Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Administrative info

- Instructor
  - Name: Arnold Meijster
  - E-mail: a.meijster@rug.nl
  - Office: Bernoulliborg 343
  - Phone: (+31 50 363)9246

- Helpdesk: **`ccrug1718@gmail.com`**

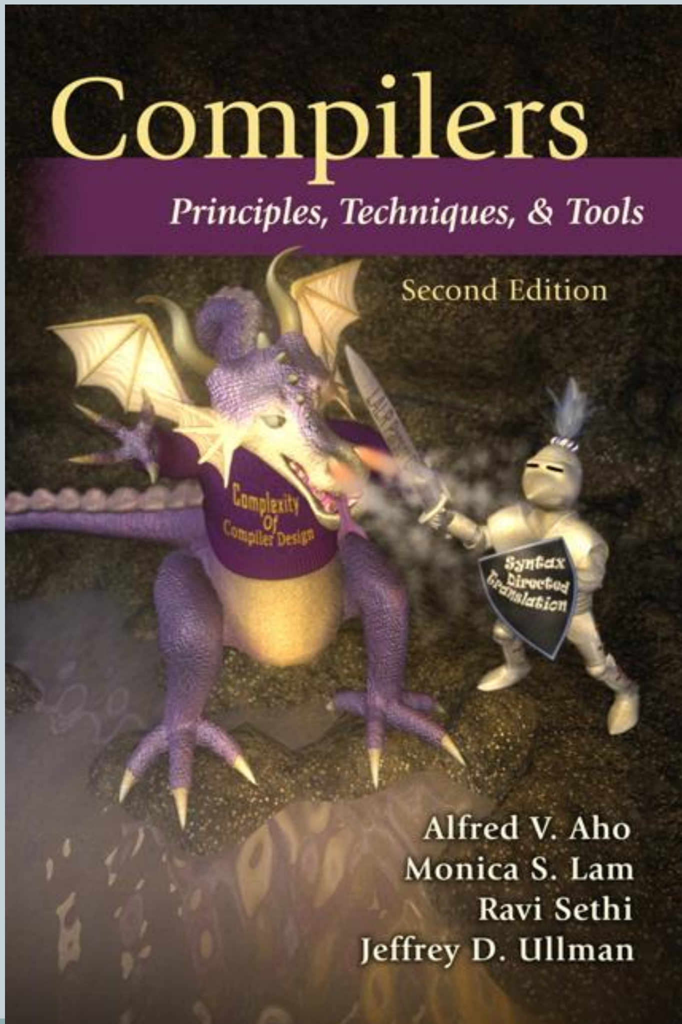
Organization

- Mondays: 9:00-10:45 lecture in BB 5161.289
  - Theory
- Wednesdays: 13:00-15:45 lecture/tutorial in BB 5161.222
  - Theory & Practice
  - Tutorial work form using computers
    - Pen and paper exercises
    - Computer exercises (bring a laptop with linux!)
- Tuesdays: 13:00-15:00 computer lab in 5161.208
  - Not in first week!

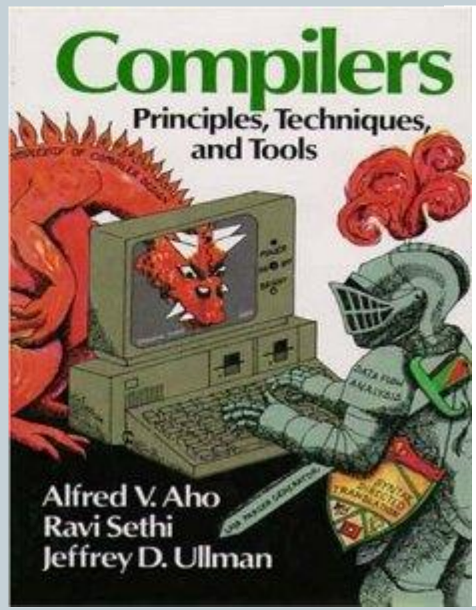
Grading

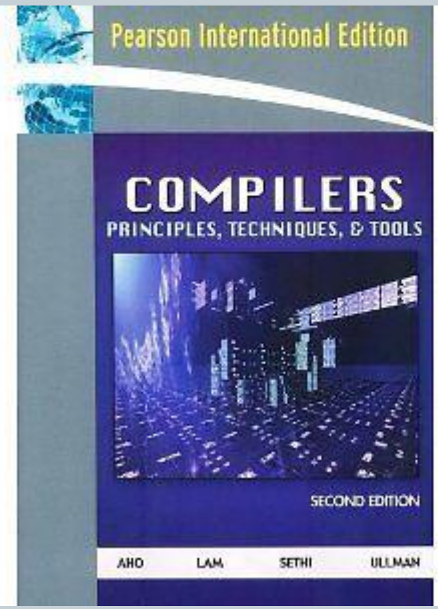
- Let L = mean of lab sessions
- Let E = grade written exam
- Final grade: F=0.6*E + 0.4*L
  - under the constraint that E,L>=4.5
  - otherwise F=Minimum(L,E)

Recommended (but not required) literature



The ‘dragonbook’: standard text on compiler construction





BUT, YOU DO NOT NEED THE BOOK!
The lecture slides + collection of weblinks (via Nestor) should suffice.

Goals

- understand the structure of a compiler
- understand how the components operate
- understand the tools involved
  - scanner generators, parser generators, code genreators, optimizers, etc.

What is a compiler?

- A program that converts a program written in some (source) language into an (equivalent) program written in some (destination) language.
  - C compiler: C source -> Assembly
  - Assembler: Assembly -> machine code
  - Source to source:
    - p2c: Pascal -> C compiler
    - f2c: Fortran -> C compiler

Why study compilers?

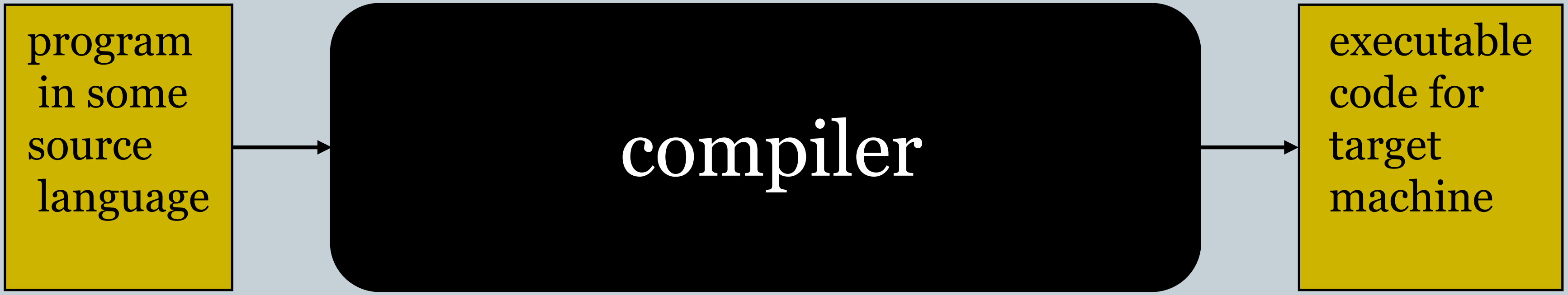
- Application of a wide range of theoretical techniques
  - Language and automata theory
  - Parsing input
  - Algorithms & Data Structures
  - Computer Architecture
- Understanding of mapping of high level programming constructs onto low level machine instructions
- Good software engineering experience!
  - Building a compiler is a major project!
  - Intel has hundreds of programmers continuously working on their compilers!

Quality characteristics of a compiler

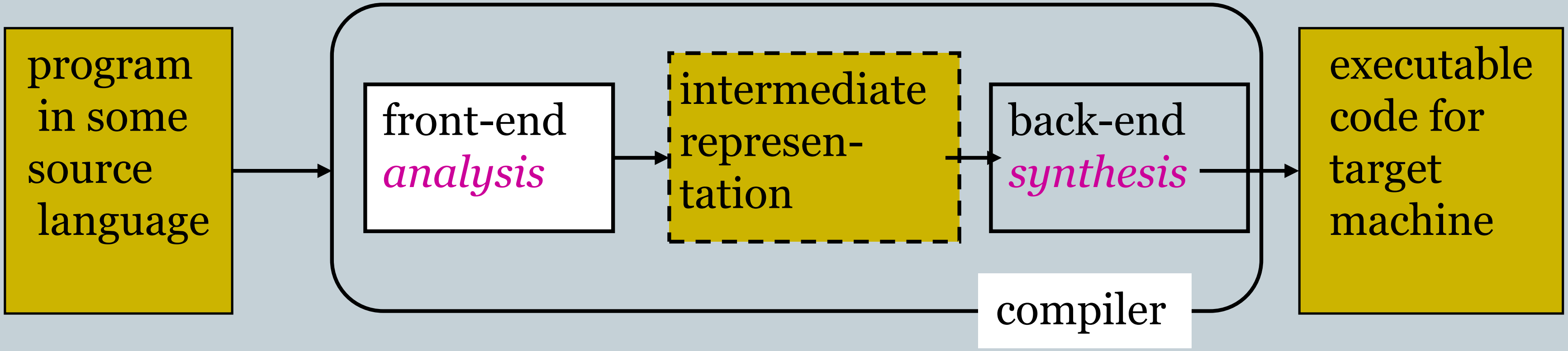
In decreasing order of importance:

- Correctness
  - preserve the meaning of the code
- Speed of target code
- Good error reporting/handling
- Speed of compilation

