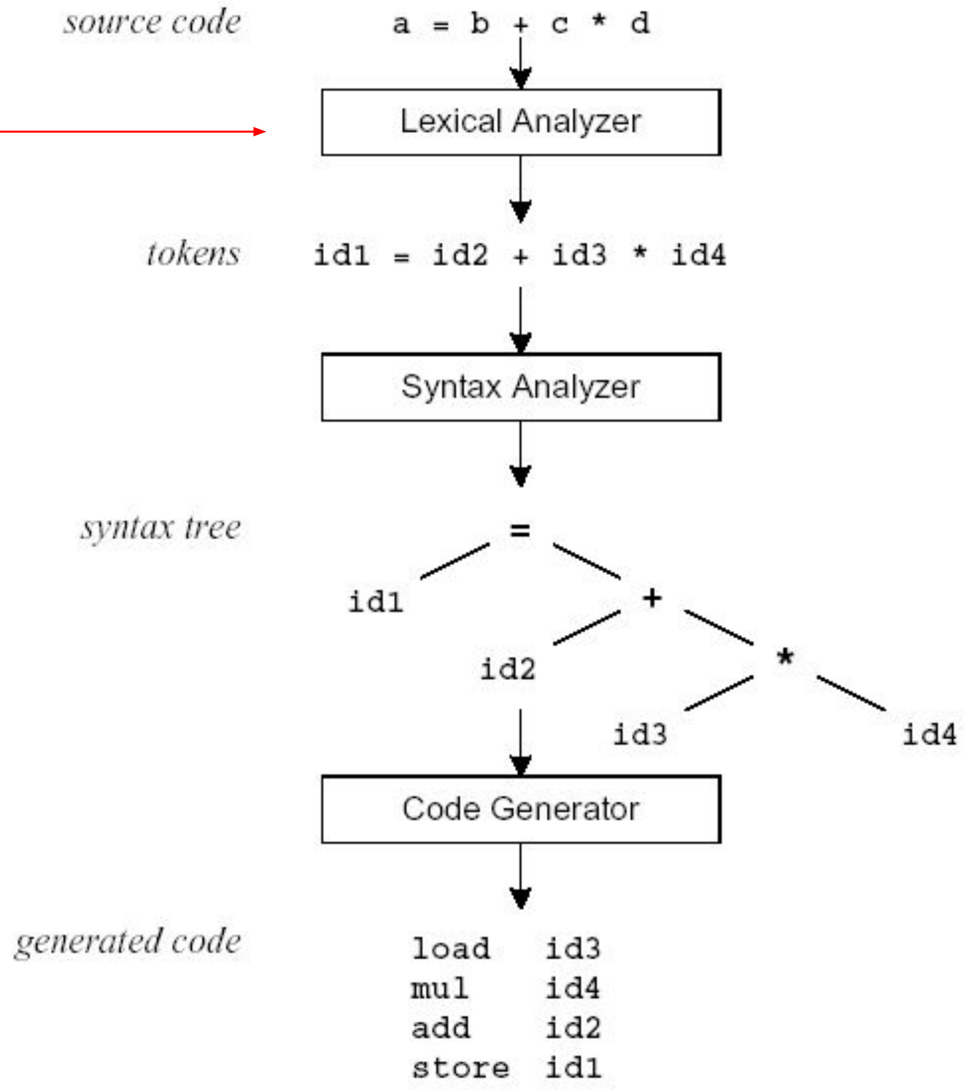
Compiler

- A **compiler** translates the code written in one language to some other language without changing the meaning of the program.
- **Compiler** design covers **basic** translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.
- Cross Compiler

- A Bootstrap Complier: **Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program. This complicated program can further handle even more complicated program and so on.
- Writing a compiler for any high level language is a complicated process. It takes lot of time to write a compiler from scratch. Hence simple language is used to generate target code in some stages.
- Suppose we want to write a cross compiler for new language X. The implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if existing compiler Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.

Compilation Sequence



What is Lex?

- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

  a = b + c * d;

  ID ASSIGN ID PLUS ID MULT ID SEMI

- Lex is an utility to help you rapidly generate your scanners

Lex – Lexical Analyzer

- Lexical analyzers **tokenize** input streams
- Tokens are the **terminals** of a language
  - » English
    - – words, punctuation marks, …
  - » Programming language
    - – Identifiers, operators, keywords, …
- Regular expressions define **terminals/tokens**

