Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 4

Outline

What is lexical analysis?

- Lexical analysis is the first step of a compiler front end.
    - It takes not-yet analyzed input ("stream of characters") and breaks it down into known units, called *tokens*.
    - Tokenization includes stripping off white space and often also comments.
- The output of the lexical analysis is the input to the parser. Tools for lexing and parsing work together closely.

Theoretical foundations

Lexical analysis builds on two rigorous formalizations.

- Tokens are specified as *regular expressions*.
- Lexers are implemented as *deterministic finite automata*.

Regular expressions and finite automata, originally developed for lexers (and compilers in general), have influenced a great many other areas in computer science.

Outline

What are tokens?

A *token* is a classification of characters. The particular classification is motivated by later processing step of the compiler.

Common tokens

Token kind	Examples
ID	x, foo, myclass, int
NUM	1,123
REAL	1.1, 10.3
FLOAT	float
CHAR	char
EQUALS	=
LPAREN	(
RPAREN	)
COMMA	,

Example (tokenization)

Consider

```
float foo (char* cp);
```

- As input to the lexer, the string is a stream of 21 characters.
- Depending on the underlying token definitions, a lexer might split this string in the following 8 tokens

FLOAT ID LPAREN CHAR STAR ID RPAREN SEMI

Reserved words

Every programming language contains a number of keywords. In most languages, keywords are *reserved words*, which cannot be used as identifiers.

Examples

- if, while, for, return, main

Reserved words often constitute tokens.

Non-tokens

Not every character sequence in a stream maps to a token.

- If subsequent compilation steps do not need, or process, certain sequences, the lexer discards them.
- Examples of non-tokens:

comments	/* as;fjasdf; */
space	newline, blank, tab
preprocessor	#include<stdio.h>

- In the presence of non-tokens, the original source cannot completely be reconstructed after lexical analysis.

In-class exercise

In-class exercise

Regular expressions: definition

Assume a finite alphabet Σ of symbols. A *regular language* over Σ is the set of *regular expressions*, inductively defined:

- Every symbol is a regular expression.
- The empty string is a regular expression.
- Given two regular expressions. The alternation of two regular expressions is a regular expression.
- Given two regular expressions. The concatenation of two regular expressions is a regular expression.
- Given a regular expression. Its Kleene closure (zero or more concatenations) is a regular expression.

Regular expressions: notation

A common notation for the operation on regular expressions:

- The alternation of two regular expressions is denoted by $|$
  - Example: the language of $a \mid b$ contains the two symbols a and b.
- The concatenation of two regular expressions has no special symbol.
  - Example: the language of $(a \mid b)c$ contains the symbols ac and bc.
- The Kleene closure (zero or more concatenations) is denoted by $*$.
  - Example: the language of $a*$ refers to the infinite set $\{``", a, aa, aaa, aaaa, aaaaa, \ldots\}$

Examples (regular expressions)

- $((a \mid b)c)*$ denotes the following language $\{``", ac, bc, acac, acbc, bcac, bcbc, acacac, \ldots\}$
- More notation for convenience
  - $[abcd]$ is a shortcut for $(a|b|c|d)$
  - $[a-z]$ is a shortcut for $(a|b|\cdots|z)$
  - $a+$ is a shortcut for $aa*$
  - $\hat{}a$ is a shortcut for “not a”

More examples (tokens)

Tokens can be specified as regular expressions.

$[a-z][a-z0-9]*$	ID
$[0-9]+$	NUM
if	IF
return	RETURN

In-class exercise

Ambiguities, again

Consider the previous token definitions. How should the token “if1” be classified? As one token or as two tokens? Rules are needed for disambiguation.

The two most common rules:

- Longest match: take the longest initial substring that matches
  - Then, “if1” is interpreted as an identifier
- Rule priority: take the first initial substring that matches
  - Then, “if1” is interpreted as two tokens: ’if’ and ’1’.
  - The order matters in which tokens are defined.

