Programs as data

From concrete syntax to abstract syntax: Lexing and parsing

Peter Sestoft

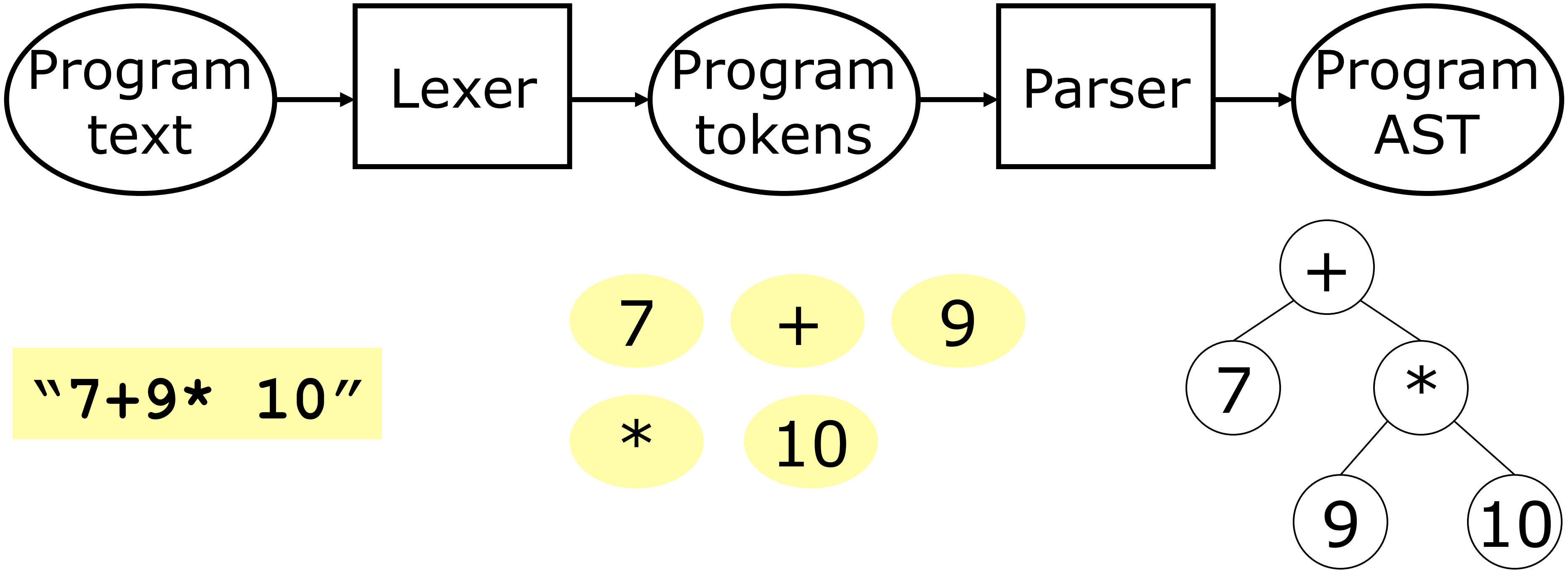
Monday 2012-09-10

Plan for today

- Lexing and parsing: From text to tokens to abstract syntax
- Lexer specifications
  - Regular expressions
  - Automata
  - The fslex lexer generator tool
- Parser specifications
  - Grammars
  - The fsyacc parser generator tool
- Anders Hejlsberg (C#) TechTalk Thu 4 Oct:
  https://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032528197&Culture=da-DK&community=0

