Lexing and Parsing

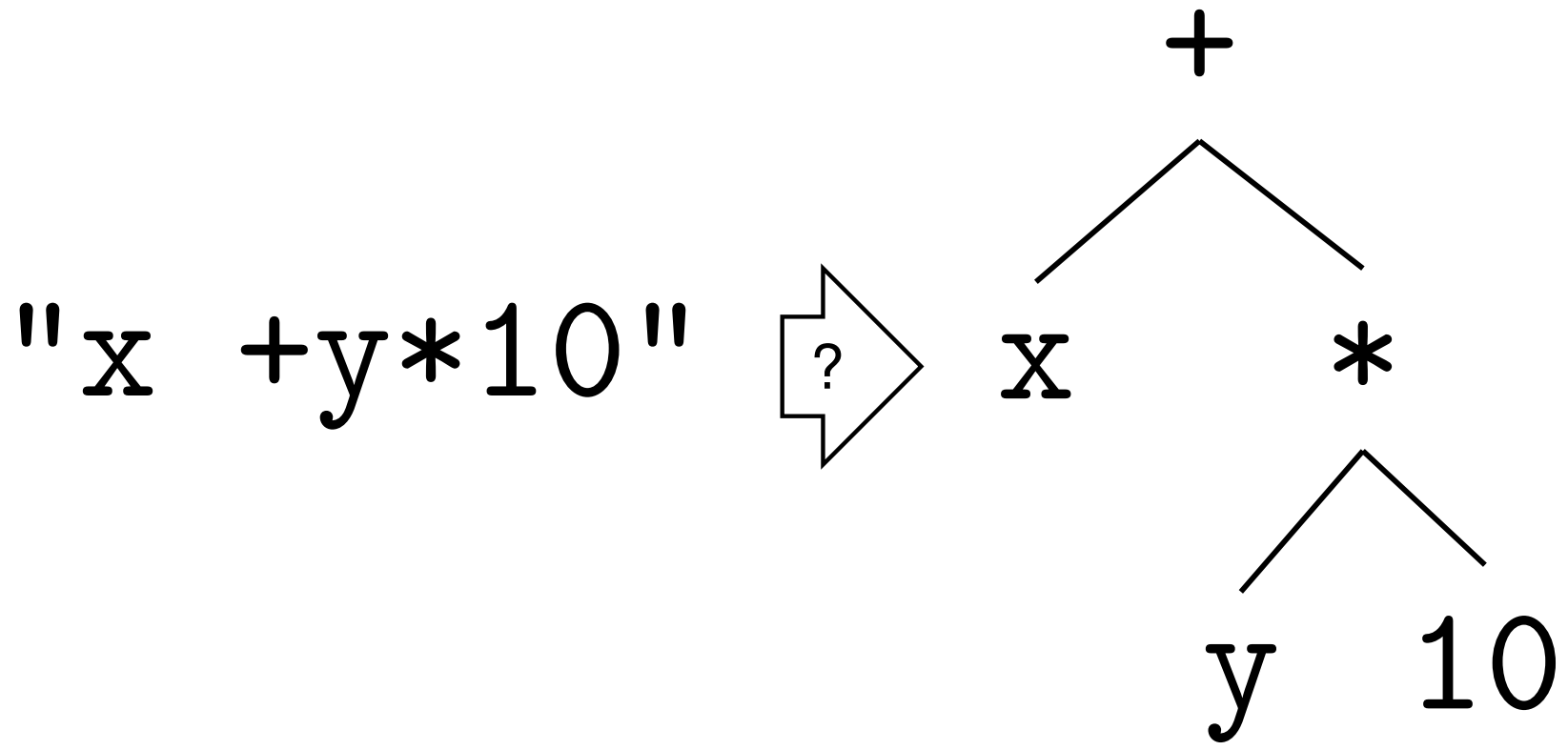
David Raymond Christiansen

2 September, 2013

Based on slides by Peter Sestoft

IT University
of Copenhagen

