Compiler Design

IT 423

Lecture - 2
The Phases of a compiler

1. Lexical Analyzer
   - Tokens

2. Syntax Analyzer
   - Parse tree

3. Semantic Analyzer

4. Intermediate Code Generator
   - Abstract parse tree with attributes
   - Non optimized intermediate code

5. Code Optimizer
   - Optimized intermediate code

6. Code Generator
   - Target program

Input: Source Program (High Level Language)
Output: Target program
The Phases of a compiler

- Lexical analysis phase scan the input and separates the characters of source language that are logically one (e.g. int), these group of characters are called tokens. The tokens are keywords, identifiers, constants, operator symbols or punctuation symbols. These tokens are passed to next phase i.e. syntax analyzer.

- Syntax analyzer groups these token into syntactic structures called expressions. These are in the form of a tree called parse tree whose leaves are tokens.

- Semantic analyzer is optional phase of compiler. Here we make syntax tree of tokens with its attributes.
The intermediate code generator uses syntactic structure to create stream if simple instructions. These can be in the form of simple statements. These can be in the form of simple statements or some of the intermediate forms like postfix, triples and quadruples etc. depending upon the languages, its strength and for which machine it will generate the code. It can be mixture of two or more forms also. It is different from machine code, as it does not require registers.

Code optimization is the optional phase of compiler which is executed only if the intermediate code needs to be optimized. Its purpose is to remove redundancies and reduce the time of execution, in order to increase the efficiency of execution of source program.
The Phases of a compiler

- The final phase, code generation produces the object code, assigning memory and allocating registers for computations.

- Table management as a bookkeeping process which keeps tracks of essential information's like name, attributes, data types etc.

- Error handlers acts when some error occurs in program, like syntax error, therefore, proper diagnostics should be called to locate and warn about the error with appropriate message and error point, like line number at which error has occurred.

- Both table management and error handler cooperate and interact with each phase of compilers.
The Phases of a compiler

- Lexical analyzer

- In a compiler, linear analysis is called lexical analyzer or scanning. For ex.

  Lexical analysis of the characters in the assignment statement

  \[\text{position} = \text{initial} + \text{rate} \times 60\]

  will be grouped into the following tokens

  - The identifier 1 ‘position’
  - The assignment symbol ‘=’
  - The identifier 2 ‘identifier’
  - The plus operator ‘+’ sign
  - The identifier 3 ‘rate’
  - The multiplication operator ‘*’ sign
  - The number ‘60’

  The blanks separating the characters of these token would normally be eliminated during lexical analysis.
The Phases of a compiler

- Syntax analyzer

Syntax analysis is also called ‘parsing’. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually the grammatical phrases of the source program are represented by a parse tree.

- Parse tree is a tree whose leaves are labeled with tokens and each of its parent-children portion forms a rule tree that graphically represents a rule.
For expression \[\text{position} = \text{initial} + \text{rate} \times 60\]
The hierarchical structure of a program is usually expressed by recursive rules. For ex. We might have the following rules as part of definition of expression:

- Any identifier is an expression
- Any number is an expression
- If expression1 and expression2 are expression, then expression1+expression2, expression1*expression2 are also expression.

A more common internal representation of this syntactic structure is given by syntax tree:
The Phases of a compiler

- **Semantic analysis**

- The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.
Suppose for ex., that all identifier in parse tree of \( [\text{position} = \text{initial} + \text{rate} \times 60] \), have been declared to be reals and that ‘60’ by itself is assumed to be an integer. Type checking of above figure reveals that ‘\( \times \)’ is applied to a real, rate and an integer 60. The general approach is to convert the integer into a real. This has been achieved in figure.
The Phases of a compiler

- Intermediate code generation
- After syntax and semantic analysis, some compilers generates an explicit intermediate representation of the source program.
- We can think of this intermediate representation as a program for an abstract machine.
- This intermediate representation should have two important properties, it should be easy to produce and easy to translate into the target program.
- The intermediate representation can have a variety of forms. In which one form is called ‘three-address code’.
- Other forms – Triples, Quadruples
Three address code consists of a sequence of instruction, each of which has at most three operands. The source program [position=initial + rate * 60] might appear in three address code

- temp1=int to real (60)
- temp2=id3*temp1
- temp3=id2+temp2
- id1=temp3
The Phases of a compiler

The intermediate form has several properties as

- Each three address instruction has at most one operator in addition to the assignment.
- The compiler must generate a temporary name to hold the value computed by the each instruction.
The Phases of a compiler

- Code optimization

- This is optional phase and attempts to improve the intermediate code. So that faster running machine code will result, some optimization are are trivial.

