



BPRD – Programs as Data

Lecture 3: Parsing (2/3)

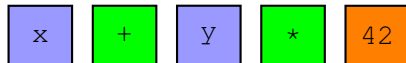
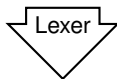
Zhoulai Fu

These slides are based on original slides by Niels Hal-
lenberg, Peter Sestoft, David Raymond Christensen and
Ahmad Salim Al-Sibahi. Thanks!!!



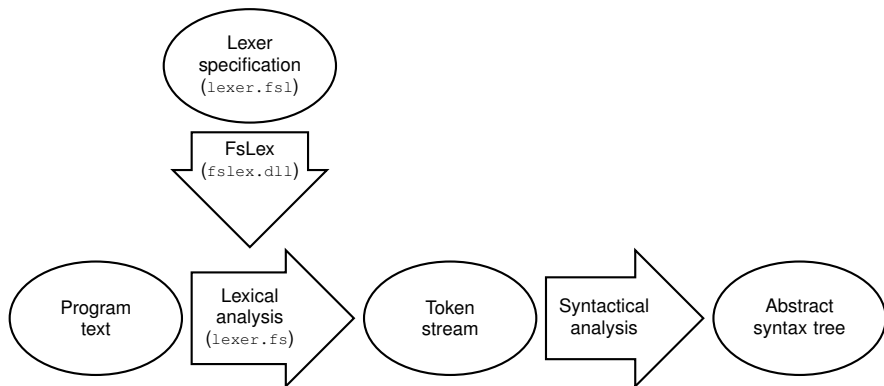
Lexical analysis (Lexer) converts LexBuffer to a token stream

"x+y*42"



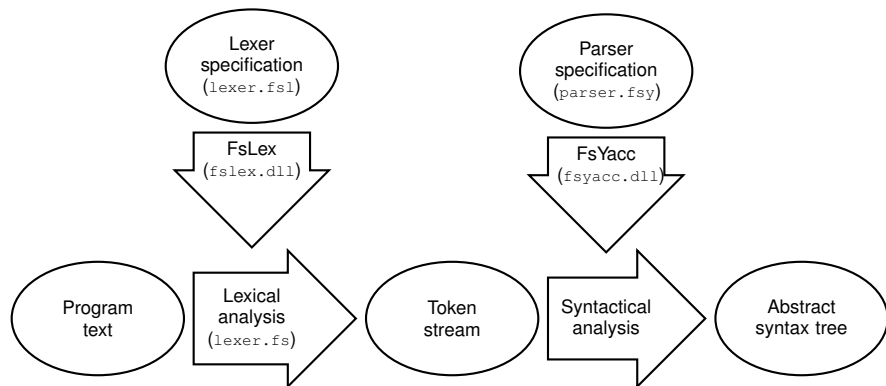


‘ A lexer generator takes as input a lexer specification and outputs a lexer.



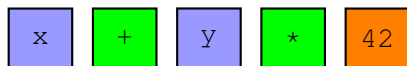
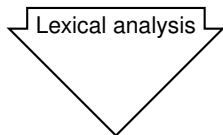


The lexer (usually) relies on an DFA, which is transformed from the regular expressions. This transformation makes sense thanks to Kleene's Theorem.

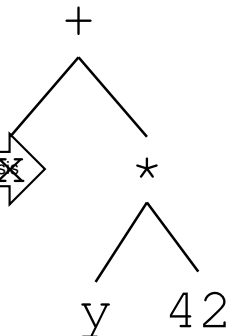




"x+y*42"



Syntactical analysis



Anecdote: How I used parsing techniques in my work



First, I used parsing techniques to derive security holes of a programs into logic constraints.

```
void Prog(double x) {  
    if (x < 1){  
        x = x + 1;  
        assert(x < 2);  
    }  
}
```

The security hole is transformed into $x < 1 \wedge x + 1 \geq 2$.

Then I solved these constraints via MCMC (Markov chain Monte Carlo), an approach in applied mathematics.

