CSc 553

Principles of Compilation

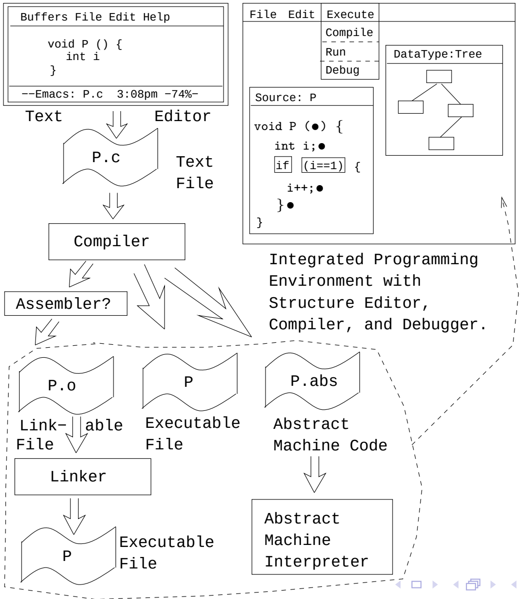
1 : Compiler Overview

Department of Computer Science
University of Arizona

collberg@gmail.com

What does a compiler do?

What's a Compiler???



Compiler Input and Output

Compiler Input

Text File
: Common on Unix.

Syntax Tree
: A structure editor uses its that 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 The compiler should produce **profiler** information. Where does my program spend most of its execution time? Try `man gprof`.

The structure of a compiler

Compiler Phases

ANALYSIS

Lexical Analysis

Syntactic Analysis

Semantic Analysis

SYNTHESIS

Intermediate Code Generation

Code Optimization

Machine Code Generation

Compiler Organization

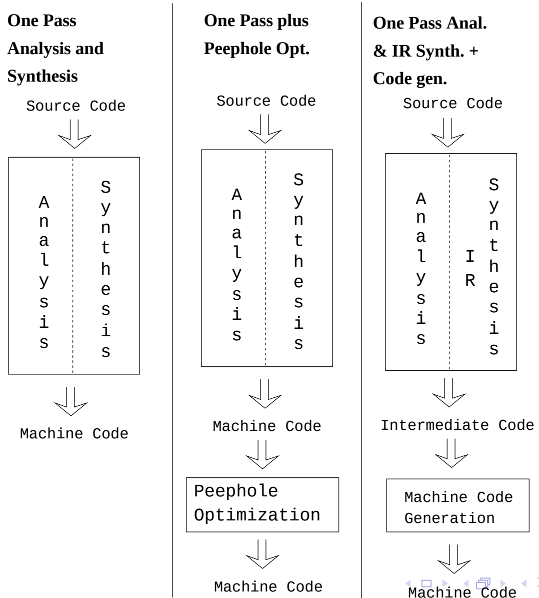
Compiler Organization I (a)

One Pass Analysis and Synthesis Fast. OK for definition-before-use languages like Pascal. No explicit intermediate representation. Target machine code is generated on-the-fly. Very little optimization is possible since we can't "look forward". Difficult to retarget, since semantic analysis and code generation are performed simultaneously.

One Pass Plus Peephole Optimization Better code generation by performing a scan over the machine code and making local improvements.

One Pass Analysis + IR Generation Machine code is produced from an explicit intermediate representation. Better chances that the front-end & back-end can be recycled.

Compiler Organization I (b)



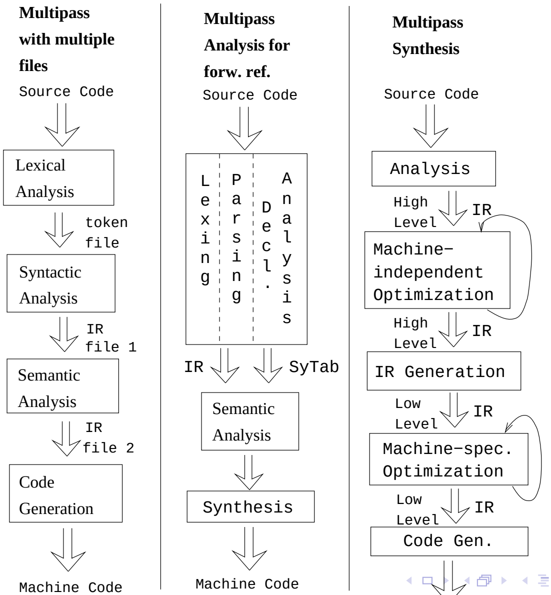
Compiler Organization II (a)