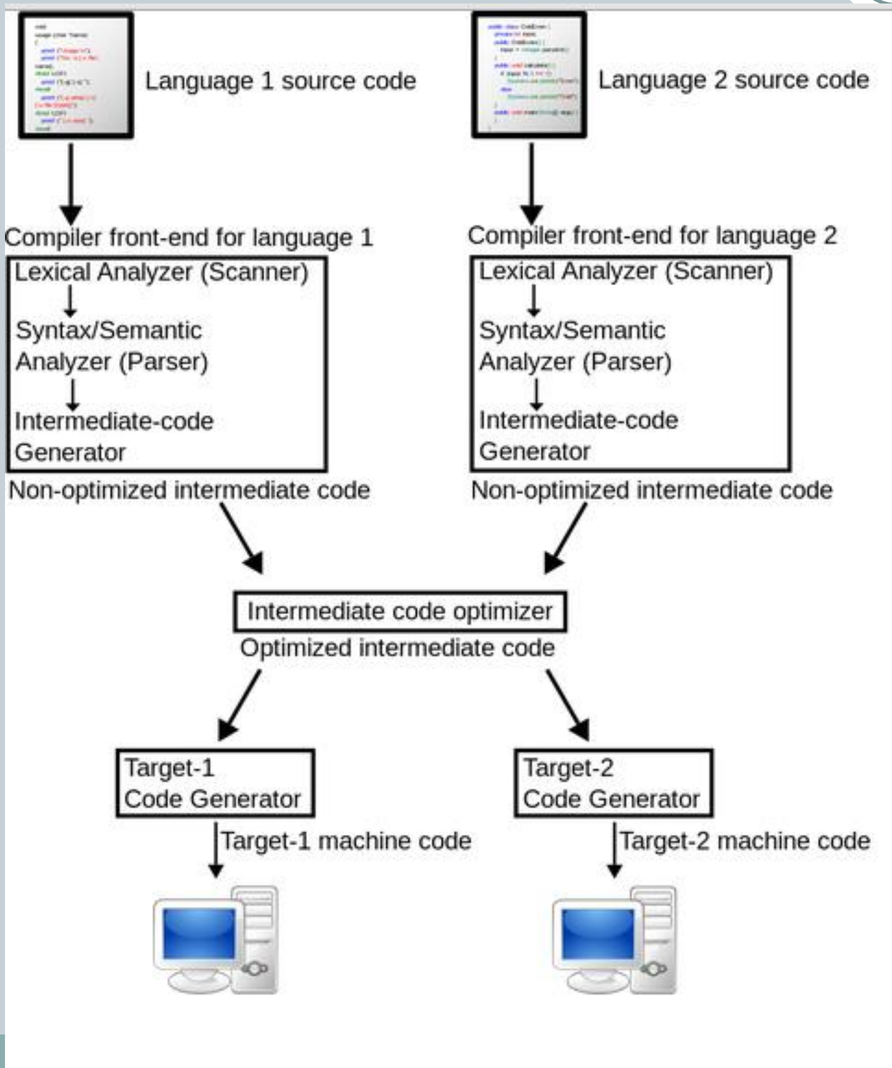
Compiler structure



Compiler structure



Why intermediate code?



Two main reasons:

1) Multiple front ends can use the same back end (gnu compilers do this, e.g. gcc / g++ / gpc / gfortran all use the same backend)

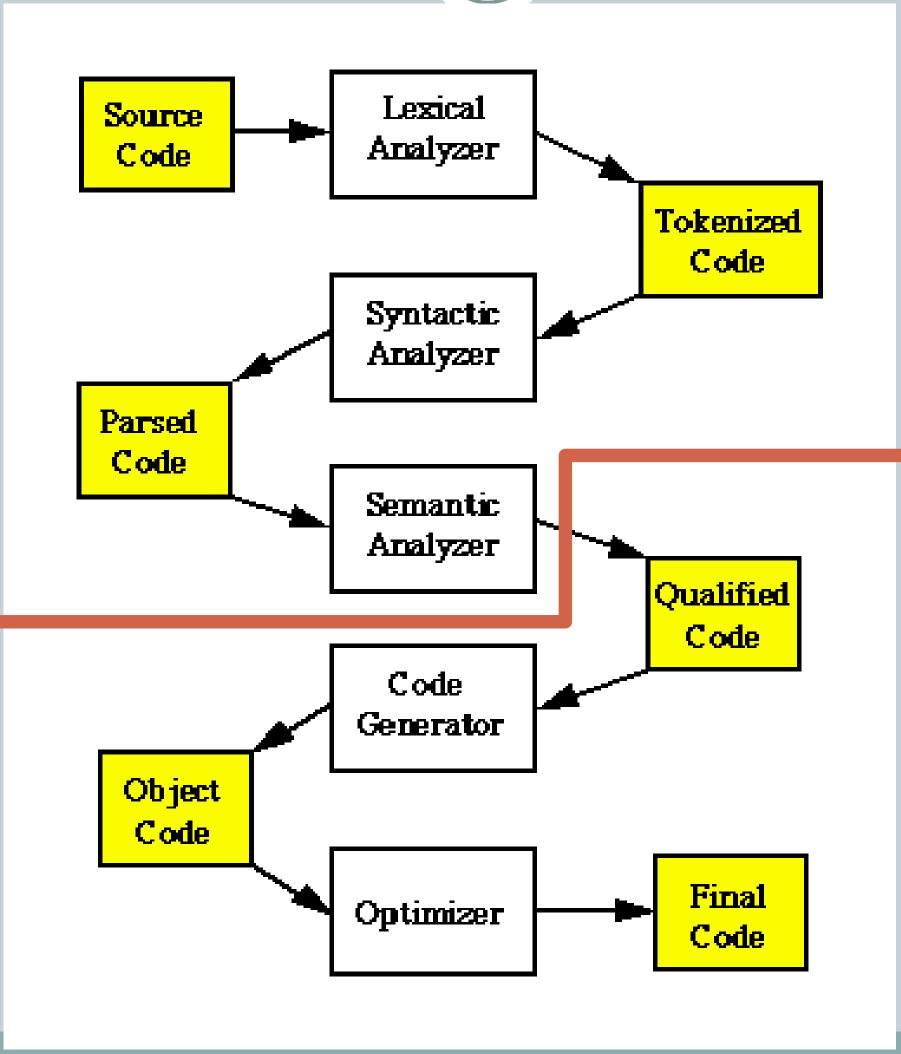
2) The compiler is easier to port to different CPUs/architectures. Only the code generator of the back end needs to be changed. Such a compiler is called a *retargetable compiler*.

Compiler Structure

- Front end
  - Recognize legal/illegal programs
    - report/handle errors
  - Generate IR (Intermediate Representation)
  - Building the front of a compiler is done with automated tools
- Back end
  - Translate IR into target code
    - instruction selection
    - register allocation
    - instruction scheduling
    - optimization
    - lots of NPC problems -- use approximations

Compiler Structure

Front end



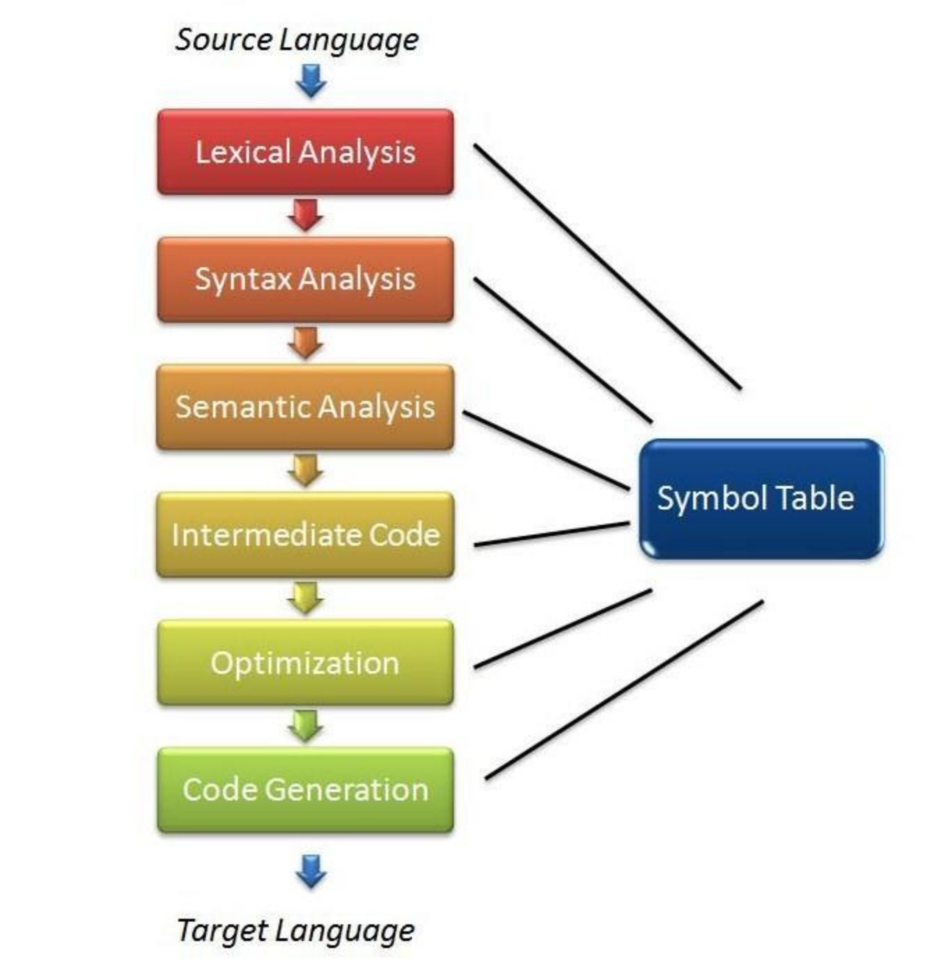
Back end

The Front End

- Scanner (a.k.a. lexical analyzer)
  - recognize "words" and convert them to tokens
  - Keeps track of line/column position
  - Serves the parser (pipeline structure, although often not strict)
- Parser (a.k.a. syntactic analyzer)
  - checks syntax
- Semantic analyzer
  - type checking
  - are variables/functions declared?
  - block structure (local/global variables)
  - ....

