What are Compilers?

Compiler is a program which translates a program written in one language (the source language) to an equivalent program in other language (the target language). Usually the source language is a high level language like Java, C, Fortran etc. whereas the target language is machine code or "code" that a computer's processor understands. The source language is optimized for humans. It is more user-friendly, to some extent platform-independent. They are easier to read, write, and maintain and hence it is easy to avoid errors. Ultimately, programs written in a high-level language must be translated into machine language by a compiler. The target machine language is efficient for hardware but lacks readability.

Compilers

. **Translates from one representation of the program to another**

. **Typically from high level source code to low level machine code or object code**

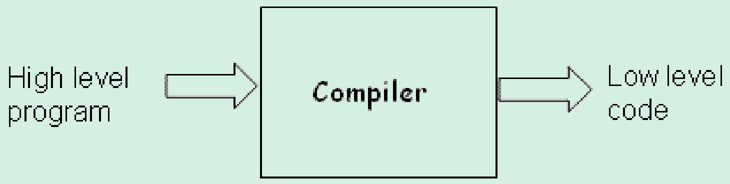
. **Source code is normally optimized for human readability**

- Expressive: matches our notion of languages (and application?!)
- Redundant to help avoid programming errors

. **Machine code is optimized for hardware**

- Redundancy is reduced
- Information about the intent is lost

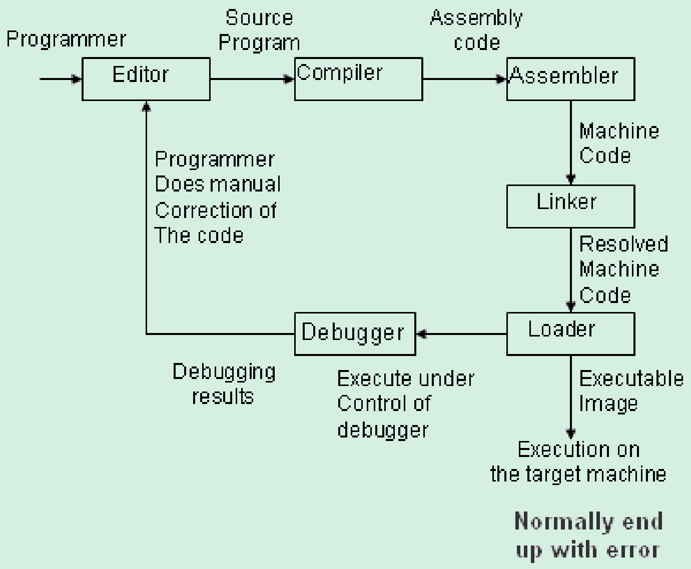
Many modern compilers share a common 'two stage' design. The "front end" translates the source language or the high level program into an intermediate representation. The second stage is the "back end", which works with the internal representation to produce code in the output language which is a low level code. The higher the abstraction a compiler can support, the better it is.



The Big picture

- Compiler is part of program development environment
- The other typical components of this environment are editor, assembler, linker, loader, debugger, profiler etc.
- The compiler (and all other tools) must support each other for easy program development

All development systems are essentially a combination of many tools. For compiler, the other tools are debugger, assembler, linker, loader, profiler, editor etc. If these tools have support for each other than the program development becomes a lot easier.



This is how the various tools work in coordination to make programming easier and better. They all have a specific task to accomplish in the process, from writing a code to compiling it and running/debugging it. If debugged then do manual correction in the code if needed, after getting debugging results. It is the combined contribution of these tools that makes programming a lot easier and efficient.

Lexical Analysis

. Recognizing words is not completely trivial. For example:
ist his ase nte nce?

. Therefore, we must know what the word separators are

. The language must define rules for breaking a sentence into a sequence of words.

. Normally white spaces and punctuations are word separators in languages.

. In programming languages a character from a different class may also be treated as word separator.