Specifying a lexer in BNFC

BNFC provides two commands specifically for lexical analysis:

- The `comment` rule takes one or two string arguments that are free text and not passed on to the parser. We have seen the two forms already:
  - Single-line comments, ex.: `comment //;`
  - Multi-line comments, ex.: `comment /* */;`
- The `token` rule allows users to define new tokens. For Haskell, it can be qualified with `position`, which provides access to the line number and the column number:

```
token UIdent (upper (letter | digit | _)*);
position token UIdent (upper (letter | digit | _)*);
```

  - `upper`, `letter`, `digit` are predefined character classes in BNFC

Tokens in BNFC

- BNFC pre-defines the following token types: Integer , Double, String , Char (for their definition, see the BNFC specification)
- For the definition of new tokens, the following regular expressions are available: `eps`, `char`, `letter`, `upper`, `lower`, `digit`
- Notation for the five basic operations: 'a', A B, A | B, A*
  Additional operators: A? (optional), A - B (difference)
  - `[''abc'']` denotes the union of the characters 'a', 'b', 'c'.
  - `{''abc''}` denotes the sequences of characters 'a', 'b', 'c'.

Among the files that `bnfc` generates are the files for lexer generators.

Outline

Lexer generators

Recall: common tools for lexical analysis are lex/flex (for C++), Alex (for Haskell), and JFlex for Java. These tools are *lexer analysis generators* (short: lexical generators).

- A lexer generator is a program that takes a specification of the lexical structure (tokens, comments, position) and returns a lexer.
- BNFC returns lexer generators. It is therefore a lexer generator generator.
- A lexer (lexical analyzer) is sometimes called “tokenizer” or “scanner”, although both are only subtasks.

Lexical analysis with flex

http://flex.sourceforge.net/manual/

The `flex` program takes an input file (or standard input) and returns a `C` source file.

The input file consists of three sections

- Definitions: introduction of names
- Rules: pairs of (regular expression, action)
- User code in `C`.

The sections are separated by %% . C-style comments are permitted.

The input file has the ending `.l`.

Example (rules section)

The core of a flex file is the second section, called rules section.
Here are two rules:

```
{DIGIT}+"."{DIGIT}* {
                printf( "A float: %s (%g)\n", yytext,
                atof( yytext ) );
            }

if|then|begin|end|procedure|function {
                printf( "A keyword: %s\n", yytext );
           }
```

- Observe the patterns. The actions are enclosed in curly parentheses.

Regular expressions (patterns)

http://flex.sourceforge.net/manual/Patterns.html#Patterns

- The regular expression is either specified directly in the rule or refers to a name from the definition section
  - The set of operators on regular expressions is quite evolved. It includes the standard operators: |, *, $+$
  - Names from the definition section are enclosed in `{`, `}` to refer to their definition.
  - The regular expression must not be intended. It ends at the first non-escaped whitespace. The action must start in the same line.
  - If a rule should be activated only in a certain state, the regular expression is prefixed by that state, enclosed in $<>$ ("start condition").

Actions

http://flex.sourceforge.net/manual/Index-of-Functions-and-Macros.html

An action is a sequence of C-statements.

- Examples of actions are pretty-printing, evaluating, AST building.
- An action is specified as regular `C` code.
- In addition, it may use macros or functions predefined by flex. Ex.:
  - `char *yytext` holds the text of the current token
  - `int yyleng` holds the length of the current token
  - `FILE *yyin`, `FILE *yyin` refer to input and output file
  - `BEGIN(s)` puts the lexer in state `s`.
- If no action is specified for a pattern, the pattern is discarded.

