Programming Languages

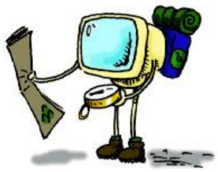
Tevfik Koşar

Lecture - II

January 19th, 2006

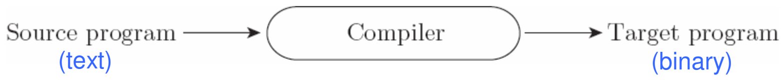
Roadmap

- Compilation
- Interpretation
- Preprocessing
- Linking
- Assembling
- Phases of Compilation
    - Scanning
    - Parsing
    - Semantic Analysis



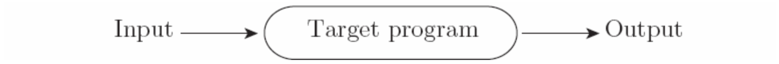
Compiler

- Translates high-level program source code (in text) into a target code (generally binary executable)



Source program (text) → Compiler → Target program (binary)

- Generated target program is standalone
  - After compilation the compiler goes away

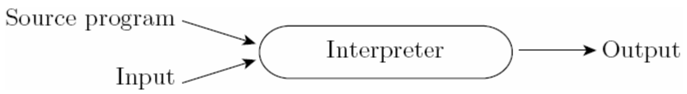


Input → Target program → Output

- Generated target program can be platform-dependant

Interpreter

- Reads and executes the source code line by line
- Stays around during execution
- Does not generate standalone executables

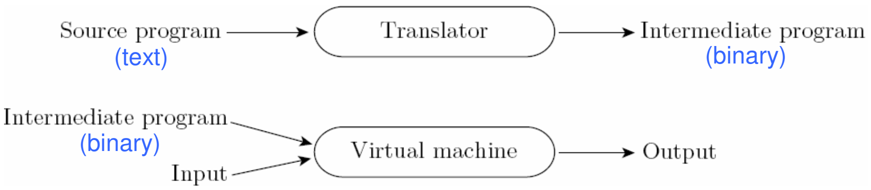


Source program → Interpreter
Input → Interpreter
Interpreter → Output

Compilation vs Interpretation

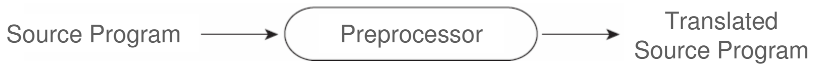
- Compilation
  - Better performance
    - Most decisions are done at compile time (eg. memory references)
    - Compile once, execute may times
- Interpretation
  - More flexible
  - Enables better diagnostics (error messages)
    - After compilation some information is lost
  - Can have source-level debugger

Hybrid Systems



- Example: Java
  - Intermediate binaries are called: “byte codes”

Preprocessors


Source Program

Preprocessor

Translated Source Program

- Preprocessor : initial translator
  - Removes comments & white space
  - Groups characters into tokens (keywords, identifiers, numbers)
  - Expends macros and abbreviations
  - Produced source can be compiled/interpreted more efficiently
    - In early versions of Basic, you had to remove comments to improve performance (reread everytime a certain part was executed)

Compilation, Interpretation & Preprocessing

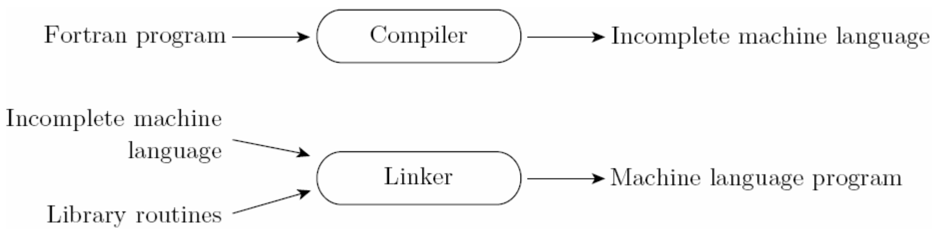
- Compilation generally produces a binary; but does NOT have to produce machine language for some sort of hardware
- Compilation is *translation* from one language into another, with full analysis of the meaning of the input
- Compilation & Interpretation entail semantic *understanding* of what is being processed; pre-processing does not
- A pre-processor will often let errors through. Compilers and Interpreters will not.

