Compiler design syllabus

The aim of the course is learning principles of the Compilers , Techniques and Tools. We introduce the basic concepts and terminology used to Design and implement Compiler.

We will cover the following topics:

- **Lexical analysis** (**Scanner ) Design:** Tokens, Patterns, and Lexemes, Attributes for Tokens, Regular Expressions, Transition Diagrams, The Lookahead Operator ,Conversion of an NFA to a DFA, DFA's for Lexical Analyzers, Minimizing the Number of States of a DFA, scanner-generator tools (lex/Flex).

- **Syntax Analysis Design:** Representative Grammars, Context-Free Grammars, Parse Trees and Derivations, Ambiguity, Eliminating Ambiguity, Elimination of Left Recursion

**Top-Down Parsing**: Recursive-Descent Parsing, FIRST and FOLLOW, LL(1) Grammars.

**Bottom-Up Parsing**: Handle Pruning, Conflicts During Shift-Reduce Parsing

- **Intermediate-Code Generatio:** Variants of Syntax Trees, Three-Address Code, Construction of basic blocks, generation of code from expressions and basic blocks, and register-allocation techniques.

-The technology of code optimization, including flow graphs, data-flow frameworks, and iterative algorithms for solving these frameworks.

Resources

**[1]**  Alfred V. Aho, Monica S. Lam,Ravi Sethi, Jeffrey D. Ullman., **“**Compilers, principles, techniques, and tools “,Addison Wesley, 2007

[2] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, Koen G. Langendoen ,” Modern Compiler Design “ , John Wiley & Sons, 2000.

**Compiler**:

A compiler is a program that translates a high-level language program into a functionally equivalent low-level language program.

**Compilation:**

Assembly level program

High level language

Assembler

Machine Code

Compiler

Assembler

Machine Code

Compiler

Computer

Machine Code

Assembler

Compiler

The compiler takes as input a source program and produces as output an equivalent sequence of machine instructions.

Things to take into account in the compiler design

**Correctness**

--- correct output of the execution

--- report error correctly if the program is not following the language syntax

**Efficiency**

---How efficiency (fast) the compiler is generating the code (Compiler time)

---How efficiency the generated code (run time)

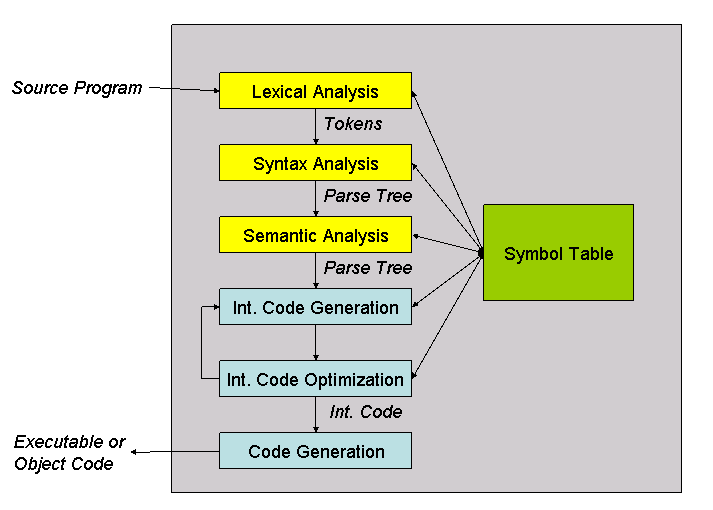
**Debuggability**

Intt a=5;

Undefined the varivle intt with line number and the column number

This process is very complex; hence, from the logical as well as an implementation point of view, it is customary to partition the compilation process into several phases.

A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation .



**The phases of a compiler**

**1-Lexical Analyzer (or Scanner):**

The lexical analysis phase reads the characters in the source program and groups them into streams of tokens; each token represents a logically cohesive sequence of characters, such as identifiers, operators, and keywords. These usual tokens are keywords such as **DO** or **IF**, identifier , such as **X** or **NUM ,** operator symbols such as **<=** or **+** , and punctuation symbols such as parentheses . The character sequence that forms a token is called a "lexeme".

Example: Const pi = 3.1416

The sub string pi is a lexeme for the token “identifier”

The output of the lexical analyzer is a stream of tokens which is passed to the next phase , the syntax analyzer or parser .The lexical analyzer is the interface between the source program and the compiler . What is called tokens depends on the language at the hand and , on some extent on the discretion of the compiler designer . There are two kinds of tokens:

Specific such as IF or a semicolon and , and classes of strings such as identifiers constants labels. We shall treat a token as a pair consisting of two parts : **a token type** and **a token value**. A token consisting of specific string such as semicolon well be treated as having a type but no value . A token such as the identifier **MAX** has a type "identifier" and a value consisting of the string MAX.

The lexical analyzer and the next phase , parser , are often grouped together into one pass. In that pass , the lexical analyzer operates either under the control of the parser or as a co routine with the parser . The parser asks the lexical analyzer for the next token , whenever the parser needs one. The lexical analyzer returns to the parser a code for the token that it found. In the case that the token is an identifier or another token with a value , the value is also passed to the parser .