Example (rules section)

http://flex.sourceforge.net/manual/Simple-Examples.html#Simple-Examples

The lexer for the following Pascal-like language simply prints each token.

```
%%
{DIGIT}+       {  printf( "An integer: %s (%d)\n", yytext,
                  atoi( yytext ) );
               }

{DIGIT}+"."{DIGIT}*    {
                  printf( "A float: %s (%g)\n", yytext,
                  atof( yytext ) );
               }

if|then|begin|end|procedure|function    {
                  printf( "A keyword: %s\n", yytext );
               }
{ID}                printf( "An identifier: %s\n", yytext );

"+"|"-"|"*"|"/"     printf( "An operator: %s\n", yytext );
"{"[^{}\n]*"}"  /* eat up one-line comments */
[ \t\n]+            /* eat up whitespace */
.                   printf( "Unrecognized symbol: %s\n",yytext );
```

Example (definition section)

The first section of a flex file contains definitions that are needed in the later two sections.

- Name definitions; their form: `name definition`
- Start conditions are listed in a line that begins with `%`.
- Indented text or text enclosed `%{..%}` is copied verbatim to the output. It contains `include` directives for header files, macro definitions, or functions definitions.

Example

```
%{
     /* need this for the call to atof() below */
     #include <math.h>
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%
```

Example (user-code section)

The user-code section is copied to the file `lex.yy.c`.

- If the lexer is a stand-alone program, it contains a `main` routine, which calls `yylex()`. If the lexer collaborates with the parser, there is no `main` routine.
  - The function `yylex()` is generated by `flex`. It encapsulates token recognition.
  - It refers to a subroutine `yywrap()`. Either define that function or include `%option noyywrap` in the definition section or link the library `libfl.a`.

```
int main( int argc, char **argv ) {
    ++argv, --argc;  /* skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();
}
```

Workflow

The output of `flex` is the file `lex.yy.c`, which contains the routine `yylex()`.

```
> flex pascal.l
> g++  lex.yy.c -o lexer
> ./lexer prog.p
```

- `flex` generates the lexer.
- The `g++` compiler the lexer.
- The newly created program `lexer` runs on input programs in the source language.

BNFC lexer

The `flex` file that `bnfc` generates is used along with a parser.

- The definition section includes `Parser.h`, and defines two functions, `YY_BUFFER_RESET` and `YY_BUFFER_APPEND`, and a few macros.
- Rules are conditional, and actions return symbols (that are defined in the parser file).
- The user-code section initializes the lexer.

Outline

What is a finite automaton?

A finite automaton is a graph with a

- finite set of nodes (“states”)
- directed edges (“transitions”) between pairs of states
- symbols (“labels”) on each edge

Finite automata are used in many areas of computer science. In compiler construction

- states often carry a name,
- there is one start state and several terminal states.

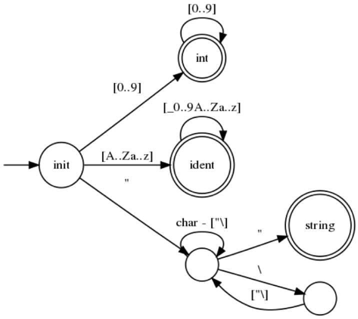
Lexers and finite automata

In the context of a lexer, finite automata are used for token recognition. Assume a token is specified as a regular expression.

- Represent that regular language as a finite automaton.
- Traverse the finite automaton (using the symbols of the input) for selecting transitions.
- If the traversal ends in a terminal state, the input is an expression of the language, thus accepted.

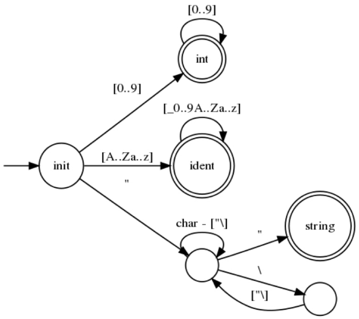
Example (finite automaton)

Consider the following finite automaton



- The start state is marked by an in-going arrow.
- Terminal states are marked by a double circle.
- Labels are symbols.

