

Programming Languages, F2003

Lecture 3, Wednesday 19 February 2003

- Standard ML compilation units
- Concrete syntax
- Regular expressions and lexing
- Context-free grammars and parsing
- Lexer and parser specifications for an expression language
- A cookbook for parser and lexer generation with `mosmlyac` and `mosm1ex`
- The parser automaton and the parse stack
- Syntax error reporting in lexers and parsers
- Lexer and parser specifications for an SQL subset

Literature: Torben Mogensen, *Basics of Compiler Design* (DIKU 2002), sections 2.1–2.5, 2.9, 3.1–3.6, 3.16.

Also, draft lecture notes *Programming Language Concepts*, chapter 3 (handout).

IT-C

Programming Languages, F2003

Page 3-1

SML structures and Moscow ML compilation units

So far we have used Moscow ML's interactive top-level, `mosml.c`.

When we modularize programs, we may use Moscow ML's batch compiler, `mosml.c`.

For modularity we declare the abstract syntax datatype `expr` in a file `Absynr.sml`:

```
datatype expr =
  CstI of int
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr list
```

This file defines a structure `Absynr`, similar to a class that has only static members. We can compile it like this:

```
mosmlc -c Absynr.sml
```

This creates a bytecode file `Absynr.uo` and an interface file `Absynr.vi`, corresponding to a Java class file.

When using `mosml.c` to compile other files, `Absynr` is loaded automatically if needed.

But in the interactive system `mosml`, you may have to explicitly load `Absynr.sml`:

```
- load "Absynr";
> val it = () : unit
```

In both cases, all names must be prefixed by `Absynr`, as in `Absynr.Var`.

Alternatively, after the declaration `open Absynr`, the prefix is not needed.

IT-C

Programming Languages, F2003

Page 3-2

Describing legal program syntax: regular expressions and context-free grammars

Programming language syntax is described on two levels, the local one and the global one:

- Lexical: the form of names, constants, keywords, operators, delimiters, ...
- Lexical structure is described by *regular expressions*.
- Grammatical: the structure of expressions, declarations, statements, programs, ...
- Grammatical structure is described by *context-free grammars*.

The description of legal program syntax serves two purposes:

- It tells programmers (and compiler writers) what a legal program must look like.
- It permits automatic construction of lexer and parser programs, to be used in a compiler.

IT-C

Programming Languages, F2003

Page 3-3

From concrete syntax to abstract syntax: lexing and parsing

A program is written as a linear character stream, usually stored in a file.

For instance, the program text `x+5 2 * wk` really is this character stream:

```
| x | + | 5 | 2 | | * | | w | k |
```

To interpret or compile this program, we must transform the character stream to an abstract syntax tree.

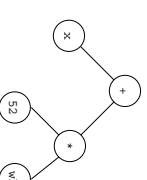
This is done in two phases:

(1) **Lexing** (or **scanning**) checks that the characters form legal tokens, and if so, produces a token stream:

```
NAME "x", PLUS, CSTINT 52, TIMES, NAME "wk"
```

A token (or lexeme) represents a simple constant, an operator, a name, a delimiter, or similar.

(2) **Parsing** checks that the token stream is legal, and if so, constructs an abstract syntax tree (AST):

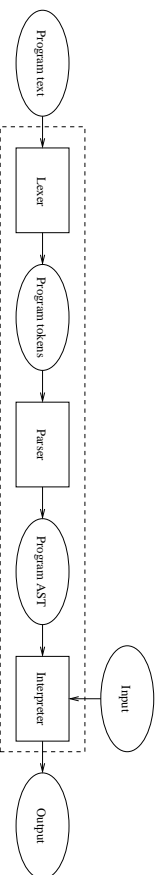


IT-C

Programming Languages, F2003

Page 3-4

From program text to token stream to abstract syntax tree



Regular expressions: compact descriptions of sets of strings

A regular expression r is a character, or empty, or sequence r_1r_2 , or iteration r_1^* , or choice $r_1|r_2$:

Regular expression r	Meaning	Language $L(r)$
a	the character a	$\{ "a" \}$
ϵ	the empty string	$\{ "" \}$
r_1r_2	r_1 followed by r_2	$\{ v_1v_2 \mid v_1 \in L(r_1), v_2 \in L(r_2) \}$
r_1^*	zero or more r_1 's	$\{ v_1v_2 \dots v_n \mid v_i \in L(r_1), n \geq 0 \}$
$r_1 r_2$	either r_1 or r_2	$L(r_1) \cup L(r_2)$

Examples:

- If r is ab then $L(r)$ is the set $\{ "ab" \}$
- If r is $a|b$ then $L(r)$ is the set $\{ "a", "b" \}$
- If r is $a|b$ then $L(r)$ is the set $\{ "aa", "ab" \}$
- If r is a^* then $L(r)$ is the infinite set $\{ "", "a", "aa", "aaa", \dots \}$
- If r is $a^*(a|b)$ then $L(r)$ is the infinite set $\{ "a", "b", "aa", "ab", "aaa", \dots \}$
- If r is $a(a|b)^*$ then $L(r)$ is $\dots ?$

Abbreviations in regular expressions

Some abbreviations make life easier:

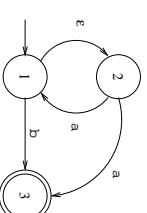
Abbreviation	Meaning	Expansion
$[aeiou]$	a or e or i or o or u	$a e i o u$
$[0-9]$	a character in range 0 to 9	$0 1 2 3 4 5 6 7 8 9$
$[a-zA-Z]$	a character in range a to z or A to Z	$a b \dots z A B \dots Z$
$r_1^?$	empty or r_1	ϵr_1
r_1^+	one or more r_1 's	$r_1r_1^*$

Examples:

- Non-negative integer constants are described by $[0-9]^+$
- Integers constants are described by $[-+]?[0-9]^+$
- Java variable names are described by $([a-zA-Z\$]|_)([a-zA-Z0-9\$]|_)^*$
- Java floating-point constants are described by $[-+]?[0-9]^+(\.[0-9]^+)?([Ee][+-]?[0-9]^+)?$
- Car license plates are described by $\dots ?$
- Internet domains (such as `www.p01.dk`) are described by $\dots ?$
- E-mail addresses are described by $\dots ?$

Regular expressions and nondeterministic finite automata (NFAs)

A *nondeterministic finite automaton* (NFA) is a graph of states (nodes) and labelled transitions (edges):



The automaton has a starting state s_0 (here $s_0 = 1$) and any number of accepting states (circled).

It *accepts* a string s if there is a path $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \dots \xrightarrow{\ell_n} s_n$ from s_0 to an accept state s_n , where the label sequence $\ell_1\ell_2 \dots \ell_n$ makes up s .

An epsilon-transition (ϵ) does not contribute to the label sequence.

Useful fact

For every regular expression r there is an NFA that accepts exactly the strings in $L(r)$.

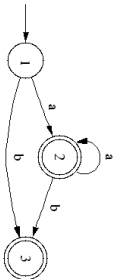
The NFA above accepts exactly $L(a^*(a|b))$.

The NFA can be constructed in a systematic way from the regular expression; see *Mogensen*.

(Also, for every NFA there is a regular expression r such that the NFA accepts exactly the strings in $L(r)$).

Regular expressions and deterministic finite automata (DFAs)

A *deterministic finite automaton* (DFA) is an NFA without ϵ -transitions, and with distinct labels on all outgoing edges from a state:



Useful facts

For every NFA there is a DFA that accepts the same set of strings.

Thus for every regular expression r there is a DFA that accepts exactly the strings in $L(r)$.

For a given string s , the next state (if any) is uniquely determined. So acceptance can be decided in time $O(|s|)$.

An NFA nfa is constructed from a regular expression r by recursion on the structure of r .

A DFA dfa is constructed from nfa by letting a state S of dfa be a non-empty set of states of nfa .

There is a transition $S \xrightarrow{a} S'$ in dfa if there are $s \in S$ and $s' \in S'$ with $s \xrightarrow{a} s'$ in a in the given nfa .

State S of dfa is accepting if there is $s \in S$ such that s is accepting in the nfa .