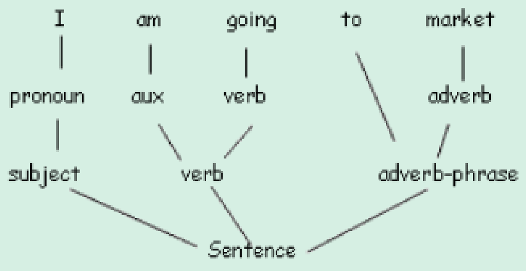
. The lexical analyzer breaks a sentence into a sequence of words or tokens: - If a == b then a = 1 ; else a = 2 ; - Sequence of words (total 14 words) if a == b then a = 1 ; else a = 2 ;

In simple words, lexical analysis is the process of identifying the *words* from an input string of characters, which may be handled more easily by a parser. These words must be separated by some predefined delimiter or there may be some rules imposed by the language for breaking the sentence into tokens or words which are then passed on to the next phase of syntax analysis. In programming languages, a character from a different class may also be considered as a word separator.

The next step

. Once the words are understood, the next step is to understand the structure of the sentence

. The process is known as syntax checking or parsing



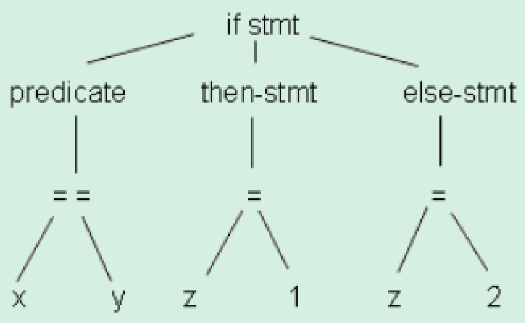
Syntax analysis (also called as parsing) is a process of imposing a hierarchical (tree like) structure on the token stream. It is basically like generating sentences for the language using language specific grammatical rules as we have in our natural language

Ex. sentence ˈsubject + object + subject The example drawn above shows how a sentence in English (a natural language) can be broken down into a tree form depending on the construct of the sentence.

Parsing

Just like a natural language, a programming language also has a set of grammatical rules and hence can be broken down into a parse tree by the parser. It is on this parse tree that the further steps of semantic analysis are carried out. This is also used during generation of the intermediate language code. Yacc (yet another compiler compiler) is a program that generates parsers in the C programming language.

- Parsing a program is exactly the same as shown in previous slide.
- Consider an expression
  if x == y then z = 1 else z = 2



Understanding the meaning

. Once the sentence structure is understood we try to understand the meaning of the sentence (semantic analysis)

. **Example**: Prateek said Nitin left his assignment at home

. **What does his refer to**? Prateek **or** Nitin ?

. **Even worse case**

Amit said Amit left his assignment at home

. **How many** Amits **are there**? **Which one left the assignment**?

Semantic analysis is the process of examining the statements and to make sure that they make sense. During the semantic analysis, the types, values, and other required information about statements are recorded, checked, and transformed appropriately to make sure the program makes sense. Ideally there should be no ambiguity in the grammar of the language. Each sentence should have just one meaning.

Semantic Analysis

. Too hard for compilers. They do not have capabilities similar to human understanding

. However, compilers do perform analysis to understand the meaning and catch inconsistencies

. Programming languages define strict rules to avoid such ambiguities

```
{ int Amit = 3;
        { int Amit = 4;
        cout << Amit;
        }
}
```

Since it is too hard for a compiler to do semantic analysis, the programming languages define strict rules to avoid ambiguities and make the analysis easier. In the code written above, there is a clear demarcation between the two instances of Amit. This has been done by putting one outside the scope of other so that the compiler knows that these two Amit are different by the virtue of their different scopes.

