# CSc 453

## Compilers and Systems Software

## 1 : Compiler Overview

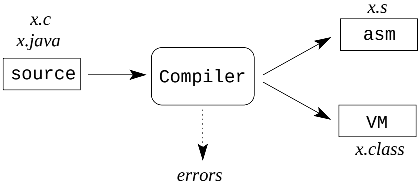
Department of Computer Science
University of Arizona

collberg@gmail.com

# What does a compiler do?

# What's a Compiler???

- At the very basic level a compiler translates a computer program from source code to some kind of executable code:




- Often the source code is simply a text file and the executable code is a resulting assembly language program: `gcc -S x.c` reads the C source file `x.c` and generates an assembly code file `x.s`. Or the output can be a virtual machine code: `javac x.java` produces `x.class`.

# What's a Language Translator???

- A compiler is really a special case of a language translator.
- A translator is a program that transforms a "program" $P_1$ written in a language $L_1$ into a program $P_2$ written in another language $L_2$.
- Typically, we desire $P_1$ and $P_2$ to be semantically equivalent, i.e. they should behave identically.

# Example Language Translators

| source language | translator | target language |
|---|---|---|
| $\LaTeX$ | $\xrightarrow{\texttt{latex2html}}$ | html |
| Postscript | $\xrightarrow{\texttt{ps2ascii}}$ | text |
| FORTRAN | $\xrightarrow{\texttt{f2c}}$ | C |
| C++ | $\xrightarrow{\texttt{cfront}}$ | C |
| C | $\xrightarrow{\texttt{gcc}}$ | assembly |
| .class | $\xrightarrow{\texttt{SourceAgain}}$ | Java |
| x86 binary | $\xrightarrow{\texttt{fx32}}$ | Alpha binary |

# What's a Language???

- A formal language is a notation for precisely communicating ideas.
- By formal we mean that we know exactly which "sentences" belong to the language and that every sentence has a well-defined meaning.
- A language is defined by specifying its syntax and semantics.
- The syntax describes how words can be formed into sentences. The semantics describes what those sentences mean.

# Example Languages

- English is a natural, not a formal language. The sentence

  Many missiles have many warheads.

  has multiple possible meanings.
- Programming languages: FORTRAN, LISP, Java, C++,...
- Text processing languages: LaTeX, troff,...

  ```
  \begin{frame}\frametitle{Example Languages}
  \begin{itemize}
     \item English is a \redtxt{natural}, not a formal
           language. The sentence
  \end{itemize}
  \end{frame}
  ```
- Specification languages: VDM, Z, OBJ,...

# Compiler Input

**Text File** Common on Unix.

**Syntax Tree** A structure editor uses its knowledge of the source language syntax to help the user edit & run the program. It can send a syntax tree to the compiler, relieving it of lexing & parsing.

# Compiler Output

**Assembly Code** Unix compilers do this. Slow, but easy for the compiler.

**Object Code** `.o`-files on Unix. Faster, since we don't have to call the assembler.

**Executable Code** Called a load-and-go-compiler.

**Abstract Machine Code** Serves as input to an interpreter. Fast turnaround time.

**C-code** Good for portability.

# Compiler Tasks

- **Static Semantic Analysis** Is the program (statically) correct? If not, produce error messages to the user.
- **Code Generation** The compiler must produce code that can be executed.
- **Symbolic Debug Information** The compiler should produce a description of the source program needed by symbolic debuggers. Try `man gdb`.
- **Cross References** The compiler may produce **cross-referencing** information. Where are identifiers declared & referenced?
- **Profiler Information** Where does my program spend most of its execution time? Try `man gprof`.

# The structure of a compiler

# Compiler Phases

| ANALYSIS | SYNTHESIS |
|---|---|
| Lexical Analysis | Intermediate Code Generation |
| Syntactic Analysis | Code Optimization |
| Semantic Analysis | Machine Code Generation |

# Compiler Phases – Lexical analysis

- The lexer reads the source file and divides the text into lexical units (tokens), such as:

  Reserved words **BEGIN**, **IF**,...
  identifiers `x`, `StringTokenizer`,...
  special characters $+, *, -,$ ^,...
  numbers 30, 3.14,...
  comments `(* text *)`,
  strings `"text"`.