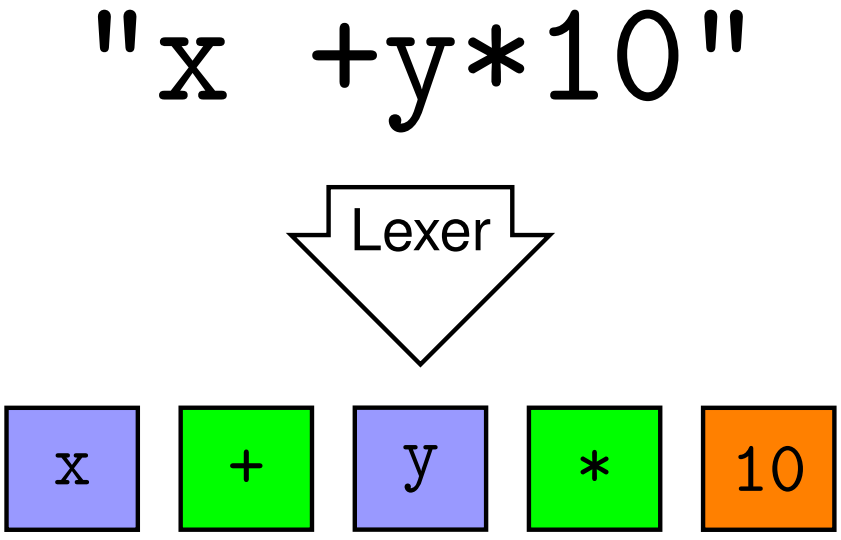
From text file to abstract syntax



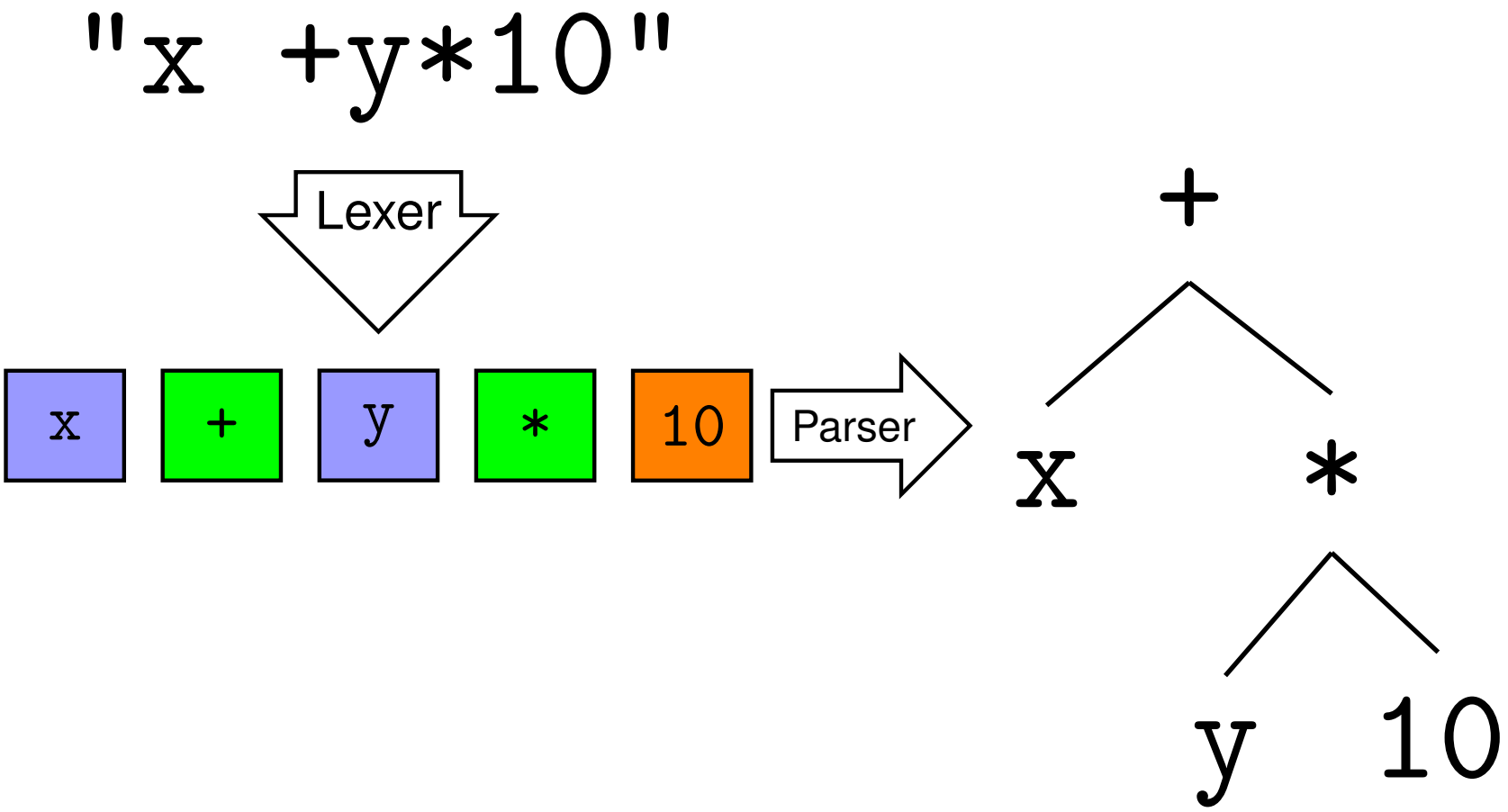
From text file to abstract syntax

```
"x +y*10"
```