- Other issues:
  - symbol table (to keep track of identifiers)
    - Typically implemented as a hashtable
  - error detection/reporting/recovery

Compiler Structure: Symbol table



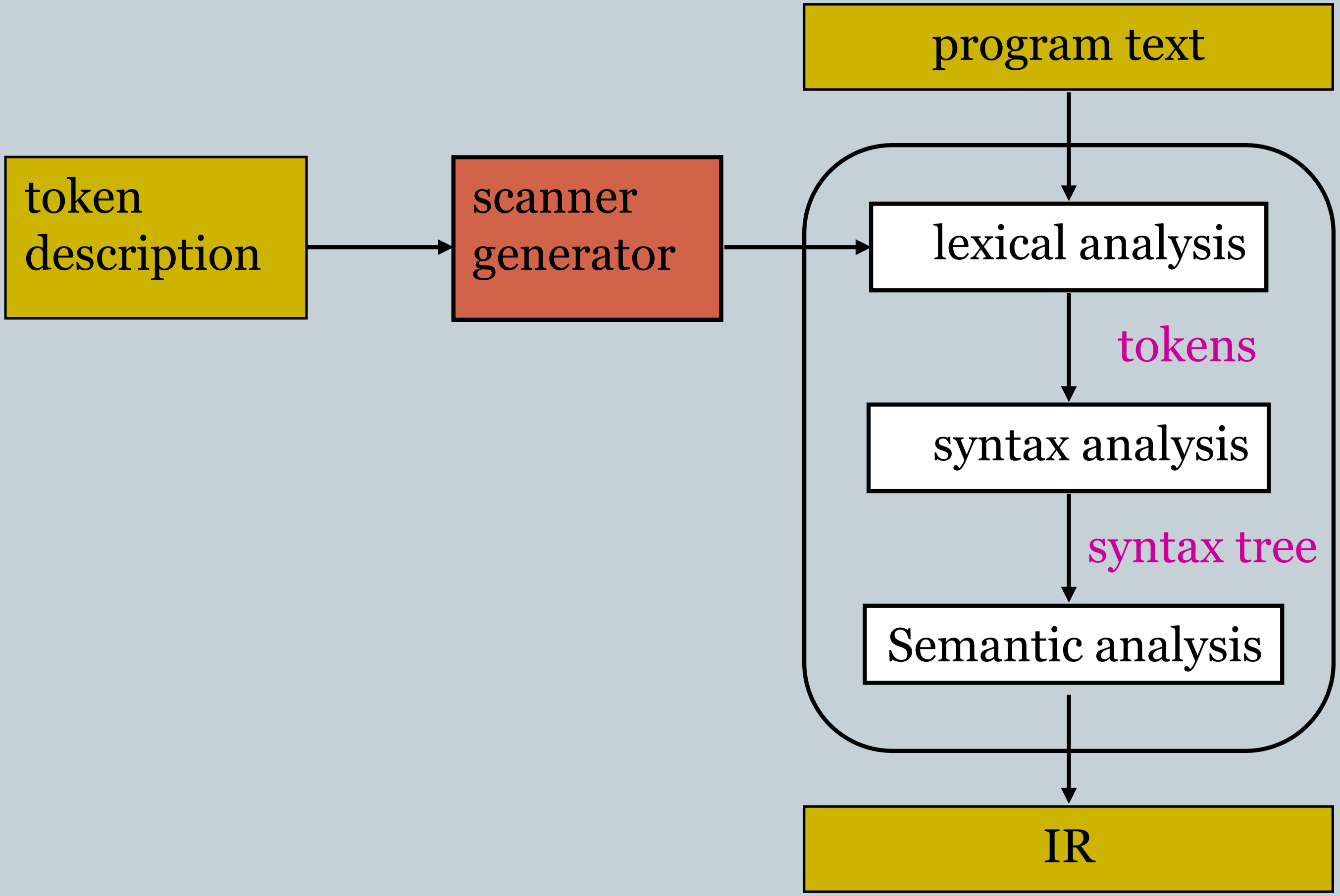
- Most compilers are not organized as a perfect pipeline. This is due to the use of a symbol table.
- Example: the first time that the scanner recognizes some identifier it inserts it in the symbol table and returns the token **`IDENTIFIER`**. The next time it finds it in the symbol table it will return **`VARIABLE`** or **`FUNCNAME`** (because the parser added info in the symbol table).

The Scanner/lexer

- Its job:
  - produce token stream
    - e.g. x = 1; becomes the token sequence IDENTIFIER EQUAL INTEGER SEMICOLON
  - retrieve/store identifier information
    - e.g. IDENTIFIER corresponds to a *lexeme* (the actual word x)
    - e.g. look up IDENTIFIER in symbol table (and return VARIABLE instead)
  - ignore white space and comments (depends on language)
  - report errors (illegal characters)

- Good news
  - the process can be automated (using scanner generators: e.g. **`lex/flex`**, **`DFASTAR`**, re2c, …)

The Front End

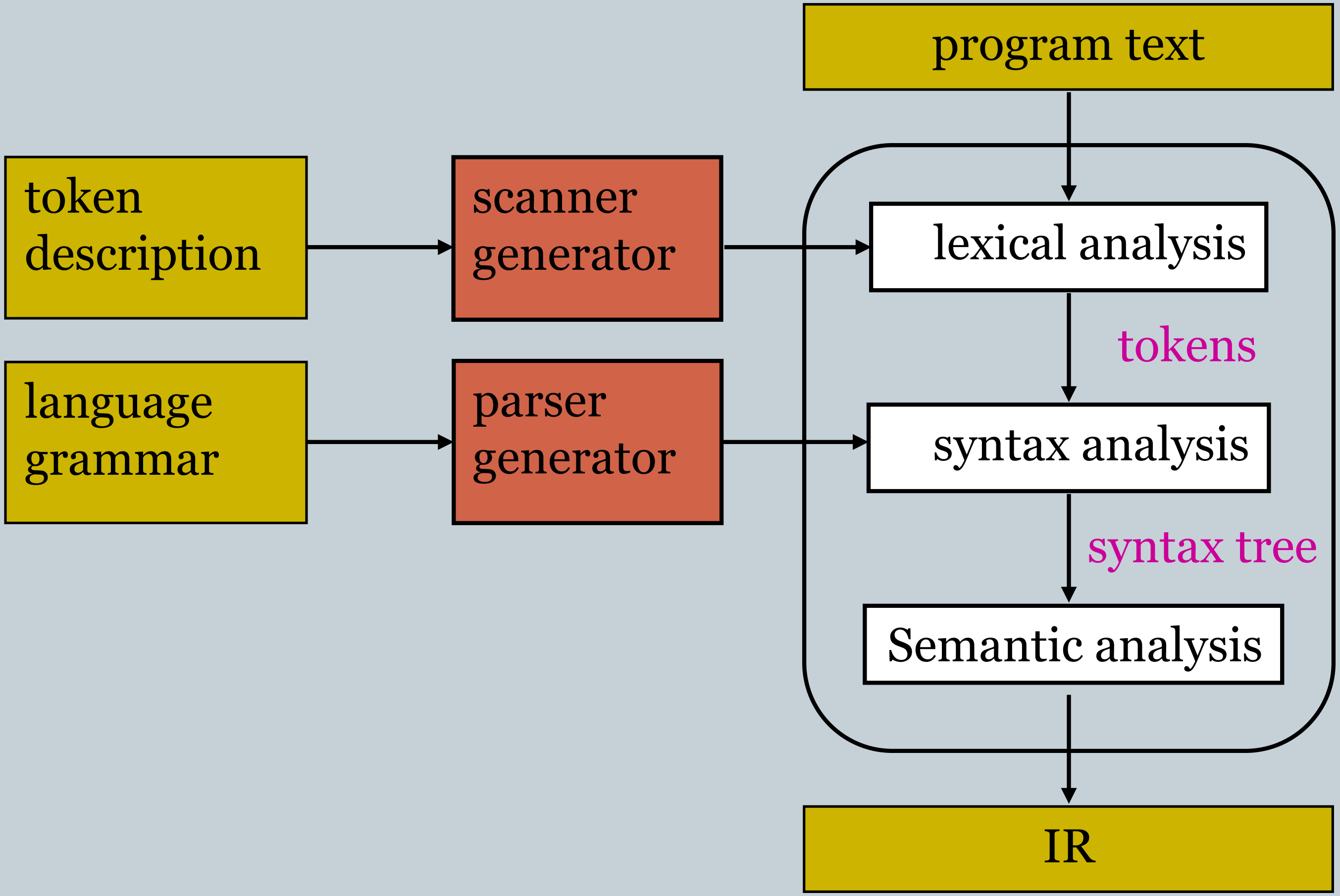


The Parser

- Its job:
  - Check and verify syntax based on specified syntax rules
    - e.g. IDENTIFIER LPAREN IDENTIFIER RPAREN makes up a function call (expression).
    - Coming soon: how context-free grammars specify syntax
  - Report syntax errors/warnings
  - Produce abstract program representation
    - often a syntax tree

- Good news
  - the process can be automated (using parser generators: **`llnextgen, bison/yacc`**, …)