Lex Source Program

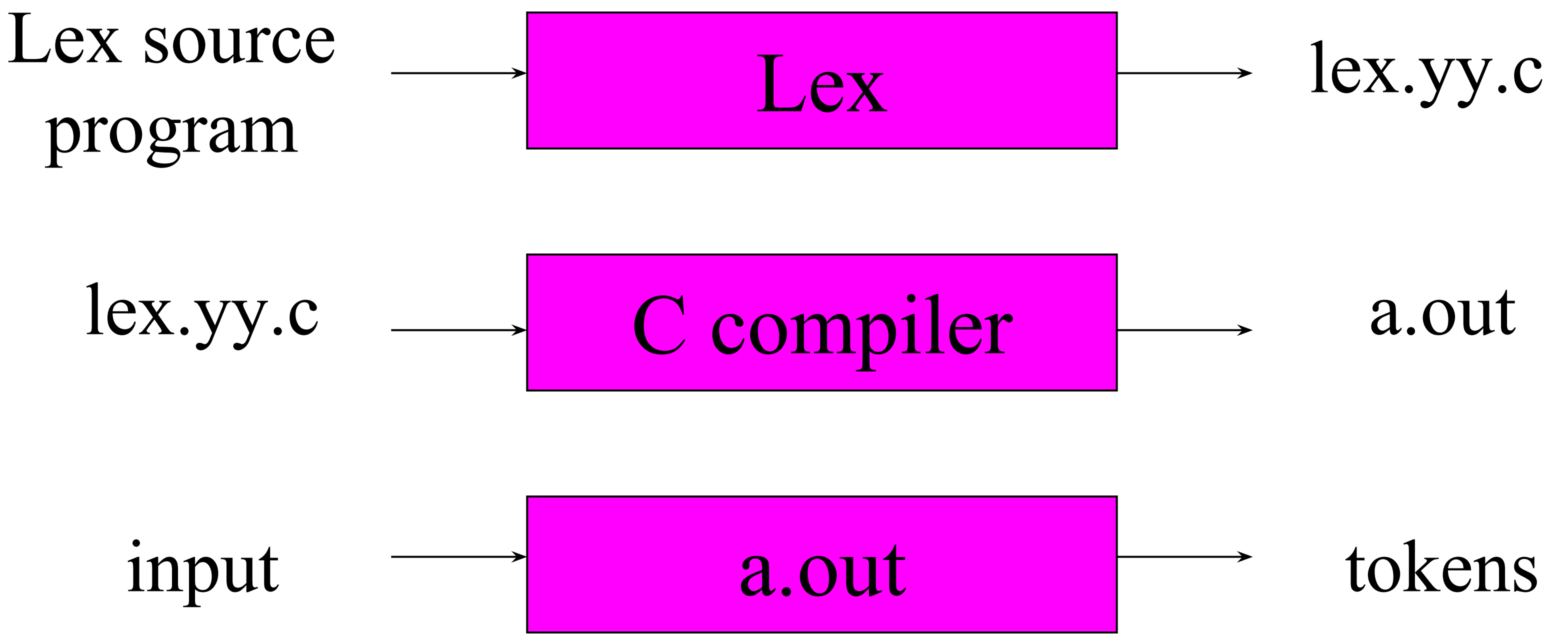
- Lex source is a table of
  - » regular expressions and
  - » corresponding program fragments

```
digit   [0-9]
letter  [a-zA-Z]
%%
{letter}({letter}|{digit})* printf("id:
%s\n", yytext);
\n             printf("new line\n");
%%
main() {
   yylex();
```

Lex Source to C Program

- The table is translated to a C program (lex.yy.c) which
  - reads an input stream
  - partitioning the input into strings which match the given expressions and copying it to an output stream if necessary

An Overview of Lex



Lex Source

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is

```
{definitions}
%%                                   (required)
{transition rules}
%%
                                     (optional)
{user subroutines}
```

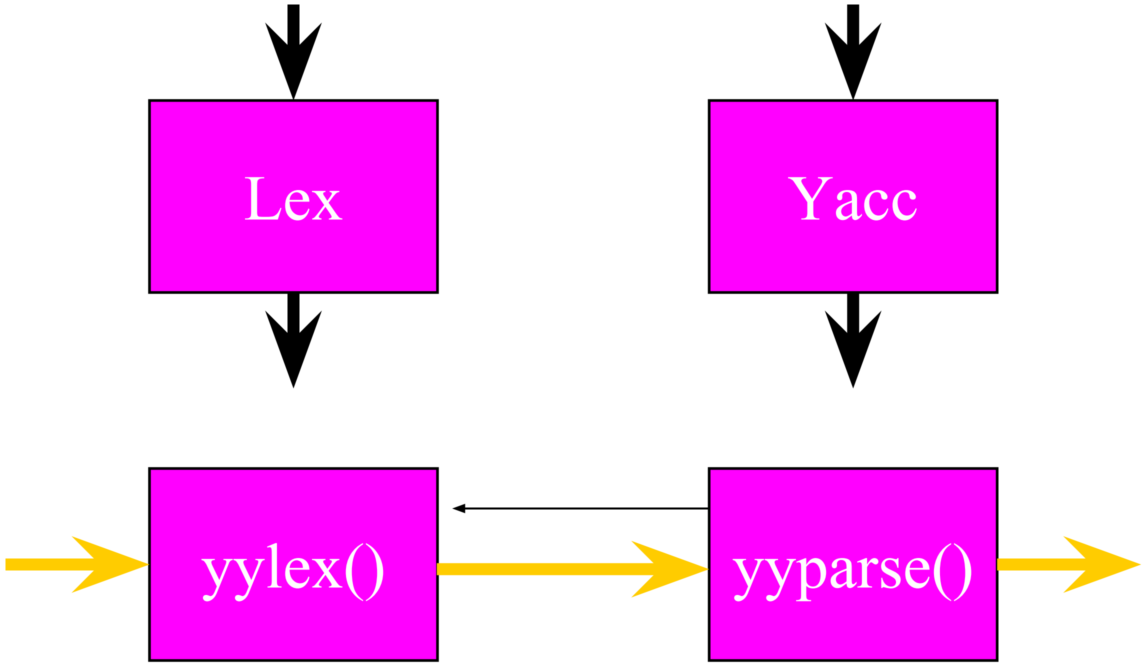
- The absolute minimum Lex program is thus

```
%%
```

Lex v.s. Yacc

- Lex
  - Lex generates C code for a lexical analyzer, or scanner
  - Lex uses patterns that match strings in the input and converts the strings to tokens
- Yacc
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Lex with Yacc



Lex Regular Expressions (Extended Regular Expressions)

- A regular expression matches a set of strings
- Regular expression
  - Operators
  - Character classes
  - Arbitrary character
  - Optional expressions
  - Alternation and grouping
  - Context sensitivity
  - Repetitions and definitions

Operators

“ \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- If they are to be used as text characters, an escape should be used

  \$ = “$”

  \\ = “\”

- Every character but *blank*, *tab* (\t), *newline* (\n) and the list above is always a text character

Character Classes []

- `[abc]` matches a single character, which may be `a`, `b`, or `c`
- Every operator meaning is ignored except `\` `-` and `^`
- e.g.

