Lecture 4: Lexical Analysis, Regular Expressions, and Finite Automata

Programming Languages Course
Aarne Ranta (aarne@chalmers.se)

Book: 3.1, 3.3, 3.4, 3.6, 3.7, 3.9

Before parsing

Parsing is done in accordance with BNF rules such as

```
Stm ::= "if" "(" Exp ")" Stm "else" Stm ;
Exp ::= Exp "+" Exp ;
Exp ::= Ident ;
Exp ::= Integer ;
```

The terminals (in quotes) are **tokens**, and in the end all rules generate and recognize lists of tokens.

Also `Ident` and `Integer` are a special kinds of tokens.

The task of **lexical analysis** is to split the input string into tokens.

Spaces

Tokens are sometimes separated by spaces (/newlines/tabs), but not necessarily:

```
if (i>0) {i=i+27;} else {i++;}
```

```
if, (, i, >, 0, ), {, i, =, i, +, 27, ;, }, else, {, i, ++, ;, }
```

Good design: it should *always* be *legal* to have spaces, newlines, or tabs between tokens - even if not necessary.

Quiz: which programming languages violate this rule?

Lexical analysis in parser (don't do like this!)

It *is* possible for the parser to read input character by character; in this case, the grammar should tell where spaces are possible/necessary.

```
Exp       ::= Exp Space "+" Space Exp ;
Space     ::= [SpaceChar] ;
SpaceChar ::= " " | "\n" | "\t" ;
```

But this really clutters the grammar!

Programming languages are usually designed in such a way that lexical analysis can be done before parsing, and parsing gets tokens as its input.

Classes of tokens

The lexer *splits* the code into tokens.

The lexer also *classifies* each token into classes. For example, the string

```
i2=i1+271
```

results in the following list of classified tokens:

```
identifier "i1"
symbol "="
identifier "i2"
symbol "+"
integer "271"
```

The lexer program

The lexer should read the source code character by character, and send tokens to the parser.

After each token, it should use the next character `c`to decide what kind of token to read.

- if `c` is a digit, collect an integer as long as you read digits
- if `c` is a letter, collect an identifier as long as you read identifier characters (digit, letter, `'`)
- if `c` is a double quote, collect a string literal as long as you read characters other than a double quote
- if `c` is space character (i.e. space, newline, or tab), ignore it and read next character

Longest match: read a token *as long as* you get characters that can belong to it.

Comments

Since comments (`/* ... */`) are not saved by the lexer, an easy way to remove them might be in a separate, earlier pass.

However, if we have string literals, we have to spare comment signs in them:

```
"no comment /* here */ please!"
```

Thus also comments belong to the lexer:

- if you read `/` and the next character is `*`, then ignore characters until you encounter `*` and `/`

The lexer here needs a 1-character **lookahead** - it must read one more than the next character

When lexing, the lexer knows if it is reading a string, and does not start reading a comment in the middle of this.

Reserved words and symbols

Reserved symbols and words: the terminals in the BNF grammar.

There is a finite number of reserved symbols and words (why?)

Reserved symbols: `+ > ++ >> >=` etc.

Longest match: if `++` is found, `+` is not returned.

This explains a detail of C++ template syntax: space is necessary in

```
vector<vector<int> >
```

Reserved words: usually similar to identifiers, longer than special symbols: `if inline while` etc

Transition diagram for reserved words

Book 3.4.1

A picture will be shown of a diagram for reserved words.

The diagram is in the book, and we will draw it on the blackboad to make it livelier.

(One could also use a hash table)

Transition diagram for reading tokens

Book 3.4.3

A picture will be shown

Implementing lexers (don't do like this!)

Transition diagrams can be hand-coded by using `case` expressions.

Book 3.4.4 gives a Java example; here is a Haskell one

```
lexCmm :: String -> [Tok]
lexCmm s = case s of
  c:cs   | isSpace c     -> lexCmm cs
  c:cs   | isAlpha c     -> getId s
  c:cs   | isDigit c     -> getInt s
  c:d:cs | isSymb [c,d]  -> TS [c,d] : lexCmm cs
  c:cs   | isSymb [c]    -> TS [c]   : lexCmm cs
  _                      -> []  -- covers end of file and unknown characters
 where
  getId s  = lx i : lexCmm cs where (i,cs) = span isIdChar s
  getInt s = TI (read i) : lexCmm cs where (i,cs) = span isDigit s
  isIdChar c = isAlpha c || isDigit c
  lx i = if isReservedWord i then TS i else TI i

isSymb s = elem s $ words "++ -- == <= >= ++ { } = , ; + * - ( ) < >"

isReservedWord w = elem w $ words "else if int main printInt return while"
```

But the code is messy, and hard to read and maintain.

