JYOTHISHMATHI INSTITUTE OF TECHNOLOGY & SCIENCE, KARIMNAGAR

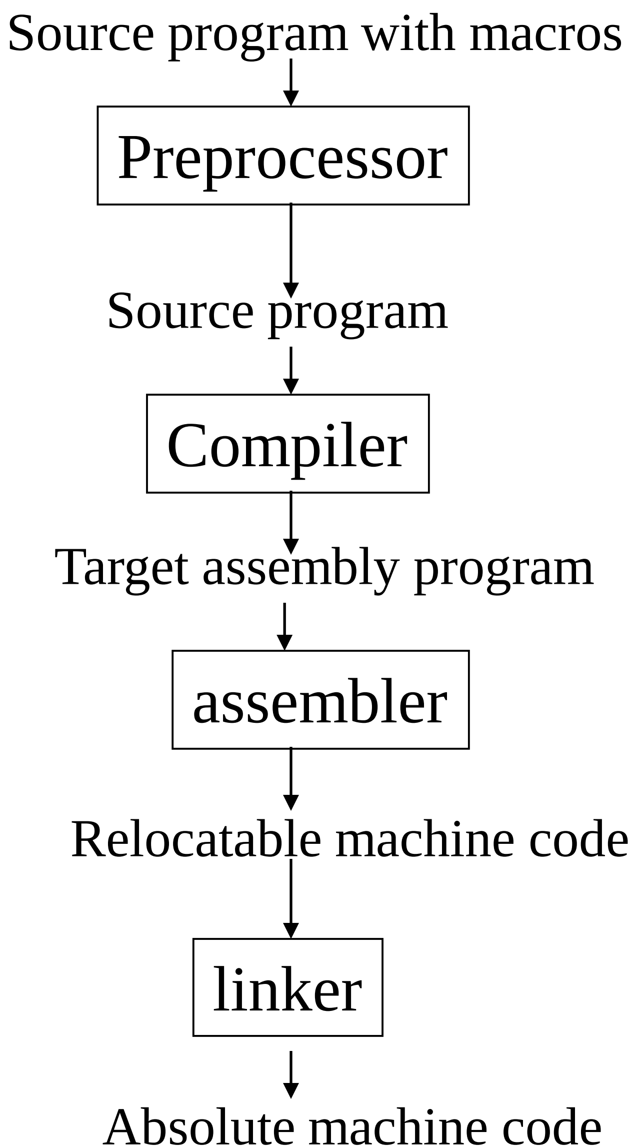
COMPILER DESIGN

G.RANJITH KUMAR
ASST. PROFESSOR
CSE DEPT.