Example (finite automata as regular expression)



The automaton represents a regular expression:

```
     digit digit*
  |  letter ('_' | letter | digit)*
  |  '"' ((char - ('\' | '"') ) | '\' ('\' | '"'))* '"'
```

Example (recognition)

Assume an input string.

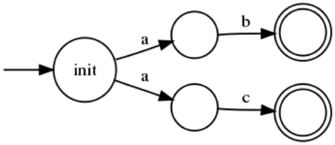
- Start from the initial state.
- Consider the first symbol of the input
  - If it is a digit, transition to state “int.”
  - If it is a character, transition to state “ident.”
  - If it is a quote, transition to the next (non-accepting) state without name.
  - If it is something else, the input is not element of the language (“illegal”). Return with error message.
- Repeat with the next symbol.

Deterministic vs. non-deterministic automata

- A deterministic automaton (DFA) is an automaton where for any given state and symbol not more than one transition exists.
- A non-deterministic automaton (NFA) is an automaton where a symbol may have more than one transition in a given state and where ϵ-transitions may exist (i.e., transitions that do not consume input).

Example (NFA)

Consider the following NFA



It represents the regular expression

$$a\ b \mid a\ c,$$

which recognizes the language $\{ab, ac\}$.

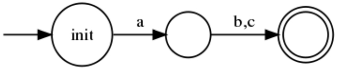
Example (DFA)

An alternative representation is the regular expression

$$a\,(b \mid c),$$

which recognizes the same language, $\{ab, ac\}$.

The corresponding DFA:



For automated processing, DFAs are preferable. For humans, NFAs are typically easier to write than DFAs.
Notice that in this case, the DFA is more compact than the equivalent NFA. Generally, that is not the case.

Major lexer algorithms

The three major algorithms of a lexer

- Step 1: convert a regular expression into an NFA
- Step 2: convert the NFA into an equivalent DFA
- Step 3: minimize the DFA

Step 1: Converting regular expressions into NFAs

Thompson's algorithm

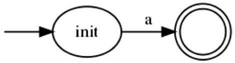
Recall the five constructors of regular expressions (regexps): concatenation, alternation, repetition, symbol, and empty.

Idea: proceed inductively

- Turn each regular expression into an NFA with exactly one start state and one end state.
- Start with NFAs for the basic regexps, symbol and empty.
- Generate NFAs for non-basic regexps by composing them from NFAs of their subexpressions, using ϵ-transitions.

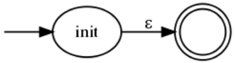
NFA generation, basic cases

- A regular expression that is just a symbol generates the following NFA:



  - “init” is a new start state and the second state is a final state.

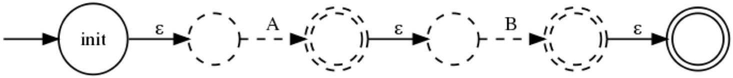
- The empty regular expression generates the following NFA:



  - “init” is a new start state and the second state is a final state.

NFAs of concatenated regular expressions

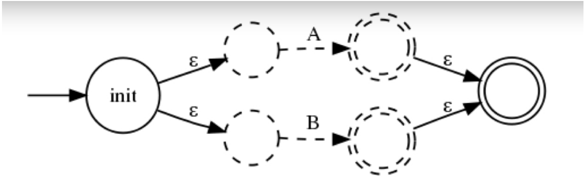
The NFA of a concatenation $A\,B$ connects the NFAs of A and B using ϵ-transitions and embedding the result in a NFA with one start and one terminal state:



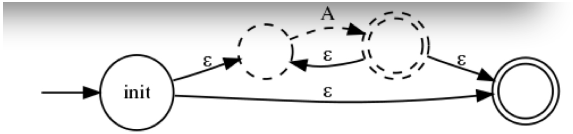
- Introduce a new start.
- Introduce an ϵ-transition from that start state to the start state of A. Merge the final state of A and the start state of B, and introduce an ϵ-transition from B to the final state.

