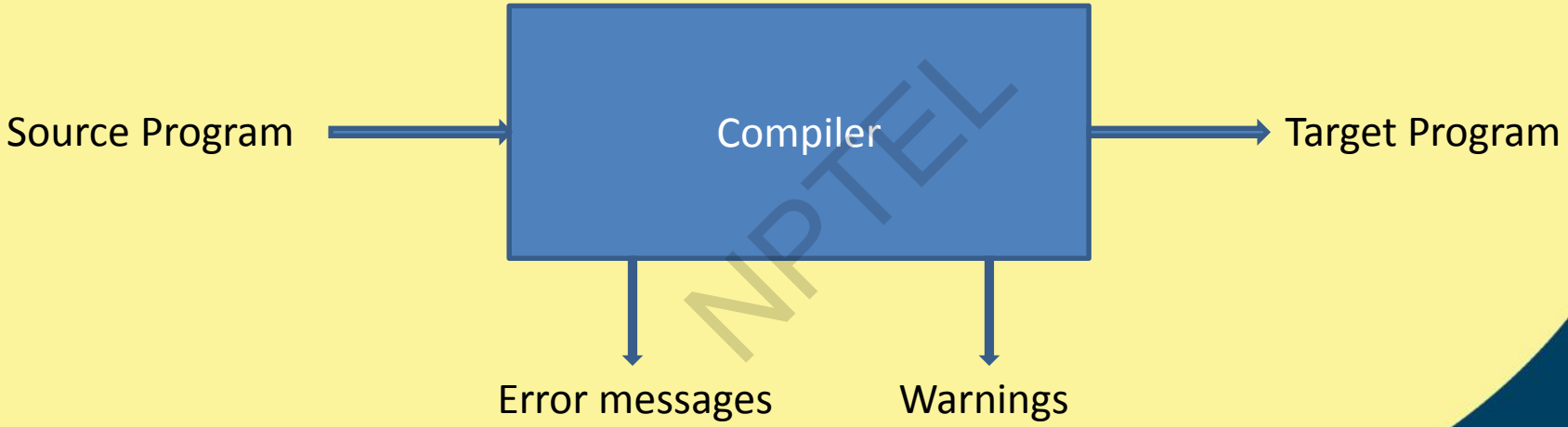
Introduction

- Compilers have become part and parcel of computer systems
- Makes user's computing requirements, specified as a piece of program, understandable to the underlying machine
- Complex transformation
- Large number of options in terms of advances in computer architecture, memory management and newer operating systems

What is a compiler

- A system software to convert source language program to target language program
- Validates input program to the source language specification – produces error messages / warnings
- Primitive systems did not have compilers, programs written assembly language, handcoded into machine code
- Compiler design started with FORTRAN in 1950s
- Many tools have been developed for compiler design automation

Compiler Input-Output



How Many Compilers ??

- Impossible to quantify number of compilers designed so far
- Large number of well known, lesser known and possibly unknown computer languages designed so far
- Equally large number of hardware-software platforms
- Efficient compilers must for survival of programming languages
- Inefficient translators developed for LISP made programs run very slowly
- Advances in memory management policies, particularly garbage collection, has paved the way for faster implementation of such translators, rejuvenating such languages

Compiler Applications

- Machine Code Generation
  - Convert source language program to machine understandable one
  - Takes care of semantics of varied constructs of source language
  - Considers limitations and specific features of target machine
  - Automata theory helps in syntactic checks – valid and invalid programs
  - Compilation also generate code for syntactically correct programs

Compiler Applications (Contd.)

- Format Converters
  - Act as interfaces between two or more software packages
  - Compatibility of input-output formats between tools coming from different vendors
  - Also used to convert heavily used programs written in some older languages (like COBOL) to newer languages (like C/C++)

Compiler Applications (Contd.)

- Silicon Compilation
  - Automatically synthesize a circuit from its behavioural description in languages like VHDL, Verilog etc.
  - Complexity of circuits increasing with reduced time-to-market
  - Optimization criteria for silicon compilation are area, power, delay, etc.