- Lexical errors (such as 'illegal character', 'undelimited character string', 'comment without end') are reported.

## Lexical Analysis of English

- The sentence

  The boy's cowbell won't play.

  would be translated to the list of tokens

  `the, boy+possessive, cowbell, will, not, play`

## Lexical Analysis of Java

- The sentence

  `x = 3.14 * (9.0+y);`

  would be translated to the list of tokens

  ```
  <ID,x>, EQ, <FLOAT,3.14>, STAR, LPAREN,
  <FLOAT,9.0>, PLUS, <ID,y>, RPAREN, SEMICOLON
  ```

# Compiler Phases – Syntactic analysis

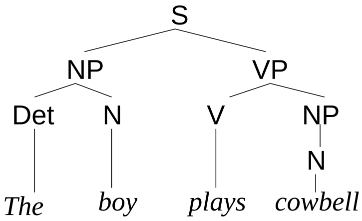
- Syntax analysis (parsing) determines the structure of a sentence.
- The compiler reads the tokens produced during lexical analysis and builds an abstract syntax tree (AST), which reflects the hierarchical structure of the input program.
- Syntactic errors are reported to the user:
  - 'missing ;',
  - 'BEGIN without END'

## Syntactic Analysis of English

- The sentence

  The boy plays cowbell.

  would be parsed into the tree



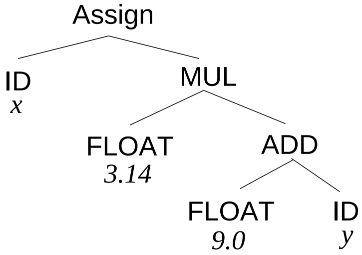

- S=sentence, NP=noun phrase, VP=verb phrase, Det=determiner, N=noun, V=verb.

## Syntactic Analysis of Java

- The sentence

```
x = 3.14 * (9.0+y);
```

would be parsed into the AST



# Compiler Phases – Semantic analysis

- The AST produced during syntactic analysis is decorated with attributes, which are then evaluated. The attributes can represent any kind of information such as expression types.
- The compiler also collects information useful for future phases.
- Semantic errors are reported:
  - 'identifier not declared',
  - 'integer type expected'.

## Semantic Analysis of English

- In the sentence

  The boy plays his cowbell.

  we determine that `his` refers to `the boy`.

## Semantic Analysis of Java

- In the sentence

```
static float luftballons = 10;
void P() {int luftballons = 99;
          System.out.println(luftballons);}
```

  the compiler must determine

  - which `luftballons` the print-statement refers to,
  - that `float luftballons=10` has the wrong type.

# Compiler Phases – IR Generation

- From the decorated AST this phase generates intermediate code (IR).
- The IR adds an extra a level of abstraction between the high level AST and the very low level assembly code we want to generate. This simplifies code generation.
- A carefully designed IR allows us to build compilers for a number of languages and machines, by mixing and matching front-ends and back-ends.

## IR Generation of English

- From the sentence

  Every boy has a cowbell.

  we could generate

$$\forall x; \texttt{boy}(x) \Rightarrow \texttt{has-cowbell}(x)$$

## IR Generation of Java

- From the sentence

  ```
  x = 3.14 * (9.0+y);
  ```

  the compiler could generate the (stack-based) IR code

  ```
  pusha x,  pushf 3.14, pushf 9.0,
  push y, add, mul, assign
  ```

# Compiler Phases – Code Optimization

- The (often optional) code optimization phase transforms an IR program into an equivalent but more efficient program.
- Typical transformations include

  common subexpression elimination only compute an expression once, even if it occurs more than once in the source),

  inline expansion insert a procedure's code at the call site to reduce call overhead,

  algebraic transformations $A := A + A$ is faster than $A := A * 2$.

# Compiler Phases – Code Generation

- The last compilation phase transforms the intermediate code into machine code, usually assembly code or link modules.
- Alternatively, the compiler generates Virtual Machine Code (VM), i.e. code for a software defined architecture. Java compilers, for example, generate class files containing bytecodes for the Java VM.