Multipass w/ Interm. Files Early compilers were severely constrained by the size of available primary storage. Therefore the compiler was often organized as a series of passes, where each pass wrote its output to an intermediate file which then became input to the next pass. Still a good design if you're not worried about speed.

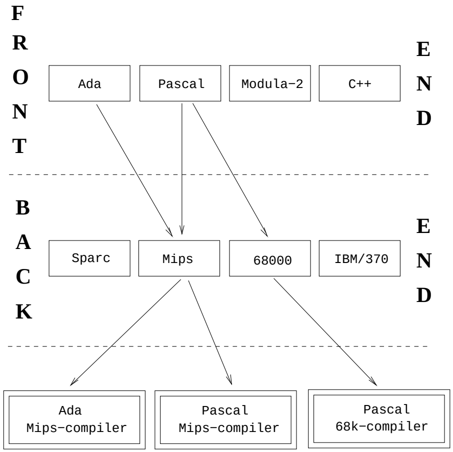
Multipass Analysis Languages that allow "use-before-declaration", require the compiler to process the program more than once..

Multipass Synthesis Highly optimizing compilers usually process the intermediate representation in several passes. Often, we separate machine-independent and machine-dependent optimizations.

Compiler Organization II (b)



Multi-Language — Multi-target Compilers



Multipass Compilation

Multi-pass Compilation I

- We are going to work with compilers with multi-pass analysis and multi-pass synthesis parts.
- These compilers are very general:
  - They can handle any language, whether free or fixed declaration order.
  - They can produce efficient code.
  - They are portable since the front- and back-ends can be reused for compilers for new languages or new architectures.
- We will assume that the parser builds a tree (an **abstract syntax tree**) that is modified during semantic analysis, and then used during code generation.

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:

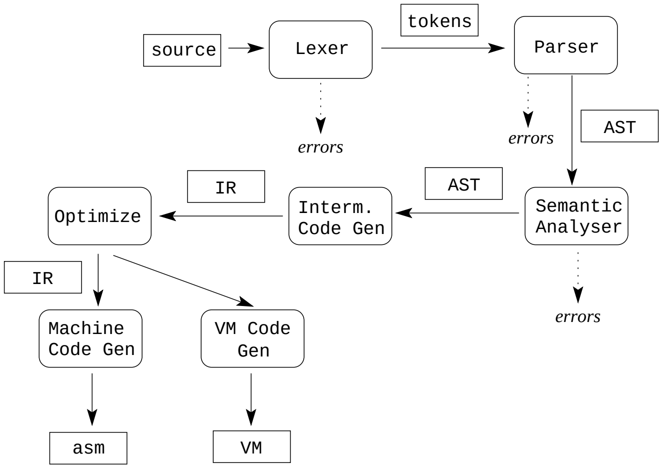
  ```
  lex x.c | parse | sem | ir | opt | codegen > x.s
  ```

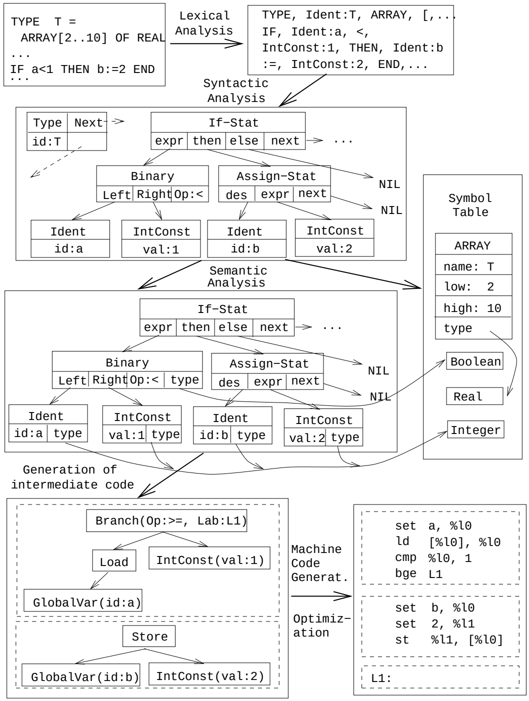
- For performance reasons the passes are usually integrated:

  ```
  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. . .