More on Semantic Analysis

. Compilers perform many other checks besides variable bindings

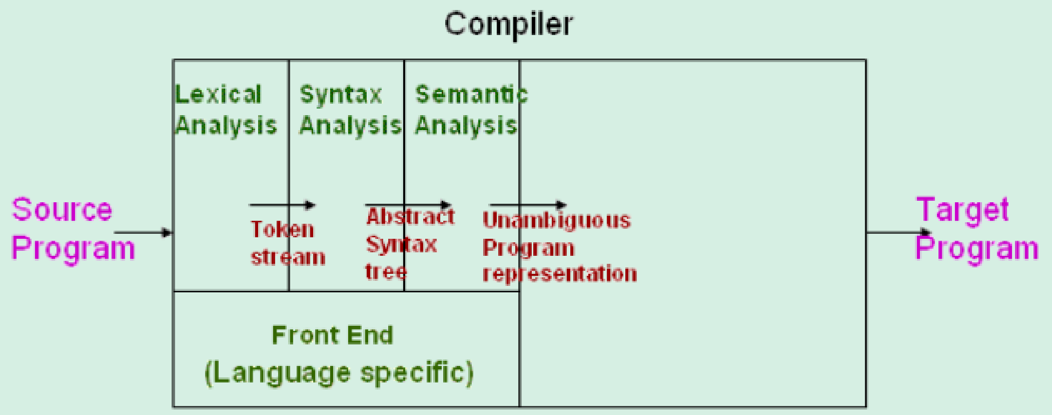
. Type checking Amit left her work at home

. There is a type mismatch between her and Amit . Presumably Amit is a male. And they are not the same person.

From this we can draw an analogy with a programming statement. In the statement:

double y = "Hello World";
The semantic analysis would reveal that "Hello World" is a string, and y is of type double,

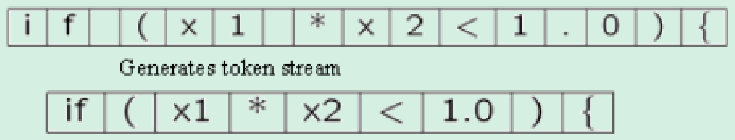
which is a type mismatch and hence, is wrong.



Till now we have conceptualized the front end of the compiler with its 3 phases, viz. Lexical Analysis, Syntax Analysis and Semantic Analysis; and the work done in each of the three phases. Next, we look into the backend in the forthcoming slides.

Front End Phases

- Lexical Analysis
  - Recognize tokens and ignore white spaces, comments

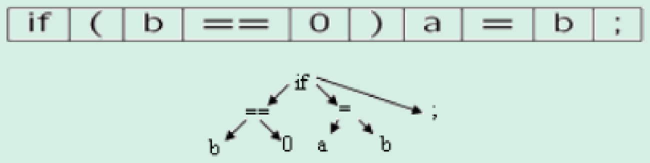


  - Error reporting
  - Model using regular expressions
  - Recognize using Finite State Automata

Lexical analysis is based on the finite state automata and hence finds the lexicons from the input on the basis of corresponding regular expressions. If there is some input which it can't recognize then it generates error. In the above example, the delimiter is a blank space. See for yourself that the lexical analyzer recognizes identifiers, numbers, brackets etc.

Syntax Analysis

. Check syntax and construct abstract syntax tree



. Error reporting and recovery

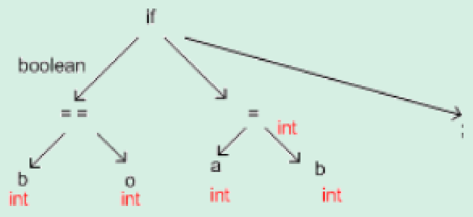
. Model using context free grammars

. Recognize using Push down automata/Table Driven Parsers

Syntax Analysis is modeled on the basis of context free grammars. Programming languages can be written using context free grammars. Based on the rules of the grammar, a syntax tree can be made from a correct code of the language. A code written in a CFG is recognized using Push Down Automata. If there is any error in the syntax of the code then an error is generated by the compiler. Some compilers also tell that what exactly is the error, if possible.

Semantic Analysis

. Check semantics
. Error reporting
. Disambiguate overloaded operators
.Type coercion
. Static checking
- Type checking
- Control flow checking
- Unique ness checking
- Name checks



Semantic analysis should ensure that the code is unambiguous. Also it should do the type checking wherever needed. Ex. int y = "Hi"; should generate an error. Type coercion can be explained by the following example:- int y = 5.6 + 1; The actual value of y used will be 6 since it is an integer. The compiler knows that since y is an instance of an integer it cannot have the value of 6.6 so it down-casts its value to the greatest integer less than 6.6. This is called type coercion.