# Multipass Compilation

# Multi-pass Compilation

- The next slide shows the outline of a typical compiler. In a unix environment each pass could be a stand-alone program, and the passes could be connected by pipes:

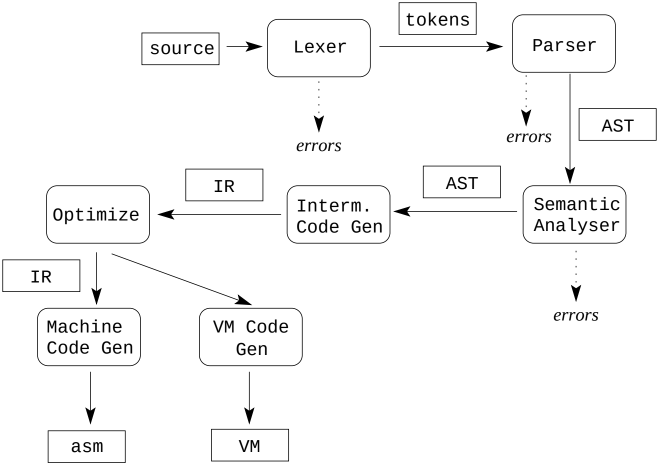
  ```
  lex x.c | parse | sem | ir | opt | codegen > x.s
  ```

- For performance reasons the passes are usually integrated:

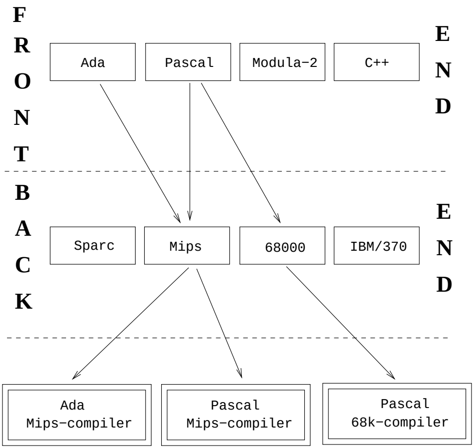
  ```
  front x.c > x.ir
  back x.ir > x.s
  ```

  The front-end does all analysis and IR generation. The back-end optimizes and generates code.

# Multi-pass Compilation...



# Mix-and-Match Compilers



# Example

- Let's compile the procedure `Foo`, from start to finish:

```
PROCEDURE Foo ();
VAR i :  INTEGER;
BEGIN
  i := 1;
  WHILE i < 20 DO
    PRINT i * 2;
    i := i * 2 + 1;
  ENDDO;
END Foo;
```

The compilation phases are:

*Lexial Analysis* ⇒ *Syntactic Analysis* ⇒ *Semantic Analysis* ⇒ *Intermediate code generation* ⇒ *Code Optimization* ⇒ *Machine code generation.*

# Example – Lexical Analysis

- Break up the source code (a text file) and into tokens.

| Source Code | Stream of Tokens |
|---|---|
| PROCEDURE Foo (); | PROCEDURE, <id,Foo>, LPAR, RPAR, SC, |
| VAR i : INTEGER; | VAR, <id,i>, COLON, <id,INTEGER>,SC, |
| BEGIN | BEGIN, <id,i>,CEQ,<int,1>,SC, |
| i := 1; | WHILE, <id,i>, LT, <int,20>,DO, |
| WHILE i < 20 DO | PRINT, <id,i>, MUL, <int,2>, SC, |
| PRINT i * 2; | <id,i>, CEQ, <id,i>, MUL, <int,2>, |
| i := i * 2 + 1; | PLUS, <int,1>, SC, ENDDO, SC, END, |
| ENDDO; | <id,Foo>, SC |
| END Foo; | |

# Example – Lexical Analysis...

- We defined the following set of tokens:

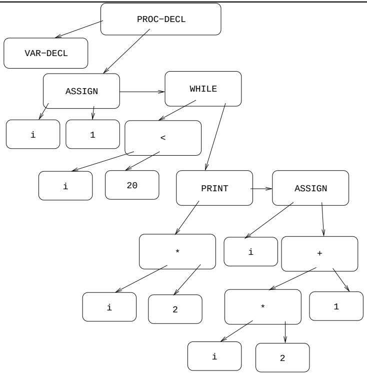
| TOKEN | STRING |
|---|---|
| PROCEDURE | "PROCEDURE" |
| <id,Foo> | identifier |
| LPAR | "(" |
| RPAR | ")" |
| SC | ";" |
| VAR | "VAR" |
| COLON | ":" |
| BEGIN | "BEGIN" |
| CEQ | ":=" |

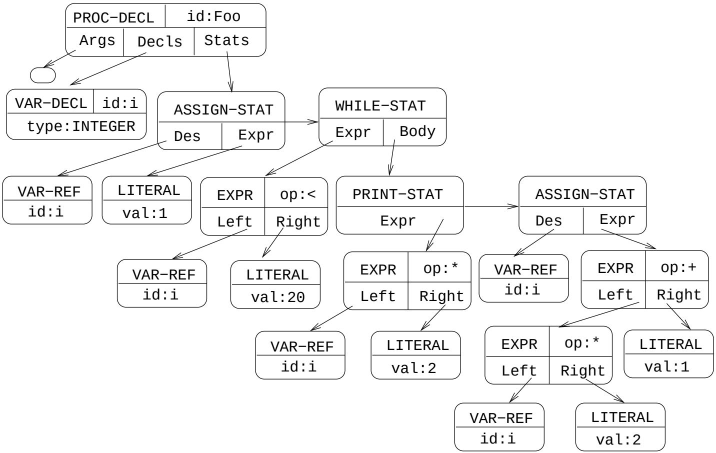
| TOKEN | STRING |
|---|---|
| <int,1> | integer literal |
| WHILE | "WHILE" |
| LT | "<" |
| DO | "DO" |
| PRINT | "PRINT" |
| MUL | "*" |
| PLUS | "+" |
| ENDDO | "ENDDO" |
| END | "END" |

# Example – Syntactic Analysis

| Stream of Tokens | Abstract Syntax Tree |
|---|---|

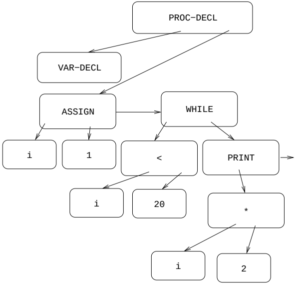
```
PROCEDURE, <id,Foo>,
LPAR,RPAR,SC,VAR,<id,i>,
COLON,<id,INTEGER>,SC,
BEGIN,<id,i>,CEQ,<int,1>,
SC,WHILE,<id,i>,LT,<int,20>,
DO,PRINT,<id,i>,MUL,<int,2>,
SC,<id,i>,CEQ,<id,i>,MUL,
<int,2>,PLUS,<int,1>,SC,
ENDDO,SC,END,<id,Foo>,SC
```