Finally, I found that “ $x = 0.999\ 999\ 999\ 999\ 999\ 9$ ” opened that door.
Demo.

Reference; "Effective floating-point analysis via weak-distance minimization". In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2019.

<http://zhoulaifu.com/wp-content/papercite-data/pdf/wd.pdf>



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



A grammar in Programming Language Theory describes the syntax of a language, namely, how to form sentences from a language's words.

Which grammar have we already used?

- Grammar in Danish, English: Subject - Verb - Object
- Regular expression: $[1 - 9][0 - 9]^*$ for positive integers
- Context-free grammar



A grammar in Programming Language Theory describes the syntax of a language, namely, how to form sentences from a language's words.

Which grammar have we already used?

- Grammar in Danish, English: Subject - Verb - Object
- Regular expression: $[1 - 9][0 - 9]^*$ for positive integers
- Context-free grammar



Definition (Context-free grammar)

A **context-free grammar**, abbreviated CFG, is a grammar whose rules have the following properties:

- The left-hand side is a single variable.
- The right-hand side is any string of variables and terminals.
- That is, every rule is of the form $A \rightarrow w$, where $w \in (V \cup \Sigma)^*$.



Example (Context-free grammar)

Let

- $V = \{E\}$
- $S = E$
- $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, +, *, (,)\}$
- R consists of the rules

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$



Example (Context-free grammars)

Find CFGs for the following languages.

- $\{a^n b^n \mid n \geq 0\}$
- $\{a^n b^m \mid n \geq m \geq 0\}$
- $\{a^n b^m \mid m \geq n \geq 0\}$



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



Definition (Yields)

A string u **yields** a string v if we can apply a grammar rule to u and get v .

- We write $u \Rightarrow v$.



Definition (Derives)

A string u **derives** a string v if there is a sequence

$$u_1, u_2, \dots, u_k,$$

with $k \geq 1$, where $u = u_1$, $v = u_k$, and

$$u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k.$$

The sequence is called a **derivation**.

- We write $u \Rightarrow^* v$.



- 1 You know what context-free grammar is and why it matters

Derivations

Leftmost and Rightmost Derivations

Ambiguity

- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



Definition (Leftmost derivation)

A **leftmost derivation** of a string is a derivation in which, at each step, the leftmost variable is replaced with a string.

Definition (Rightmost derivation)

A **rightmost derivation** of a string is a derivation in which, at each step, the rightmost variable is replaced with a string.



Example (Leftmost and rightmost derivations)
Using the grammar

$$S \rightarrow SS \mid \mathbf{aSb} \mid \mathbf{bSa} \mid \varepsilon,$$

find leftmost and rightmost derivations of **abab**.



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



- Some grammars provide more than one way to derive a string.
- For example, **abab** can be derived in two different ways using the grammar rules

$$S \rightarrow SS \mid \mathbf{aSb} \mid \mathbf{bSa} \mid \varepsilon.$$



Definition (Ambiguous grammar)

A grammar is **ambiguous** if its language contains a string that has more than one leftmost derivation under that grammar.

Definition (Inherently ambiguous language)

A language is **inherently ambiguous** if every grammar for that language is ambiguous.



- Consider again the grammar rules

$$S \rightarrow SS \mid \mathbf{aSb} \mid \mathbf{bSa} \mid \varepsilon.$$

- Suppose that, for every string, we need to associate each **a** in the string with a unique **b** in the string.
- Does the grammar show us how to do this?



Example (Ambiguous grammar)

- Consider the grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}.$$

- Derive the string $\mathbf{a} + \mathbf{b} * \mathbf{c}$ in two different ways.
- Is this grammar ambiguous?



Example (Unambiguous grammar)

The same language can be derived unambiguously from the following grammar.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



See PLC page 45.

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
- Terminal symbols (from lexer)
- Grammar rules, or Productions (called A–I)
- Start symbol (the nonterminal Main)



See PLC page 45.

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**
- Terminal symbols (from lexer)
- Grammar rules, or Productions (called A–I)
- Start symbol (the nonterminal Main)



See PLC page 45.