Examples

- Interpreted Languages:
  - Java
  - Scheme
  - Prolog
  - Python
  - Most Scripting Languages
- Compiled Languages
  - C / C++
  - Pascal
  - Fortran
  - Ada

Linking

- Compiler uses a *linker* program to merge the appropriate *library* of subroutines (e.g., math functions such as sin, cos, log, etc.) into the final program:
  - eg. Fortran Compiler



Fortran program → Compiler → Incomplete machine language

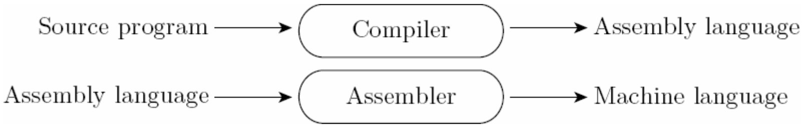
Incomplete machine language → Linker

Library routines → Linker

Linker → Machine language program

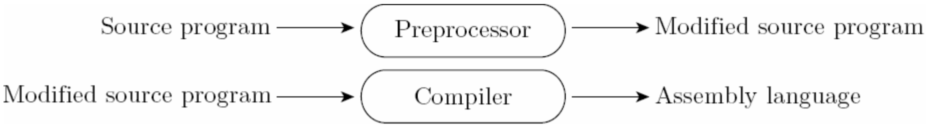
Assembling

- Many compilers generate assembly language instead of a machine language



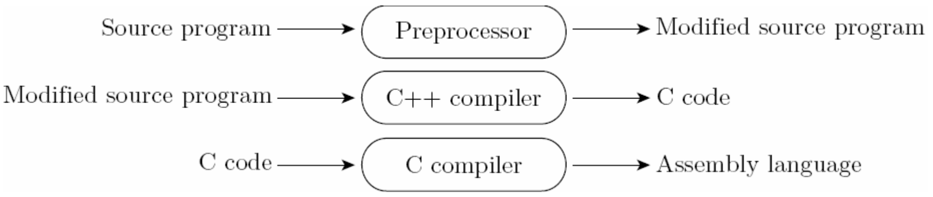
- Facilitates debugging
  - Assembly is easier to read
- Isolates compiler from changes in the format of machine language files
  - only assembler need to be changed, and it is shared by many compilers

C Compiler



- C preprocessor
  - Removes comments & extends macros
  - It can also delete portions of code, which allows several versions of a program to be built from the same source
    - eg. Adding & removing debugging information

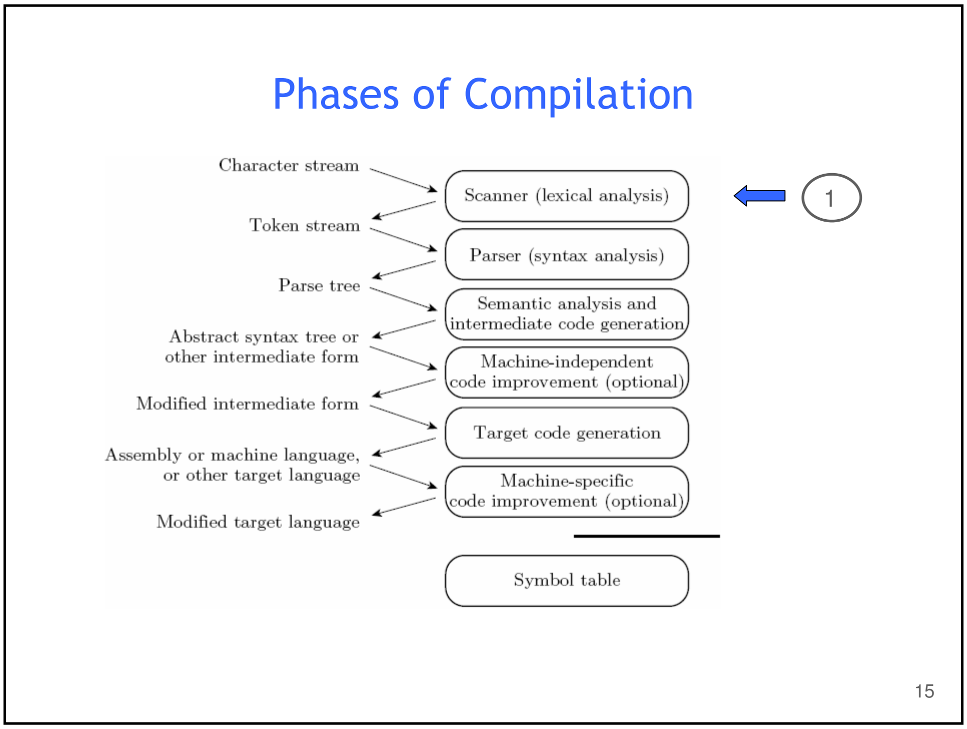
Early C++ Compiler



- Early C++ compilers were generating C code
- Complete error check was performed
- If no errors, C compiler was invoked by the C++ compiler
  - Programmers were unaware of this fact