Code Optimization

. No strong counter part with English, but is similar to editing/précis writing

. Automatically modify programs so that they

- Run faster
- Use less resources (memory, registers, space, fewer fetches etc.)

. Some common optimizations

- Common sub-expression elimination
- Copy propagation
- Dead code elimination
- Code motion
- Strength reduction
- Constant folding

. Example: x = 15 * 3 is transformed to x = 45

There is no strong counterpart in English, this is similar to precise writing where one cuts down the redundant words. It basically cuts down the redundancy. We modify the compiled code to make it more efficient such that it can - Run faster - Use less resources, such as memory, register, space, fewer fetches etc.

Example: see the following code,

```
int x = 2;

int y = 3;

int *array[5];

for (i=0; i<5;i++)

        *array[i] = x + y;
```

Because **x** and **y** are invariant and do not change inside of the loop, their addition doesn't need to be performed for each loop iteration. Almost any good compiler optimizes the code. An optimizer moves the addition of **x** and **y** outside the loop, thus creating a more efficient loop. Thus, the optimized code in this case could look like the following:

```
int x = 5;

int y = 7;

int z = x + y;

int *array[10];

for (i=0; i<5;i++)

        *array[i] = z;
```

Code Generation

. Usually a two step process

- Generate intermediate code from the semantic representation of the program
- Generate machine code from the intermediate code

. The advantage is that each phase is simple

. Requires design of intermediate language

. Most compilers perform translation between successive intermediate representations

. Intermediate languages are generally ordered in decreasing level of abstraction from highest (source) to lowest (machine)

. However, typically the one after the intermediate code generation is the most important

The final phase of the compiler is generation of the relocatable target code. First of all, Intermediate code is generated from the semantic representation of the source program, and this intermediate code is used to generate machine code.

Intermediate Code Generation

. Abstraction at the source level identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)

. Abstraction at the target level memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems

. Code generation is mapping from source level abstractions to target machine abstractions

. Map identifiers to locations (memory/storage allocation)

. Explicate variable accesses (change identifier reference to relocatable/absolute address

. Map source operators to opcodes or a sequence of opcodes

. Convert conditionals and iterations to a test/jump or compare instructions

. Layout parameter passing protocols: locations for parameters, return values, layout of activations frame etc.

. Interface calls to library, runtime system, operating systems

By the very definition of an intermediate language it must be at a level of abstraction which is in the middle of the high level source language and the low level target (machine) language. Design of the intermediate language is important. The IL should satisfy 2 main properties :

. easy to produce, and

. easy to translate into target language.

Post translation Optimizations

. **Algebraic transformations and re-ordering**

- **Remove/simplify operations like**

  . Multiplication by 1

  . Multiplication by 0

  . Addition with 0

- **Reorder instructions based on**

  . Commutative properties of operators

  . For example x+y is same as y+x (always?)

Instruction selection

- **Addressing mode selection**
- **Opcode selection**
- **Peephole optimization**

Some of the different optimization methods are :

1) Constant Folding - replacing y= 5+7 with y=12 or y=x*0 with y=0

2) Dead Code Elimination - e.g.,

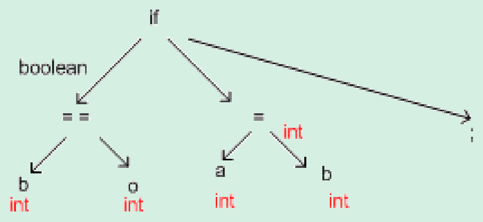
```
If (false)
        a = 1;
else
        a = 2;
with a = 2;
```

3) Peephole Optimization - a machine-dependent optimization that makes a pass through low-level assembly-like instruction sequences of the program(called a peephole), and replacing them with a faster (usually shorter) sequences by removing redundant register loads and stores if possible.