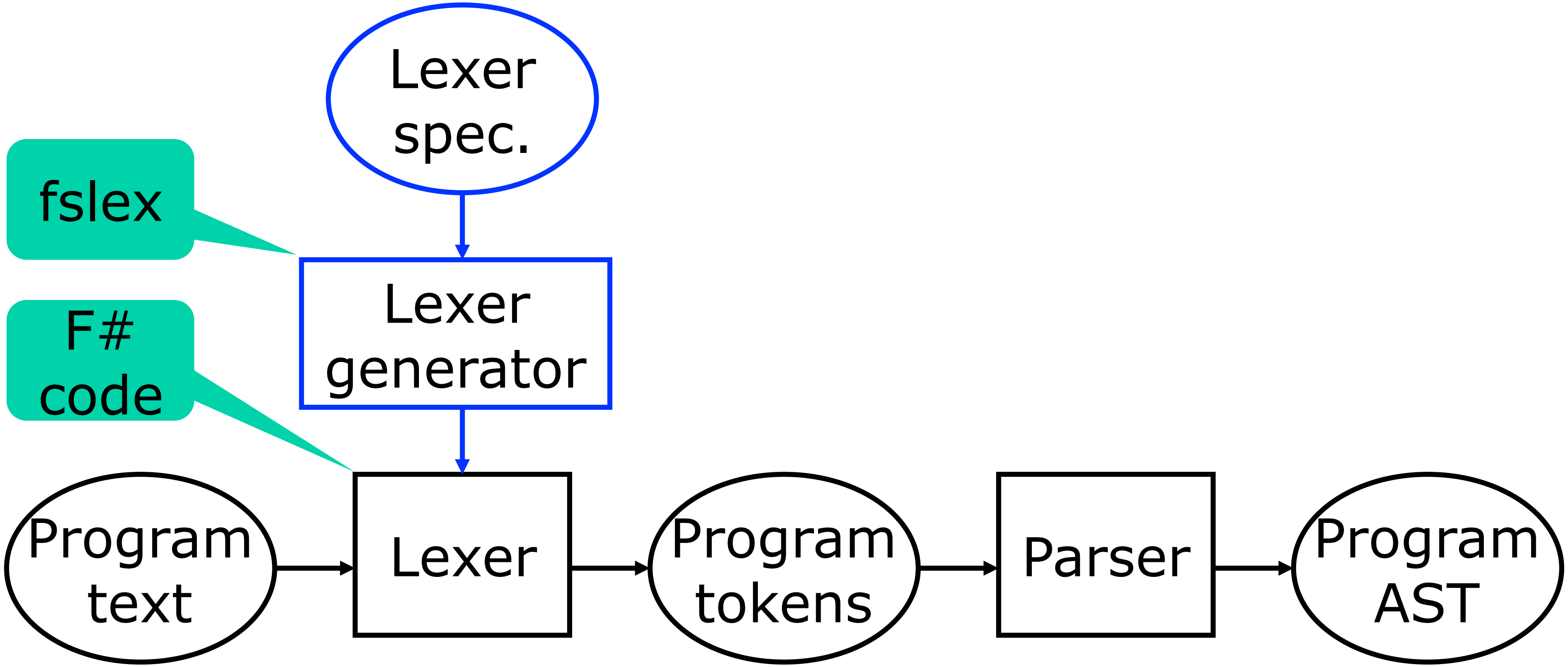
From text file to abstract syntax

- Abstract syntax is very tiresome
  `Prim("+",CstI 7,Prim("*",CstI 9,CstI 10))`
- Programmers want to write source code: text!



Lexers and lexer generators

- A *lexer* converts a character stream to a token stream
- A *lexer specification* is a description of tokens
- A *lexer generator* takes as input a lexer specification, and generates a lexer



Regular expressions (r.e.)

- A regular expression describes a *set of strings*

R.E. r	Meaning	Language L(r)
a	symbol a	{ "a" }
ε	empty sequence	{ "" }
r_1 r_2	r_1 followed by r_2	$\{ s_1 s_2 \mid s_1 \in L(r_1), s_2 \in L(r_2) \}$
r *	zero or more r	$\{ s_1 \ldots s_n \mid s_i \in L(r), n \geq 0 \}$
r_1 | r_2	r_1 or else r_2	$L(r_1) \cup L(r_2)$

- ab* represents { "a", "ab", "abb", ... }
- (ab)* represents { "", "ab", "abab", ... }
- (a|b)* represents { "", "a", "b", "ab", "ba" ... }
- (a|b)c* represents ?