The Front End



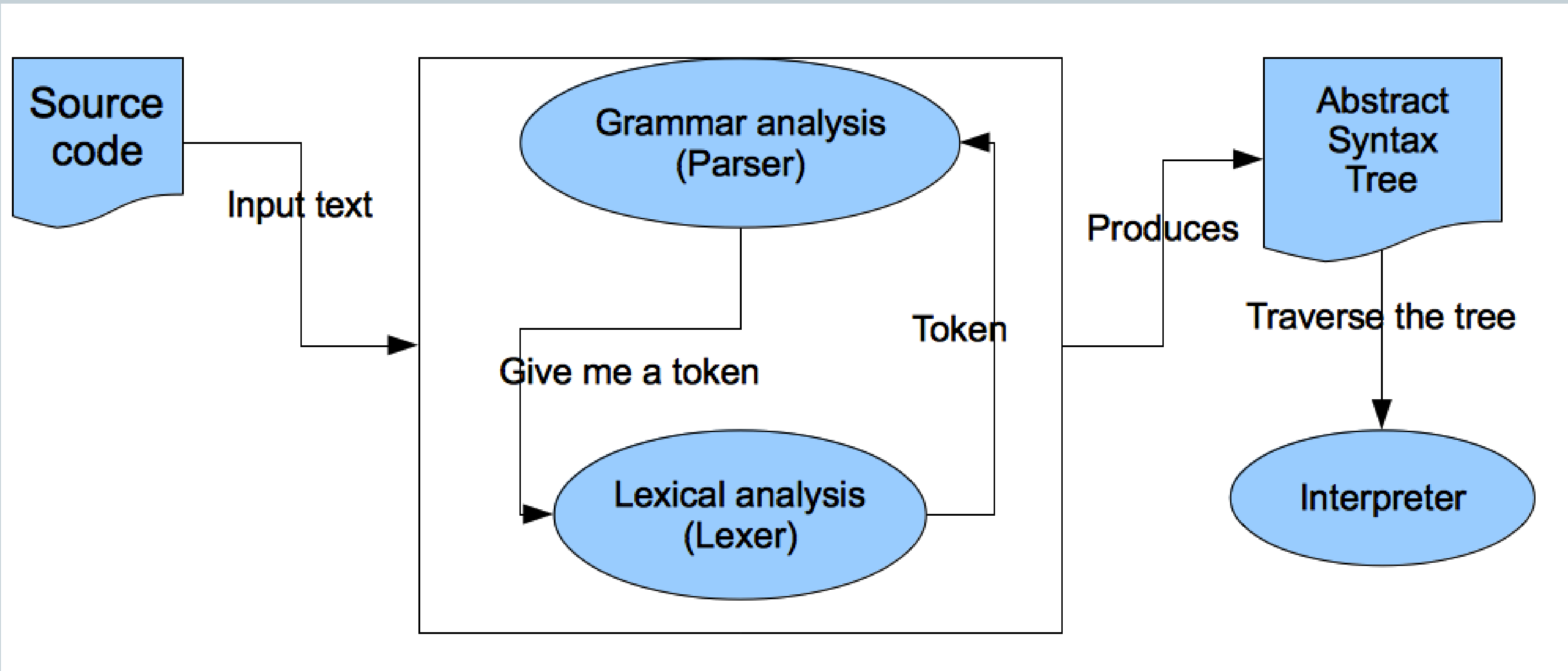
Why the separation Lexer/Parser?

- Simplicity of design / separation of concerns
  - e.g. a parser that has to incorporate skipping whitespace/comments is more complex.
  - Input specific peculiarities are only of concern to the lexer.
- Efficiency
  - The parser can process the input at a courser level (tokens instead of characters)
  - The parser should not backtrack (extensively)
  - The scanner uses buffering techniques

Semantic analyzer

- Its job:
  - Check the meaning (semantics) of the program
    - e.g. In x=y, is y defined before being used? Are x and y declared?
    - e.g. In x=y, are the types of x and y such that you can assign one to the other?
  - Understand block structure (local/global variables)
  - Report errors/warnings
  - Produce intermediate representation
- Bad news
  - the process cannot be automated

An interpreter (no back end)



Lexical Analysis

The rest of todays lecture is about lexical Analysis.

C Code

```c
while (count <= 100) { /** some loop */
   count++;
   // Body of while continues
   ...
```

tokenizing

Tokens

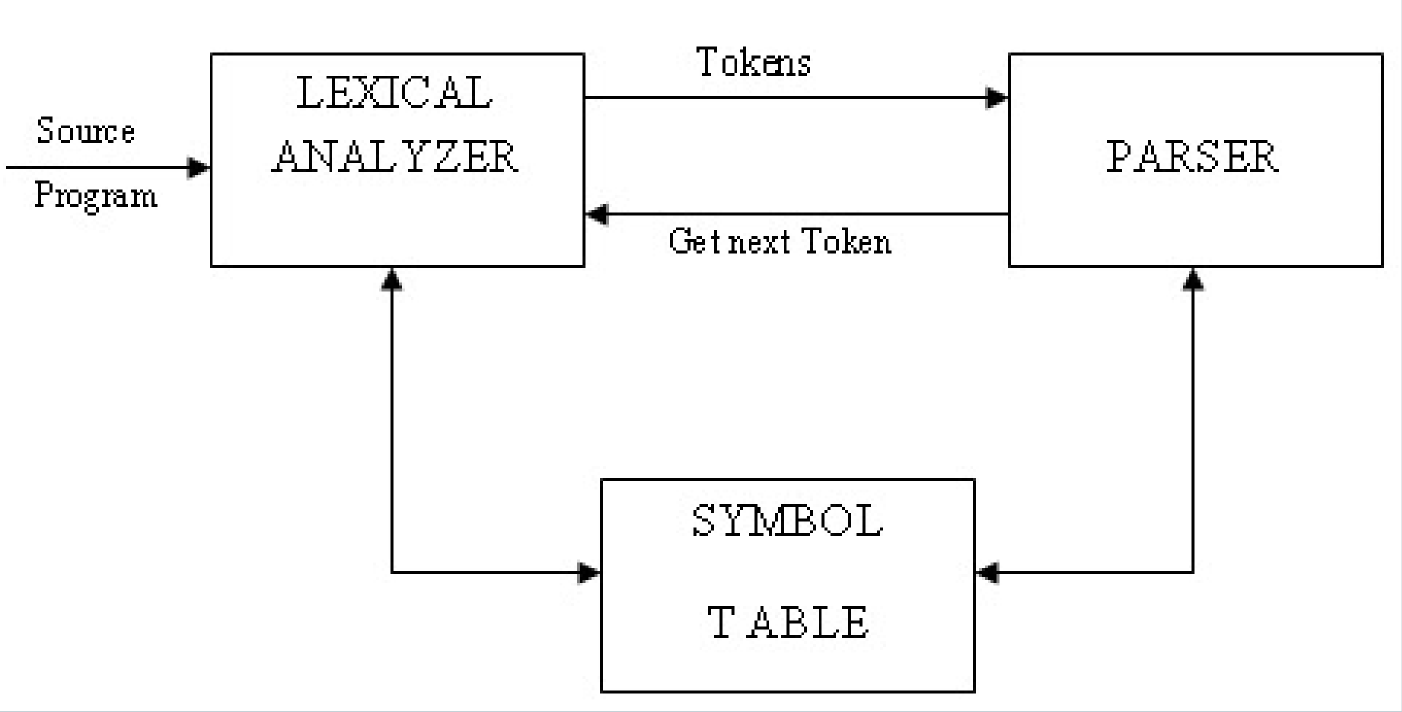
```
while
(
count
<=
100
)
{
count
++
;
...
```

Lexical analysis

- Role of the lexical analyzer:
  - Read input characters and group them into *lexemes*.
  - Produce a sequence of *tokens* (one token for each lexeme)
  - Interact with the symbol table
    - When an identifier is recognized, insert it in the symbol table
    - Deletions are done by the parser. When?
  - Skipping *whitespace* (space, newline, tab)
  - Skipping comments, like /* .. */
  - Keeping track of line/column for error messages
  - Possibly, expansion of macros
    - in C, the C-preprocessor takes care of this
  - The parser calls the scanner: **getNextToken()**

Lexical analysis

The parser calls the Scanner: **getNextToken()**



Tokens/Lexemes

```
printf("score=%d\n", score);
```

Token	Lexeme
IDENTIFIER (or FUNCNAME)	printf
LPAR	(
STRING	"score=%d\n"
COMMA	,
IDENTIFIER (or VARNAME)	score
RPAR	)
SEMICOLON	;

Tokens/Lexemes

We have some freedom in choosing granularity of tokens.

E.g. consider the C statement: `y = x >= 0 && x < 42;`

Token	Lexeme
IDENTIFIER	y
EQUAL	=
IDENTIFIER	x
GREQ	>=
LOGICAND	&&
IDENTIFIER	x
LESSTHAN	<
INTCONST	42
SEMICOLON	;

Token	Lexeme
IDENTIFIER	y
EQUAL	=
IDENTIFIER	x
COMPARE	>=
LOGICOP	&&
IDENTIFIER	x
COMPARE	<
INTCONST	42
SEMICOLON	;

Tokens/Lexemes

Beware! The following example is not a good choice of tokens!

The scanner is not the problem, but this yields problems in the parser!

```
while(x < 42) x++;
```

Token	Lexeme
IDENTIFIER	while
LPAR	(
IDENTIFIER	x
COMPARE	<
INTCONST	42
RPAR	)
IDENTIFIER	x
INCREMENT	++
SEMICOLON	;

Token attributes

Besides a token, the scanner returns (usually) also:

- Lexeme (string)
- Line/Column location
- Id: Index/pointer to entry in symbol table

In the symbol table entry symbol information is stored:

- Lexeme
- Type
- Line of first appearance (for error reporting)
- …

```
typedef struct {
  int type;
  char *lexeme;
  file_pos position;
  …
} SymtabEntry;
```

Lexical error reporting

- Very few errors can be detected by a lexer!
  - typos are not detected
    - `fi (x < 42) x = 42; /* fi yields an IDENTIFIER */`
  - It can detect illegal characters!
    - usual solution, *abort*
    - or *panic mode recovery* (delete offending chars)
    - fancy: repair input by insertions and deletions. This is complicated and expensive. Usually not worth the effort.

