Chapter 1: Lexical Syntax

References:

- Güting/Erwig: Übersetzerbau.
  Kapitel 1: Einführung (Seiten 1–18),
  Kapitel 2: Lexikalische Analyse (Seiten 19–40).
- Kernighan/Ritchie:
  The C Programming Language, 2nd Ed.
- Fischer/LeBlanc: Crafting a Compiler with C.
  Chapter 1: Introduction (pages 1–22)
  Chapter 3: Scanning — Theory and Practice (pages 50–90)

Overview

1. Introduction to Compilers
2. Task of Lexical Analysis
3. Regular Expressions
4. Lexical Syntax of C
5. The Scanner-Generator lex

Task of a Compiler (1)

- Translation from a programming language A into a programming language B.
- A machine for language B exists (hardware, interpreter, compiler to implemented language C).
- A is more convenient for the programmer than B.

  At least for certain tasks. Different applications might require different languages.

- The computer understands language A via the compiler.

  A compiler $A \rightarrow B$ transforms a machine for B into a machine for A.

Task of a Compiler (2)

Exercise:

- Actually, three languages are involved in a compiler specification: The compiler translates A into B, and is itself written in a language C.
- Do you think it makes any sense to write a compiler for a language in the language itself (e.g. write a C-compiler in C)?

Compilation Phases

- Lexical Analysis (Scanner)
- Syntactic Analysis (Parser)
- Semantic Analysis (includes Type Checking)
- Generation of Intermediate Code
- Code Optimization
- Code Generation

Overview

1. Introduction to Compilers
2. Task of Lexical Analysis
3. Regular Expressions
4. Lexical Syntax of C
5. The Scanner-Generator lex

Task of Lexical Analysis (1)

- Input: Program text as sequence of characters (e.g. ASCII).
- Output: Sequence of tokens (word symbols).
- White space and comments are removed.

  White Space: Spaces, Tabulator characters, line breaks, etc.

- Format-free language: Arbitrary sequence of white space characters (including line breaks and comments) is allowed between tokens.

  Some tokens require white space to separate them (e.g. `else x`), others don't (`x=5`).

Task of Lexical Analysis (2)

- Example Input:

```
if(total_amount >= 100)
        /* Good customer */
        shipping = 0;
else
        shipping = 5.95;
```

- Example Output:

```
[if] [(] [Identifier] [Bin-Op] [Integer] [)]
[Identifier] [Bin-Op] [Integer] [;]
[else]
[Identifier] [Bin-Op] [Real] [;]
```

Task of Lexical Analysis (3)

- Some tokens have besides their type also a value that the scanner passes to the parser:
  - Data type constants (e.g. Integer) have a value.

    Usually the scanner does the transformation from the sequence of digits into the internal binary representation.
  - Identifiers have a name.

    The scanner may already enter all identifiers into a table (the "symbol table") and return a pointer to the symbol table entry. However, this may also be done later. Then the scanner only returns the string of characters that are the name of the identifier.
  - If all operators are treated as the same token, an additional value differentiates between them.

    It is also possible to use a different token for each operator.

Scanner vs. Parser

- The syntax of tokens can be described with a simpler formalism (regular expressions) than the overall syntax (requires a context-free grammar).
- The very efficient scanner reduces the input size for the parser, which needs a more powerful and slightly slower algorithm.
- It is a general principle of software engineering to separate complex programs (like compilers) into modules that are relatively independent.

Overview

1. Introduction to Compilers
2. Task of Lexical Analysis
3. Regular Expressions
4. Lexical Syntax of C
5. The Scanner-Generator lex

Formal Languages

- An alphabet Σ is a finite and non-empty set.

  The elements of Σ are called characters.
- A word w over an alphabet Σ is a finite sequence $c_1\, c_2\, \ldots\, c_n$ of characters $c_i \in \Sigma$.
- ϵ is the empty word (length 0).
- Σ^* is the set of all words over Σ.
- A formal language over Σ is any subset $\mathcal{L} \subseteq \Sigma^*$.
- The concatenation of two words $v = a_1 \ldots a_n$ and $w = b_1 \ldots b_m$ is the word $v\, w = a_1 \ldots a_n\, b_1 \ldots b_m$.

Regular Languages

- $\emptyset$ and $\{\epsilon\}$ are regular languages.
- For every $c \in \Sigma$, $\{c\}$ is a regular language.
- If $\mathcal{L}_1$ and $\mathcal{L}_2$ are regular languages, then also

$$\mathcal{L}_1\mathcal{L}_2 := \{w_1\, w_2 \mid w_1 \in \mathcal{L}_1,\, w_2 \in \mathcal{L}_2\}$$

  and $\mathcal{L}_1 \cup \mathcal{L}_2$ are regular languages.