**Symbol Table**

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various source language constructs. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code. For example , a compiler needs to know whether a variable represents an integer or real number , what size an array has , how many an arguments a function expects , and so forth .

We required several capabilities of the symbol table we need to be able to:

1- Determine if a given name is in the table, the symbol table routines are

concerned with saving and retrieving tokens.

**insert(s,t)** : this function is to add a new name to the table

**Lookup(s)** : returns index of the entry for string s, or 0 if s is not found.

2- Access the information associated with a given name, and add new

information for a given name.

3- Delete a name or group of names from the table.

For example consider tokens **begin ,** we can initialize the symbol-table using the function**: insert("begin",1)**

**2-Syntax Analyzer (or Parser):**

Groups tokens together into syntactic structures. For example the three tokens A+B might be grouped into a syntactic structure called an expression . Expressions might be combined to form statements.

The function of this phase is to check that the tokens appearing in its input , which is the output of lexical analyzer , occur in patterns that are permitted by the specification for the source language. For example

A+/B is not acceptable as an expression .Also this phase specify the parse tree for each grouped of tokens that belong to the same syntactic structure so that determine the order in which the sequence of instructions is executed.

**3-Semantic Analyzer :**

The routines perform semantic error checking:

• Check variables declared before use

• Check variables initialised before reference

• Test type restrictions in expressions and assignments.

• Check number of arguments in function calls,

**4-Intermediate Code Generation:**

Uses the structure produced by the syntactic analyzer to create a stream of simple instructions. The intermediate code can have a variety of forms. For example, a three-address code (TAC) representation for the following expression the intermediate code will be will be:

x-x is 0 ;

x-0 is x

(x \* 2) - x is x

temp1 = sub i32 %x , %x

temp2 = sub i32 %x , 0

temp3 = mul i32 %x , 2

temp4 = sub i32 %temp3 , % x

where temp1,temp2,temp3 and temp4 are compiler-generated temporaries.

**Example:**

For this statement

**while** A>B & A<=2\*B-5 **do**

A:=A+B;

The lexical analyzer may produce the sequence tokens

**While** [id,n1] > [id,n2] & [id,n1]<=[const,n3] \*[id,n2]-[const,n4] **do**

[id,n1]:=[id,n1]+[id,n2];

Where n1,n2,n3,n4 stand for pointers to the symbol table entries for A,B,2,5 respectively .

The second phase ,parser , may produce the following parse tree :

Statement

While-statement

While condition do statement

condition & condition assignment

relation relation location := exp

id(A) exp + exp

id(A) id(B)

exp relop exp exp relop exp

id(A) > id(B) id(A) <= exp - exp

exp \* exp cons(5)

cons(2) id(B)

This parse tree can be translated into intermediate code like the following code :

L1: if A > B goto L2

Goto L3

L2: T1:=2\*B

T2:=T1-5

If A <= T2 goto L4

goto L3

L4: A:=A+B

goto L1

L3:

**5-Code Optimization:**

This phase is designed to improve the intermediate code so that the ultimate object program runs and/or takes less space . Its output is another intermediate code program that does the same job as the original , but perhaps in a way that saves time and/or space .

**Local Optimization:**

**Common sub expression**

local optimization is the elimination of **common sub expression** , for example

A:=B+C+D

E:=B+C+F

might be evaluated as

T1:=B+C

A:=T1+D

E:=T1+F

Taking advantage of the common sub expression .

**Code Motion**

Having found invariant statements within a loop, in this case the statements will be moved outside loop body.

For(i=0;i<m;i++){

x=20;

.

.

}

----------------------------------------

x=20;

For(i=0;i<m;i++){

.

.

}

**6-Code Generation**

This final phase involves selecting memory locations for each variable used by the program. Then, each intermediate instruction is translated into a sequence of machine instructions that performs the same task.

**6-Error Handling**:

One of the most important function of a compiler is the detection and reporting of errors in the source program . Errors can be encountered by all of the phases of a compiler .

**Types of errors**

The syntax and semantic phases usually handle a large fraction of errors

detected by compiler.

1. Lexical error: The lexical phase can detect errors where the characters remaining in the input do not form any token of the language . Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of the source program. Example : If the string fi is encountered in a C program for the first time in context:

fi ( a== f(x)….

A lexical analyzer cannot tell whether **fi** is a misspelling of the keyword **if** or an undeclared function name. since **fi** is a valid identifier, the lexical

analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

2- syntax error: The syntax phase can detect Errors where the token stream violates the structure rules (syntax) of the language.

3- semantic error: During semantic analysis the compiler tries to detect

constructs that have the right syntactic structure but no meaning to the

operation involved, e.g., if we try to add two identifiers, one of which is the name of an array, and the other the name of a procedure.

4-runtime error , a constant that is large to fit in a word of the target machine or division by zero.

Compiler tools:

A number of tools that have been developed specifically to help construct compilers . these tools are range from scanner and parser generators to complex systems, like compiler-compilers , compiler-generators or translator-written systems , which a compiler from some specification of a source language and target machine . The input specification for these systems may contain:

1- A description of the lexical and syntactic structure of the source language

2- A description of what output is to be generated for each source language construct

3-A description of the target machine .