Input buffering in lexers

- We need to buffer input.
  - E.g. After having read < we need to read an extra character to see if the lexeme is <=
  - If the next character is not =, then we should backtrack and memorize the character!
  - Beware: can we run out of buffer space?

Theory: Alphabet, strings and languages

- An *alphabet* is a finite set of characters.
  - e.g. the alphabet {'0', '1'} is the set of binary characters
  - e.g. the ASCII character set (or the unicode character set)
- A *string* over an alphabet is a finite sequence of chars drawn from that alphabet.
- The *empty string* is denoted by ε (sometimes λ).
- The *length* of a string *s* is denoted as $|s|$.
  - e.g. |while|=5 and |ε|=0.
- A *language* is a set of strings over an alphabet.
  - Note that this is not the daily-life notion of a language!

String concatenation

- Let $x =$ 'ba', $y =$ '$nana$'

- The notation xy (or $x \cdot y$) denotes x appended with y
  - x y = x · y = '$banana$'
  - Similar notation as product, which suggests the notation
    - $s^0 = \epsilon$, $s^i = s \cdot s^{i-1}$ for $i > 0$

Operations on languages

- *Union*: $A \cup B = \{x \mid x \in A \vee x \in B\}$
- *Concatenation*: $A \cdot B = \{x \cdot y \mid x \in A \wedge y \in B\}$
  - $A^0 = \{\epsilon\}, \; A^i = A \cdot A^{i-1}$
- *Kleene closure*: concatenate zero or more times
  - $L^* = \cup_{i=0}^{\infty} L^i = \{x \mid x \in L^i \text{ where } i \geq 0\}$
  - example: $L = \{ab, c\}$, $L^* = \{\epsilon, ab, c, abab, abc, cab, cc, \ldots\}$
  - $L^+ = \cup_{i=1}^{\infty} L^i$, i.e. at least once.
  - Note that $\epsilon \in L^+ \Leftrightarrow \epsilon \in L \Leftrightarrow L^+ = L^*$

Examples

Let $L = \{a, b, c, .., z, A, B, .., Z\}$ and $D = \{0,1,2, .., 9\}$

- $L \cup D$ is the set of letters and digits.
- L^3 is the set of all length 3 strings of letters
- $L \cdot D$ is the set with 520 strings of length 2, each consisting of a letter followed by a digit
- $L(L \cup D)^*$ is the set of all strings beginning with a letter followed by zero or more letters and digits.

Regular expressions (regexps)

- Notation to describe all languages that can be built from the operations *Union*, *Concatenation* and *closure* applied to some alphabet.

- Example: identifiers in C
  - $(L \cup \{\_\}) \cdot (L \cup D \cup \{\_\})^*$
  - Regexp notation: (L | '_')(L | D | '_')*

Regular expressions (regexps)

Regular expressions over the alphabet Σ are defined recursively:

- ϵ is the regular expression that denotes the language $L(\epsilon) = \{\epsilon\}$.
- If $a \in \Sigma$, then a is the regular expression for the language $L(a) = \{a\}$.

And if r and s are regular expressions, then:

- $r|s$ is the regular expressions that denotes the language $L(r) \cup L(s)$.
- rs is the regular expressions that denotes the language $L(r) \cdot L(s)$.
- r^* is the regular expressions that denotes the language $L(r)^*$.

Priority: * highest, concatentation second highest, | lowest

Example: ab*|cd = (a(b*))|(cd)

Examples

- $a|b$ denotes the language $\{a, b\}$
- $(a|b)(a|b)$ denotes the language $\{aa, ab, ba, bb\}$
- a^* denotes the language $\{\epsilon, a, aa, aaa, aaaa, \dots\}$
- $a|a^*b$ denotes the language $\{a, b, ab, aab, aaab, \dots\}$

Laws of regexps

- $r|s = s|r$
- $r|(s|t) = (r|s)|t = r|s|t$
- $r(st) = (rs)t = rst$
- $r(s|t) = rs|rt$
- $(s|t)r = sr|tr$
- $\epsilon r = r\epsilon = r$
- $r^* = (r|\epsilon)^*$
- $r^{**} = r^*$
- $r^*r^* = r^*$

Regular definitions

- Sequence of definitions of the form:
  - $d_i \rightarrow r_i$ where each d_i is a unique symbol (not in alphabet) and r_i a regular expression over the alphabet.

Example: unsigned numbers

  - unsigned $number \rightarrow digits\ fraction\ exponent$
  - $digits \quad \rightarrow digit(digit)^*$
  - $digit \quad \rightarrow 0|1|2|3|4|5|6|7|8|9$
  - $fraction \rightarrow \epsilon|\ .digits$
  - $exponent \rightarrow \epsilon|\ (E|e)(+|-|\epsilon)digits$

Regular definitions (extensions)

- One or more instances: notation +
- Zero or one instance (optional) : notation ?
- Classes $[a_1|a_2| \dots |a_n]$: notation $[a_1 - a_n]$

Example: unsigned numbers

- $unsigned\ number \rightarrow digits\ (.\,digits)?\ ([Ee][+\,-]?\ digits)?$
- $digits \rightarrow digit^{+}$
- $digit \rightarrow [0-9]$

Recognition of tokens

Consider the following grammar for IF-THEN-ELSE statements:

stmt	->	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
expr	->	*term* **relop** *term*
	|	*term*
term	->	**identifier**
	|	**number**

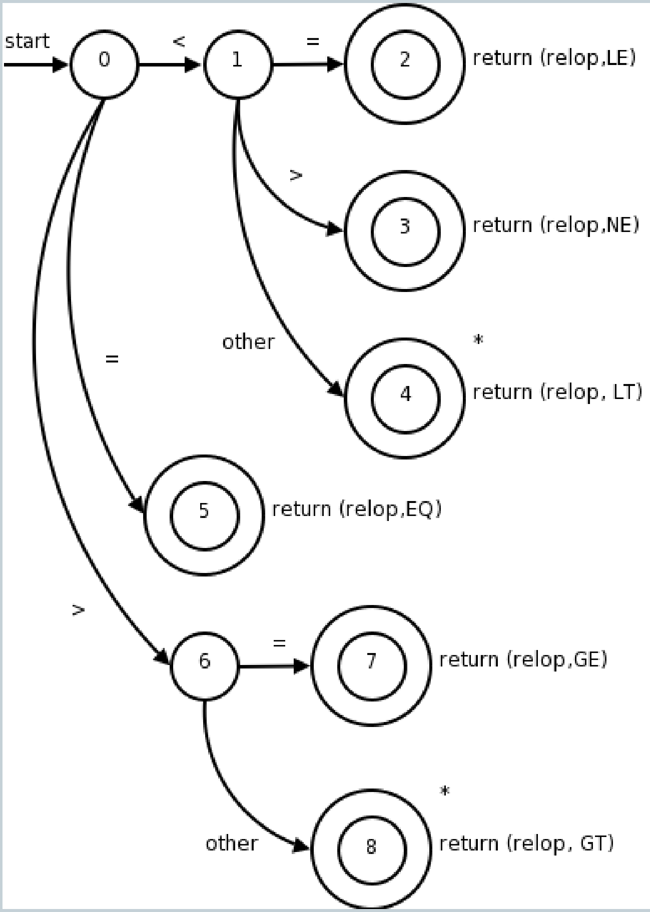
Recognition of tokens

- We need to recognize the following tokens:
  - $number \rightarrow digits\ (.\,digits)?\ ([\mathrm{Ee}][+\,-])?\ digits)?$
    - $digit \rightarrow$[0-9]
    - $digits \rightarrow digit^{+}$
  - $identifier \rightarrow letter(letter|digit)^{*}$
    - $letter \rightarrow$[A-Za-z]
  - $if \rightarrow \mathrm{if}$
  - $then \rightarrow \mathrm{then}$
  - $else \rightarrow \mathrm{else}$
  - $relop \rightarrow$ < | > | <= | >= | = | <>
- We also want to skip whitespace (no token is returned):
  - $ws \rightarrow$(blank |tab | newline)$^{+}$

Transition diagrams

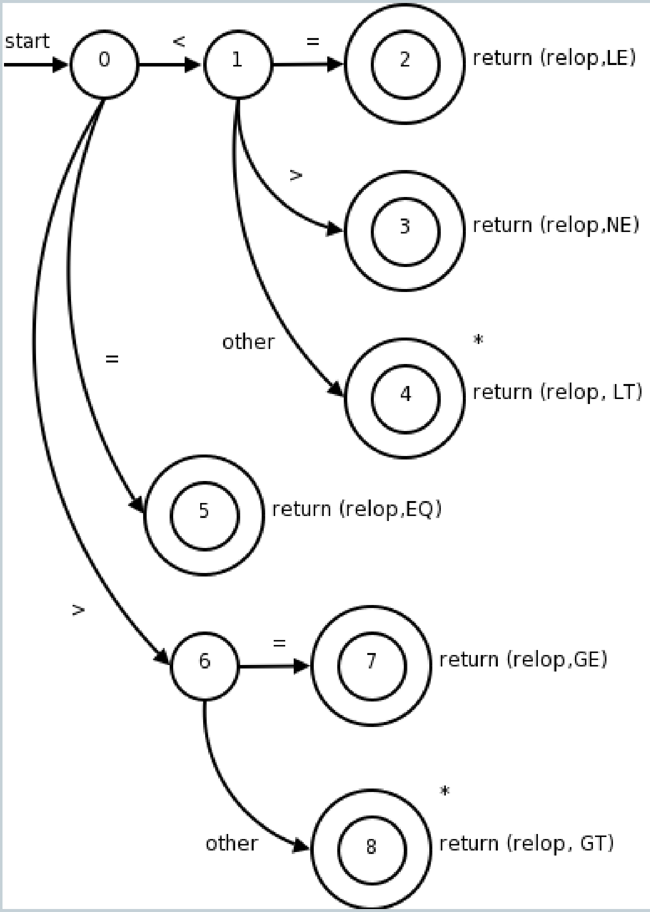
Regexps can be converted into *transition diagrams*.