Regular expression abbreviations

Abbrev	Meaning	Expansion
[aeiou]	set	a | e | i | o | u
[0-9]	range	0 | 1 | … | 9
[a-zA-Z]	ranges	a | … | z | A | … | Z
r?	zero or one r	ε | r
r+	one or more r	r r*

Examples and joint exercises

- Write regular expressions for
  - Non-negative integer constants
  - Integer constants
  - Floating-point constants: **`3.14   3E8   +6.02E23`**
  - Java variable names: **`xy   x12   _x   $x12   ...`**

Lexer specification (ExprLex.fsl)

Tokens: constant, name, +, -, *, =, (,), eof:

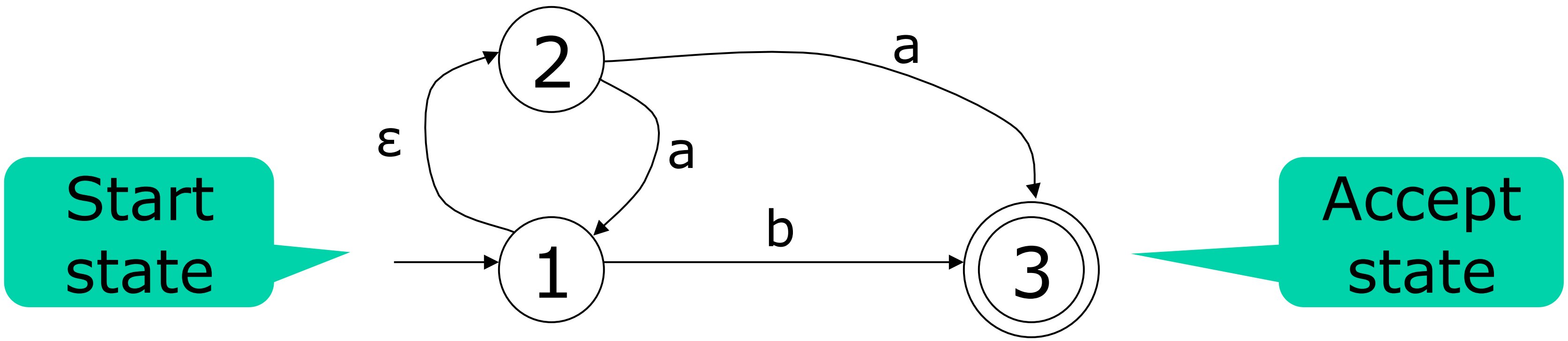
```
rule Token = parse
  | [' ' '\t' '\n' '\r']  { Token lexbuf }
  | ['0'-'9']+            { CSTINT (...) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                          { keyword (...) }
  | '+'                   { PLUS  }
  | '-'                   { MINUS }
  | '*'                   { TIMES }
  | '='                   { EQ    }
  | '('                   { LPAR  }
  | ')'                   { RPAR  }
  | eof                   { EOF   }
  | _                     { lexerError lexbuf "Bad char" }
```

Regular expressions

Corresponding tokens

Finite automata (FA), finite state machines

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



- An FA *accepts* string s if there is a path from start to an accept state such that the labels make up s
- Epsilon (ε) does not contribute to the string
- This automaton is *nondeterministic*: an NFA
- It accepts string "b"
- Does it accept "a" or "aa" or "ab" or "aba"?

Regular expressions and finite automata

- *For every regular expression r there is a finite automaton that recognizes exactly the strings described by r*
- The converse is also true
  – What r.e. does our automaton represent?
- Construction:

  Regular expression

  => Nondeterministic finite automaton (NFA)

  => Deterministic finite automaton (DFA)
- Gives a very efficient way of determining whether a given string is described by a regular expression

From regular expression to NFA

- Recursively, by case on the form of the regular expression:

‘a’