Lexer specification for an expression language: the rules

The DFAs are usually constructed by a *lexer generator* from a *lexer specification*.

A lexer specification uses regular expressions to describe the various kinds of tokens.

In this lexer specification, a token is a non-negative integer constant, or a name or keyword, or a special symbol:

```
rule Token = parse
  | ['0'-'9']+          { Token lexbuf }
                        { case Int.fromString (getLexeme lexbuf) of
                          NONE => LexerError lexbuf "internal error"
                          | SOME i => CSTINT i
                        }
  | ['a'-'z','A'-'Z'] ['a'-'z','A'-'Z','0'-'9']*
                        { keyword (getLexeme lexbuf) }
  | '+'                { PLUS }
  | '-'                { MINUS }
  | '*'                { TIMES }
  | '='                { EQ }
  | '('                { LPAR }
  | ')'                { RPAR }
  | eof                { EOF }
  | _                  { LexerError lexbuf "Illegal symbol in input" }
;
```

Given the lexer specification, a lexer generator builds a DFA that *recognizes* and *classifies* tokens.

Context-free grammars

A *context-free grammar* G consists of

- terminal symbols: identifiers x , integer constants 1,2, etc.; usually tokens as produced by a lexer
- nonterminal symbols A , denoting grammar classes
- a start symbol S , which is a nonterminal
- rules (or productions) of the form $A ::= \text{thseq}$ where *thseq* is a sequence of terminals or nonterminals

An example grammar for our expression language:

```
Main ::= Expr EOF           rule 1
Expr ::= NAME               rule 2
      | INT                 rule 3
      | - INT               rule 4
      | ( Expr )            rule 5
      | let NAME = Expr in Expr end rule 6
      | Expr * Expr         rule 7
      | Expr + Expr         rule 8
      | Expr - Expr         rule 9
```

The grammar has terminal symbols NAME, INT, -, (,), let, =, in, end, *, +, EOF.

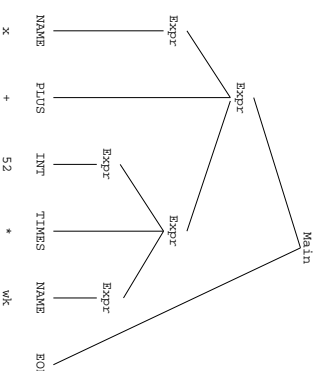
It has nonterminals Main and Expr, its start symbol is Main, and it has 9 rules.

Derivation: the grammar as a string generator

The string "x+52*wk EOF" or "NAME + INT * NAME EOF" is derivable as follows:

```
Main => Expr EOF           by rule 1
      => Expr + Expr EOF  by rule 8
      => Expr + Expr * Expr EOF by rule 7
      => Expr + Expr * NAME EOF by rule 2
      => Expr + INT * NAME EOF by rule 3
      => NAME + INT * NAME EOF by rule 2
```

The derivation as a tree:



Parsing is Inverse derivation

A string s is *derivable* from G if there is a sequence of rule applications that transforms G 's start symbol to s . The language $L(G)$ defined by a grammar G is the set of strings derivable from G .

Recognition: given a string s , is s derivable from G ; that is, is $s \in L(G)$?

Parsing: given a string s , find a derivation from G , if any.

We shall work with machine-generated bottom-up parsers called LR -parsers.

They read input from the *Left* and do a bottom-up reconstruction of the derivation tree that would result from always doing derivations from the *Rightmost* nonterminal.

We use the $LR(1)$ -parser class, which considers the next move by looking at only one additional input symbol.

Most hand-written parsers are top-down LL -parsers, also called *recursive descent* parsers.

They read input from the *Left* and do a top-down reconstruction of the derivation tree that would result from always doing derivations from the *Leftmost* nonterminal.

$LR(1)$ -parsers are more powerful than $LL(1)$ -parsers, but too complicated to write by hand.

We shall use an $LR(1)$ -parser generator `mosm1.yacc` for SML. There are such tools for all major languages.