`[ab]`	=> `a` or `b`
`[a-z]`	=> `a` or `b` or `c` or … or `z`
`[-+0-9]`	=> all the digits and the two signs
`[^a-zA-Z]`	=> any character which is not a letter

Arbitrary Character .

- To match almost character, the operator character . is the class of all characters except newline
- [\40-\176] matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde~)

Optional & Repeated Expressions

- a?	=> zero or one instance of a
- a*	=> zero or more instances of a
- a+	=> one or more instances of a

- E.g.
  ab?c => ac or abc
  [a-z]+ => all strings of lower case letters
  [a-zA-Z][a-zA-Z0-9]* => all alphanumeric strings with a leading alphabetic character

Precedence of Operators

- Level of precedence
  - Kleene closure (*), ?, +
  - concatenation
  - alternation (|)
- All operators are left associative.
- Ex: a*b|cd* = ((a*)b)|(c(d*))

Pattern Matching Primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line / complement
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
[ab]	a or b
a{3}	3 instances of a
“a+b”	literal “a+b” (C escapes still work)

Recall: Lex Source

- Lex source is a table of
  - » regular expressions and
  - » corresponding program fragments (actions)

```
…
%%
<regexp> <action>
<regexp> <action>
…
%%
```

```
a = b + c;


a operator: ASSIGNMENT  b + c;
```

```
%%
"="  printf("operator:
ASSIGNMENT");
```

Transition Rules

- regexp <one or more blanks> action (C code);
- regexp <one or more blanks> { actions (C code) }

- A null statement ; will ignore the input (no actions)

```
[ \t\n]    ;
```

  - » Causes the three spacing characters to be ignored

```
a = b + c;
d = b * c;

↓↓

a=b+c;d=b*c;
```

Transition Rules (cont'd)

- Four special options for actions:
  |, ECHO;, BEGIN, and REJECT;
- | indicates that the action for this rule is from the action for the next rule

```
» [ \t\n]      ;
» " "          |
"\t"      |
"\n"      ;
```

- The unmatched token is using a default action that ECHO from the input to the output

Transition Rules (cont'd)

- REJECT
  - Go do the next alternative

```
…
%%
pink     {npink++; REJECT;}
ink {nink++; REJECT;}
pin {npin++; REJECT;}
. |
\n   ;
%%
…
```

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - » the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - » the default output of default main() is **stdout**.
- **cs20: %./a.out < inputfile > outfile**

- E.g.

```
[a-z]+        printf("%s", yytext);
[a-z]+        ECHO;
[a-zA-Z]+     {words++; chars += yyleng;}
```

Lex Library Routines

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

Review of Lex Predefined Variables

Name	**Function**
char *yytext	pointer to matched string
int yyleng	length of matched string
FILE *yyin	input stream pointer
FILE *yyout	output stream pointer
int yylex(void)	call to invoke lexer, returns token
char* yymore(void)	return the next token
int yyless(int n)	retain the first n characters in yytext
int yywrap(void)	wrapup, return 1 if done, 0 if not done
ECHO	write matched string
REJECT	go to the next alternative rule
INITAL	initial start condition
BEGIN	condition switch start condition

User Subroutines Section

- You can use your Lex routines in the same ways you use routines in other programming languages.

```
%{
   void foo();
%}
letter[a-zA-Z]
%%
{letter}+foo();
%%
…
void foo() {
   …
}
```

User Subroutines Section (cont’d)

- The section where main() is placed

```
%{
  int counter = 0;
%}
letter  [a-zA-Z]

%%
{letter}+ {printf(“a word\n”);
  counter++;}

%%
main() {
  yylex();
```

Usage

- To run Lex on a source file, type
  ```
  lex scanner.l
  ```
- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type
  ```
  cc lex.yy.c -ll
  ```
- To run the lexical analyzer program, type
  ```
  ./a.out < inputfile
  ```

Versions of Lex

- AT&T -- lex
  http://www.combo.org/lex_yacc_page/lex.html
- GNU -- flex
  http://www.gnu.org/manual/flex-2.5.4/flex.html
- a Win32 version of flex :
  http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html
  or Cygwin :
  http://sources.redhat.com/cygwin/

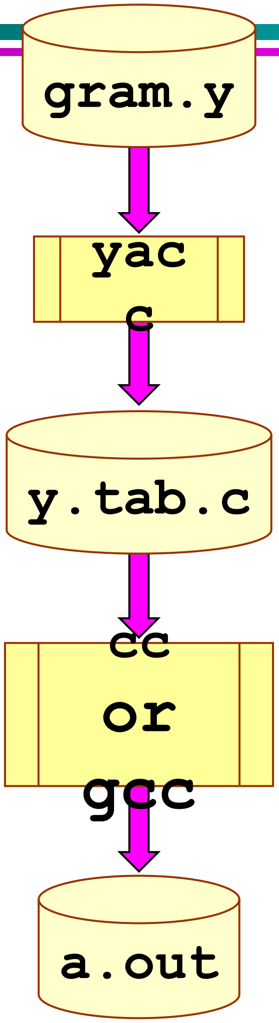
- Lex on different machines is not created equal.