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
- **Terminal symbols (from lexer)**
- Grammar rules, or Productions (called A–I)
- Start symbol (the nonterminal Main)



See PLC page 45.

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
- Terminal symbols (from lexer)
- **Grammar rules, or Productions (called A–I)**
- Start symbol (the nonterminal Main)



See PLC page 45.

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
- Terminal symbols (from lexer)
- Grammar rules, or Productions (called A–I)
- **Start symbol (the nonterminal Main)**

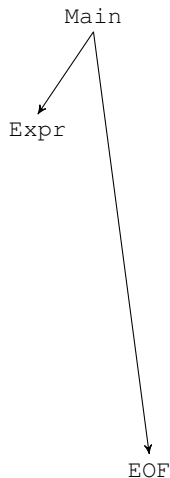
Derivation: grammar as string generator



Main

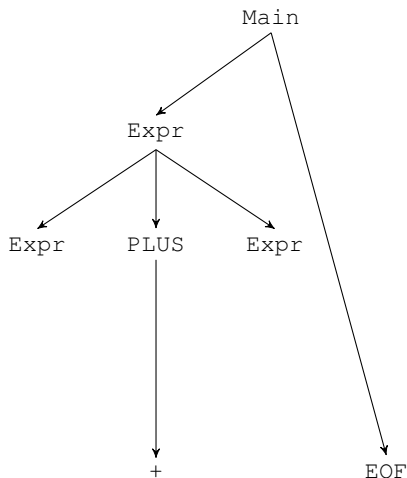
Main	
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



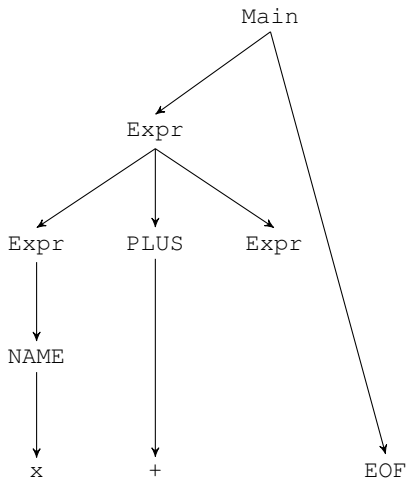
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



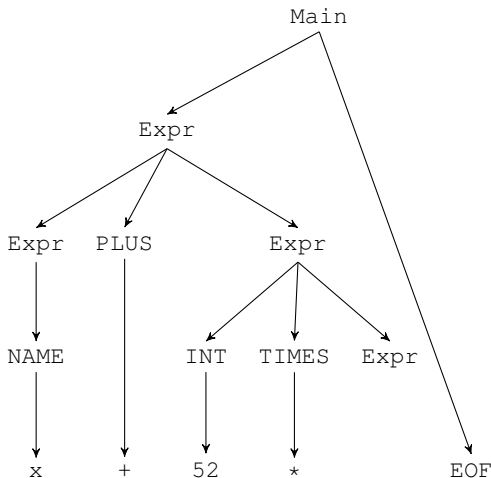
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



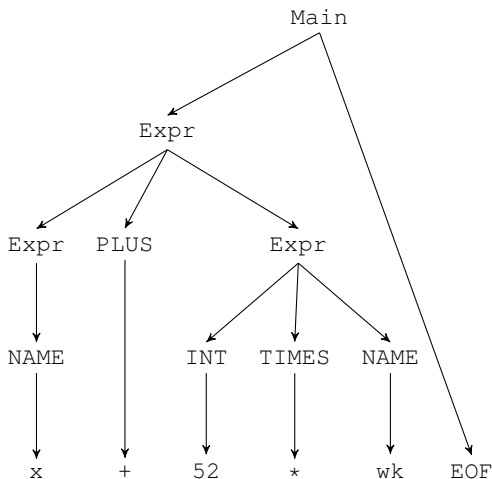
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