IT-C

Parser specification for an expression language: header and declarations (`expr/Exprpar.grm`)

The header opens the `Absyn` (abstract syntax) structure for the expression language.

The declarations list the tokens (nonterminal symbols), give the precedence and associativity of operators, declare `Main` to be the start symbol, and declares the nonterminal symbols and their type.

```
% {
  open Absyn;
}

%token <int> CSTINT
%token <string> NAME
%token PLUS MINUS TIMES EQ
%token END IN LET
%token LPAR RPAR
%token EOF

%left MINUS PLUS           /* lowest precedence */
%left TIMES                /* highest precedence */

%start Main
%type <Absyn.expr> Main Expr
%%
... grammar rules ...
```

IT-C

Ambiguity of grammars: precedence and associativity of operators

A grammar is *ambiguous* if there is some string s for which there is more than one derivation.

For instance, $1 + 2 * 3$ may be derived in two ways: we have not specified the precedence of $+$ and $*$. Also, $1 - 2 - 3$ may be derived in two ways: we have not specified the associativity of $-$.

A grammar G can sometimes be made unambiguous by changing the grammar (without changing $L(G)$).

In practice, operator precedence and associativity declarations are often used to resolve ambiguity.

Usually operator $*$ has *higher precedence* than operator $+$, so it binds more strongly.

Usually the operator $-$ is *left associative*, so $1 - 2 - 3$ must be read as $(1 - 2) - 3$.

In SML, the operators `::` and `@` are right associative, so `1 :: 2 :: []` means `1 :: (2 :: [])`.

Which operators in Java are right associative?

What does it mean for an operator to be non-associative?

Which operators in Java are non-associative?

IT-C

Parser specification for an expression language: grammar rules with semantic actions

Every rule has a 'semantic action' within `{ ... }` that computes the result of parsing the left-hand side.

This result is called a *synthesized attribute*, usually it is the abstract syntax tree for the construct.

```
Main:
;
  Expr EOF      { $1 }
;
Expr:
;
  NAME          { Var $1 }
  | CSTINT      { CstI $1 }
  | MINUS CSTINT { CstI (~ $2) }
  | LPAR Expr RPAR { $2 }
  | LET NAME EQ Expr IN Expr END { Let($2, $4, $6) }
  | Expr TIMES Expr { Prim("**", [$1, $3]) }
  | Expr PLUS Expr { Prim("+", [$1, $3]) }
  | Expr MINUS Expr { Prim("-", [$1, $3]) }
;
```

A `Main` must be an `Expr` followed by `end-of-file`, and it returns `$1`, the abstract syntax for the `Expr`.

An `Expr` is a variable (`NAME`), or an integer constant, or a negative integer, or a parenthesized expression, etc.

In each case, the abstract syntax of the parsed construct is returned.

For instance, in `Let($2, $4, $6)`, the `$2` is the string in the `NAME`, `$4` is the first `Expr`, etc.

IT-C

Outline of a lexer specification

```
{
  ... header ...
}

rule E1 =
  parse regexp { action }
  | ...
  | regexp { action }
and E2 =
  parse ...
and ...
;
```

The { ... header ... } contains SML declarations.

It usually opens the Lexing structure and the relevant parser structure, to get access to the token datatype.

The rule part declares rules for getting the next token.

Each action part must contain an SML expression of type token.

Lexer specification for an expression language: the header

The header part must open the built-in Lexing structure and the relevant parser structure.

It also declares an exception and a function for error reporting, and other auxiliary functions.

```
{
  open Lexing Exprpar;

  exception LexicalError of string * int * int (* (message, loc1, loc2) *)

  fun lexerError lexbuf s =
    raise LexicalError (s, getLexemeStart lexbuf, getLexemeEnd lexbuf);

  (* Function to classify names as keywords or identifiers: *)
  fun keyword s =
    case s of
      "let"   => LET
    | "in"    => IN
    | "end"   => END
    | _      => NAME s;
}

... rule section left out as it was shown before ...
```

A cookbook approach to parser and lexer generators

File `Abdsyn.sml` contains the declaration of the abstract syntax datatype. Compile it by running

```
mosmlc -c Abdsyn.sml
```

File `Exprpar.grm` contains a parser specification. Generate a parser by running

```
mosmlyac -v Exprpar.grm
```

If successful, the result is an SML structure in file `Exprpar.sml` and signature in `Exprpar.sig`.