Yacc - Yet Another Compiler-Compiler

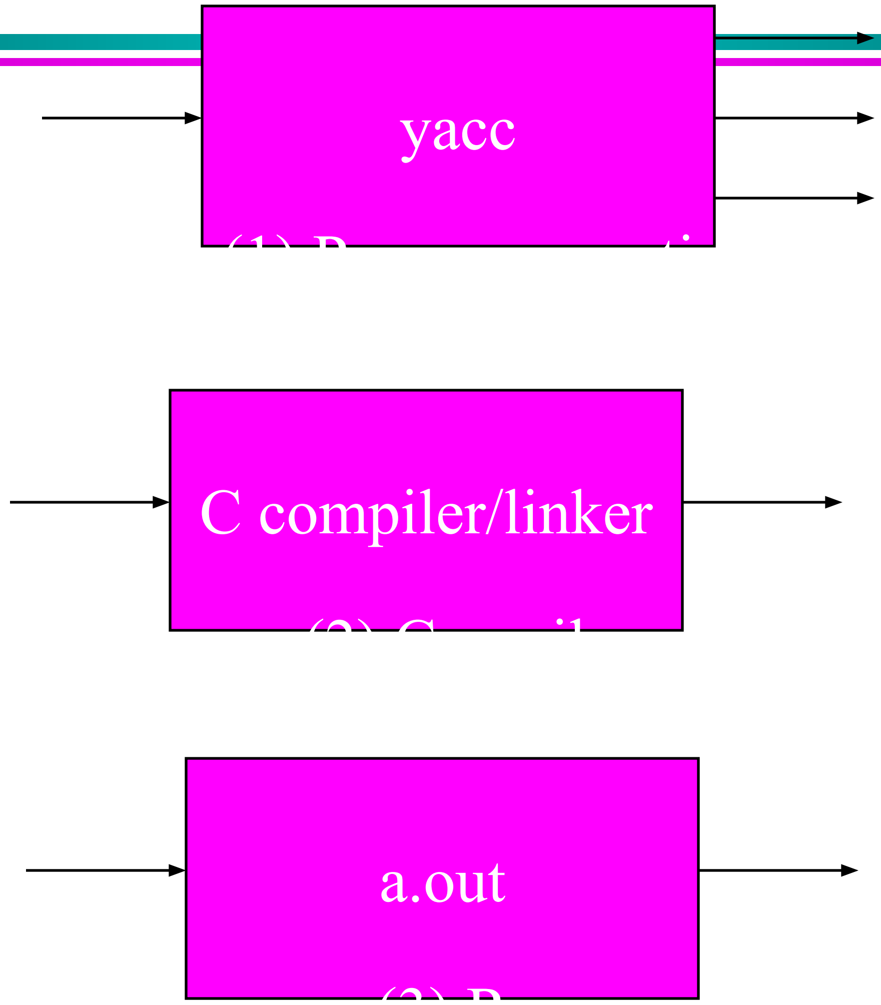
Introduction

- What is **YACC** ?
  - » **Tool which will produce a parser for a given grammar**.
  - » YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of the language produced by this grammar.

How YACC Works



File containing desired


yacc
C compiler/linker
a.out

An YACC File Example

```
%{
#include <stdio.h>
%}
%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression            { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

int main(void)
{
   yyparse();
   return 0;
}
```

Works with Lex

LEX

yylex()

YACC

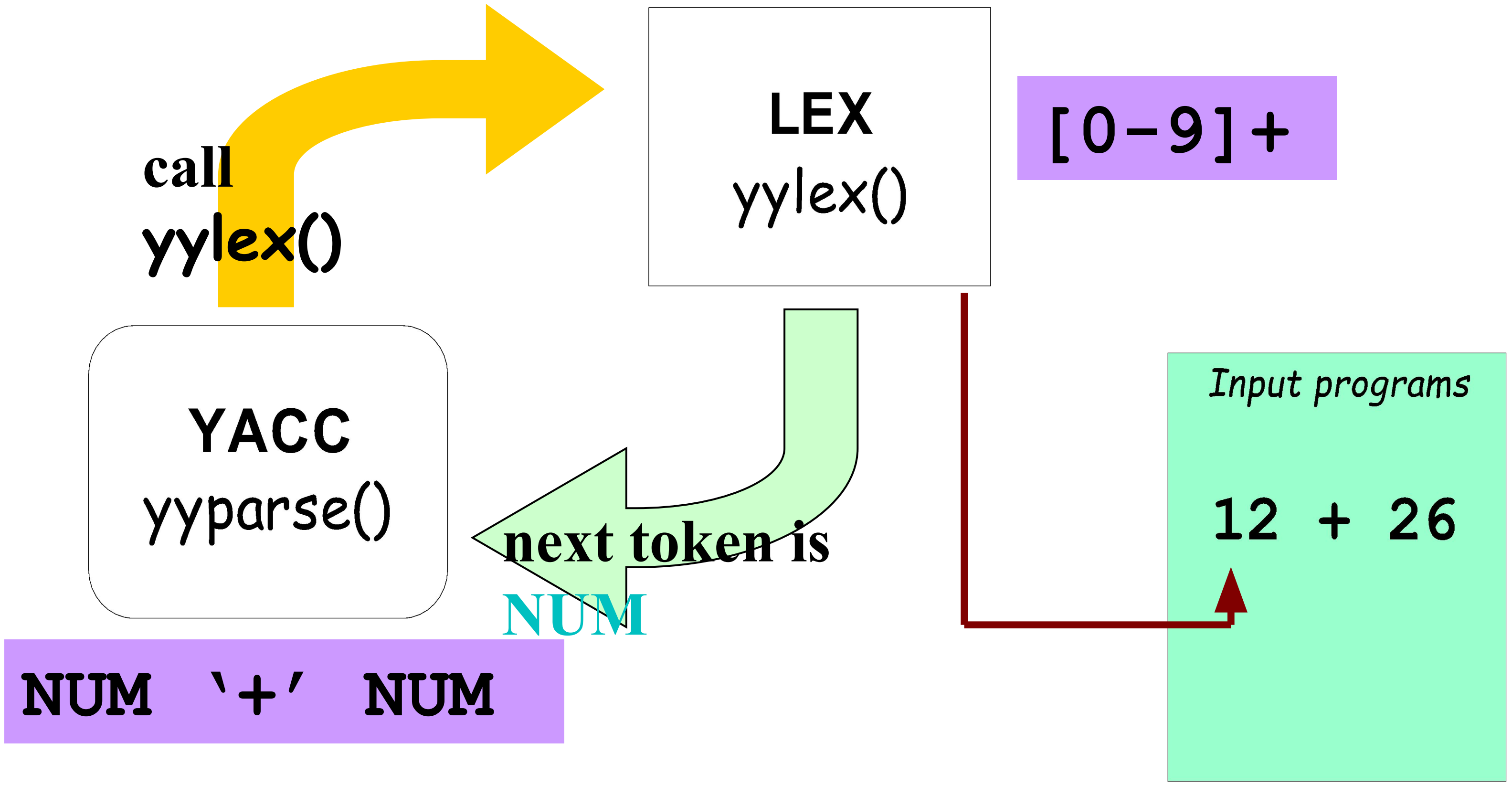
yyparse()