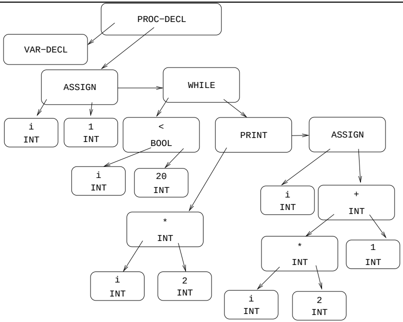


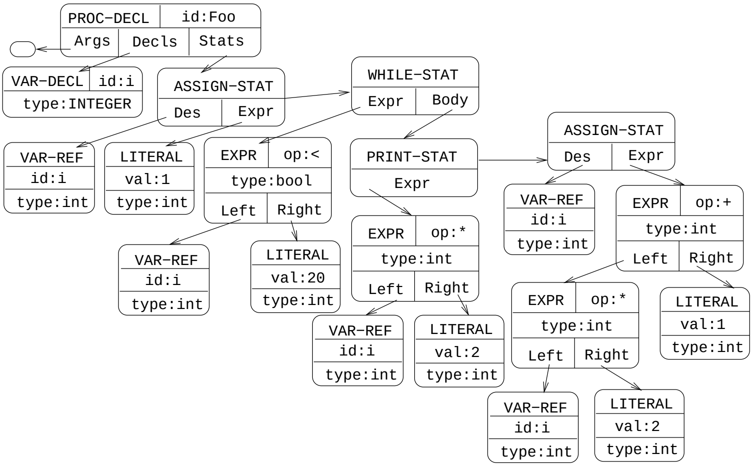

PROC−DECL
id:Foo
Args
Decls
Stats
VAR−DECL
id:i
type:INTEGER
ASSIGN−STAT
Des
Expr
WHILE−STAT
Expr
Body
VAR−REF
id:i
LITERAL
val:1
EXPR
op:<
Left
Right
PRINT−STAT
Expr
ASSIGN−STAT
Des
Expr
VAR−REF
id:i
LITERAL
val:20
EXPR
op:*
Left
Right
VAR−REF
id:i
EXPR
op:+
Left
Right
VAR−REF
id:i
LITERAL
val:2
EXPR
op:*
Left
Right
LITERAL
val:1
VAR−REF
id:i
LITERAL
val:2

# Example – Semantic Analysis

| Abstract Syntax Tree | Decorated Abstract Syntax Tree |
| --- | --- |

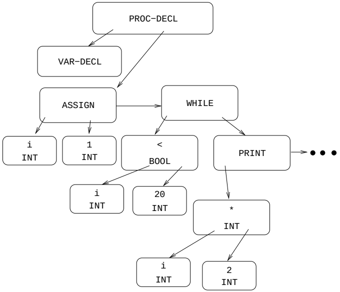



PROC-DECL
id:Foo
Args
Decls
Stats
VAR-DECL
id:i
type:INTEGER
ASSIGN-STAT
Des
Expr
WHILE-STAT
Expr
Body
VAR-REF
id:i
type:int
LITERAL
val:1
type:int
EXPR
op:<
type:bool
Left
Right
PRINT-STAT
Expr
ASSIGN-STAT
Des
Expr
VAR-REF
id:i
type:int
EXPR
op:+
type:int
Left
Right
VAR-REF
id:i
type:int
LITERAL
val:20
type:int
EXPR
op:*
type:int
Left
Right
VAR-REF
id:i
type:int
LITERAL
val:2
type:int
EXPR
op:*
type:int
Left
Right
LITERAL
val:1
type:int
VAR-REF
id:i
type:int
LITERAL
val:2
type:int

# Example – IR Generation

Decorated Abstract Syntax Tree




Intermediate Code

| | | | | |
|---|---|---|---|---|
| [1] | ASSIGN | i | 1 | |
| [2] | BRGE | i | 20 | [9] |
| [3] | MUL | $t_1$ | i | 2 |
| [4] | PRINT | $t_1$ | | |
| [5] | MUL | $t_2$ | i | 2 |
| [6] | ADD | $t_3$ | $t_2$ | 1 |
| [7] | ASSIGN | i | $t_3$ | |
| [8] | JUMP | [2] | | |
| [9] | | | | |

# Example – IR Generation...

| Intermediate Code | | | | | Intermediate Code Definition | |
|---|---|---|---|---|---|---|
| [1] | ASSIGN | i | 1 | | $\boxed{\text{ASSIGN } A, B}$ | $A := B;$ |
| [2] | BRGE | i | 20 | [9] | $\boxed{\text{BRGE } A, B, C}$ | IF $(A \geq B)$ THEN continue at instruction $C$; |
| [3] | MUL | $t_1$ | i | 2 | $\boxed{\text{MUL } A, B, C}$ | $A := B * C;$ |
| [4] | PRINT | $t_1$ | | | $\boxed{\text{ADD } A, B, C}$ | $A := B + C;$ |
| [5] | MUL | $t_2$ | i | 2 | $\boxed{\text{SHL } A, B, C}$ | $A$:=shift $B$ left $C$ steps; |
| [6] | ADD | $t_3$ | $t_2$ | 1 | $\boxed{\text{PRINT } A}$ | Print $A$ and a newline; |
| [7] | ASSIGN | i | $t_3$ | | $\boxed{\text{JUMP } A}$ | Continue at instruction $A$; |
| [8] | JUMP | [2] | | | | |
| [9] | | | | | | |