Roadmap

- Compilation
- Interpretation
- Preprocessing
- Linking
- Assembling
- Phases of Compilation
  - Scanning
  - Parsing
  - Semantic Analysis



Phases of Compilation

Character stream

Scanner (lexical analysis)

1

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis and intermediate code generation

Abstract syntax tree or other intermediate form

Machine-independent code improvement (optional)

Modified intermediate form

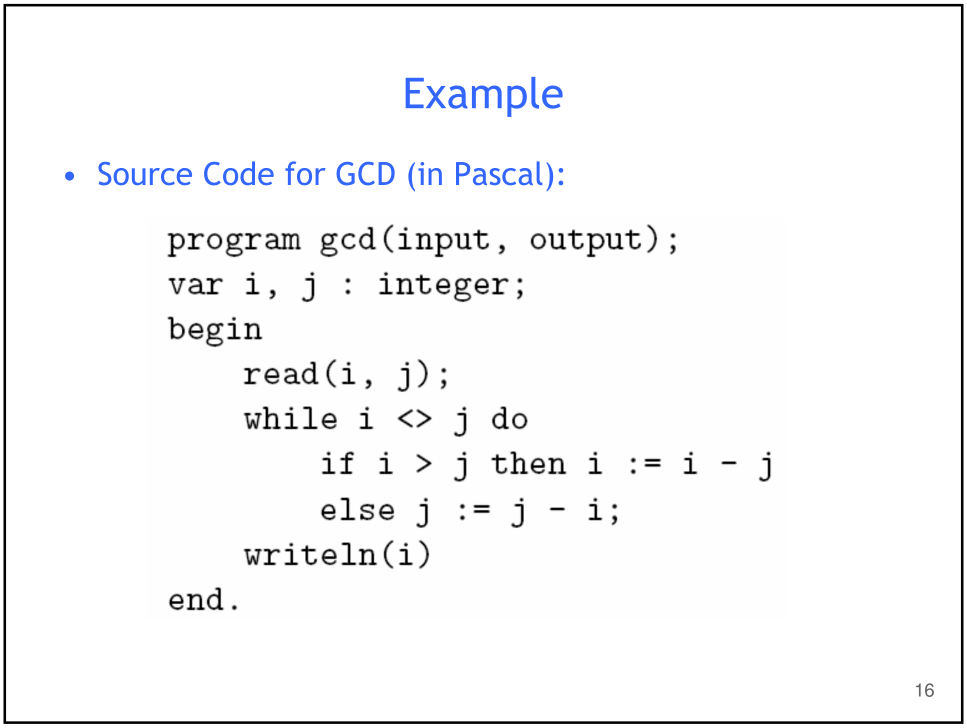
Target code generation

Assembly or machine language, or other target language

Machine-specific code improvement (optional)

Modified target language

Symbol table



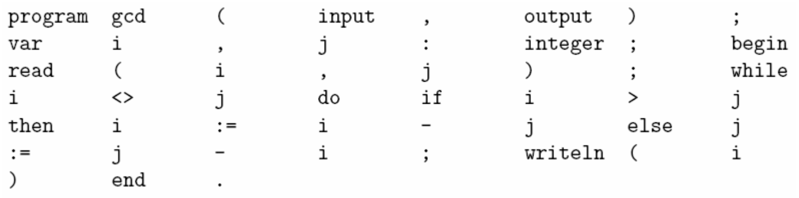
Example

- Source Code for GCD (in Pascal):

```
program gcd(input, output);
var i, j : integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
end.
```