- E.g. The transition diagram for a *relop*
- Nodes/circles are called *states*.
- One state is labeled *start*, it is the initial state.
- *Edges* denote the possible transitions, labeled with the corresponding character(s).
- A state defines what we have 'seen' between the beginning of the lexeme and the current character position.

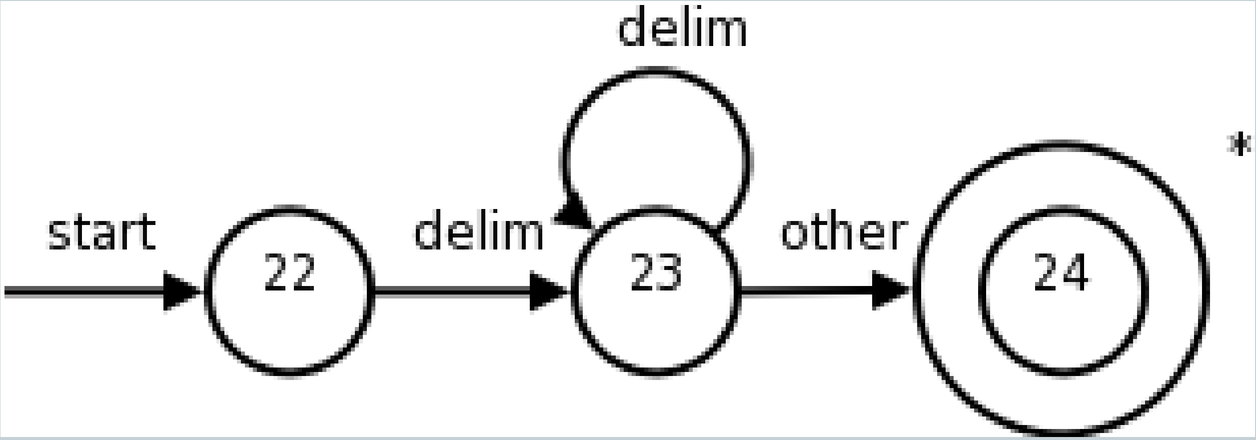


Transition diagrams

- Being in some state *s*, we can accept a character *a* if there is an outgoing edge from *s* labeled with an *a*. In that case, the lexer advances the current character pointer, and gets in the next state.
- States with double circles are *accepting states* (or *final states*). In these states a lexeme has been recognized.
- An accepting state can have an *action* associated with it, e.g. return some token.
- In an accepting state that is labeled with a star (*) (or multiple stars), the lexer needs to retract the character pointer one (or multiple) position(s).

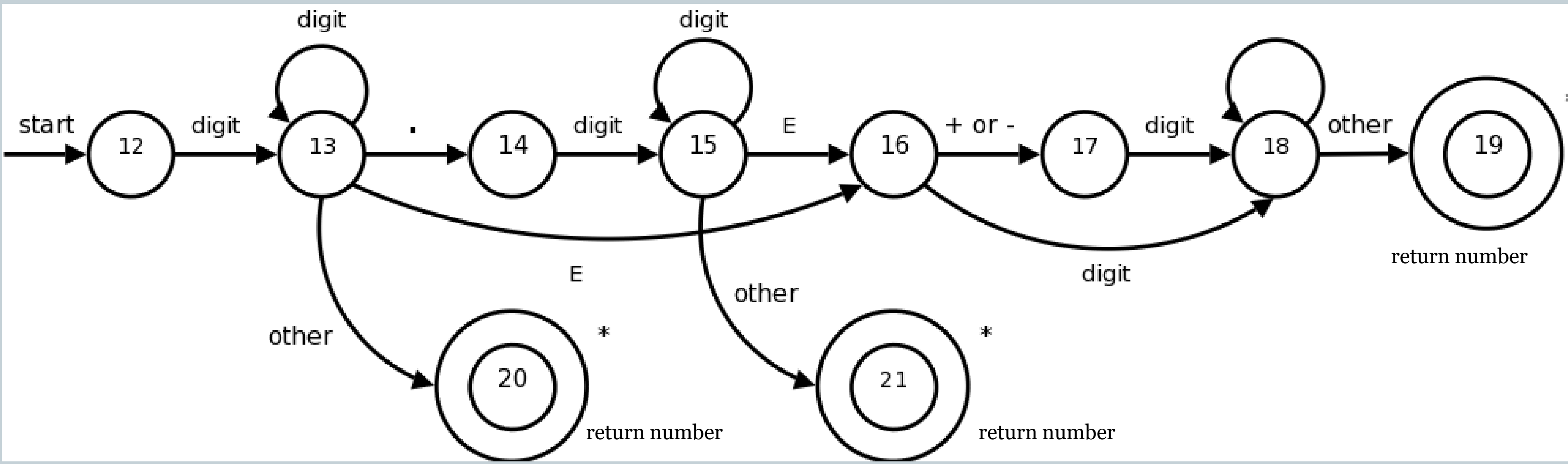


Recognizing/Skipping whitespace



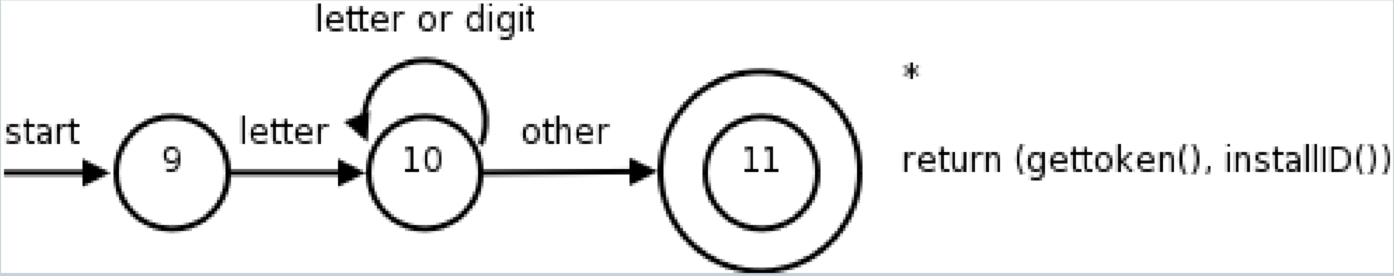
delim = tab | space | newline

Recognizing unsigned numbers



- $number \rightarrow digits\ (.digits)?\ (\mathrm{E}[+\ -])?\ digits)?$
- $digits \rightarrow digit^{+}$
- $digit \rightarrow [0\text{-}9]$