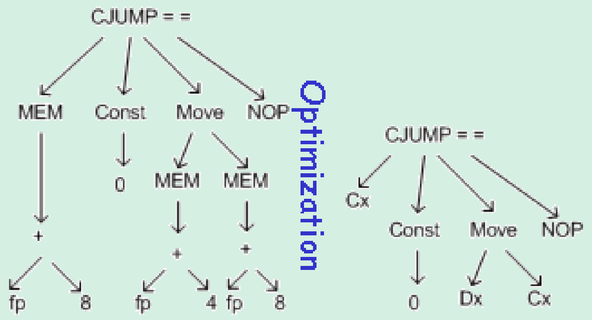
4) Flow of Control Optimizations

5) Strength Reduction - replacing more expensive expressions with cheaper ones - like pow(x,2) with x*x

6) Common Sub expression elimination - like a = b*c, f= b*c*d with temp = b*c, a= temp, f= temp*d;



Intermediate code generation

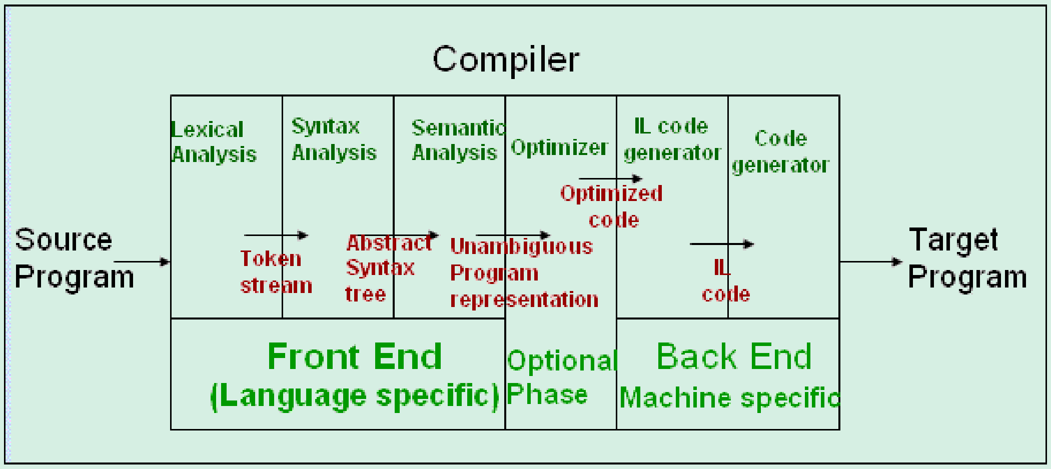


Code Generation

CMP Cx, 0 CMOVZ Dx,Cx

There is a clear intermediate code optimization - with 2 different sets of codes having 2 different parse trees.The optimized code does away with the redundancy in the original code and produces the same result.

Compiler structure



These are the various stages in the process of generation of the target code

from the source code by the compiler. These stages can be broadly classified into

. the Front End (Language specific), and

. the Back End (Machine specific)parts of compilation.

. **Information required about the program variables during compilation**

- Class of variable: keyword, identifier etc.
- Type of variable: integer, float, array, function etc.
- Amount of storage required
- Address in the memory
- Scope information