NFAs of alternative and repetition

- The NFA of an alternative $A \mid B$ is formed by including a new start state and forking from it to the NFAs of A and B:



- The NFA of a repetition $A*$ is formed by including a new state, forking from it directly to the new terminal state and to the initial state of the NFA for A:



Finite automata, formally

A *finite automaton* is a tuple (Σ,S ,F,i,t) where

- Σ is a finite set of symbols (“alphabet”)
- S is a finite set of states
- $F \subseteq S$ is the subset of final states
- $i \in S$ denotes the initial state
- t: $S \times \Sigma \to \mathcal{P}(S)$ denotes the transition function.

NFA and DFA, formally

The notion of NFAs and DFAs can be formalized.

- A non-deterministic automaton is a finite automaton where the domain of the transition function is extended by the ϵ symbol.

$$t : S \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(S)$$

- An automaton is *deterministic* if $t(s, a)$ returns a single element. An automaton is *non-deterministic* if there exists a transition with $a = \epsilon$ or states and symbols s, a exist such that $t(s, a)$ returns a set.

Step 2: Converting NFAs to DFAs

Every NFA can be converted into a DFA for the same language.

Idea: subset construction

- For every state s and symbol a compute the new state $\sigma(s, a)$

$$\sigma(s, a) = \{s_1, \ldots, s_n\},$$

  the set of all states for which a transition exists from s under a.
- The transitions of $\sigma(s, a)$ for a symbol b are all the transitions for b from a state $s_i \in \sigma(s, a)$.
- A state $\sigma(s, a)$ is final if it contains a s_i that is a final state.

Conversion algorithm

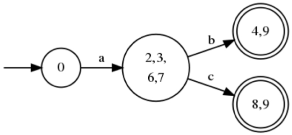
- Let s_o be the start state. Compute $\sigma(s_0, a)$ for all symbols a.
- Repeat: compute $\sigma(\sigma(s_0, a), a), \sigma(\sigma(s_0, a), b), \sigma(\sigma(\sigma(s_0, a), a), a), \ldots$

Note: the size of the DFA can be exponential in the size of the NFA.

Step 3: Minimization

Minimization is an optimization of DFAs that reduces the number of states by eliminating states with no *distinguishing string*.

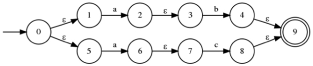
- A *distinguishing string* for two states s and t is a sequence of symbols that
  - ends in a terminal state if s is taken as start state
  - ends in a non-accepting state if t is taken as start state
- In the following DFA, the states 0 and $2, 3, 6, 7$ are distinguished by the input *ab*. The states $4, 9$ and $8, 9$ are not distinguished.



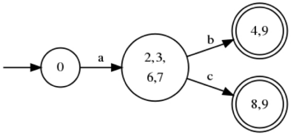
- Minimization reduces the automaton by states that are not distinguishable in the sense above.

Example (compiling a regular expression)

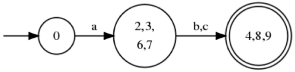
- NFA generation



- Determination



- Minimization



Correspondence theorem

The following notations and formalizations are equivalent:

- Regular languages
- Regular expressions
- Finite automata

Limits of regular expressions

Nested comments present a challenge for lexers.

- Consider the comment `a /* b /* c */ d */ e`.
- A standard lexer returns `a d */ e`.

The general problem behind: the language of balanced parentheses is not a regular language.

- Informally, “finite automata cannot count.”
- Need *context-free grammars* instead (see lecture on parsing).

Summary

- Tokens; regular languages and regular expressions; constructors
- Token recognition: ambiguities; longest match/priorities
- The `flex` lexer generator; work flow
- DFA, NFA, Thomson's algorithm, correspondence theorem
- Limits of regexps

References

- IPL, Ch.3.1-3.4