Recognizing identifiers/keywords

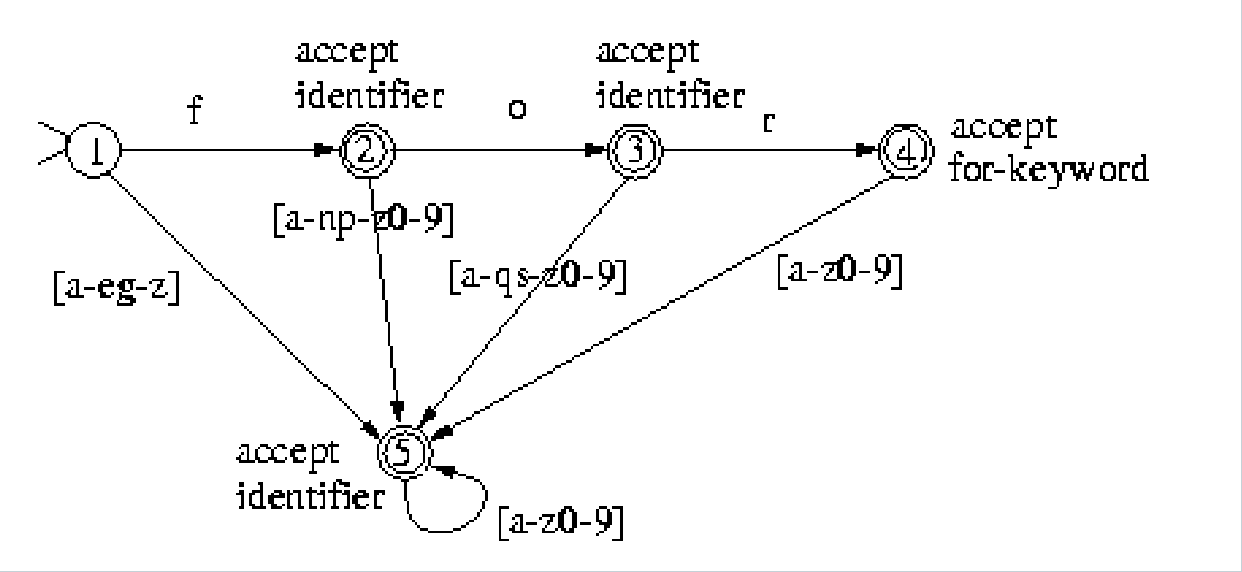


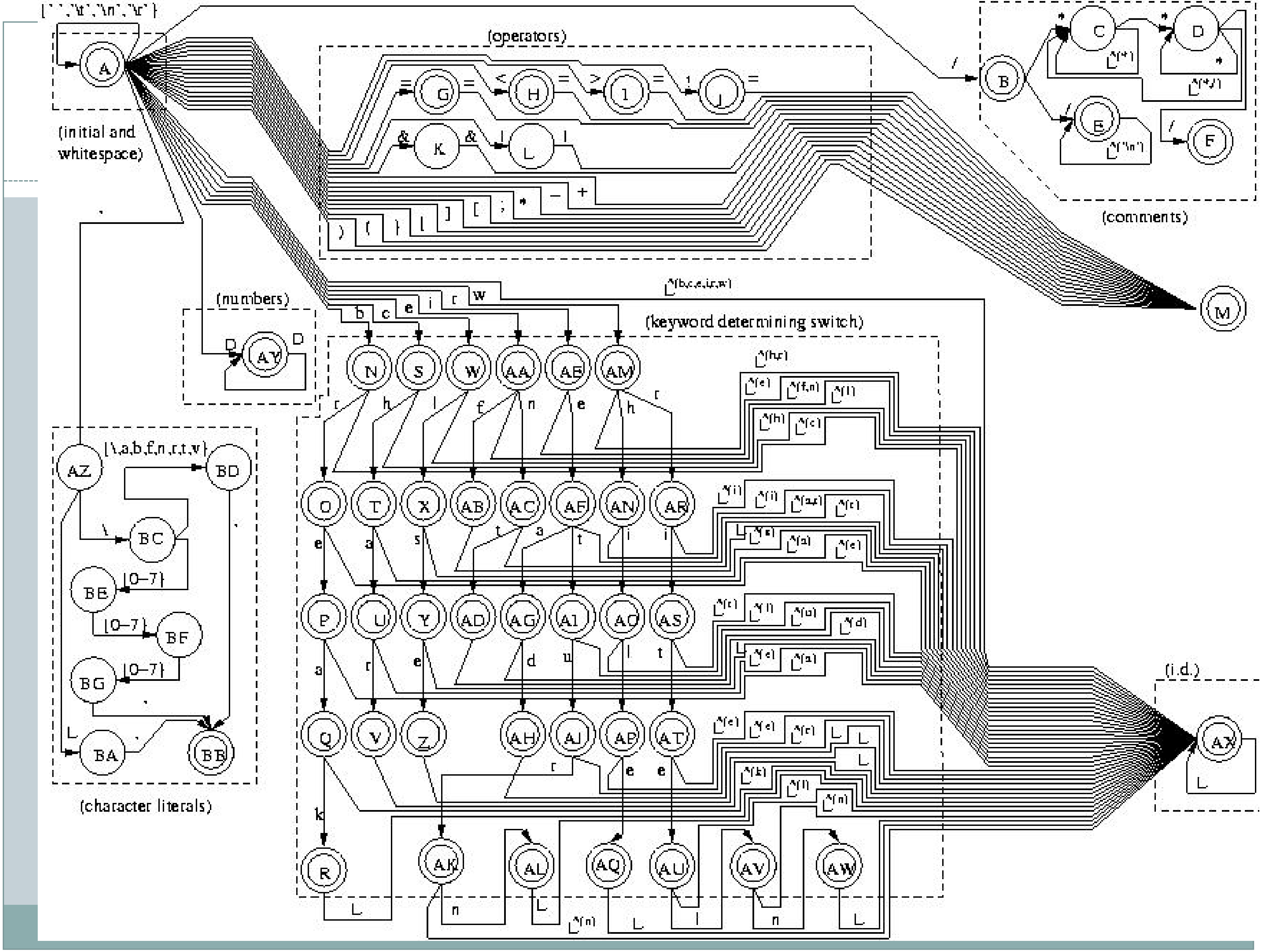
- The above transition diagram accepts identifiers.

- However, it also accepts keywords like **for**, ..
  - Solution 1: Insert all keywords in the symbol table before we start. `installID()` inserts lexeme/identifier in the symbol table only if it is not in the table already.

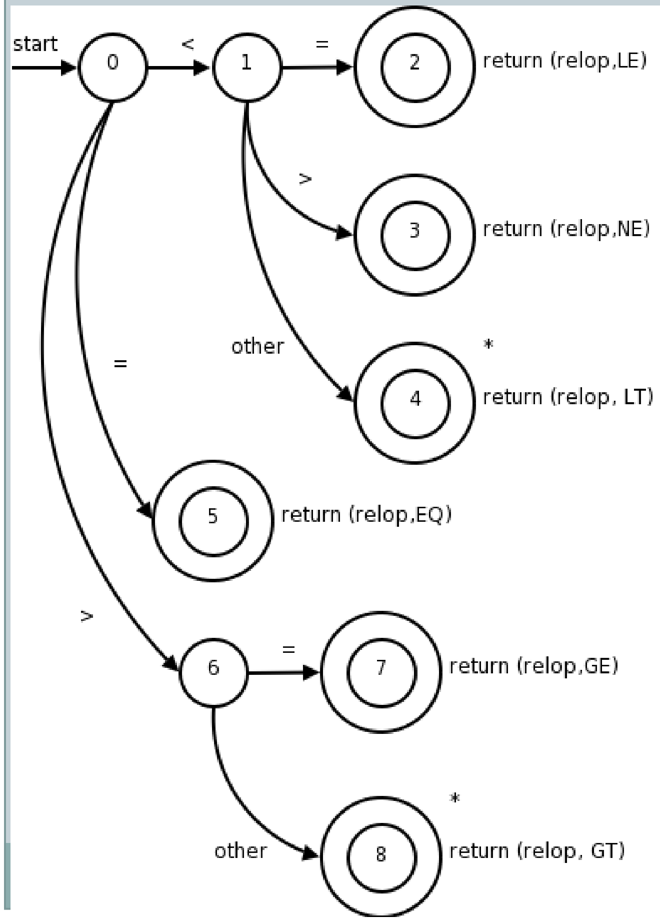
  - Solution 2: Create separate transition diagrams for each keyword. This makes the transition diagram non-deterministic, but this is not a problem (we discuss this later).

Recognizing identifiers/keywords (solution 2)




(initial and whitespace)
(operators)
(comments)
(numbers)
(keyword determining switch)
(character literals)
(i.d.)

Implementation of transition diagrams

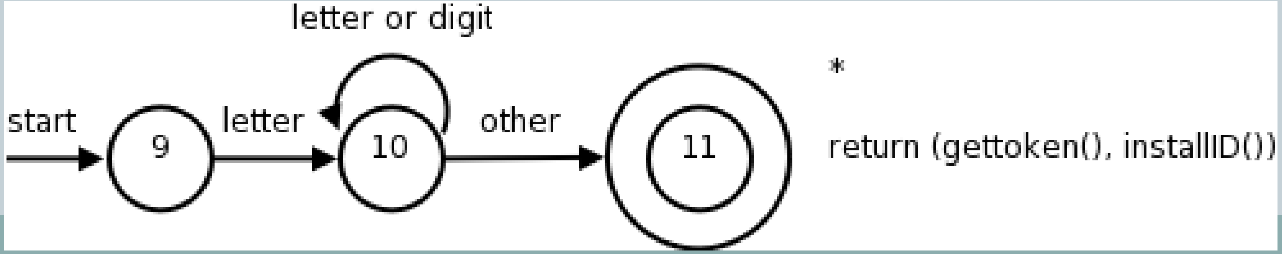



```
state = 0;
token nexttoken(){
   while(1) {
   switch (state) {
   case 0:   c = nextchar();
      /* c is lookahead character */
      if (c== blank || c==tab || c== newline) {
         state = 0;
         lexeme_beginning++;
         /* advance beginning of lexeme */
      }
      else if (c == '<') state = 1;
      else if (c == '=') state = 5;
      else if (c == '>') state = 6;
      else state = fail();
      break;
      … /* cases 1-8 here */
```

returns char pointed by input pointer; input pointer ++

Implementation of transition diagrams (Cont.)


```
case 9:   c = nextchar();
   if (isletter(c)) state = 10;
   else state = fail();
   break;
case 10;  c = nextchar();
   if (isletter(c)) state = 10;
   else if (isdigit(c)) state = 10;
   else state = 11;
   break;
case 11;  retract(1); install_id();
   return ( gettoken() );
   ……
```

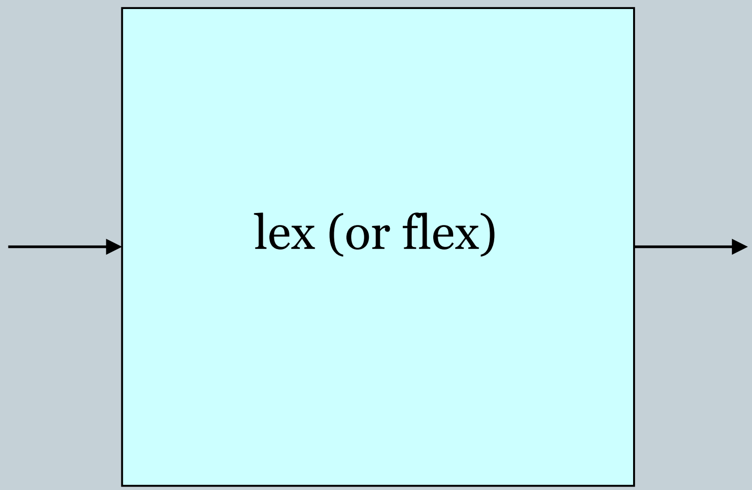


Handcoded vs. Generated Lexers

- Why handcode?
  - People claim it is faster (I doubt it)
  - Some languages, like Fortran, have very strange lexical rules.
    - E.g. DO I=1,10 and DOI=1,10 are the same.
    - In Fortran blanks are ignored ! So, at the comma the scanner can decide that DO is a keyword. If the comma was a dot, this would have been an assignment. A scanner generator can not handle this!
- Why generate?
  - Faster development
  - Higher level of abstraction
  - Easier to maintain/modify/extend
  - Scanner generator detects conflicts in the input specification
  - Hand coding is error prone (the transition diagram must be a DFA)

lex (and flex)