Regular expressions

A good way of specifying and documenting the lexer is transition diagrams.

More concisely, we can use **regular expressions**:

```
  Tokens = Space (Token Space)*
  Token  = TInt | TId | TKey | TSpec
  TInt   = Digit Digit*
  Digit  = '0' | '1' | '2' |'3' | '4' |'5' | '6' |'7' | '8' | '9'
  TId    = Letter IdChar*
  Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
  IdChar = Letter | Digit
  TKey   = 'i' 'f' | 'e' 'l' 's' 'e' | ...
  TSpec  = '+''+' | '+' | ...
  Space  = (' ' | '\n' | '\t')*
```

What is more, these expressions can be compiled into a program that performs the lexing! The program implements a transition diagram - more precisely, a **finite state automaton**.

The syntax and semantics of regular expressions

name	notation	semantics	verbally
symbol	'a'	{a}	the symbol 'a'
sequence	A B	`{xy	x : A, y : B}`	A followed by B
union	`A	B`	A U B	A or B
closure	`A*`	`{A^n	n = 0,1,2,..}`	any number of A's
empty	eps	{[]}	the empty string

This is BNFC notation.

The semantics is a **regular language**, a set of strings of symbols. One often writes *L(E)* for the language denoted by the expression *E*.

There are variants of the notation, especially for symbols and empties. Quoting symbols is not so common, since it lengthens the expressions - but it is very good for clarity!

More notations for regular expressions

name	notation	definition/semantics	verbally
symbol range	`['c'..'t']`	'c'|...|'t'	any symbol from 'c' to 't'
symbol	`char`	`{x	x is a character}`	any character
letter	`letter`	`['A'..'Z']	['a'..'z']`	any letter
digit	`digit`	`['0'..'9']`	any digit
option	`A?`	`A	eps`	optional A
n-sequence	`A^n`	`{x1...xn	xi : A`	n A's
nonempty closure	`A+`	`A A*`	any positive number of A's
difference	`A - B`	`{x	x : A & not (x : B)}`	the difference of A and B

The symbol range and n-sequence are not supported by BNFC.

"Any character" in BNFC is ASCII character 0..255.

State-of-the-art lexer implementation

Define each token type by using regular expressions

Compile the regular expressions into a code for a transition diagram (finite-state automaton).

Compile the resulting code into a binary.

```
    regexp in BNFC file

             |
             v          bnfc program

    Flex/Jlex/Alex file
```

```
            |
            v            flex/jlex/alex program

    C/C++/Java/Haskell file

            |
            v            gcc/javac/ghc program

    binary file
```

Pros and cons of hand-written lexer code

+ full control on the program (lookahead)

+ usually smaller program

+ can be quick to create (if the lexer is simple)

+ can be made more powerful than regular expressions

- difficult to get right (if the lexer is complex)

- much extra work to include position information etc.

- may compromise performance (lookahead)

- not self-documenting

- not portable

- lexer generators are state-of-the-art in compilers

Thus the usual trade-offs between hand-made and generated code hold.

In this course, we use generated lexers in the labs.

A look at lexer generation tools

You can take a look at the web pages, manuals, and bnfc output for

Alex: Haskell

JLex: Java

Flex: C, C++

The BNF Converter web page has links to all these tools.

Lexer definitions in BNFC

Reserved word and symbols: terminals in the rules.

Built-in types

```
    Integer Double String Char
```

Comments

```
    comment "/*" "*/" ;
    comment "//" ;
```

Token

```
    token Id (letter (letter | digit | '_')*)
```

Position token (for better error messages in later phases)

```
    position token Id (letter (letter | digit | '_')*)
```

Built-in lexical types of BNFC

type	definition
Ident	letter (letter | digit | '_' | '\'')*
Integer	digit+
Double	digit+ '.' digit+ ('e' '-'? digit+)?
String	'"' ((char - ["\"\\"]) | ('\\' ["\"\\nt"]))* '"'
Char	'\'' ((char - ["'\\"]) | ('\\' ["'\\nt"])) '\''

Finite-state automata

A **Nondeterministic Finite-state Automaton** (NFA) is a tuple (Q,S,q0,F,d) where

- Q, a finite set of states
- S, a finite set of input symbols
- q0 : Q, the start state
- F < Q, the set of final states
- d : Q x S' -> P(Q), transition function (S' = S U {eps})

A **Deterministic Finite-state Automaton** (DFA) is like NFA. except that the value of the transition function d is a single state in Q (instead of a subset of Q) and there are no **epsilon transitions** (transitions that don't consume a symbol). In other words,

- d : Q x S -> Q, transition function

It is customary to draw diagrams with circles for states and arrows for the transition function: you can find them in the course book, and we will draw them on the blackboard.