- If $\mathcal{L}$ is a regular language, then also

$$\mathcal{L}^* := \{w_1\, w_2\, \ldots w_n \mid w_i \in \mathcal{L}\}$$

  is a regular language.

- Nothing else is a regular language.

Regular Expressions (Theory)

- $\emptyset$ is a regular expression (regexp) that describes $\emptyset$.
- ϵ is a regexp that describes $\{\epsilon\}$.
- Every $c \in \Sigma$ is a regexp that describes $\{c\}$.
- If E_1 and E_2 are regexps that describe $\mathcal{L}_1$ and $\mathcal{L}_2$, then
  - $E_1 E_2$ is a regexp that describes $\mathcal{L}_1 \mathcal{L}_2$.
  - $E_1 | E_2$ is a regexp that describes $\mathcal{L}_1 \cup \mathcal{L}_2$.
- If E is a regexp that describes $\mathcal{L}$, then E^* is a regexp that describes $\mathcal{L}^*$.
- Nothing else is a regular expression.

Small Problem

- If one sees regular expressions as character strings over an extended alphabet (and not as operator trees), the structure must be defined.
- E.g. Does `ab|c` mean `a(b|c)` or `(ab)|c`?
- Usually one assumes that concatenation binds stronger than "|", i.e. `ab|c` means `(ab)|c`.
- The star binds strongest, i.e. `ab*` means `a(b*)`.
- One can always use parentheses `(...)` to enforce a particular structure (and in case of doubt).

  More about binding strengths, tree structure: see below.

RegExps in Practice (1)

- $\emptyset$ and ϵ are not used in practice.
- Many abbreviations are introduced:
  - E + means one or more iterations of E.

    I.e. it is an abbreviation for $E\,E$*.
  - E ? means zero or one occurrences of E.

    I.e. it is an abbreviation for $(E \mid \epsilon)$.
  - [c_1 c_2 ... c_n] means one of the characters c_i.

    I.e. it is an abbreviation for $(c_1|c_2|\ldots|c_n)$. In addition, instead of a single character, one can also use character ranges: c_1-c_n is an abbreviation for $(c_1|c_2|\ldots|c_n)$, where c_2, ..., c_{n-1} are all characters that have an ASCII code between the ASCII codes of c_1 and c_n. E.g. [0-9] are all digits, [a-zA-Z] are all letters.

RegExps in Practice (2)

- Abbreviations, continued:
  - [^$c_1 \ldots c_n$] ("complement") matches any character in $\Sigma - \{c_1, \ldots, c_n\}$.

    The complement can also be used with character ranges, e.g. `[^a-zA-Z0-9]` means any non-alphanumeric character.
  - . means any character in Σ (except newline).

    This is an abbreviation for $(c_1|c_2|\ldots|c_n)$, where c_1, c_2, ..., c_n are all characters in $\Sigma - \{\text{newline}\}$.
- Possibilities to specify the context of a match in a larger text:
  - ^E: E at the beginning of a line.
  - E\$: E at the end of a line.

Meta-Characters (1)

- The above definition assumes that the characters | and $*$ are not contained in Σ.
- But in practice, Σ is often the set of all ASCII characters, and regular expressions must also be sequences of ASCII characters.
- Meta-characters are characters that have a special meaning in regular expressions, e.g. |, $*$.

  In the scanner generator lex, the following characters are meta-characters: `.`, `$`, `^`, `[`, `]`, `-`, `?`, `*`, `+`, `|`, `(`, `)`, `/`, `{`, `}`, `<`, `>`, `"`, `\`.

Meta-Characters (2)

- Therefore the definition that c in a regular expression describes the language $\{c\}$ only holds for non-meta-characters.
- To use meta-characters in their literal meaning, they must be "escaped". This can be done by prefixing them with a backslash "\".
- Alternatively, a sequence of characters in double quotes "$c_1\, c_2\, \ldots\, c_n$" describes the language consisting of the word $c_1\, c_2\, \ldots\, c_n$.

  The c_i can be meta-characters except the double quote and the backslash. These two characters must be prefixed with a backslash.

Exercises

- Write a regular expression that describes words over the alphabet $\Sigma = \{0, 1\}$ that contain an even number of 1.
- Write a regular expression for integer constants: They should start with an optional minus "–" (at most one) and then contain a sequence of digits (at least one).