Example

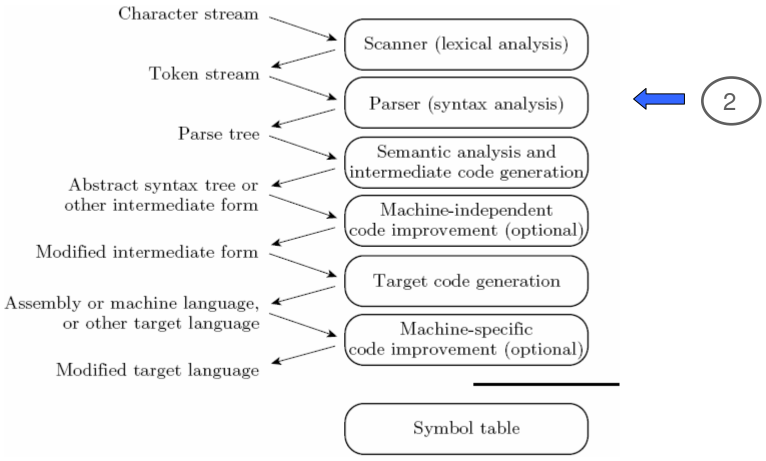
- After Scanning (Lexical Analysis):
  - Characters are grouped in to tokens (smallest meaningful units of the program)
    - Eg. identifiers, variables, punctuation, operators ..

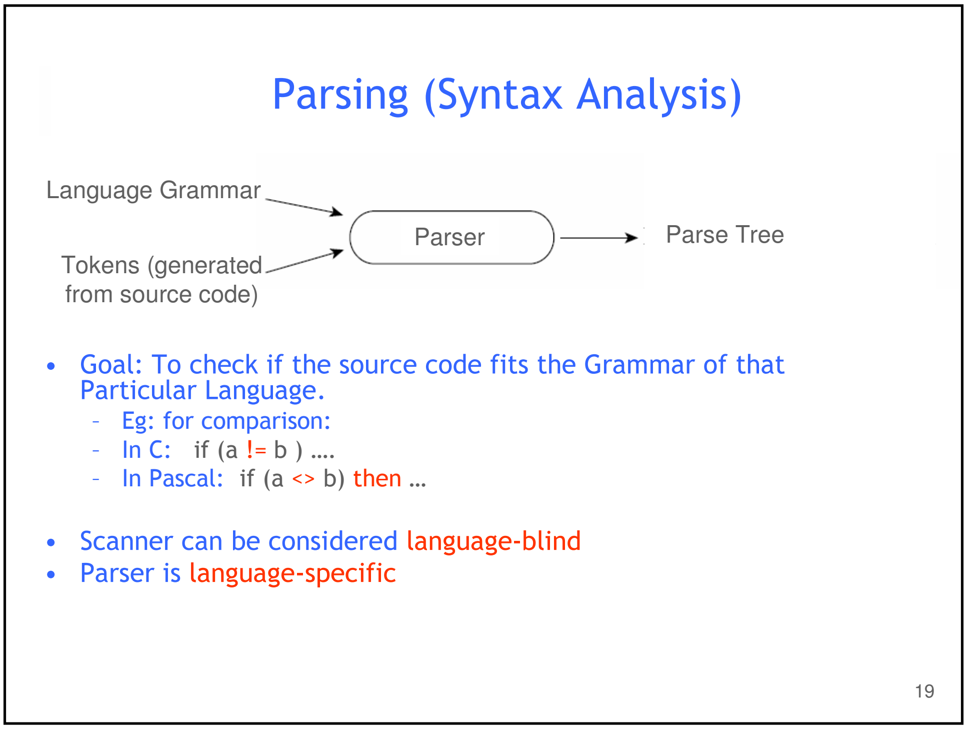


```
program  gcd      (        input    ,        output   )        ;
var      i        ,        j        :        integer  ;        begin
read     (        i        ,        j        )        ;        while
i        <>       j        do       if       i        >        j
then     i        :=       i        -        j        else     j
:=       j        -        i        ;        writeln  (        i
)        end      .
```

- Purpose of Scanning (Lexical Analysis):
  - Simplify the task for parser by reducing the input size

Phases of Compilation


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
2
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Assembly or machine language, or other target language
Machine-specific code improvement (optional)
Modified target language
Symbol table