How to work ?

Input programs

12 + 26

Works with Lex



YACC File Format

```
%{
        C declarations
%}
        yacc declarations
%%
        Grammar rules
%%
        Additional C code
```

» **Comments enclosed in /* ... */ may appear in any of the sections.**

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

It is a terminal

由 expr 開始parse

Start Symbol

- The first non-terminal specified in the grammar specification section.
- To overwrite it with %start declaraction.

```
%start  non-terminal
```

Rules Section

- This section defines grammar
- Example

```
expr : expr '+' term | term;
term : term '*' factor | factor;
factor : '(' expr ')' | ID | NUM;
```

Rules Section

- Normally written like this
- Example:

```
expr    : expr '+' term
        | term
        ;
term    : term '*' factor
        | factor
        ;
factor  : '(' expr ')'
        | ID
        | NUM
        ;
```

The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

The Position of Rules

$1

```
expr : expr '+' term       { $$ = $1 + $3; }
     | term                { $$ = $1; }
     ;
term : term '*' factor     { $$ = $1 * $3; }
     | factor              { $$ = $1; }
     ;
factor : '(' expr ')'      { $$ = $2; }
       | ID
       | NUM
       ;
```

The Position of Rules

```
expr : expr '+' term       { $$ = $1 + $3; }
     | term                { $$ = $1; }
     ;
term : term '*' factor     { $$ = $1 * $3; }
     | factor              { $$ = $1; }
     ;
factor : '(' expr ')'      { $$ = $2; }
       | ID
       | NUM
       ;
```

$2

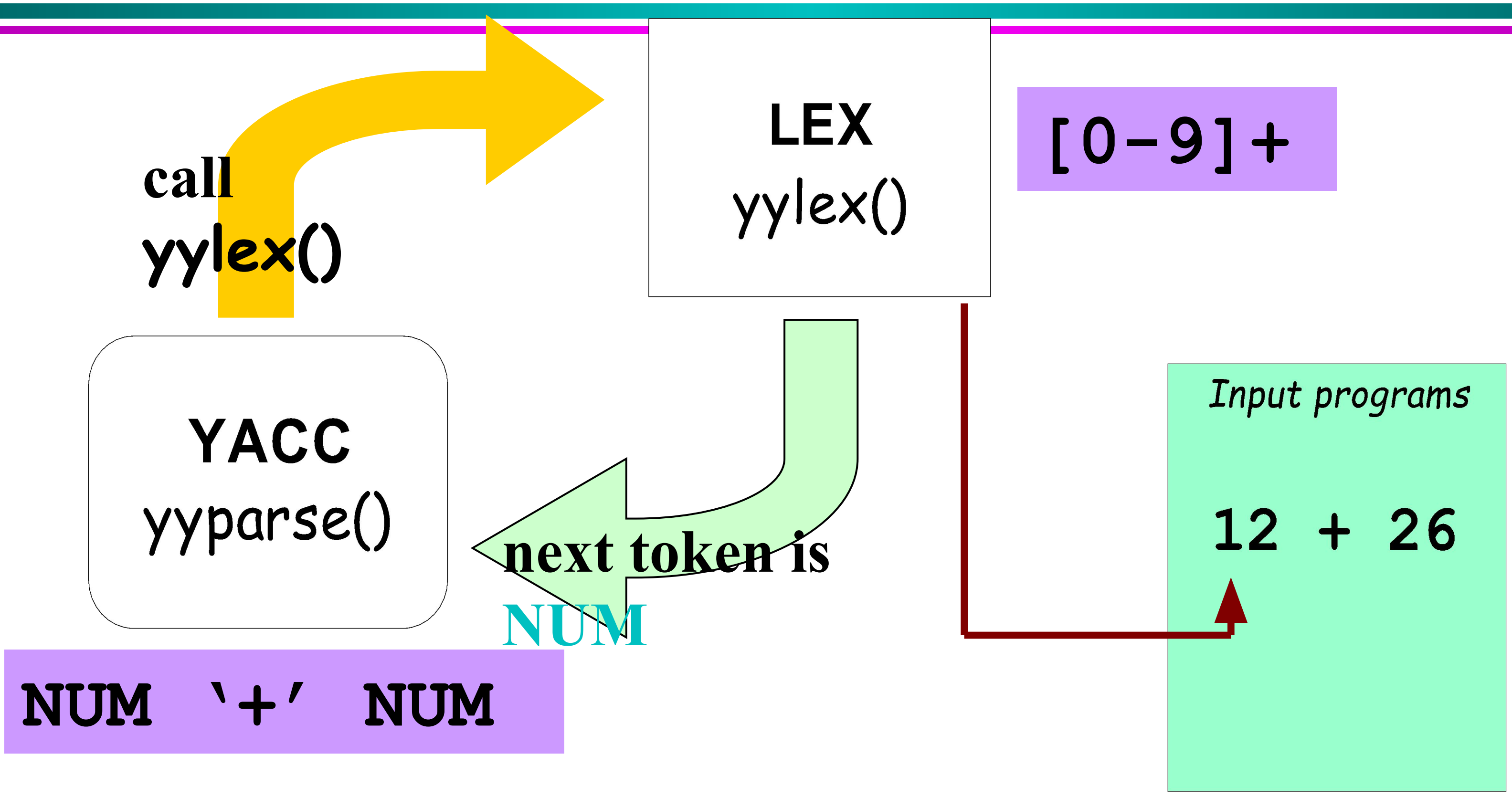
The Position of Rules

```
expr : expr '+' term      { $$ = $1 + $3; }
     | term               { $$ = $1; }
     ;
term : term '*' factor    { $$ = $1 * $3; }
     | factor             { $$ = $1; }
     ;
factor : '(' expr ')'     { $$ = $2; }
       | ID
       | NUM
       ;
```