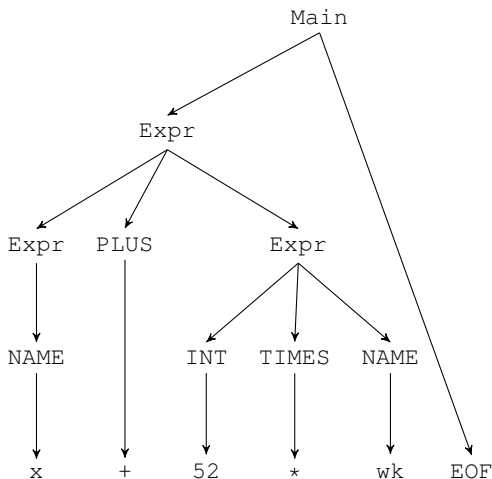
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF

Derivation: grammar as string generator



	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF
C	x + 52 * EXPR EOF
B	x + 52 * wk EOF



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar**
- 4 You know how to use FsYacc and FsLex to generate a parser



Parsing

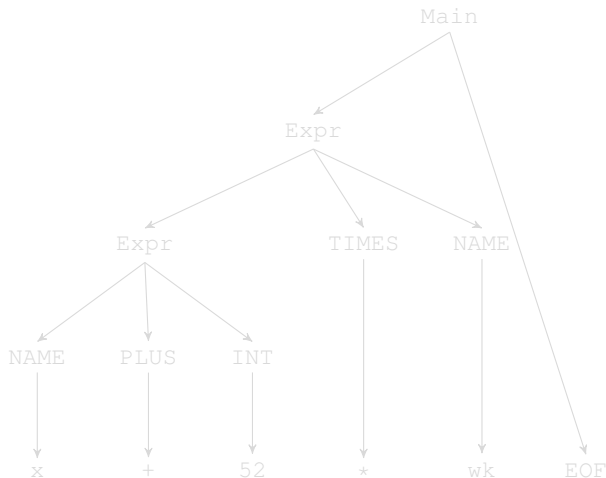
Given a grammar and a string - Determine whether the string can be derived - and if possible, then reconstruct the derivation steps

There are many systematic ways to do this:

- Hand-written top-down parsers (1970)
- Generated bottom-up parsers (1974)
 - Write parser specification
 - Use tool to generate parser

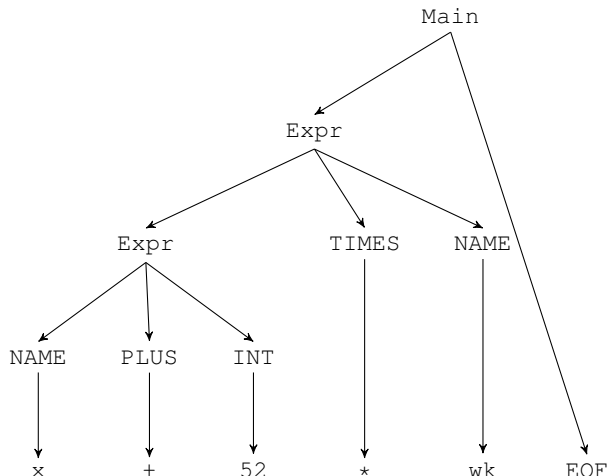


A grammar is *ambiguous* if there exists a string with more than one derivation tree.





A grammar is *ambiguous* if there exists a string with more than one derivation tree.





Leftmost derivation

Always expand the leftmost nonterminal.

See first example.

Rightmost derivation

Always expand the rightmost nonterminal.

See second example.

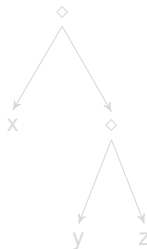


How to read $x \diamond y \diamond z$?

\diamond is left-associative



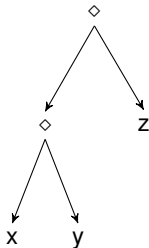
\diamond is right-associative



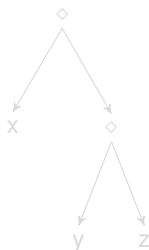


How to read $x \diamond y \diamond z$?