ε

$r_1\ r_2$

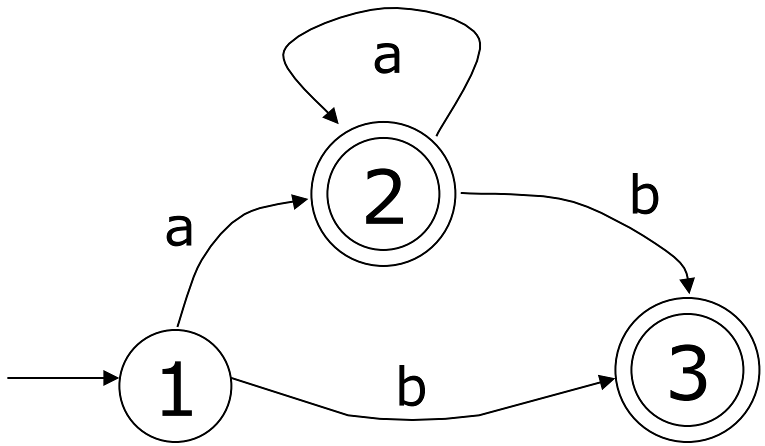
r*

$r_1 \mid r_2$

Exercises: Make NFAs for (ab)* and (a|b)*

From NFA to DFA

- A *deterministic* FA has no ε-transitions, and distinct labels on all transitions from a state



Multiple accept states OK

- A DFAs is easy to implement with a 2D table:

  `nextstate = table[currentstate][nextsymbol]`

- Decides in *linear time* whether it accepts string s
- *For every NFA there is a corresponding DFA*
  - DFA state = epsilon-closed set of NFA states
  - There is a DFA transition from S_1 to S_2 on x if there is an NFA state in S_1 with a transition to an NFA state in S_2 on x

Example NFA to DFA constructions

- ε-closure(s) = { t | t reachable from s on ε }
- Make DFA from NFA for (ab)*
- Make DFA from NFA for (a|b)*

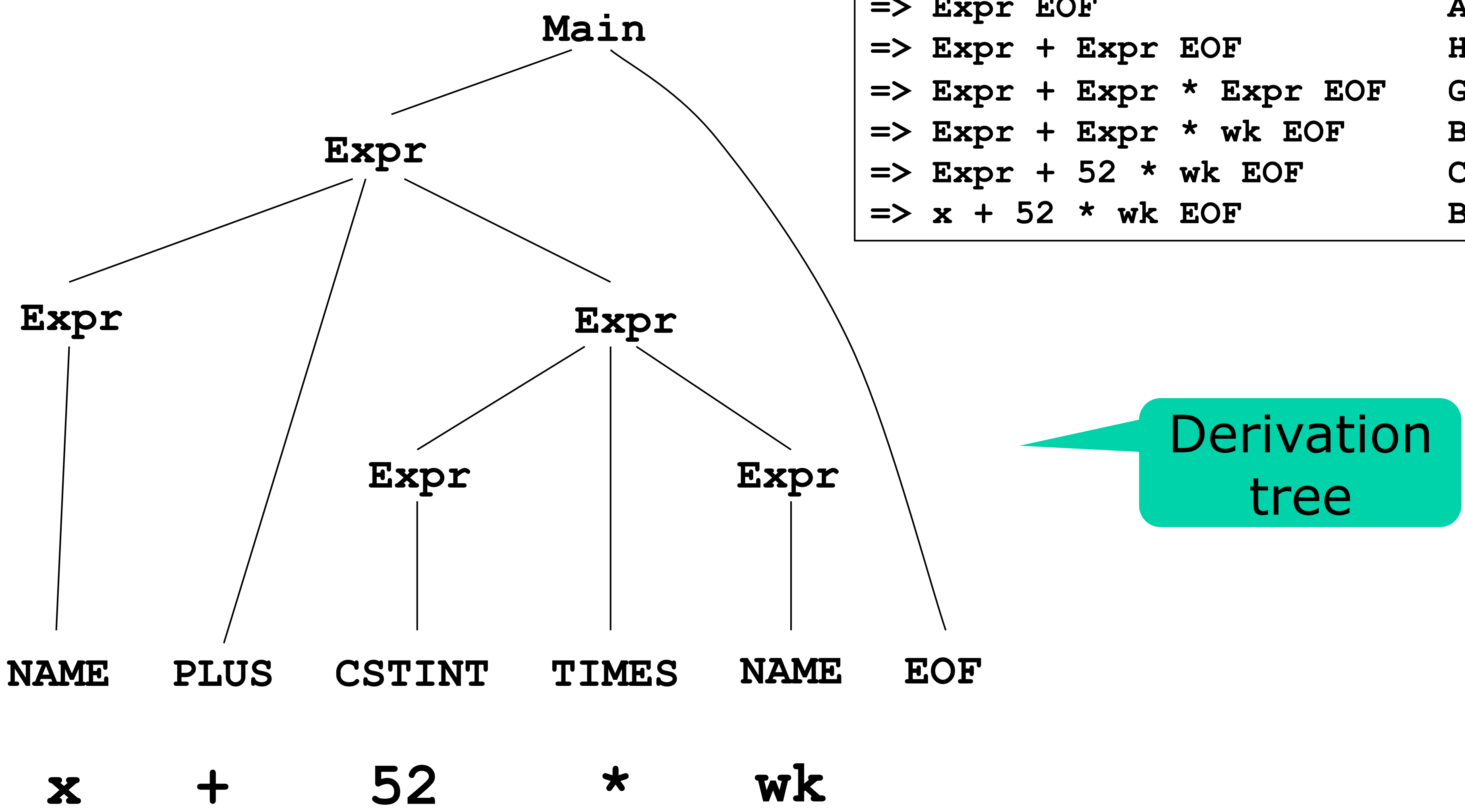
- More tricky:
  Exercise 1.2 from Mogensen ICD 2011
  (equals exercise 2.2 from Mogensen 2010):
  - (i) Make NFA for a*(a|b)aa
  - (ii) Convert this NFA to an equivalent DFA