. **Location to store this information**

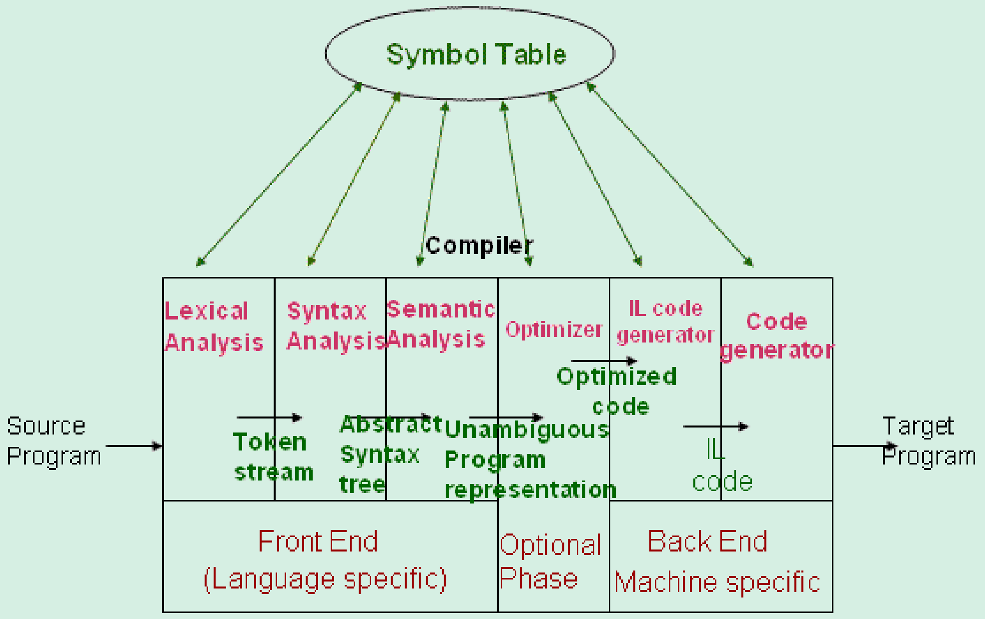
- Attributes with the variable (has obvious problems)
- At a central repository and every phase refers to the repository whenever information is required

. **Normally the second approach is preferred**

- Use a data structure called symbol table

For the lexicons, additional information with its name may be needed. Information about whether it is a keyword/identifier, its data type, value, scope, etc might be needed to be known during the latter phases of compilation. However, all this information is not available in a straight away. This information has to be found and stored somewhere. We store it in a data structure called Symbol Table. Thus each phase of the compiler can access data from the symbol table & write data to it. The method of retrieval of data is that with each lexicon a symbol table entry is associated. A pointer to this symbol in the table can be used to retrieve more information about the lexicon

Final Compiler structure



Advantages of the model

. Also known as Analysis-Synthesis model of compilation

- Front end phases are known as analysis phases

- Back end phases are known as synthesis phases

. Each phase has a well defined work

. Each phase handles a logical activity in the process of compilation

The Analysis-Synthesis model:

The front end phases are Lexical, Syntax and Semantic analyses. These form the "analysis phase" as you can well see these all do some kind of analysis. The Back End phases are called the "synthesis phase" as they synthesize the intermediate and the target language and hence the program from the representation created by the Front End phases. The advantages are that not only can lots of code be reused, but also since the compiler is well structured - it is easy to maintain & debug.

Advantages of the model .

. Compiler is retargetable

. Source and machine independent code optimization is possible.

. Optimization phase can be inserted after the front and back end phases have been developed and deployed

. Also known as Analysis-Synthesis model of compilation

Also since each phase handles a logically different phase of working of a compiler parts of the code can be reused to make new compilers. E.g., in a C compiler for Intel & Athlon the front ends will be similar. For a same language, lexical, syntax and semantic analyses are similar, code can be reused. Also in adding optimization, improving the performance of one phase should not affect the same of the other phase; this is possible to achieve in this model.

Issues in Compiler Design

. Compilation appears to be very simple, but there are many pitfalls

. How are erroneous programs handled?

. Design of programming languages has a big impact on the complexity of the compiler