- For ex., a natural algorithm generates the intermediate code using an instruction for each operator in the three representation after semantic analysis, even through there is better way to perform the same calculation using the two instruction.
  - \( \text{temp1} = \text{id3} \times 60.0 \)
  - \( \text{id1} = \text{id2} + \text{temp1} \)
The Phases of a compiler

- Code generation
- The final phase of the compiler is the generation of target code, consisting normally to relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then intermediate code instruction are translated into a sequence of machine instruction that perform the same task.
The Phases of a compiler

- For ex., using register 1 & 2, the translation of the code of optimized code might become:
  - MOV F id3,R2
  - MUL F #60.0, R2
  - MOV F id2, R1
  - ADD F R2,R1
  - MOV F R1,id1
The Phases of a compiler

- **Symbol table manager**

- Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

- Symbol table management is a mechanism that associates each identifier with relevant information, such as name, type and scope.

- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

- Most of this information is collected during analysis.
The Phases of a compiler

Error Detection

Each phase can encounter errors. However after detecting an error, a phase must some how deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.

A compiler that stop when it find the first error is not as helpful as it could be.

The syntax & semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

the lexical phase can detect errors when the characters remaining in the input do not form any token of the language.
The Phases of a compiler

Slide 20
Front end and Back end in Compiler

- **Front end**
  - The front end consists of those phases or parts of phases, that depend primarily on the source language and largely independent of the target language.
  - These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of the intermediate code.
  - The front end also includes the error handling that goes along with each of these phases.
Back end

The back end includes those portion of the compiler that depend on the target machine and generally these portion do not depend on the source language, just the intermediate language.

In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling, and symbol table operation.
### Front end and Back end in Compiler

<table>
<thead>
<tr>
<th>Front end</th>
<th>Back end</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tasks</strong></td>
<td><strong>Memory allocation.</strong></td>
</tr>
<tr>
<td>• Find validity of a source statement.</td>
<td>• Code generation</td>
</tr>
<tr>
<td>• Analyze the source statement lexically, syntactically, and semantically.</td>
<td></td>
</tr>
<tr>
<td>• Construct suitable representation of source statement</td>
<td></td>
</tr>
<tr>
<td><strong>Phases of compiler</strong></td>
<td><strong>Code generation</strong></td>
</tr>
<tr>
<td>• Lexical analysis</td>
<td>• Code optimization</td>
</tr>
<tr>
<td>• Syntax analysis</td>
<td></td>
</tr>
<tr>
<td>• Semantic analysis</td>
<td></td>
</tr>
<tr>
<td>• Generation of intermediate code</td>
<td></td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td><strong>Error handling</strong></td>
</tr>
<tr>
<td>The intermediate representation produced by front end has two components</td>
<td>• Symbol table management</td>
</tr>
<tr>
<td>(i). Table of information</td>
<td></td>
</tr>
<tr>
<td>(ii). Intermediate Code</td>
<td></td>
</tr>
<tr>
<td>The output comprises of relocatable object code ready for the loader, to</td>
<td></td>
</tr>
<tr>
<td>load and run the program</td>
<td></td>
</tr>
</tbody>
</table>
Phases and Passes of Compiler

- Each individual unique step in compilation process is called a phase such as lexical analysis, syntax analysis and so on.

- Different phases may be combined into one or more than one group, this group of phases makes the concept of passes.

- If all the phases are combined into one group then this is called as one pass compiler otherwise more than one pass constitute the multipass compiler.
### Difference between one-pass and multipass compiler

<table>
<thead>
<tr>
<th>Single-pass (One pass)</th>
<th>Multi-pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>In a single-pass compiler all the phases of compiler design are grouped in one pass.</td>
<td>In a multi-pass compiler the different phases of a compiler design are grouped into multiple phases.</td>
</tr>
<tr>
<td>A one-pass compiler is a compiler that passes through the source code of each compilation unit only once.</td>
<td>A multi-pass compiler is a type of compiler that processes the source code of a program several times.</td>
</tr>
<tr>
<td>A one pass compiler does not “look back” at code previously processed.</td>
<td>Each pass takes the result of the previous pass as the input, and creates an intermediate output</td>
</tr>
</tbody>
</table>