  If you want a more difficult example, exclude leading zeros, e.g. 007.

Named RegExps

- Tools like `lex` (see below) support user-defined abbreviations (macros), e.g. one can define

```
digit   →  [0-9]
letter  →  [a-zA-Z]
```

- Then one can use these names in place of the corresponding regular expressions. They are marked by enclosing them in `{...}`:

```
alphanum    →  {letter}|{digit}
identifier  →  {letter}{alphanum}*
```

- Of course, all names must be defined before their use, i.e. recusion is excluded.

  Otherwise it would already be a context-free grammar.

Implementation

- For a given regular expression, a finite automaton can be constructed that accepts the language decribed by the regular expression.

  It is easy to construct a non-deterministic finite automaton with ϵ-transitions, a unique start-state and a unique final state for a given regular expression. Then one transforms the non-deterministic automaton into a deterministic one. As far as I know, this is also the way in which scanner-generators like lex work.

- One can also write a scanner by hand. Then the state of the automaton is encoded by the current position in the program.

  One usually uses a variable `nextchar` that contains the next non-processed character.

Overview

1. Introduction to Compilers

2. Task of Lexical Analysis

3. Regular Expressions

4. Lexical Syntax of C

5. The Scanner-Generator lex

White Space

- White space: Any sequence of blanks (spaces), horizontal and vertical tabs, line breaks (linefeed and carriage return), formfeeds and comments.
- White space is ignored except that it separates tokens.

  E.g. it is not possible to have a comment or line break within an identifier.

- A comment starts with the characters /* and ends with the characters */.
- Comments do not nest.

Identifiers

- Identifiers are names for variables, procedures, types, enumeration constants, . . .
- Identifiers are sequences of letters and digits that start with a letter. The underscore "_" counts as a letter: `[a-zA-Z_][a-zA-Z_0-9]*`
- Upper and lower case letters are different, i.e. the case is significant.
- At least 31 characters are significant.

  I.e. the compiler may treat `abcdefghijklmnopqrstuvwxyzABCDE0` and `abcdefghijklmnopqrstuvwxyzABCDE1` as the same name (because it only stores 31 characters in the symbol table). The linker might treat fewer characters as significant (important only for external names).

Reserved Words

- As an exception to the syntax rule for identifiers, the following names are reserved as keywords and cannot be used otherwise:

```
auto      double  int       struct
break     else    long      switch
case      enum    register  typedef
char      extern  return    union
const     float   short     unsigned
continue  for     signed    void
default   goto    sizeof    volatile
do        if      static    while
```

- One also should avoid names starting with "_".

Integer Constants (1)

- An integer constant is a sequence of digits.

  Sometimes one uses the word "literal" instead of constant to avoid the confusion with named constants (`pi`).

- If it starts with a zero, it is understood as octal (then the digits 8 and 9 cannot be used).

  E.g. $123 = 1*100+2*10+3*1$ and $0123 = 1*64+2*8+3*1 = 83$.

- It is also possible to write the constant in the hexadecimal system, then it has to start with `0x` or `0X`.

  The additional digits are `a`/`A` (10), `b`/`B` (11), `c`/`C` (12), `d`/`D` (13), `e`/`E` (14), `f`/`F` (15). E.g. `0xFF` $= 15 * 16 + 15 = 255$.

Integer Constants (2)

- C has different integer types that are explained in more detail below. E.g. if integers are 16-bit, the possible range is (usually) `-32768..+32767`.
- C also supports unsigned integers, which would have the range `0..65535` on a 16-bit machine. They are marked with the suffix `u/U`, e.g. `65535U`.

  Actually, if the value of a sequence of digits does not fit into an `int`, C understands it automatically as `unsigned int`. After that, it tries `long` and `unsigned long`.

- Long constants (e.g. 32-bit integers) are marked with the suffix `l/L`: `-2147483648L..+2147483647L`.

Floating Point Constants

- A floating point constant consists of
  - an integer part (sequence of digits),
  - a decimal point ".",
  - a fraction part (sequence of digits),
  - an `e` or `E`,
  - an optionally signed integer (exponent),
  - an optional type suffix: `f`, `F`, `l`, `L`.
- Either the integer part or the fraction part (not both) may be missing, either the decimal point or the `e` and the exponent (not both) may be missing. Exercise: Write a regular expression.

Character Constants (1)

- Character constants consist of a character in single quotes, e.g. `'a'`.

  The character cannot be a single quote, a line break, or a backslash.