Context-free grammar (CFG), example

```
Main ::= Expr EOF                                 rule A
Expr ::= NAME                                     rule B
      |  CSTINT                                   rule C
      |  - CSTINT                                 rule D
      |  ( Expr )                                 rule E
      |  let NAME = Expr in Expr end              rule F
      |  Expr * Expr                              rule G
      |  Expr + Expr                              rule H
      |  Expr - Expr                              rule I
```

- *Nonterminal* symbols: **`Main`**, **`Expr`**
- *Terminal* symbols, or *tokens*: **`NAME`**, **`CSTINT`**, **`MINUS`**, **`LPAR`**, **`RPAR`**, ...
- Grammar *rules*, or *productions*: A to I
- *Start symbol* (a nonterminal): **`Main`**

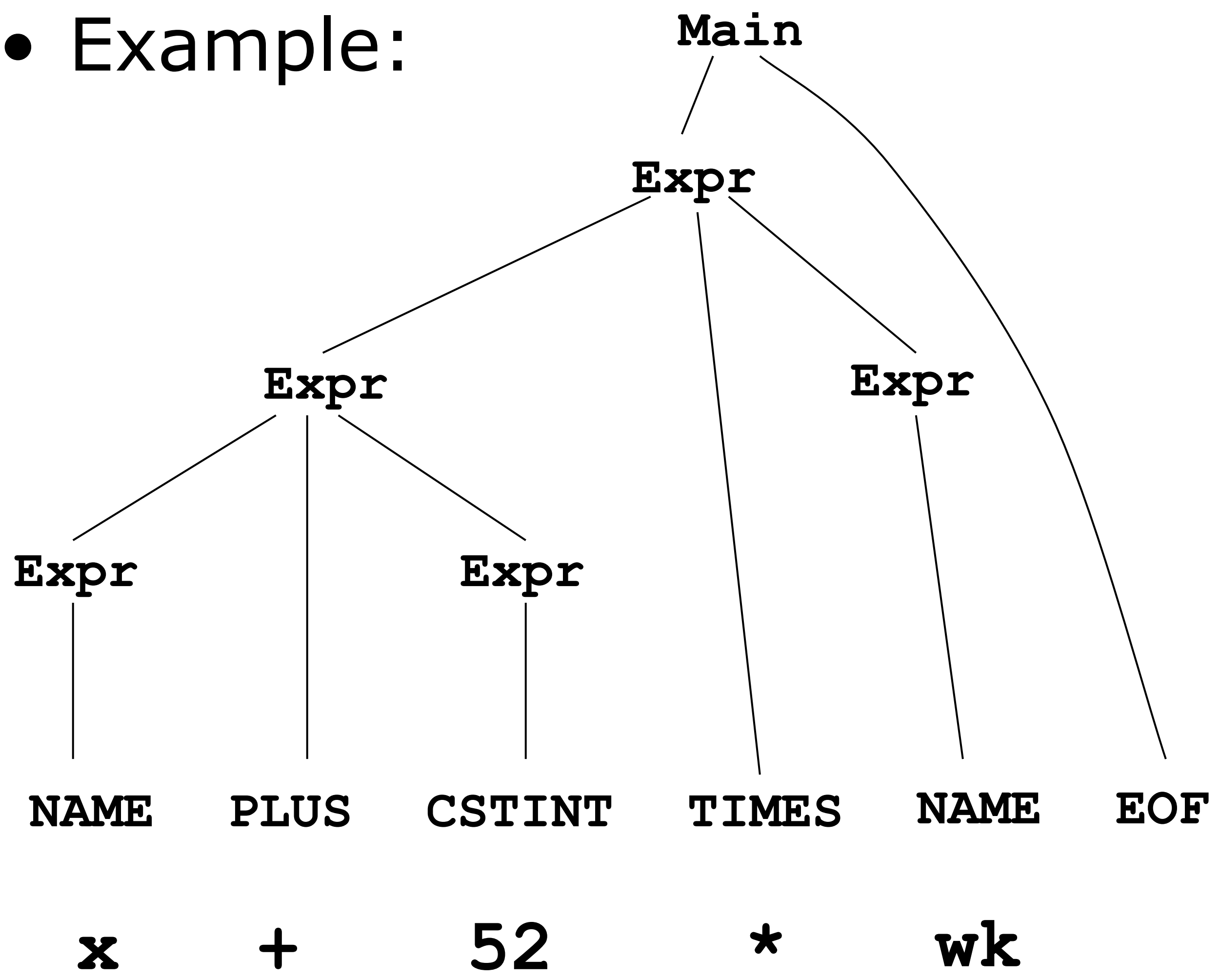
Derivation: grammar as string generator




```
Main
=> Expr EOF                    A
=> Expr + Expr EOF             H
=> Expr + Expr * Expr EOF      G
=> Expr + Expr * wk EOF        B
=> Expr + 52 * wk EOF          C
=> x + 52 * wk EOF             B
```

Grammar ambiguity

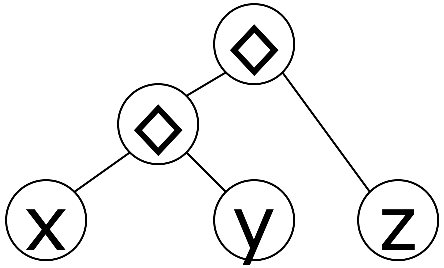
- A grammar is *ambiguous* if there is a string that has more than one derivation tree
- Example:

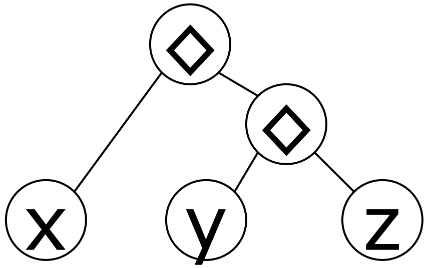


Operator associativity and precedence

- **Associativity**: How should we read x ◇ y ◇ z ?

◇ left-assoc.

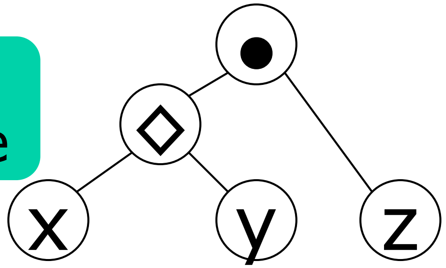


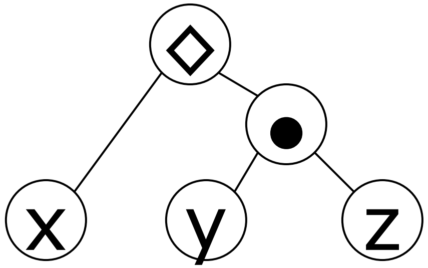


◇ right-assoc.