```
TYPE  T =
 ARRAY[2..10] OF REAL
...
IF a<1 THEN b:=2 END
...
```

```
TYPE, Ident:T, ARRAY, [,...
IF, Ident:a, <,
IntConst:1, THEN, Ident:b
:=, IntConst:2, END,...
```

```
set  a, %l0
ld   [%l0], %l0
cmp  %l0, 1
bge  L1

set  b, %l0
set  2, %l1
st   %l1, [%l0]

L1:
```

Example

Example I

- Let's go through the compilation of a 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:

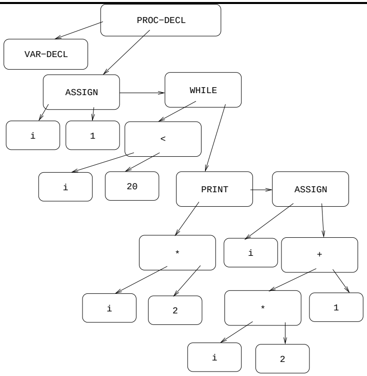
  Lexial Analysis ⇒ *Syntactic Analysis* ⇒
  Semantic Analysis ⇒ *Intermediate code generation*
  ⇒ *Code Optimization* ⇒ *Machine code generation.*

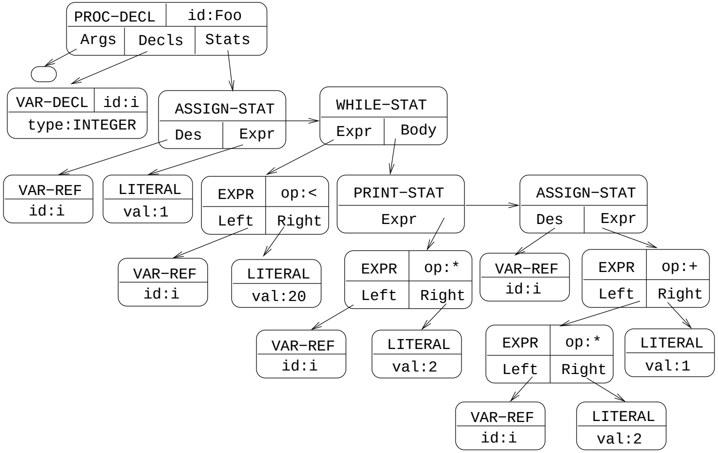
Example II – Lexical Analysis

- 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>, PLU`
`    i := i * 2 + 1;`	`<int,1>, SC, ENDDO, SC, END, <id,Foo>`
`  ENDDO;`	
`END Foo;`	

Example III/A – Syntactic Analysis

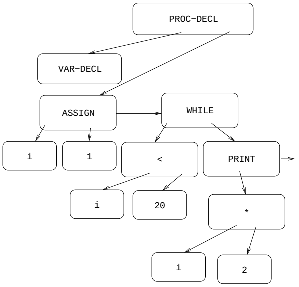
Stream of Tokens	Abstract Syntax Tree
PROCEDURE, <id,Foo>, LPAR,RPAR,SC,VAR,<id,i>, COLON,<id,INTEGER>,SC, BEGIN,<id,i>,CEQ,<int,1>, SC,WHILE,<id,i>,LT,<int,20>, DO,PRINT,<id,i>,MUL,<int,2>, SC,<id,i>,CEQ,<id,i>,MUL, <int,2>,PLUS,<int,1>,SC, ENDDO,SC,END,<id,Foo>,SC	

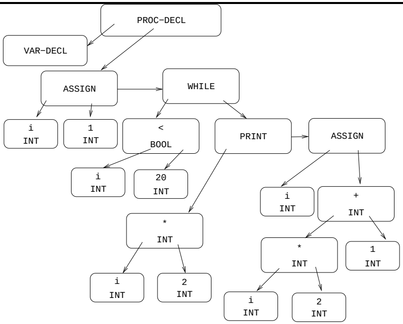


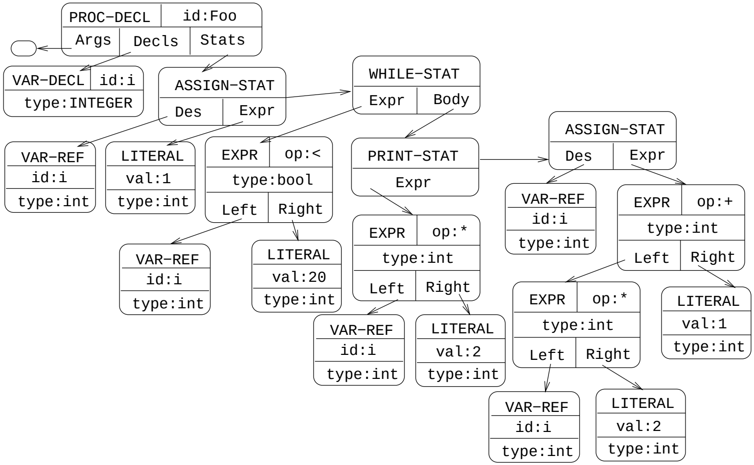
PROC−DECL
id:Foo
Args
Decls
Stats
VAR−DECL
id:i
type:INTEGER
ASSIGN−STAT
Des
Expr
WHILE−STAT
Expr
Body
VAR−REF
id:i
LITERAL
val:1
EXPR
op:<
Left
Right
PRINT−STAT
Expr
ASSIGN−STAT
Des
Expr
VAR−REF
id:i
LITERAL
val:20
EXPR
op:*
Left
Right
VAR−REF
id:i
EXPR
op:+
Left
Right
VAR−REF
id:i
LITERAL
val:2
EXPR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

Example IV/A – Semantic Analysis

Abstract Syntax Tree	Decorated Abstract Syntax Tree