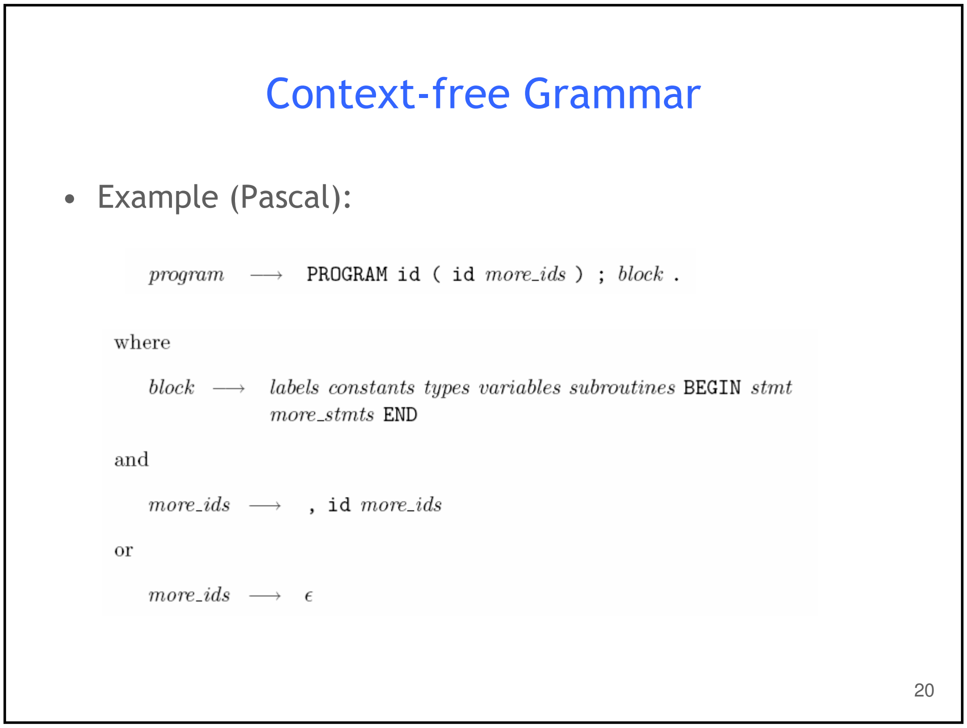


Parsing (Syntax Analysis)

Language Grammar → Parser → Parse Tree

Tokens (generated from source code) → Parser

- Goal: To check if the source code fits the Grammar of that Particular Language.
  - Eg: for comparison:
  - In C: if (a != b) ….
  - In Pascal: if (a <> b) then …
- Scanner can be considered language-blind
- Parser is language-specific



Context-free Grammar

- Example (Pascal):

$$\textit{program} \longrightarrow \texttt{PROGRAM id (id } \textit{more\_ids} \texttt{) ; } \textit{block} \texttt{ .}$$

where

$$\textit{block} \longrightarrow \textit{labels constants types variables subroutines } \texttt{BEGIN } \textit{stmt more\_stmts } \texttt{END}$$

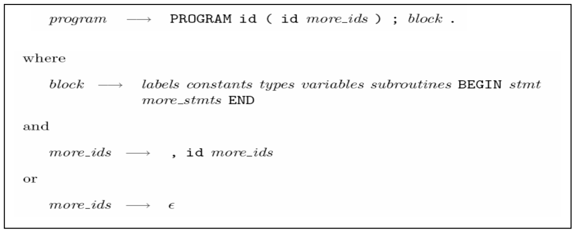
and

$$\textit{more\_ids} \longrightarrow \texttt{, id } \textit{more\_ids}$$

or

$$\textit{more\_ids} \longrightarrow \epsilon$$

Parsing Example



program ⟶ PROGRAM id (id *more_ids*) ; *block* .

where

block ⟶ *labels constants types variables subroutines* BEGIN *stmt* *more_stmts* END