$
3

Default: $$ = $1;

Communication between LEX and YACC



LEX and YACC需要一套方法確認 token的身份

Communication between LEX and YACC

yacc -d gram.y

Will produce:

y.tab.h

Communication between LEX and YACC

yacc –d xxx.y

scanner.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id          [_a-zA-Z][_a-zA-Z0-9]*
%%
int         { return INT; }
char        { return CHAR; }
float       { return FLOAT; }
{id}        { return ID;}
```

```
# define CHAR
  258
# define FLOAT
```

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token  CHAR, FLOAT, ID,
  INT
```

YACC

- Rules may be recursive
- Rules may be ambiguous*
- Uses bottom up Shift/Reduce parsing
  - » Get a token
  - » Push onto stack
  - » Can it reduced (How d
    - – If yes: Reduce
    - – If no: Get an her token
- Yacc cannot look ahead more than one token

Phrase -> cart_animal AND CART
| work_animal AND PLOW
…

Yacc Example

- Taken from Lex & Yacc
- Simple calculator

```
a = 4 + 6
a
a=10
b = 7
c = a + b
c
c = 17
$
```

Grammar

```
expression ::= expression '+' term |
               expression '-' term |
               term

term       ::= term '*' factor |
               term '/' factor |
               factor

factor     ::= '(' expression ')' |
               '-' factor  |
               NUMBER |
               NAME
```

Parser (cont'd)

```
statement_list:  statement '\n'
              |  statement_list statement '\n'
          ;

statement:    NAME '=' expression { $1->value = $3; }
         |    expression        { printf("= %g\n", $1); }
      ;

expression: expression '+' term { $$ = $1 + $3; }
          | expression '-' term { $$ = $1 - $3; }
          | term
          ;
```

parser.y

Parser (cont'd)

```
term:          term '*' factor { $$ = $1 * $3; }
     |         term '/' factor { if ($3 == 0.0)
                                    yyerror("divide by zero");
                                  else
                            $$ = $1 / $3;
                               }
     | factor
     ;

factor: '(' expression ')' { $$ = $2; }
      | '-' factor         { $$ = -$2; }
      | NUMBER             { $$ = $1; }
      | NAME               { $$ = $1->value; }
      ;
%%
```

parser.y

Scanner

```
%{
#include "y.tab.h"
#include "parser.h"
#include <math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
    yylval.dval = atof(yytext);
  return NUMBER;
  }

[ \t] ;       /* ignore white space */
```

Scanner (cont’d)

```
[A-Za-z][A-Za-z0-9]*  { /* return symbol pointer */
                          yylval.symp = symlook(yytext);
                          return NAME;
                      }

"$"{ return 0; /* end of input */ }

\n|″=‟|″+″|″-″|″*″|″/″  return yytext[0];
%%
```

YACC Command

- Yacc (AT&T)
  - » yacc –d *xxx.y*
- Bison (GNU)
  - » bison –d **–y** *xxx.y*

產生y.tab.c, 與yacc相同
不然會產生xxx.tab.c

Precedence / Association

```
expr: expr '-' expr
     | expr '*' expr
     | expr '<' expr
     | '(' expr ')'
      ...
     ;
```

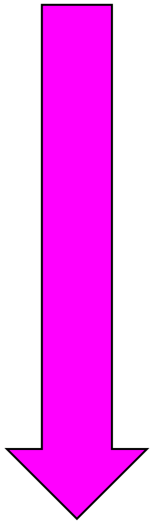
(1) 1 – 2 - 3

(2) 1 – 2 * 3

1. 1-2-3 = **(1-2)-3**? **or** 1-(2-3)?
   Define '-' operator is left-association.
2. 1-2*3 = 1-(2*3)
   Define "*" operator is precedent to "-" operator

Precedence / Association

```
%right '=`
%left  '<' '>' NE LE GE
%left  '+' '-`
%left  '*' '/'
```



Precedence / Association

[redacted]

```
expr   : expr '+' expr  { $$ = $1 + $3; }
    | expr '-' expr  { $$ = $1 - $3; }
    | expr '*' expr  { $$ = $1 * $3; }
    | expr '/' expr
              {
                 if($3==0)
                    yyerror("divide 0");
                 else
                    $$ = $1 / $3;
              }
    |  '-' expr %prec UMINUS {$$ = -$2; }
```

Shift/Reduce Conflicts

- **shift/reduce conflict**
  - » occurs when a grammar is written in such a way that a decision between shifting and reducing can not be made.
  - » ex: IF-ELSE ambigious.
- To resolve this conflict, yacc will choose to shift.