- **Precedence**: How should we read x ◇ y • z ?

◇ higher precedence





• higher precedence

- What Java/C# operators
  - are left-associative?
  - are right-associative?
  - have different precedence than others?

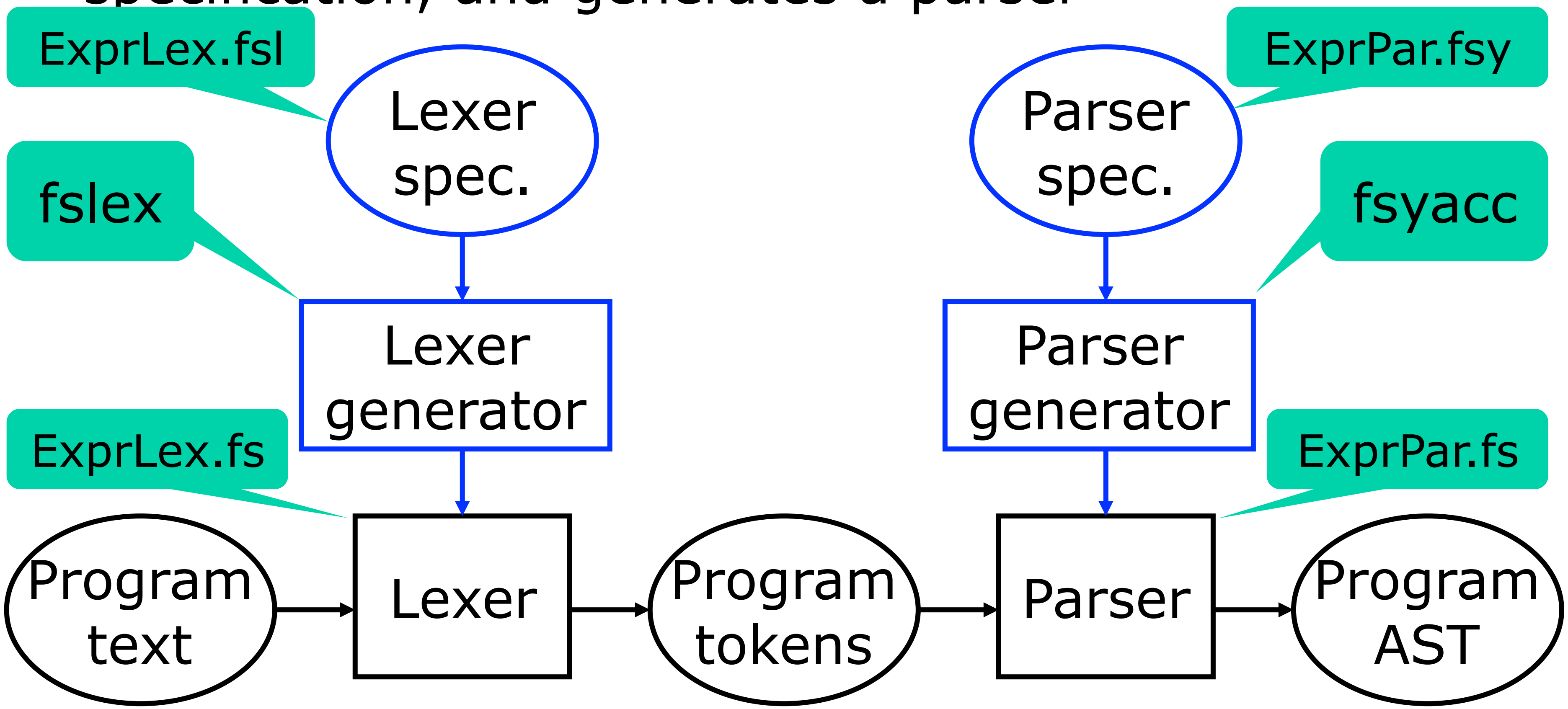
Parsing is inverse derivation

- Parsing: Given a grammar and a string
  - Determine whether the string can be derived
  - If yes, reconstruct the derivation steps