Phases of compiler

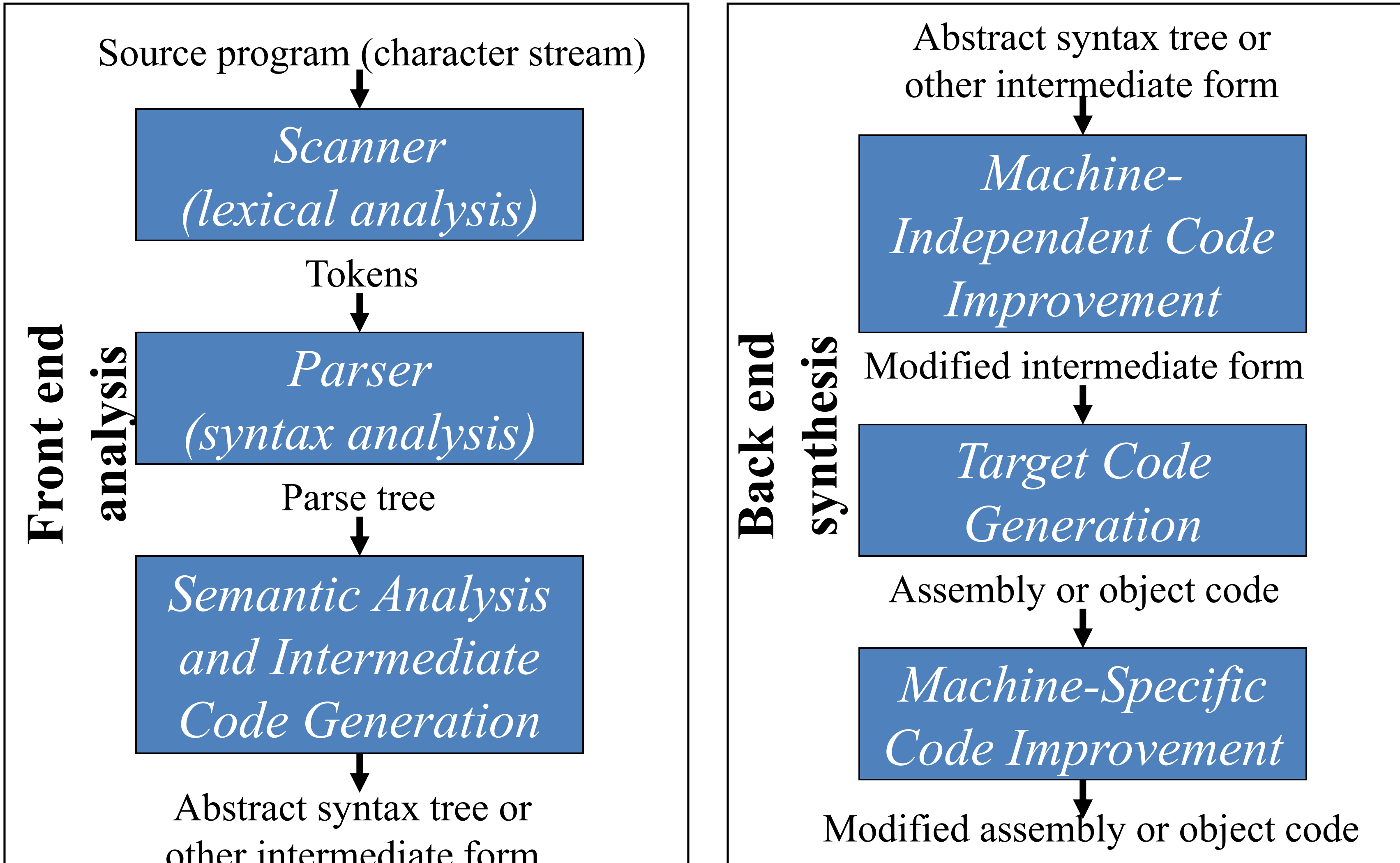
Overview

- Compiler phases
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Intermediate (machine-independent) code generation
  - Intermediate code optimization
  - Target (machine-dependent) code generation
  - Target code optimization


Source program with macros
Preprocessor
Source program
Compiler
Target assembly program
assembler
Relocatable machine code
linker
Absolute machine code

- What is a compiler?
  - A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language).
  - Traditionally, the source language is a high level language and the target language is a low level language (machine code).

Compiler Front- and Back-end



Scanner: Lexical Analysis

- Lexical analysis breaks up a program into tokens
  - Grouping characters into non-separatable units (tokens)
  - Changing a stream to characters to a stream of tokens

```
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.
```

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
:=	i	-	i	;	writeln	(	i
)	**end**	.					

Scanner: Lexical Analysis

- What kind of errors can be reported by lexical analyzer?

```
A = b + @3;
```

Parser: Syntax Analysis

- Checks whether the token stream meets the grammatical specification of the language and generates the syntax tree.
- A grammar of a programming language is typically described by a context free grammer, which also defines the structure of the parse tree.

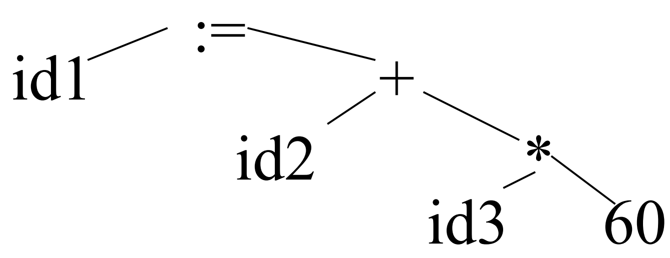
Context-Free Grammars

- A context-free grammar defines the syntax of a programming language
- The syntax defines the syntactic categories for language constructs
  - Statements
  - Expressions
  - Declarations
- Categories are subdivided into more detailed categories
  - A Statement is a
    - For-statement
    - If-statement
    - Assignment

Parsing examples

- Pos = init + / rate * 60 → id1 = id2 + / id3 * const → syntax error (exp ::= exp + exp cannot be reduced).

- Pos = init + rate * 60 → id1 = id2 + id3 * const →



Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.
- A program without grammatical errors may not always be correct program.
  - *pos = init + rate * 60*
  - What if *pos* is a class while *init* and *rate* are integers?
  - This kind of errors cannot be found by the parser
  - Semantic analysis finds this type of error and ensure that the program has a meaning.