From text file to abstract syntax



From text file to abstract syntax



Plan for today

Lexer specifications

- Regular expressions
- The fslex lexer generation tool
- Automata

Parser specifications

- Grammars
- Parsing
- The fsyacc parser generation tool

Parsing Algorithms

- Top-down
- Bottom-up

Languages and automata

Plan for today

Lexer specifications

- Regular expressions
- The fslex lexer generation tool
- Automata

Parser specifications

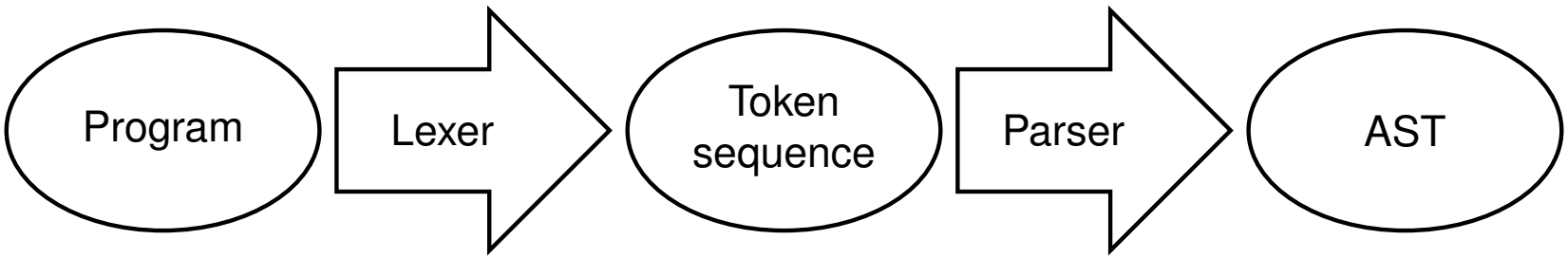
- Grammars
- Parsing
- The fsyacc parser generation tool

Parsing Algorithms

- Top-down
- Bottom-up

Languages and automata

Lexers and lexer generators



Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
`a`	Character `a`	$\{"\texttt{a}"\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\mathtt{a}"\}$
ε	Empty string	$\{""\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\mathrm{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\mathrm{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\mathrm{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \mid r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\texttt{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \mid r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

EXAMPLES

$\texttt{ab}^*$ represents $\{"\texttt{a}", "\texttt{ab}", "\texttt{abb}", \ldots\}$

$(\texttt{ab})^*$ represents $\{"", "\texttt{ab}", "\texttt{abab}", \ldots\}$

$(\texttt{a}|\texttt{b})^*$ represents $\{"", "\texttt{a}", "\texttt{b}", "\texttt{aa}", "\texttt{ab}", "\texttt{ba}", \ldots\}$

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
a	Character a	$\{"\texttt{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \mid r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

$\texttt{ab}^*$ represents $\{"\texttt{a}", "\texttt{ab}", "\texttt{abb}", \ldots\}$

$(\texttt{ab})^*$ represents $\{"", "\texttt{ab}", "\texttt{abab}", \ldots\}$

$(\texttt{a}|\texttt{b})^*$ represents $\{"", "\texttt{a}", "\texttt{b}", "\texttt{aa}", "\texttt{ab}", "\texttt{ba}", \ldots\}$

Exercise

What does $(\texttt{a}|\texttt{b})\texttt{c}^*$ represent?

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–Z]	Ranges	

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–Z]	Ranges	0|1|...|8|9|a|b|...|y|z

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–Z]	Ranges	0|1|...|8|9|a|b|...|y|z
r?	Zero or one *r*	

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–z]	Ranges	0|1|...|8|9|a|b|...|y|z
r?	Zero or one *r*	*r*|ε

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–z]	Ranges	0|1|...|8|9|a|b|...|y|z
r?	Zero or one r	$r \mid \varepsilon$
r^+	One or more r	