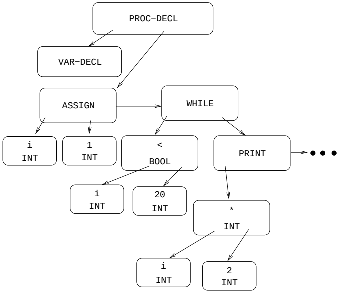




PROC−DECL
id:Foo
Args
Decls
Stats
VAR−DECL
id:i
type:INTEGER
ASSIGN−STAT
Des
Expr
WHILE−STAT
Expr
Body
VAR−REF
id:i
type:int
LITERAL
val:1
type:int
EXPR
op:<
type:bool
Left
Right
PRINT−STAT
Expr
ASSIGN−STAT
Des
Expr
VAR−REF
id:i
type:int
LITERAL
val:20
type:int
EXPR
op:*
type:int
Left
Right
VAR−REF
id:i
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 V/A – Intermediate Code 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 V/B – Intermediate Code Generation

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]				

Intermediate Code Definition

$\boxed{\text{ASSIGN } A, B}$	$A := B$;
$\boxed{\text{BRGE } A, B, C}$	IF $(A \geq B)$ THEN continue at instruction C;
$\boxed{\text{MUL } A, B, C}$	$A := B * C$;
$\boxed{\text{ADD } A, B, C}$	$A := B + C$;
$\boxed{\text{SHL } A, B, C}$	A:=shift B left C steps;
$\boxed{\text{PRINT } A}$	Print A and a newline;
$\boxed{\text{JUMP } A}$	Continue at instruction A;

Example VI – Code Optimization

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

Example VII – Machince 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 the Dragon Book:

Introduction	Chapter 1
A Simple Syntax-Directed Translator	Chapter 2
A Complete Front-End	Appendix A

Summary I

- 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).
- 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.

Summary II

- Some languages (APL, LISP, Smalltalk, Java, ICON, Perl, Awk) are traditionally *interpreted* (executed in software by an *interpreter*) rather than compiled to machine code.
- 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.

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; ...