# Example – Code Optimization

| Intermediate Code | | | | |
|---|---|---|---|---|
| [1] | ASSIGN | i | 1 | |
| [2] | BRGE | i | 20 | [9] |
| [3] | MUL | $t_1$ | i | 2 |
| [4] | PRINT | $t_1$ | | |
| [5] | MUL | $t_2$ | i | 2 |
| [6] | ADD | $t_3$ | $t_2$ | 1 |
| [7] | ASSIGN | i | $t_3$ | |
| [8] | JUMP | [2] | | |
| [9] | | | | |

| Optimized Intermediate Code | | | | |
|---|---|---|---|---|
| [1] | ASSIGN | i | 1 | |
| [2] | BRGE | i | 20 | [8] |
| [3] | SHL | $t_1$ | i | 1 |
| [4] | PRINT | $t_1$ | | |
| [5] | ADD | $t_2$ | $t_1$ | 1 |
| [6] | ASSIGN | i | $t_2$ | |
| [7] | JUMP | [2] | | |
| [8] | | | | |

# Example – Machine Code Generation

| | | Intermediate Code | | | | | MIPS Machine Code | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | `.data` | |
| | | | | | | `_i:` | `.word 0` | |
| [1] | ASSIGN | i | 1 | | | | `.text` | |
| [2] | BRGE | i | 20 | [8] | | | `.globl main` | |
| [3] | SHL | $t_1$ | i | 1 | | `main:` | `li` | `$14, 1` |
| [4] | PRINT | $t_1$ | | | | `$32:` | `bge` | `$14, 20, $33` |
| [5] | ADD | $t_2$ | $t_1$ | 1 | | | `sll` | `$a0, $14, 1` |
| [6] | ASSIGN | i | $t_2$ | | | | `li` | `$v0, 1` |
| [7] | JUMP | [2] | | | | | `syscall` | |
| [8] | | | | | | | `addu` | `$14, $a0, 1` |
| | | | | | | | `b` | `$32` |
| | | | | | | `$33:` | `sw` | `$14, _i` |

# Summary

# Readings and References

- Read Louden:

  Introduction pp. 1–18, 21–27.

- or the Dragon Book:

  Introduction pp. 1–24
  A Simple Compiler pp. 25–82
  Some Compilers pp. 733–744

- or the Tiger Book:

  Introduction pp. 1–15

# Summary

- The structure of a compiler depends on
  1. the complexity of the language we're working on (higher complexity $\Rightarrow$ more passes),
  2. the quality of the code we hope to produce (better code $\Rightarrow$ more passes),
  3. the degree of portability we hope to achieve (more portable $\Rightarrow$ better separation between front- and back-ends).
  4. the number of people working on the compiler (more people $\Rightarrow$ more independent modules).

# Summary...

- Some highly retargetable compilers for high-level languages produce C-code, rather than machine code. This C-code is then compiled by the native C compiler to machine code.
- Some languages (APL, LISP, Smalltalk, Java, ICON, Perl, Awk) are traditionally interpreted (executed in software by an interpreter) rather than compiled to machine code.

# Summary. . .

- Some interpreters use dynamic compilation (or jitting), switching between
  1. interpreting the virtual machine code,
  2. translating the virtual machine code to native machine code,
  3. executing the native machine code,
  4. optimizing the native and/or virtual machine code, and
  5. throwing native code away if it is no longer needed or takes up too much room.

  All this is done dynamically at runtime.

# Exercises

# Exercise

- Consider this little Java class:

```
class M {
   public static void main(String args[]) {
      int x = 6;
      while (x != 42) x += 6;
   }
}
```

- Show the result after lexical analysis (what tokens do you need to compile Java?)!
- Show the AST after syntax analysis of the token stream (what AST nodes does Java nead?)!
- Show the AST after semantic analysis!
- Define an intermediate code and generate it from the AST!