- There are many systematic ways to do this:
- Hand-written top-down parsers (1970)
  - Example, next week
- Generated bottom-up parsers (1974)
  - Write parser specification
  - Use tool to generate parser

Parser specification and generator

- A *parser* converts a token stream to an abstract syntax tree
- A *parser specification* describes well-formed streams
- A *parser generator* takes as input a parser specification, and generates a parser



Parser specification part 1: tokens, associativity and precedence

```
%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 */
```

a token may carry a value

token declarations

order gives precedence

associativity: left, right, nonassoc

Parser specification (ExprPar.fsy)

- A semantic action computes the result of parsing a given construct

```
%start Main
%type <Absyn.expr> Main
%%
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)   } ;
```

Putting together lexer and parser

- File **`Expr/Parse.fs`**:

```
let fromString (str : string) : expr =
    let lexbuf = Lexing.LexBuffer<char>.FromString(str)
    in try
         ExprPar.Main ExprLex.Token lexbuf
       with
         | exn -> failwith "Lexing or parsing error ... "
```

- From string to lexbuffer to tokens to abstract syntax tree:
  - **`ExprPar.Main`** = entry point in parser
  - **`ExprLex.Token`** = tokenizer in lexer

Command line use of fslex and fsyacc

- Build the lexer and parser as files **ExprLex.fs** and **ExprPar.fs**
- Compile as modules together with Absyn.fs and Parse.fs:

```
fsyacc --module ExprPar ExprPar.fsy
fslex --unicode ExprLex.fsl
fsi -r FSharp.PowerPack Absyn.fs ExprPar.fs ExprLex.fs Parse.fs
```

- Open the Parse module and experiment:

```
open Parse;;
fromString "x + 52 * wk";;
```

fsyacc and fslex with Visual Studio

- Visual Studio 2010 can run fslex and fsyacc for you, and compile the resulting .fs files
- Requires F# PowerPack:
  - Install from http://fsharppowerpack.codeplex.com/
  - Project > Add Reference > .NET > FSharp.PowerPack
  - Edit the XML file ExprProject.fsproj like this:

```
<Import Project="$(MSBuildExtensionsPath32)\..\FSharpPowerPack-2.0.0.0
                                        \bin\FSharp.PowerPack.targets" />
 <ItemGroup>
   <Compile Include="Absyn.fs" />
   <Compile Include="ExprPar.fs" />
   <Compile Include="ExprLex.fs" />
   <FsYacc Include="ExprPar.fsy">
     <OtherFlags>--module ExprPar</OtherFlags>
   </FsYacc>
   <FsLex Include="ExprLex.fsl">
     <OtherFlags>--unicode</OtherFlags>
   </FsLex>
   <Compile Include="Parse.fs" />
 </ItemGroup>
```

A single line

Project files in build order

How to run fsyacc

How to run fslex

Joint exercises

- How change the lexer and/or parser to accept brackets [ ] in addition to parens ( )?
- How change the lexer and/or parser to accept the division operator (/) also?
- How change lexer and parser to accept the syntax **`{ x <- 2 in x * 3 }`** instead of **`let x = 2 in x * 3 end`**
- How change the lexer and parser to accept function calls such as **`max(x, y)`**?

Reading and homework

- This week's lecture:
  - PLC chapter 3
  - Mogensen ICD 2011 sections 1.1-1.8, 2.1-2.5
    or Mogensen 2010 sections 2.1-2.7, 2.9, 3.1-3.6
  - Exercises 3.2, 3.3, 3.4, 3.5, 3.6, 3.7

- Next week's lecture:
  - PLCSD chapter 4
  - Mogensen ICD 2011 sections 2.11, 2.12, 2.16
    or Mogensen 2010 sections 3.12, 3.17