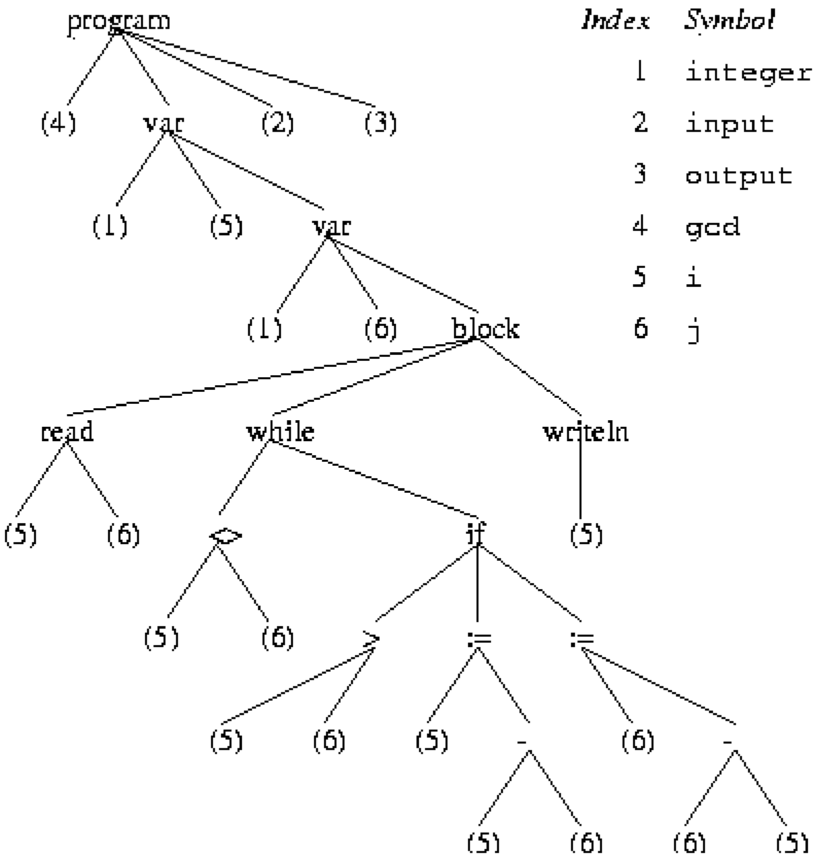
Semantic Analysis

- Static semantic checks (done by the compiler) are performed at compile time
  - Type checking
  - Every variable is declared before used
  - Identifiers are used in appropriate contexts
  - Check subroutine call arguments
  - Check labels
- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  - Array subscript values are within bounds
  - Arithmetic errors, e.g. division by zero
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn't been initialized
  - When a check fails at run time, an exception is raised

Semantic Analysis and Strong Typing

- A language is strongly typed "if (type) errors are always detected"
  - Errors are either detected at compile time or at run time
  - Examples of such errors are listed on previous slide
  - Languages that are strongly typed are Ada, Java, ML, Haskell
  - Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp
- Strong typing makes language safe and easier to use, but potentially slower because of dynamic semantic checks
- In some languages, most (type) errors are detected late at run time which is detrimental to reliability e.g. early Basic, Lisp, Prolog, some script languages

Code Generation and Intermediate Code Forms



Index	*Symbol*
1	integer
2	input
3	output
4	gcd
5	i
6	j

- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers

Example AST for the gcd program in Pascal

Code Generation and Intermediate Code Forms

- Other intermediate code forms
  - intermediate code is something that is both close to the final machine code and easy to manipulate (for optimization). One example is the three-address code:

    dst = op1 op op2

  - The three-address code for the assignment statement:

    temp1 = 60

    temp2 = id3 + temp1

    temp3 = id2 + temp2

    id1 = temp3

- Machine-independent Intermediate code improvement

    temp1 = id3 * 60.0

    id1 = id2 + temp1

Target Code Generation and Optimization

- From the machine-independent form assembly or object code is generated by the compiler

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

- This machine-specific code is optimized to exploit specific hardware features

Summary

- Compiler front-end: lexical analysis, syntax analysis, semantic analysis
  - Tasks: understanding the source code, making sure the source code is written correctly
- Compiler back-end: Intermediate code generation/improvement, and Machine code generation/improvement
  - Tasks: translating the program to a semantically the same program (in a different language).