YACC Declaration Summary

`%start'

Specify the grammar's start symbol

`%union'

Declare the collection of data types that semantic values may have

`%token'

Declare a terminal symbol (token type name) with no precedence or
associativity specified

`%type'

Declare the type of semantic values for a nonterminal symbol

YACC Declaration Summary

`%right'

Declare a terminal symbol (token type name) that is right-associative

`%left'

Declare a terminal symbol (token type name) that is left-associative

`%nonassoc'

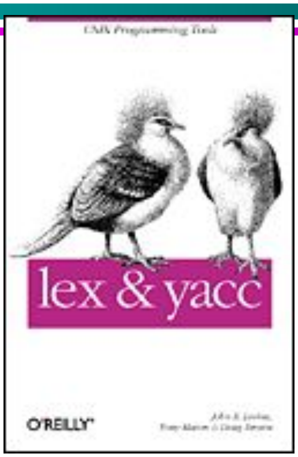
Declare a terminal symbol (token type name) that is nonassociative

(using it in a way that would be associative is a syntax error,

ex: x *op*. y *op*. z is syntax error)

Reference Books

- lex & yacc, 2nd Edition
  - » by John R.Levine, Tony Mason & Doug Brown
  - » O'Reilly
  - » ISBN: 1-56592-000-7



- Mastering Regular Expressions
  - » by Jeffrey E.F. Friedl
  - » O'Reilly
  - » ISBN: 1-56592-257-3

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

A typical compilation process

$$\begin{tikzcd}[row sep=small]
\text{Source program with macros} \arrow[d] \\
\boxed{\text{Preprocessor}} \arrow[d] \\
\text{Source program} \arrow[d] \\
\boxed{\text{Compiler}} \arrow[d] \\
\text{Target assembly program} \arrow[d] \\
\boxed{\text{assembler}} \arrow[d] \\
\text{Relocatable machine code} \arrow[d] \\
\boxed{\text{linker}} \arrow[d] \\
\text{Absolute machine code}
\end{tikzcd}$$

Try g++ with –v, -E, -S flags on linprog.

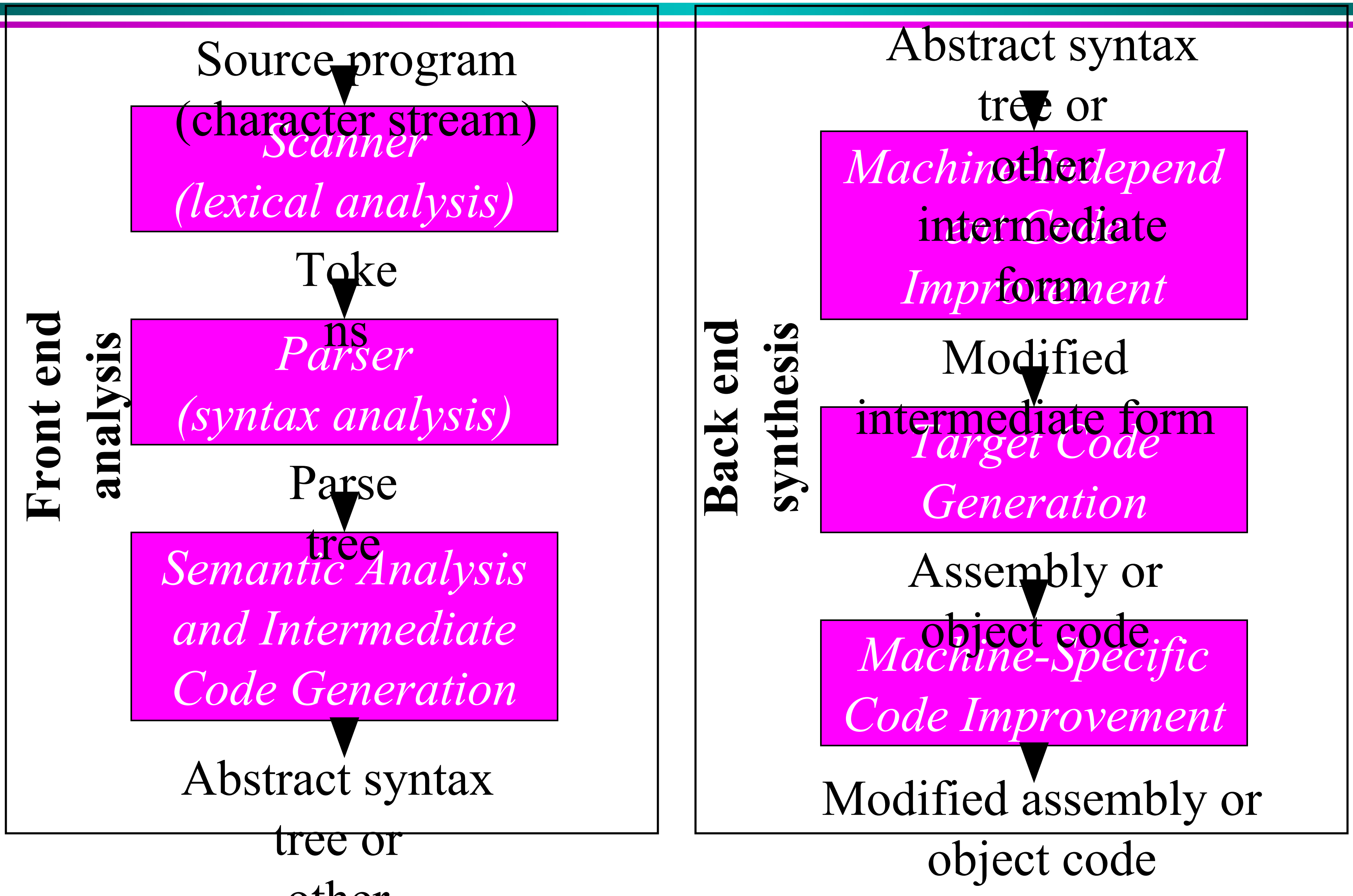
- What is a compiler?
  - » A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language).
    - – Two components
      - Understand the program (make sure it is correct)
      - Rewrite the program in the target language.
  - » Traditionally, the source language is a high level language and the target language is a low level language (machine code).