. M*N vs. M+N problem

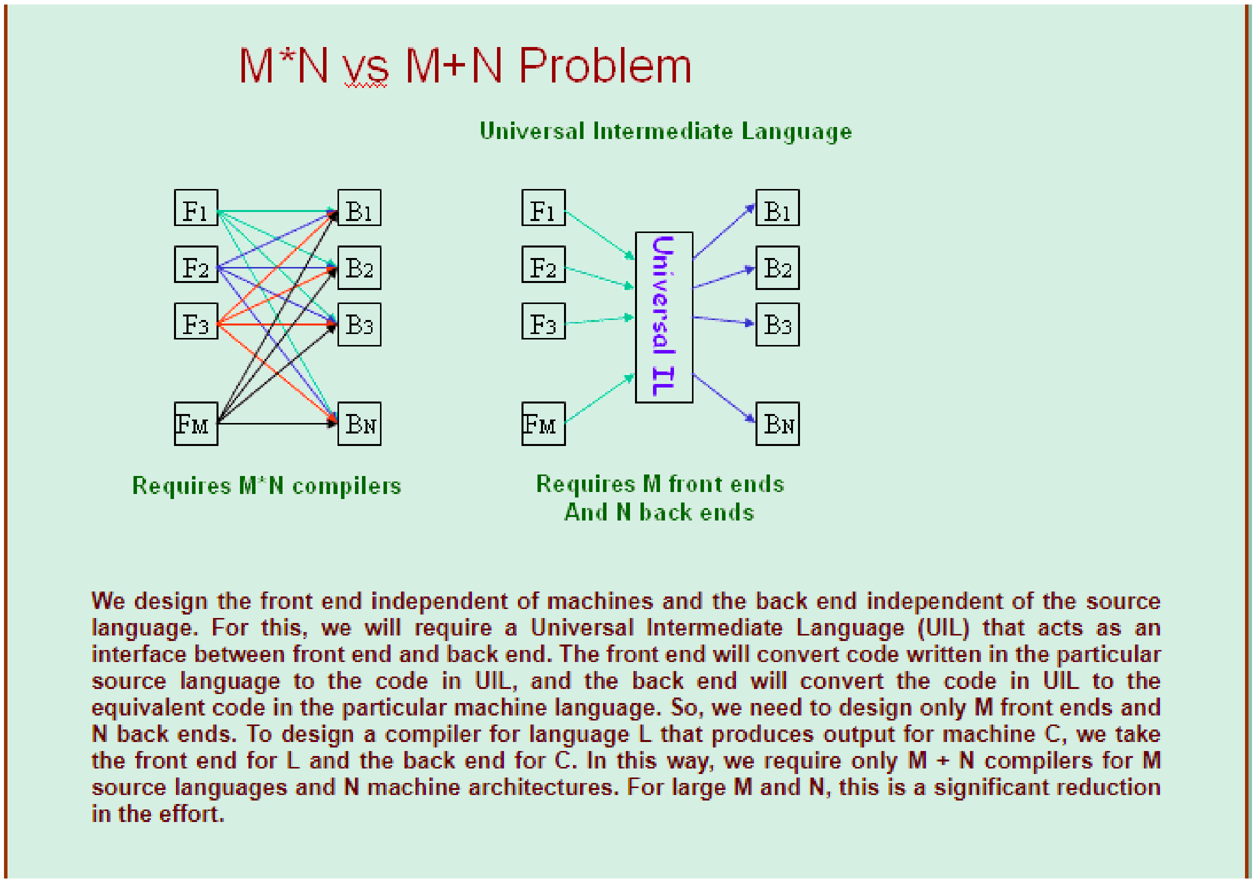
- Compilers are required for all the languages and all the machines

- For M languages and N machines we need to develop M*N compilers

- However, there is lot of repetition of work because of similar activities in the front ends and back ends

- Can we design only M front ends and N back ends, and some how link them to get all M*N compilers?

The compiler should fit in the integrated development environment. This opens many challenges in design e.g., appropriate information should be passed on to the debugger in case of erroneous programs. Also the compiler should find the erroneous line in the program and also make error recovery possible. Some features of programming languages make compiler design difficult, e.g., Algol68 is a very neat language with most good features. But it could never get implemented because of the complexities in its compiler design.



M*N vs M+N Problem

We design the front end independent of machines and the back end independent of the source language. For this, we will require a Universal Intermediate Language (UIL) that acts as an interface between front end and back end. The front end will convert code written in the particular source language to the code in UIL, and the back end will convert the code in UIL to the equivalent code in the particular machine language. So, we need to design only M front ends and N back ends. To design a compiler for language L that produces output for machine C, we take the front end for L and the back end for C. In this way, we require only M + N compilers for M source languages and N machine architectures. For large M and N, this is a significant reduction in the effort.

Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.