Compile them by running

```
mosmlc -c -liberal Exprpar.sig Exprpar.sml
```

File `ExprLex.lex` contains a lexer specification. Generate a lexer by running

```
mosmllex ExprLex.lex
```

If successful, the result is an SML program in file `ExprLex.sml`. Compile it by running

```
mosmlc -c ExprLex.sml
```

Finally, file `parse.sml` contains several functions that use the generated lexer and parser. Load it with

```
mosml parse.sml
```

Using the generated lexer and parser

Function `parsep : string -> expr` takes a string and creates a lexer buffer (a character stream).

Then it calls the parser (`Exprpar.Main`) to get an abstract syntax tree of type `expr`.

The parser uses the lexer (`ExprLex.Token`) to read tokens from the lexer buffer (character stream).

```
fun parsep str =
  let val lexbuf = Lexing.createLexerString str
      val expr  = Exprpar.Main ExprLex.Token lexbuf
  in
    Parsing.clearParser();
    expr
  end
handle exn => (Parsing.clearParser(); raise exn);
```

What are the expected results of

```
parsep "x+52 * wk" ;
parsep "10+11+12" ;
parsep "10+z" ;
parsep "10+ * 8" ;
parsep "(10+z)" ;
parsep "10 ?? z" ;
```

The `expr / Exprpar . output` file: how does the parser work?

The grammar (extended with auxiliary start and end symbols `$accept` etc):

```

0 $accept : %entry% $end
1 Main : Expr EOF
2 Expr : NAME
3       | CSTINT
4       | MINUS CSTINT
5       | LPAR Expr RPAR
6       | LET NAME EQ Expr IN Expr END
7       | Expr TIMES Expr
8       | Expr PLUS Expr
9       | Expr MINUS Expr
10 %entry% : '\001' Main

```

The second half of the `Exprpar . output` file describes the states of a deterministic finite automaton.

This parser automaton has a *parse stack*, containing automaton state numbers and grammar symbols.

Some sample parser automaton states from `Exprpar . output`

State 4: Parsing `Expr`: has seen `let`; expects `NAME`, then `EQ`, then `Expr` etc.

If next input is `NAME`, read (shift) it, and go to state 10; otherwise terminate with a parse error:

```

state 4
Expr : LET . NAME EQ Expr IN Expr END (6)
NAME shift 10
. error

```

State 5: Parsing `Expr`: has seen left parenthesis; expects `Expr` and then right parenthesis.

If next input is a constant or `let` or `'(` or `'-'` or a `NAME`, read (shift) it and go to state 3, 4, 5, 6, or 7; else error:

```

state 5
Expr : LPAR . Expr RPAR (5)
CSTINT shift 3
LET shift 4
LPAR shift 5
MINUS shift 6
NAME shift 7
. error
Expr goto 11

```

If we succeed parsing the `Expr`, go to state 11.

Another parser automaton state

State 20: Parsing `Expr`: we have seen `Expr PLUS Expr`; we expect `TIMES` or `PLUS` or `MINUS` or something that finishes the `Expr`.

If next input is `TIMES`, read (shift) it and go to state 16.

If next input is `end` or `EOF` or `in` or `MINUS` or `PLUS` or `'(`, do not read it, but reduce `Expr PLUS Expr` to `Expr` by grammar rule 8:

```

state 20
Expr : Expr . TIMES Expr (7)
Expr : Expr . PLUS Expr (8)
Expr : Expr . (8)
Expr : Expr . MINUS Expr (9)
TIMES shift 16
END reduce 8
EOF reduce 8
IN reduce 8
MINUS reduce 8
PLUS reduce 8
RPAR reduce 8

```

Parsing "`x+52*wk EOF`"