- Instead of a character, one can use the escape sequences listed on the next slide.

  On UNIX, where C originated, a single linefeed terminates a line. The standard library functions on most Windows systems translate the linefeed into the sequence linefeed, carriage return that is used as line break on these systems. Therefore on both systems it should suffice to print a `'\n'` to start a new line.

- To specify a character by code, one can use one, two, or three octal digits, e.g. `'\0'`.

Character Constants (2)

`\n`	Newline/Linefeed (LF)
`\t`	Horizontal tab (TAB, HT)
`\v`	Vertical tab (VT)
`\b`	Backspace (BS)
`\c`	Carriage Return (CR)
`\f`	Formfeed (FF)
`\a`	Audible alert (BEL)
`\\`	Backslash (\\)
`\?`	Question mark (?)
`\'`	Single quote (')
`\"`	Double quote (")
`\`*ooo*	Character with code *ooo* (in octal)
`\x`*hh*	Character with code *hh* (in hexadecimal)

String Constants

- A string constant is a sequence of characters in double quotes, e.g. `"abc"`.
- The escape sequences listed above can also be used inside string constants, e.g. `"a line\n"`.
- String constants that are adjacent in the program text (i.e. separated only by white space) are merged into a single string.

  In this way, one can also write string constants that are longer than an input line.

Operators (1)

- + (addition), - (subtraction), / (division),
  * (multiplication, dereference),
  % (remainder/mod).
- == (equal), != (unequal), < (less-than),
  > (greater-than), <= (less-than-or-equal),
  >= (greater-than-or-equal).
- && (logical and), || (logical or), ! (logical not).
- & (bit-and, reference), | (bit-or), ^ (bit-XOR),
  ~ (bit-complement), << (left shift), >> (right shift).

Operators (2)

- = assigment.
- ++ (increment), -- (decrement).
- +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= (assigment abbreviations).
- , (sequence).
- . (selection of structure component),
  -> (dereference+structure component selection).

Other Tokens

- `;` (end of statement).
- `{`, `}` (begin and end).
- `(`, `)` (parentheses).
- `[`, `]` (array parentheses).
- `?`, `:` (conditional expressions).

  `:` is also used in the `switch`-statement.

Overview

1. Introduction to Compilers
2. Task of Lexical Analysis
3. Regular Expressions
4. Lexical Syntax of C
5. The Scanner-Generator lex

Usage of lex (1)

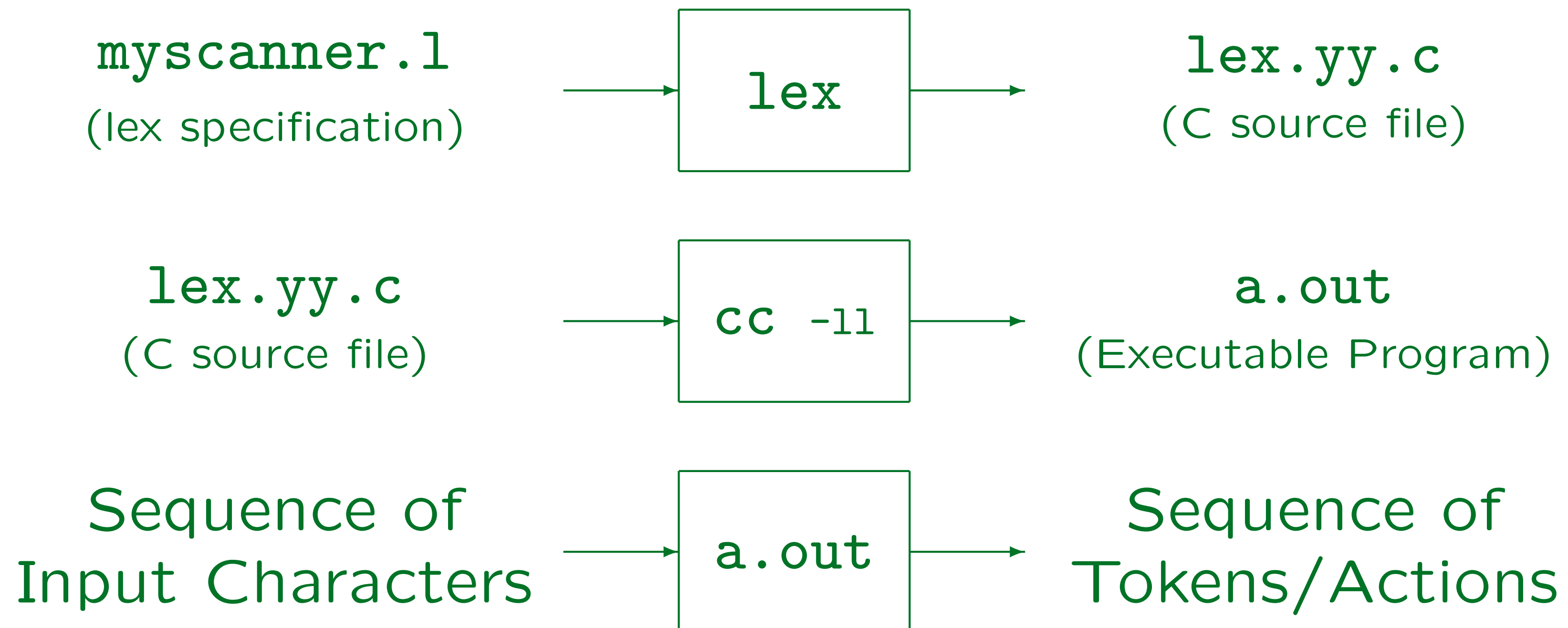
- `lex` translates a specification of the scanner into a C-program that implements the scanner.