Compiling regular expressions into finite-state automata

1) **NFA generation**. Compile the expression into an NFA, using transitions with the epsilon symbol e. (Book 3.7.4). This translation is **compositional**: simply traverse the syntax tree of the regular expression. The size of the automaton is linear in the size of the expression: O(|r|).

But NFAs are, in comparison to DFAs, - more difficult to implement - less efficient to run: O(|r| * n), where n is the input length.

2) **Determinization**. Transform the NFA into a DFA. (Book 3.7.1). Here you go through states, and for each symbol create a state collecting those states that the symbol can lead to (subset construction).

The DFA can still be bigger than necessary, i.e. have more states than needed.

3) **Minimization**. Merge equivalent states (Book 3.9.6): states beginning from which the automaton recognizes the same set of strings.

The resulting DFA permits analysis that is linear in the length of the input string: O (n). But its size is still exponential, at worst, in the length of the regular expression: O (2^|r|).

From regular expressions to NFA's

Book 3.7.4.

A typical example of **compilation schemes**:

```
NFA(eps) =

NFA('c') =

NFA(A B) =

NFA(A | B) =

NFA(A*) =
```

Notice: the compilation would be much simpler, if the operators were only between symbols.

Example:

```
'a' 'b' | 'a' 'c'
```

From NFA's to DFA's

Book 3.7.1

Subset construction with epsilon closure

Typical example of back-end optimization: of time

Example (again):

```
'a' 'b' | 'a' 'c'
```

Minimizing DFA's

Book 3.9.6

Merging equivalent states

A further example of back-end optimization: of space

Example (once more):

```
'a' 'b' | 'a' 'c'
```

Equivalence between regular expressions into finite-state automata

It is conversely possible to express any finite-state automaton by a regular expression. A decompilation algorithm is given e.g. in Hopcroft & Ullman, //Introduction to Automata Theory, Languages, and Computation// (1979).

Often it is easier to construct the expression that the automaton, sometimes vice-versa.

Example (easy with automata): any sequence of 0's and 1's with an even number of 0's. Construct an automaton with two states, corresponding to the number of 0's being even or odd.

Example (easy with expressions): algebraic reasoning (and simplification) of regular expressions, with laws such as

```
A eps = A
A | A = A
A**   = A*
```

Example of reasoning on automata

(Reminder: language = set of strings.)

Proposition. The language `` `{ a^n b^n | n = 0,1,2,...}` `` is not regular.

Proof (Book 4.2.7). We show that any language `` `a^n b^n` `` needs at least *n+1* states. It follows that there can be no finite number of states to accept the language where *n* can be arbitrary.

Consider the states `s_0, s_1, ...., s_n` reached after reading each relevant number of `a`'s. Each of these states must have a different follow-up, and hence be a distinct state: if we had `s_i = s_j`, then the automaton would accept `a^i b^j`.

Example of exponential DFA size

A sequence of *a*'s and *b*'s where the n'th last symbol is *a*.

```
(a|b)* a (a|b)^n
```

The number of states in the DFA is exponential in `n`.

Show this for n=1.

Limitations of regular languages

Java, C, and Haskell are not regular languages.

We cannot write their parser by using regular expressions.

This is because the languages have rules of the form `a^n X b^n` e.g. for matching left and right parentheses. A machine with finitely many states cannot remember that exactly *n* left parentheses have been read, if *n* is arbitrary (cf. the proposition on `a^n b^n` above).

Regular and context-free languages

Every regular language is also a context-free language (i.e. can be described by a context-free grammar).

The inverse is not true. It holds for grammars with no recursion, and also for grammars with just **left recursion**

```
C ::= C X      ===>  C = Y X*
C ::= Y
```

or **right recursion**

```
    C ::= X C       ===>  C = X* Y
    C ::= Y
```

But it does not hold for languages with **middle recursion**:

```
    C ::= X C Z     ===>  C = X^n Y Z^n
    C ::= Y
```