Compiler Applications (Contd.)

- Query Optimization
  - In the domain of database query processing
  - Optimize search time
  - More than one evaluation sequence for each query
  - Cost depends upon relative sizes of tables, availability of indexes
  - Generate proper sequence of operations suitable for fastest query processing

Compiler Applications (Contd.)

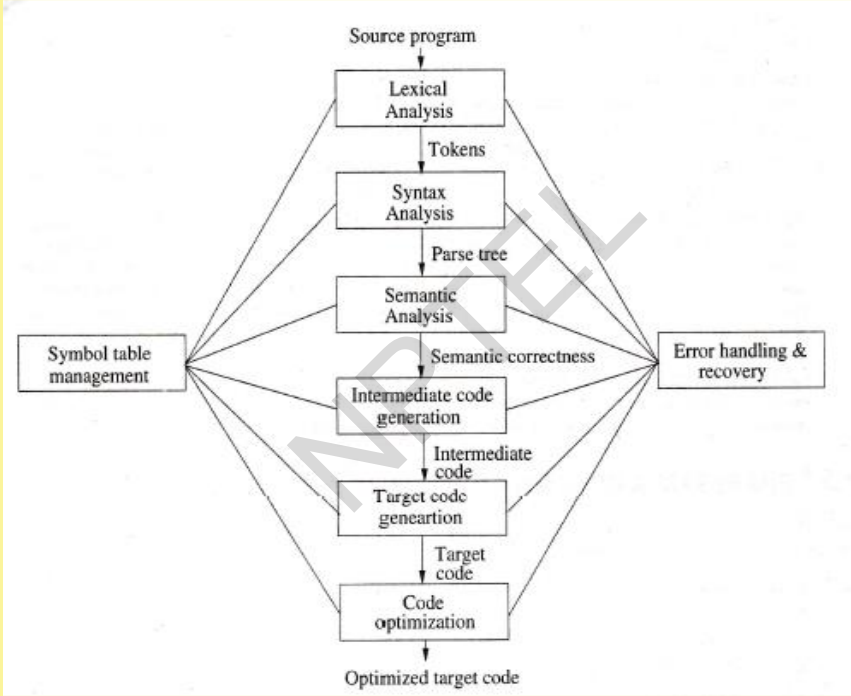
- Text Formatting
  - Accepts an ordinary text file as input having formatting commands embedded
  - Generates formatted text
  - Example *troff*, *nroff*, *LaTex* etc.

Phases of a Compiler

- Conceptually divided into a number of phases
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis
  - Intermediate Code Generation
  - Target Code Generation
  - Code Optimization
  - Symbol Table Management
  - Error Handling and Recovery

Does not exist any hard demarcation between the modules.
Work in hand-in-hand interspersed manner

Phases of a Compiler



Lexical Analysis

- Interface of the compiler to the outside world
- Scans input source program, identifies valid words of the language in it
- Also removes cosmetics, like extra white spaces, comments etc. from the program
- Expands user-defined macros
- Reports presence of foreign words
- May perform case-conversion
- Generates a sequence of integers, called *tokens* to be passed to the syntax analysis phase – later phases need not worry about program text
- Generally implemented as a *finite automata*

Syntax Analysis

- Takes words/tokens from lexical analyzer
- Works hand-in-hand with lexical analyzer
- Checks syntactic (grammatical) correctness
- Identifies sequence of grammar rules to derive the input program from the start symbol
- A *parse tree* is constructed
- Error messages are flashed for syntactically incorrect program