  The input file often has the extension "`.l`", e.g. "myscanner.l". The output file is usually called "`lex.yy.c`".

- The main part of the specification is a sequence of rules. Each rule is a pair that consists of a regular expression and a C-statement.

- `lex` mainly generates a procedure `yylex`. This procedure reads input characters until one of the regular expressions is complete and then executes the corresponding C-statement.

Usage of lex (2)



Example (1)

- A small calculator for integers with the operations $+$, $-$, $*$, and $/$ might use this specification:

```
%%
[0-9]+   return(1);
"+"      return(2);
"-"      return(3);
"*"      return(4);
"/"      return(5);
"("      return(6);
")"      return(7);
"\n"     return(0);
[ \t]+   ;
.        return(-1);
```

Example (2)

- The generated procedure `yylex` will
  - return the value 1 if an integer constant (sequence of digits) was detected in the input.
  - return the values 2–7 to encode the operators and parentheses.
  - return the value 0 if the input line ends.

    We assume that the input expression is then evaluated.
  - skip any spaces and tabulator characters.
  - return the value −1 for any other character

    "." would match any character, but if more than one regular expression match the same input, `lex` executes the action part of the one that comes first in the specification.

Example (3)

- The "%%" separates sections of the specification.
- The three sections of a `lex`-specification are:

```
Declarations
%%
Rules (Regular Expressions and Actions)
%%
Auxillary Procedures
```

- The declaration section can contain (e.g.)
  - C-code that is copied to `lex.yy.c`

    before the declaration of the procedure `yylex`.
  - auxillary regular expressions (abbreviations).

Example (4)

- E.g. the following introduces symbolic constants for the token codes and the abbreviation “digit”:

```
%{
#define TOK_NUMBER 1
#define TOK_PLUS   2
...
%}
digit    [0-9]
%%
{digit}+    return(TOK_NUMBER);
"+"         return(TOK_PLUS);
...
```

Example (5)

- The third section contains C code that is copied unchanged to `lex.yy.c`.
- E.g. auxillary procedures and the main program may be defined in this section.
- `lex` makes the characters of the matched token available in the character array `yytext`.
- The length of this token is stored in the variable `yyleng`.

Example (6)

- E.g. the following main program (declared in the third section) will print the token codes and their characters (in order to test the scanner):

```
#include <stdio.h>
int main()
{   int tok;
    tok = yylex();
    while(tok != 0) {
        printf("%d: '%s'\n", tok, yytext);
        tok = yylex();
    }
    return(0);
}
```

More about Rule Syntax

- In the rules section, the regular expression and the C code is delimited by at least one space or tabulator character.

  The regular expression may not contain spaces except inside ", [...], or escaped with \.

- The regular expression must start in the first column, lines that start with a space are considered as C code.

- The action part may extend over several lines, in which case the C statements should be enclosed in {...}.

More about Ambiguity

- `lex` always matches the longest possible sequence of input characters.
- E.g. consider keywords and identifiers:

```
if                            return(TOK_IF);
...
[a-zA-Z_][a-zA-Z0-9_]*        return(TOK_ID);
```

- If the input is `ifx ...`, `lex` will report an identifier.
- If the input is `if(...`, `lex` will report the keyword.

  It is important that the rule for the keyword comes before the rule for the identifier, since that would also match.