# Historical Notes

# The First Compiler

- FORTRAN I was the first "high-level" programming language. It's designers also wrote the first real compiler and invented many of the techniques that we use today.
- The FORTRAN manual can be found here:

  `http://www.fh-jena.de/~kleine/history`.
- The excerpt on the next few slides is taken from

  > *John Backus*, The history of FORTRAN I, II, and III, *History of Programming Languages, The first ACM SIGPLAN conference on History of programming languages, 1978.*

Before 1954 almost all programming was done in machine language or assembly language. Programmers rightly regarded their work as a complex, creative art that required human inventiveness to produce an efficient program. Much of their effort was devoted to overcoming the difficulties created by the computers of that era: the lack of index registers, the lack of builtin floating point operations, restricted instruction sets (which might have AND but not OR, for example), and primitive input- output arrangements. Given the nature of computers, the services which "automatic programming" performed for the programmer were concerned with overcoming the machine's shortcomings. Thus the primary concern of some "automatic programming" systems was to allow the use of symbolic addresses and decimal numbers...

Another factor which influenced the development of FORTRAN was the economics of programming in 1954. The cost of programmers associated with a computer center was usually at least as great as the cost of the computer itself. ... In addition, from one quarter to one half of the computer's time was spent in debugging. ...
This economic factor was one of the prime motivations which led me to propose the FORTRAN project ... in late 1953 (the exact date is not known but other facts suggest December 1953 as a likely date). I believe that the economic need ... provided for our constantly expanding needs over the next five years without ever askinging us to project or justify those needs in a formal budget.

It is difficult for a programmer of today to comprehend what "automatic program- ming" meant to programmers in 1954. To many it then meant simply providing mnemonic operation codes and symbolic addresses, to others it meant the simple'process of obtaining subroutines from a library and inserting the addresses of operands into each subroutine. ... We went on to raise the question "...can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?" ...

In view of the widespread skepticism about the possibility of producing efficient programs with an automatic programming system and the fact that inefficiencies could no longer be hidden, we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programs almost as efficient as hand coded ones and do so on virtually every job.

As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs. Of course one of our goals was to design a language which would make it possible for engineers and scientists to write programs themselves for the 704. ... Very early in our work we had in mind the notions of assignment statements, subscripted variables, and the DO statement....

The language described in the "Preliminary Report" had variables of one or two characters in length, function names of three or more characters, recursively defined "expressions", subscripted variables with up to three subscripts, "arithmetic formulas" (which turn out to be assignment statements), and "DO-formulas".

One much-criticized design choice in FORTRAN concerns the use of spaces: blanks were ignored, even blanks in the middle of an identifier. There was a common problem with keypunchers not recognizing or properly counting blanks in handwritten data, and this caused many errors. We also regarded ignoring blanks as a device to enable programmers to arrange their programs in a more readable form without altering their meaning or introducing complex rules for formatting statements.

Section I was to read the entire source program, compile what instructions it could, and file all the rest of the information from the source program in appropriate tables. ...

Using the information that was filed in section I, section 2 faced a completely new kind of problem; it was required to analyze the entire structure of the program in order to generate optimal code from DO statements and references to subscripted variables.

...

section 4, ... analyze the flow of a program produced by sections I and 2, divide it into "basic blocks" (which contained no branching), do a Monte Carlo (statistical) analysis of the expected frequency of execution of basic blocks--by simulating the behavior of the program and keeping counts of the use of each block--using information from DO statements and FREQUENCY statements, and collect information about index register usage ... Section 5 would then do the actual transformation of the program from one having an unlimited number of index registers to one having only three.

The final section of the compiler, section 6, assembled the final program into a relocatable binary program...

Unfortunately we were hopelessly optimistic in 1954 about the problems of debugging FORTRAN programs (thus we find on page 2 of the Report: "Since FORTRAN should virtually eliminate coding and debugging...")

Because of our 1954 view that success in producing efficient programs was more important than the design of the FORTRAN language, I consider the history of the compiler construction and the work of its inventors an integral part of the history of the FORTRAN language; ...