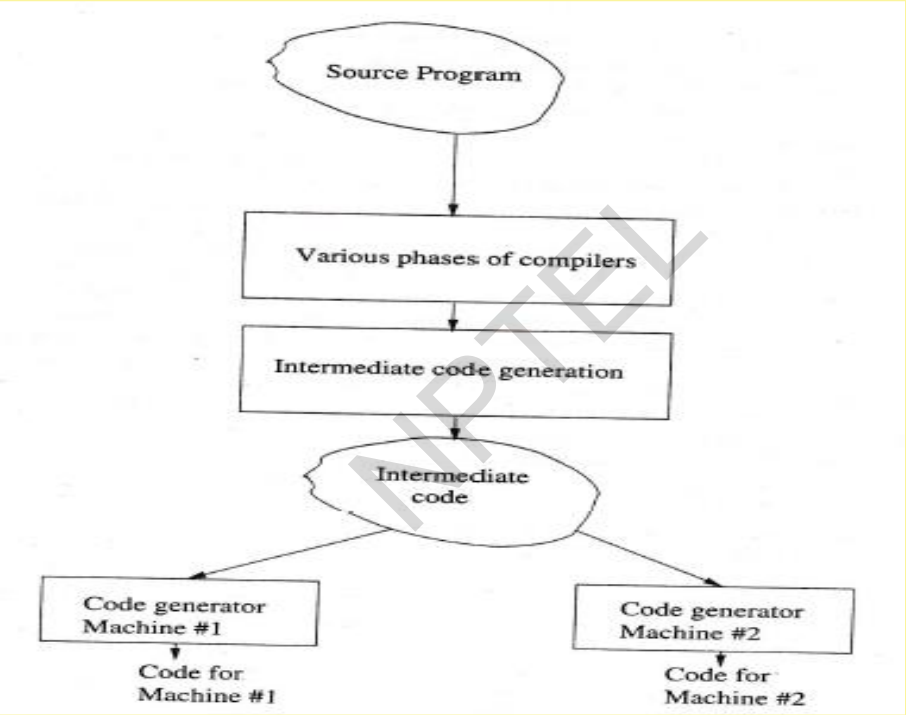
Semantic Analysis

- Semantics of a program is dependent on the language
- A common check is for types of variables and expressions
- Applicability of operators to operands
- *Scope rules* of the language are applied to determine types – *static scope* and *dynamic scope*

Intermediate Code Generation

- Optional towards target code generation
- Code corresponding to input source language program is generated in terms of some hypothetical machine instructions
- Helps to retarget the code from one processor to another
- Simple language supported by most of the contemporary processors
- Powerful enough to express programming language constructs

Targeting Different Machines



Target Code Generation

- Uses template substitution from intermediate code
- Predefined target language templates are used to generate final code
- Machine instructions, addressing modes, CPU registers play vital roles
- Temporary variables (user defined and compiler generated) are packed into CPU registers

Code Optimization

- Most vital step in target code generation
- Automated steps of compilers generate lot of redundant codes that can possibly be eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and single exit
- *Local optimizations* restrict within a single basic block
- *Global optimization* spans across the boundaries of basic blocks
- Most important source of optimization are the *loops*
- Algebraic simplifications, elimination of load-and-store are common optimizations

Symbol Table Management

- Symbol table is a data structure holding information about all symbols defined in the source program
- Not part of the final code, however used as reference by all phases of a compiler
- Typical information stored there include name, type, size, relative offset of variables
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical

Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one

Challenges in Compiler Design

- Language semantics
- Hardware platform
- Operating system and system software
- Error handling
- Aid in debugging
- Optimization
- Runtime environment
- Speed of compilation

Language Semantics

- “case” – fall through or not
- “loop” – index may or may not remember last value
- “break” and “next” modify execution sequence of the program

Hardware Platform

- Code generation strategy for accumulator based machine cannot be similar to a stack based machine
- CISC vs. RISC instructions sets

Operating System and System Software

- Format of file to be executed is depicted by the OS (more specifically, *loader*)
- Linking process can combine object files generated by different compilers into one executable file

Error Handling

- Show appropriate error messages – detailed enough to pinpoint the error, not too verbose to confuse
- A missing semicolon may be reported as “<line no> ; expected” rather than “syntax error”
- If a variable is reported to be undefined at one place, should not be reported again and again
- Compiler designer has to imagine the probable types of mistakes, design suitable detection and recovery mechanism
- Some compilers even go to the extent of modifying source program partially, in order to correct it