$$\begin{tikzcd}
\text{Source program} \arrow[r] & \boxed{\text{compiler}} \arrow[r] \arrow[d] & \text{Target program} \\
& \text{Error message} &
\end{tikzcd}$$

Compilation Phases and Passes

- Compilation of a program proceeds through a fixed series of phases
  - Each phase use an (intermediate) form of the program produced by an earlier phase
  - Subsequent phases operate on lower-level code representations
- Each phase may consist of a number of passes over the program representation
  - Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  - Single-pass compilers need less memory to operate
  - Java and ADA are multi-pass

Compiler Front- and Back-end


Front end analysis
Source program (character stream)
Scanner (lexical analysis)
Tokens
Parser (syntax analysis)
Parse tree
Semantic Analysis and Intermediate Code Generation
Abstract syntax tree or other
Back end synthesis
Abstract syntax tree or other intermediate form
Machine-Independent Code Improvement
Modified intermediate form
Target Code Generation
Assembly or object code
Machine-Specific Code Improvement
Modified assembly or object code

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 syntax error is produced by the compiler when the program does not meet the grammatical specification.
  - » For grammatically correct program, this phase generates an internal representation that is easy to manipulate in later phases
    - – Typically a syntax tree (also called a parse tree).
- A grammar of a programming language is typically described by a context free grammar, 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

<statement> ::= *<for-statement>* | *<if-statement>* | *<assignment>*

<for-statement> ::= **for (** *<expression>* **;** *<expression>* **;** *<expression>* **)** *<statement>*

<assignment> ::= *<identifier>* **:=** *<expression>*

Example: Micro Pascal

<Program> ::= **program** <id> **(** <id> <More_ids> **)** **;**
<Block> **.**

<Block> ::= <Variables> **begin** <Stmt>
<More_Stmts> **end**

<More_ids> ::= **,** <id> <More_ids>
| ε

<Variables> ::= **var** <id> <More_ids> **:** <Type> **;**
<More_Variables>
| ε

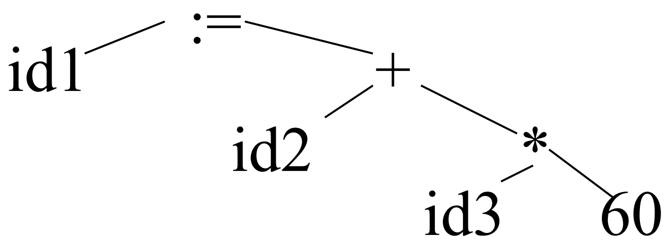
<More_Variables> ::= <id> <More_ids> **:** <Type> **;**
<More_Variables>
| ε

<Stmt> ::= <id> **:=** <Exp>
| **if** <Exp> **then** <Stmt> **else** <Stmt>

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.

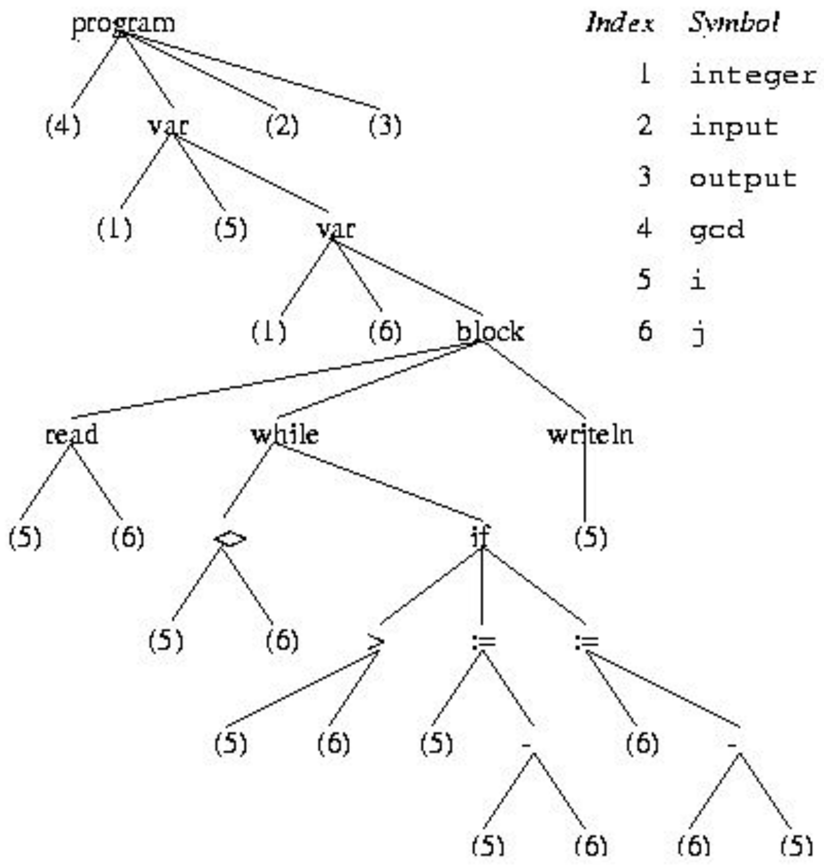
Semantic Analysis

- 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

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 = (int to float)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