\diamond is left-associative



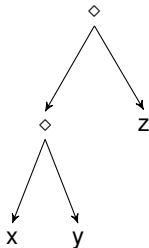
\diamond is right-associative



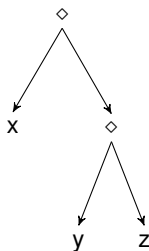


How to read $x \diamond y \diamond z$?

\diamond is left-associative



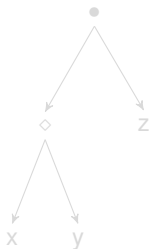
\diamond is right-associative



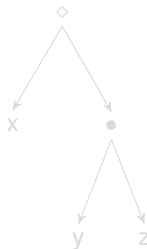


How to read $x \diamond y \bullet z$?

\diamond has higher precedence



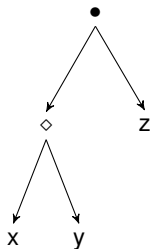
\bullet has higher precedence



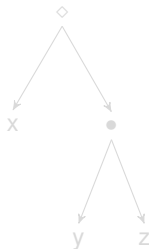


How to read $x \diamond y \bullet z$?

\diamond has higher precedence



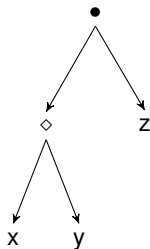
\bullet has higher precedence



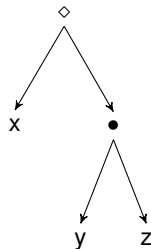


How to read $x \diamond y \bullet z$?

\diamond has higher precedence



\bullet has higher precedence





- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser



FsLexYacc:

<http://fsprojects.github.io/FsLexYacc/index.html>

Example:

<https://www.itu.dk/people/sestoft/plc/expr.zip>

File `Absyn.fs` defines the type representing the abstract syntax tree that the parser builds.

```
module Absyn

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



```
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  }
| '('                  { LPAR   }
| ')'                  { RPAR   }
| eof                  { EOF     }
| _                    { lexerError lexbuf "Bad char" }
```



```
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  }
| '('                 { LPAR   }
| ')'                 { RPAR   }
| eof                 { EOF     }
| _                   { lexerError lexbuf "Bad char" }
```

Regular Expressions



```
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 }
| '('                  { LPAR }
| ')'                  { RPAR }
| eof                  { EOF }
| _                    { lexerError lexbuf "Bad char" }
```

F# to construct token



```
%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 */
```



```
%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 */
```

Token specifications - is expanded to a datatype



```
%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 */
```

Tokens carrying data



```
%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 */
```

Precedence: %left, %right, and %nonassoc allowed



```
%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 */
```

Ordering of groups defines precedence levels



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1            } B
    | CSTINT                { CstI $1            } C
    | MINUS CSTINT          { CstI (- $2)        } D
    | LPAR Expr RPAR        { $2                } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6)    } F
    | Expr TIMES Expr       { Prim("*", $1, $3) } G
    | Expr PLUS Expr        { Prim("+", $1, $3) } H
    | Expr MINUS Expr       { Prim("-", $1, $3) } I
```



```
%start Main
%type <Absyn.expr> Main
%%
```

Main:

```
Expr EOF { $1 } A
```

Expr:

```
NAME { Var $1 } B
| CSTINT { CstI $1 } C
| MINUS CSTINT { CstI (- $2) } D
| LPAR Expr RPAR { $2 } E
| LET NAME EQ Expr IN Expr END { Let($2, $4, $6) } F
| Expr TIMES Expr { Prim("*", $1, $3) } G
| Expr PLUS Expr { Prim("+", $1, $3) } H
| Expr MINUS Expr { Prim("-", $1, $3) } I
```