Aid in Debugging

- Debugging helps in detecting logical mistakes in a program
- User needs to control execution of machine language program from the source language level
- Compiler has to generate extra information regarding the correspondence between source and machine instructions
- Symbol table also needs to be available to the debugger
- Extra debugging information embedded into the machine code

Optimization

- Needs to identify set of transformations that will be beneficial for most of the programs in a language
- Transformations should be safe
- Trade-off between the time spent to optimize a program vis-a-vis improvement in execution time
- Several levels of optimizations are often used
- Selecting a debugging option may disable any optimization that disturbs the correspondence between the source program and object code

Runtime Environment

- Deals with creating space for parameters and local variables
- For languages like FORTRAN, it is static – fixed memory locations created for them
- Not suitable for languages supporting recursion
- To support recursion, stack frames are used to hold variables and parameters

Speed of Compilation

- An important criteria to judge the acceptability of a compiler to the user community
- Initial phase of program development contains lots of bugs, hence quick compilation may be the objective, rather than optimized code
- Towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code significantly

Compilation – An Example

Consider the following program:

```
program
        var X1, X2: integer; {integer variable declaration}
        var XR1: real; {real variable declaration}
begin
        XR1 := X1 + X2 * 10; {assignment statement}
end.
```

Lexical Analysis

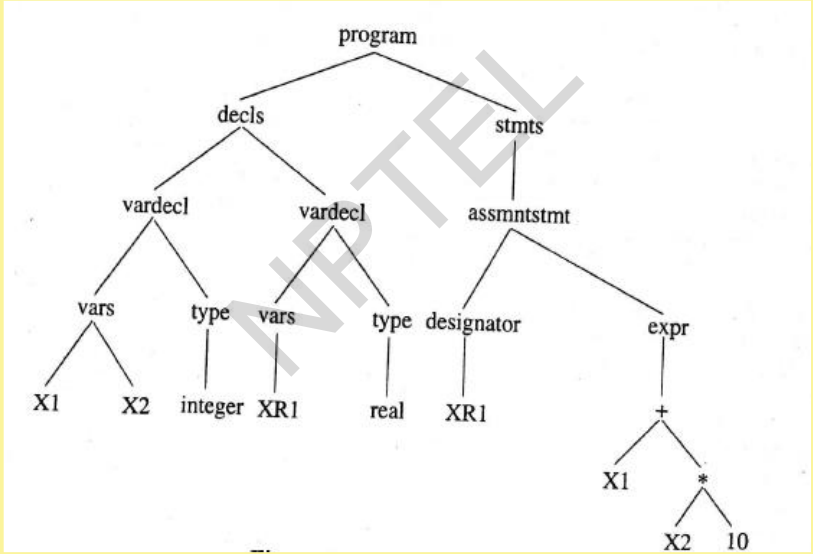
- Produces the following sequence of tokens:

  program, var, X1, ‘ , ’, X2, ‘ : ’, integer, ‘;’, var, XR1, ‘:’, real, ‘;’, begin, XR1, ‘:=’, X1, ‘+’, X2, ‘*’, 10, ‘;’, end, ‘.’

- Whitespaces and comments are discarded and removed by the Lexical Analyzer

Syntax Analysis

- Assuming the program to be grammatically correct, the following parse tree is produced



Code Generation

- Assuming a stack oriented machine with instructions like PUSH, ADD, MULT, STORE, the code looks like

```
PUSH X2
PUSH 10
MULT
PUSH X1
ADD
PUSH @XR1    @ symbol returns the address of a variable
STORE
```

Symbol Table

Name	Class	Type
X1	Variable	Integer
X2	Variable	Integer
XR1	Variable	Real

Conclusion

- Seen an overview of the compiler design process
- Different phases of a compiler have been enumerated
- Challenges faced by the compiler designer have been noted
- An example compilation process has been illustrated

FREE ONLINE EDUCATION
swayam
शिक्षित भारत, उन्नत भारत

NPTEL ONLINE CERTIFICATION COURSES

Thank you!