Input	Parse stack (top on right)	Action
<code>x+52*wk EOF</code>	<code>#0 \001 #1</code>	shift #1
<code>x+52*wk EOF</code>	<code>#0 \001 #1 x #7</code>	shift #7
<code>+52*wk EOF</code>	<code>#0 \001 #1 Expr</code>	reduce 2
<code>+52*wk EOF</code>	<code>#0 \001 #1 Expr #9</code>	goto #9
<code>52*wk EOF</code>	<code>#0 \001 #1 Expr #9 + #15</code>	shift #15
<code>*wk EOF</code>	<code>#0 \001 #1 Expr #9 + #15 52 #3</code>	shift #3
<code>*wk EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr</code>	goto #20
<code>wk EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20</code>	shift #16
<code>EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20 * #16</code>	shift #7
<code>EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20 * #16 wk #7</code>	reduce 2
<code>EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20 * #16 Expr</code>	goto #21
<code>EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20 * #16 Expr #21</code>	reduce 7
<code>EOF</code>	<code>#0 \001 #1 Expr #9 + #15 Expr #20</code>	goto #20
<code>EOF</code>	<code>#0 \001 #1 Expr #9</code>	reduce 8
<code>EOF</code>	<code>#0 \001 #1 Expr #9</code>	goto #9
<code>EOF</code>	<code>#0 \001 #1 Expr #9 EOF #13</code>	shift #13
<code>EOF</code>	<code>#0 \001 #1 Expr #9 EOF #13</code>	reduce 1
<code>EOF</code>	<code>#0 \001 #1 Main #8</code>	goto #8
<code>EOF</code>	<code>#0 \001 #1 Main #8</code>	reduce 10
<code>EOF</code>	<code>#0 %entry%</code>	goto #2
<code>EOF</code>	<code>accept #2</code>	accept

Notation: `#0`, `#1`, etc are parser automaton states; `0`, `1`, etc are grammar rule numbers. **Note reduce order.**

Syntax error reporting in lexers and parsers

The lexer and parser are far more useful if they can pinpoint errors in the source program.

This error reporting machinery distinguishes parse errors from lexical errors:

```
fun parseExprReport file stream lexbuf =
  let val expr =
      Exprpar.Main ExprLex.Token lexbuf
    handle
      Parsing.ParseError f =>
        let val pos1 = Lexing.getStart lexbuf
            val pos2 = Lexing.getEnd lexbuf
        in
          Location.errMsg (file, stream, lexbuf)
            (Location.loc(pos1, pos2))
            "Syntax error."
        end
      | ExprLex.LexicalError(msg, pos1, pos2) =>
        if pos1 >= 0 andalso pos2 >= 0 then
          Location.errMsg (file, stream, lexbuf)
            (Location.loc(pos1, pos2))
            ("Lexical error: " ^ msg)
        else
          (Location.errPrompt ("Lexical error: " ^ msg ^ "\n\n");
           raise Fail "Lexical error");
    in
      Parsing.clearParser();
    expr
  end
handle exn => (Parsing.clearParser()); raise exn);
```

Parse conflicts

The expression grammar would be ambiguous were it not for the precedence and associativity declarations.

Removing them makes `mosml_yacc` report parse conflicts; for instance:

```
20: shift/reduce conflict (shift 14, reduce 8) on MINUS
20: shift/reduce conflict (shift 15, reduce 8) on PLUS
20: shift/reduce conflict (shift 16, reduce 8) on TIMES
state 20
  Expr : Expr . TIMES Expr (7)
  Expr : Expr . PLUS Expr (8)
  Expr : Expr PLUS Expr . (8)
  Expr : Expr . MINUS Expr (9)
  MINUS shift 14
  PLUS shift 15
  TIMES shift 16
  END reduce 8
  EOF reduce 8
  IN reduce 8
  RPAR reduce 8
```

The first conflict is: should `11 + 22 - 33` be parsed as `(11+(22-33))` or as `((11+22)-33)?`

The second is: should `11 + 22 + 33` be parsed as `(11+(22+33))` or as `((11+22)+33)?`

The third is: should `11 + 22 * 33` be parsed as `(11+(22*33))` or as `((11+22)*33)?`

In general, `mosml_yacc` may report conflicts even on unambiguous grammars; it has limited lookahead.