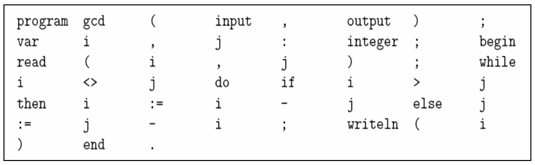
and

more_ids ⟶ , id *more_ids*

or

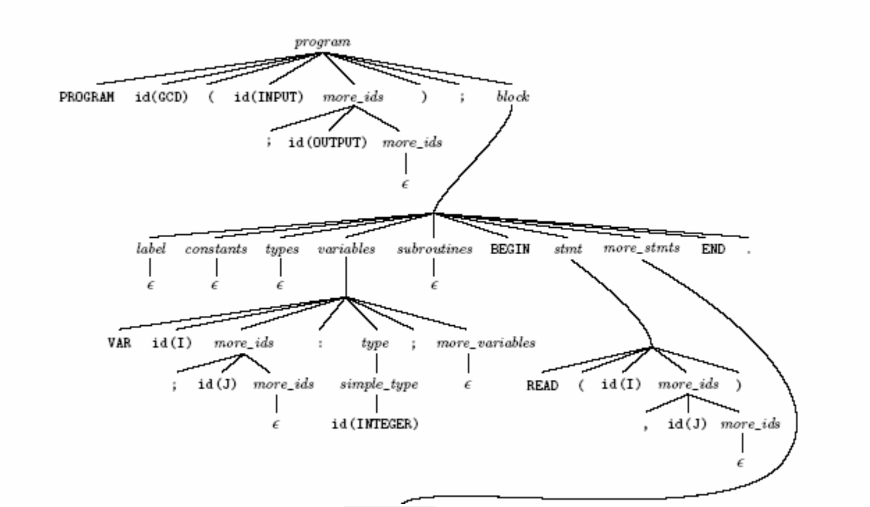
more_ids ⟶ ϵ

+



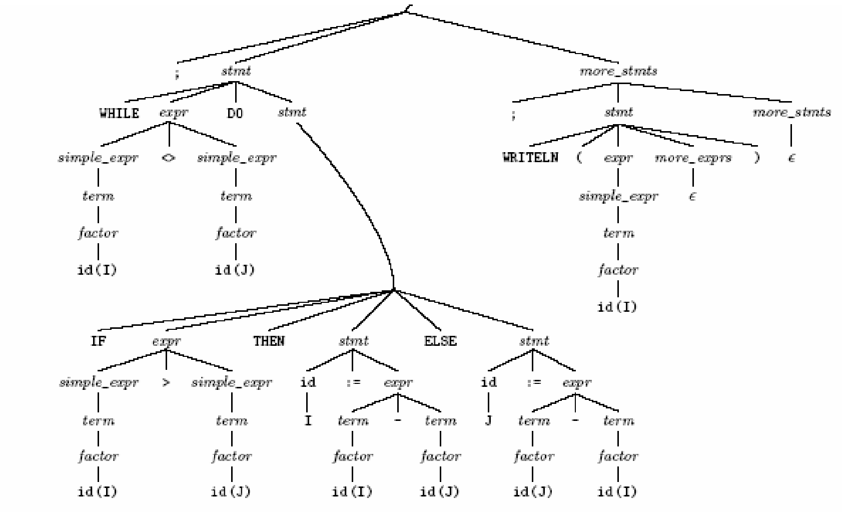
program	gcd	(	input	,	output	)	;
var	i	,	j	:	integer	;	begin
read	(	i	,	j	)	;	while
i	<>	j	do	if	i	>	j
then	i	:=	i	-	j	else	j
:=	j	-	i	;	writeln	(	i
)	end	.					

Parse Tree

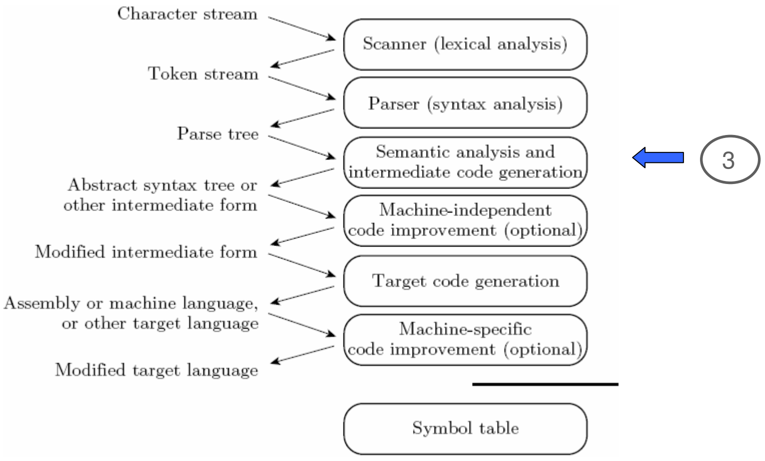


Next slide

Parse Tree (cont.)