Input: a set of regular expressions + actions

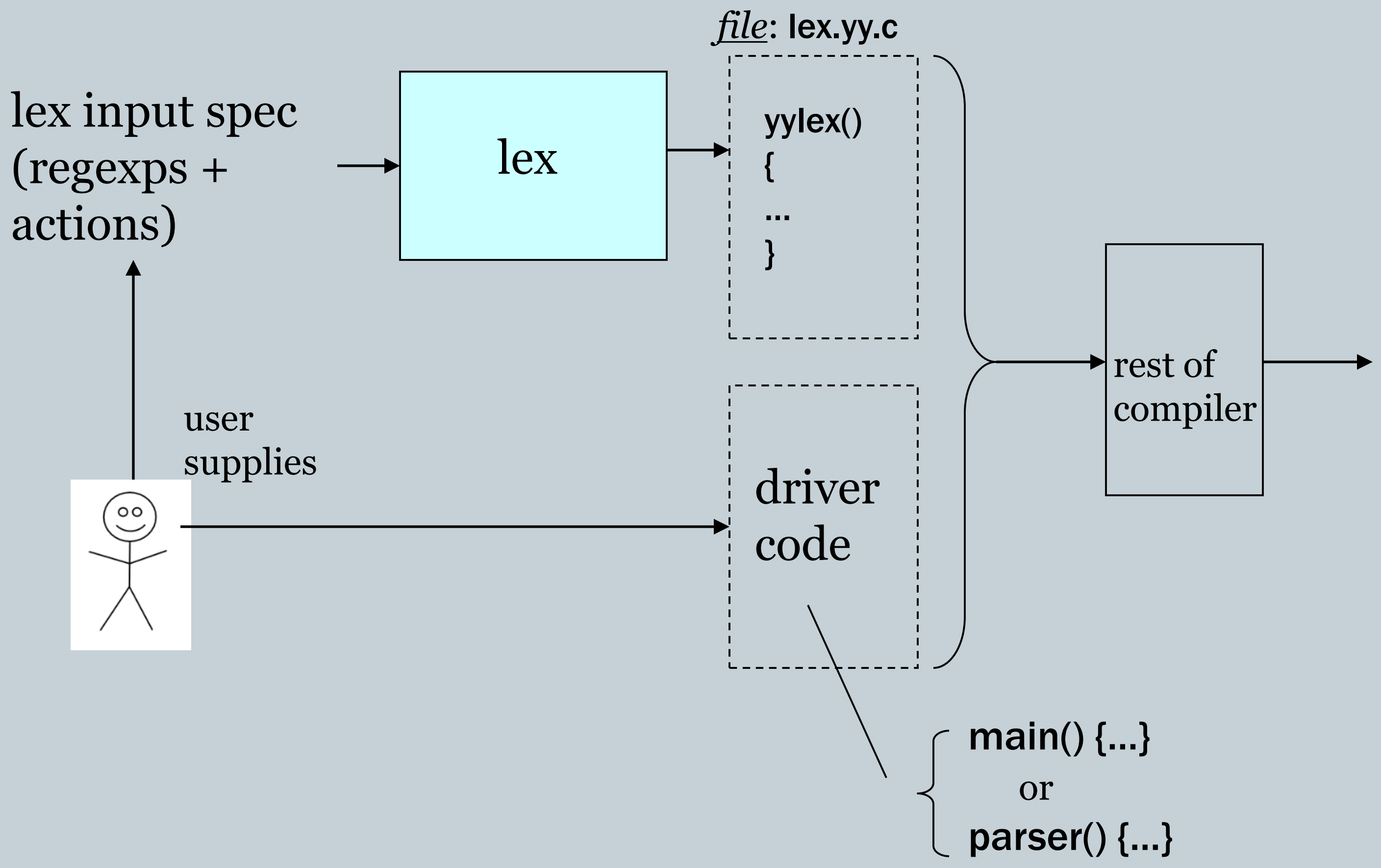


Output: C code implementing a scanner:

function: **`yylex()`**

file: **`lex.yy.c`**

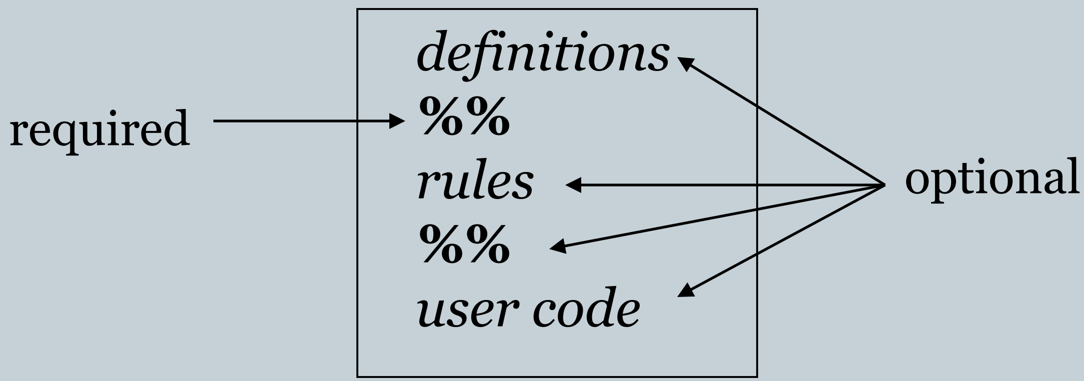
Using (f)lex



file: lex.yy.c
lex input spec (regexps + actions)
lex
yylex()
{
...
}
rest of compiler
user supplies
driver code
main() {...}
or
parser() {...}

(f)lex: input format

An input file has the following structure:



Shortest possible legal flex input:

```
%%
```

(f)lex: Definitions

- A series of:
  - *name definitions*, each of the form

    name *definition*

    e.g.:

    ```
    DIGIT            [0-9]
    CommentStart     "/*"
    ID               [a-zA-Z][a-zA-Z0-9]*
    ```

  - stuff to be copied verbatim into the flex output
    - enclosed in `%{ … }%`
    - e.g., variable declarations, **#include**s

definitions
%%
rules
%%
user code

(f)lex: Rules

- The *rules* part of the input contains a set of rules.

```
definitions
%%
rules
%%
user code
```

- Each rule has the form

  pattern   *action*

  where:
  - *pattern* describes regexp input pattern to be matched.
  - *action* to be performed when the pattern is matched.

Matching the Input

- If no rule matches, the default is to copy the next character to **stdout**.

- When more than one pattern can match the input, the scanner behaves as follows:
  - the longest match is chosen (first priority);
  - if multiple longest matches occur, the rule listed first in the flex input file is chosen;

Putting it all together

- Scanner implemented as a function

```
int yylex();
```

  - return value indicates type of token found (tokens are integers)
  - the lexeme matched is available in `yytext` (a char *)
  - the length of the lexeme is in `yyleng`

- Scanner and parser need to agree on token type encodings
  - If you use Yacc (parser generator), this is easy.
  - Let yacc generate the token type encodings
    - yacc places these in a file `y.tab.h`
  - use `#include y.tab.h` in the definitions section of the flex input file.

(f)lex: Example

A flex program to read a file of positive integers (and other characters to be ignored) and compute their sum:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
digit     [0-9]
%%
```

Definition for a digit
(could have used builtin definition [:digit:] instead)

rules

```
{digit}+    { return atoi(yytext); }
<<EOF>>     { return -1; }
.|\n        { /* . Accepts anything but a newline */  return 0; }

%%
```

Rule to match a number and return its value to the calling routine

user code

```
void main(int argc, char *argv[]) {
   int val, sum = 0;
   while ( (val = yylex()) >= 0 ) {
      sum += val;
   }
   printf("sum= %d\n", sum);
   return 0;
}
```

Driver code
(could have been in a separate file)

Beware of longest match problem!

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

longest match:

```
#include <stdio.h> /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf(“Error!\n”);  /* no arguments */
  }
  printf(“%d args given\n”, argc);
  return 0;
}
```

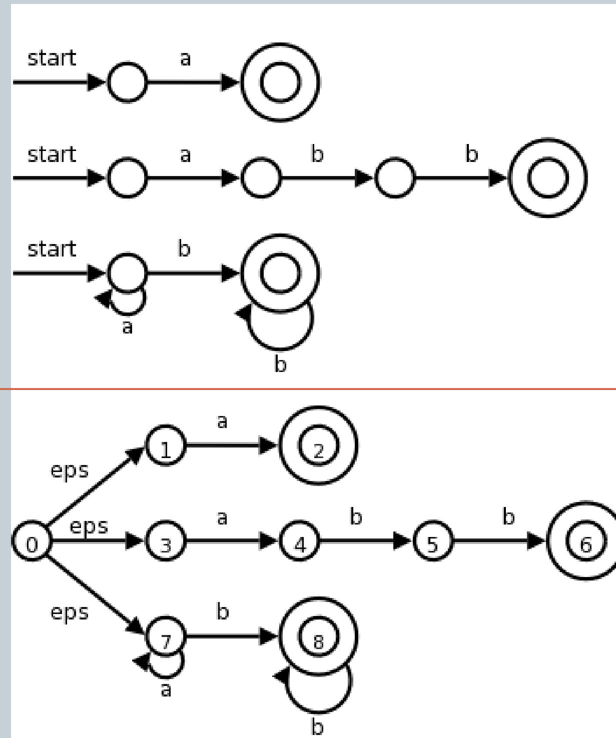
Flex Pattern Matching Primitives

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

Lex's extended regular expressions

- \c escapes for most operators
- "s" match C string as-is
- r{m,n} match r between m and n times
- r/s match r when s follows
- ^r match r when at beginning of line
- r$ match r when at end of line

How (F)lex produces a lexer



Regular expressions → NFA → DFA → Minimal DFA

Schedule

- Wednesday: tutorial
  - Extensive tutorial on Flex
  - You will make some simple scanners yourself
  - Make sure you have a laptop with flex + C-compiler installed

- Next week: lab
  - Make a couple of lexers

- Next week: Monday
  - Theory on finite automata
  - Introduction to parsing