Non-terminals



```
%start Main
```

```
%type <Absyn.expr> Main
```

```
%%
```

```
Main:
```

```
    Expr EOF                                { $1                                } A
```

```
Expr:
```

```
    NAME                                    { Var $1                                } B
```

```
  | CSTINT                                { CstI $1                                } C
```

```
  | MINUS CSTINT                          { CstI (- $2)                            } D
```

```
  | LPAR Expr RPAR                        { $2                                    } E
```

```
  | LET NAME EQ Expr IN Expr END          { Let($2, $4, $6)                        } F
```

```
  | Expr TIMES Expr                      { Prim("*", $1, $3) } G
```

```
  | Expr PLUS Expr                      { Prim("+", $1, $3) } H
```

```
  | Expr MINUS Expr                    { Prim("-", $1, $3) } I
```

Start symbol



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1                } B
    | CSTINT                 { CstI $1             } C
    | MINUS CSTINT           { CstI (- $2)         } D
    | LPAR Expr RPAR         { $2                } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6) } F
    | Expr TIMES Expr        { Prim("*", $1, $3) } G
    | Expr PLUS Expr         { Prim("+", $1, $3) } H
    | Expr MINUS Expr        { Prim("-", $1, $3) } I
```

Semantic actions



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1            } B
    | CSTINT                 { CstI $1          } C
    | MINUS CSTINT          { CstI (- $2)    } D
    | LPAR Expr RPAR        { $2                } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6) } F
    | Expr TIMES Expr       { Prim("*", $1, $3) } G
    | Expr PLUS Expr       { Prim("+", $1, $3) } H
    | Expr MINUS Expr      { Prim("-", $1, $3) } I
```

Arguments count from left



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1            } B
    | CSTINT                 { CstI $1          } C
    | MINUS CSTINT           { CstI (- $2)       } D
    | LPAR Expr RPAR         { $2              } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6)   } F
    | Expr TIMES Expr        { Prim("*", $1, $3) } G
    | Expr PLUS Expr         { Prim("+", $1, $3) } H
    | Expr MINUS Expr        { Prim("-", $1, $3) } I
```

Arguments count from left



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1                } B
    | CSTINT                { CstI $1                } C
    | MINUS CSTINT          { CstI (- $2)         } D
    | LPAR Expr RPAR        { $2                } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6)    } F
    | Expr TIMES Expr       { Prim("*", $1, $3) } G
    | Expr PLUS Expr       { Prim("+", $1, $3) } H
    | Expr MINUS Expr      { Prim("-", $1, $3) } I
```

Arguments count from left



```
%start Main
%type <Absyn.expr> Main
%%
Main:
    Expr EOF                { $1                } A
Expr:
    NAME                    { Var $1            } B
    | CSTINT                { CstI $1           } C
    | MINUS CSTINT          { CstI (- $2)       } D
    | LPAR Expr RPAR        { $2               } E
    | LET NAME EQ Expr IN Expr END { Let($2, $4, $6)   } F
    | Expr TIMES Expr       { Prim("*", $1, $3) } G
    | Expr PLUS Expr       { Prim("+", $1, $3) } H
    | Expr MINUS Expr      { Prim("-", $1, $3) } I
```

Type annotation - type of value returned by your semantic actions for the `Main` non-terminal. That is the values you have written to the right.



From string to lexbuffer to tokens to abstract syntax tree.

From file `Expr/Parse.fs`:

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



From string to lexbuffer to tokens to abstract syntax tree.

From file `Expr/Parse.fs`:

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

Entry point in parser



From string to lexbuffer to tokens to abstract syntax tree.

From file `Expr/Parse.fs`:

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

Entry point in lexer



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

```
$ dotnet fsyacc.dll --module ExprPar ExprPar.fsy
$ dotnet fslex.dll --unicode ExprLex.fsl
$ fsharp -r FsLexYacc.Runtime.dll Absyn.fs
      Expr.fs ExprPar.fs ExprLex.fs Parse.fs
```

- All DLL files above refer to their actual paths.
- Open the Parse module and experiment:

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



- 1 You know what context-free grammar is and why it matters
 - Derivations
 - Leftmost and Rightmost Derivations
 - Ambiguity
- 2 You get an intuition of concepts in context-free grammar presented above, and in particular, how to generate a language with in practice
- 3 You know how to Parse with a grammar
- 4 You know how to use FsYacc and FsLex to generate a parser