Phases of Compilation



Character stream

Scanner (lexical analysis)

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis and intermediate code generation

3

Abstract syntax tree or other intermediate form

Machine-independent code improvement (optional)

Modified intermediate form

Target code generation

Assembly or machine language, or other target language

Machine-specific code improvement (optional)

Modified target language

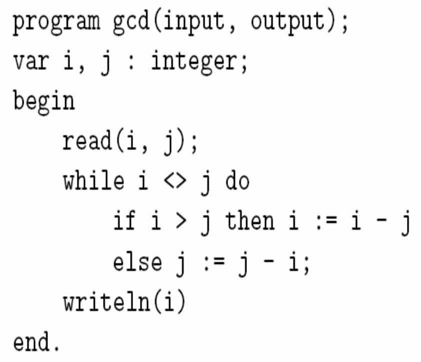
Symbol table

Semantic Analysis

- Discovery of the meaning of the program
- Creates a symbol table which maps each identifier to the information known about it
  - eg. type, scope (the portion of the program it is valid)
- Semantic Analyzer checks to make sure that:
  - Every identifier is declared before it is used
  - No identifier is used in an inappropriate context
    - Assigning incompatible types to each other.
  - Subroutine calls have the correct number and types of arguments

Example

- Source Code

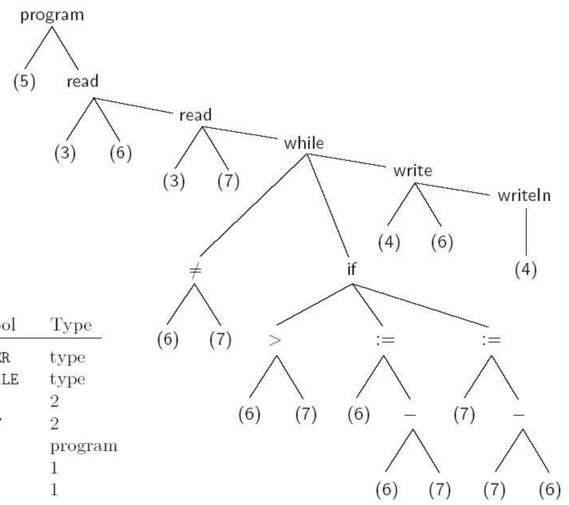


```
program gcd(input, output);
var i, j : integer;
begin
    read(i, j);
    while i <> j do
        if i > j then i := i - j
        else j := j - i;
    writeln(i)
end.
```

- Symbol Table

Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

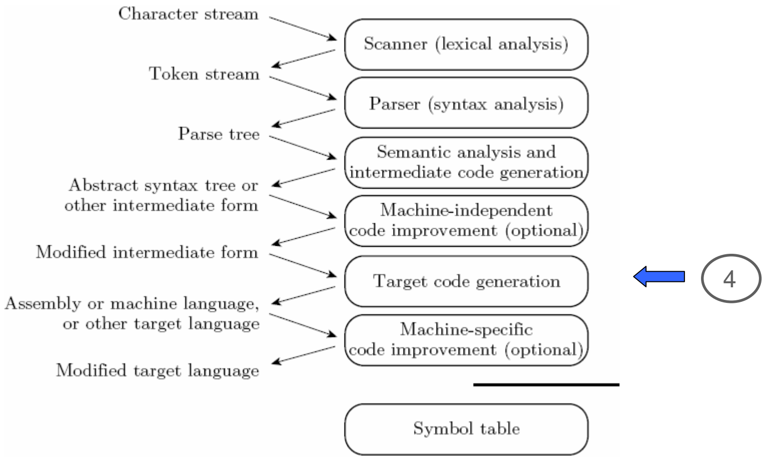
Syntax Tree



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Syntax tree and symbol table for the GCD program.

Phases of Compilation



Character stream

Scanner (lexical analysis)

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis and intermediate code generation

Abstract syntax tree or other intermediate form

Machine-independent code improvement (optional)

Modified intermediate form

Target code generation

4

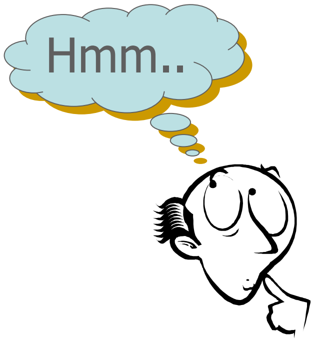
Assembly or machine language, or other target language

Machine-specific code improvement (optional)

Modified target language

Symbol table

Any Questions?



Announcements

- Reading Assignment: Sections 2.1 & 2.2
- HW 1 will be out next Tuesday and will be due 1 week
- Please send your course schedules to me
- Make sure you are in the mailing list