Regular expression abbreviations

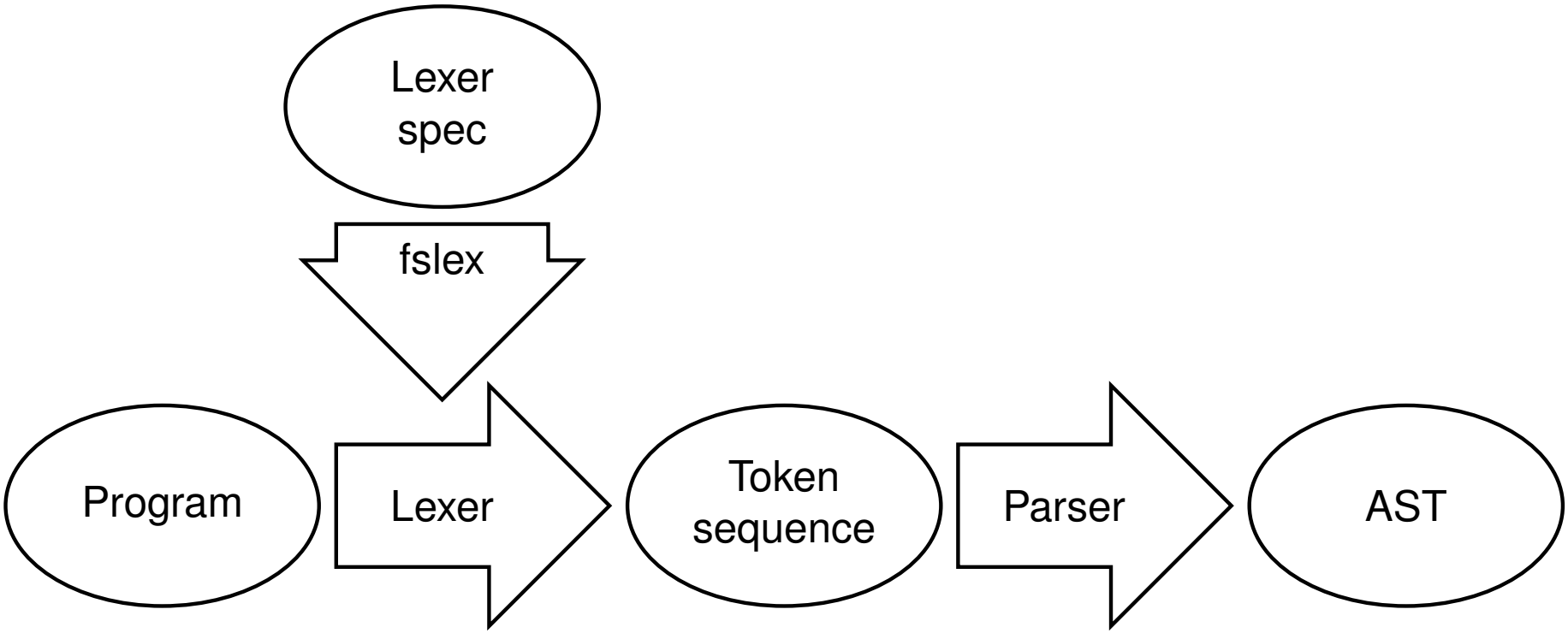
Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–z]	Ranges	0|1|...|8|9|a|b|...|y|z
r?	Zero or one r	$r \mid \varepsilon$
r^+	One or more r	$r\, r^*$

Five-minute 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 and generator



Lexer specifications: ExprLex.fsl

```
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" }
```

Lexer specifications: ExprLex.fsl

```
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" }
```

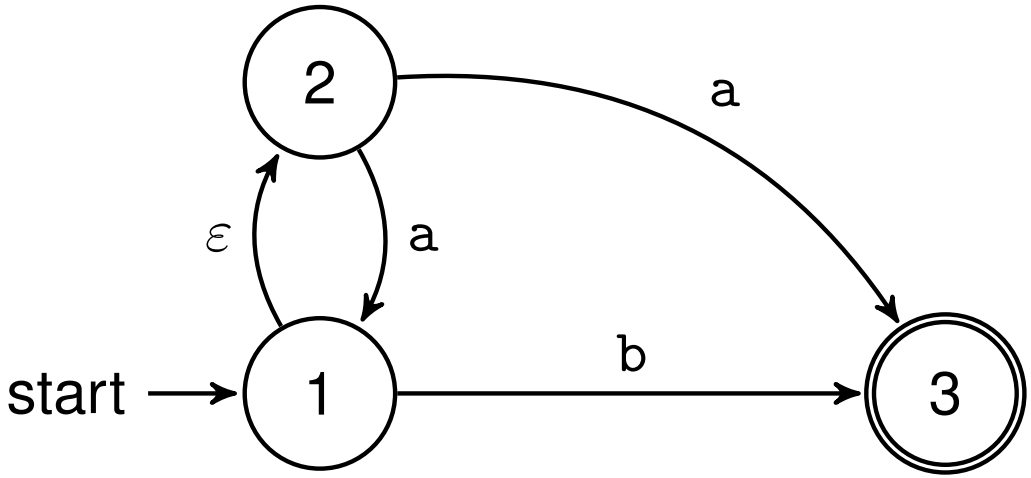
Regular Expressions

Lexer specifications: ExprLex.fsl

```
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

Finite State Automata



Regular expressions and finite automata

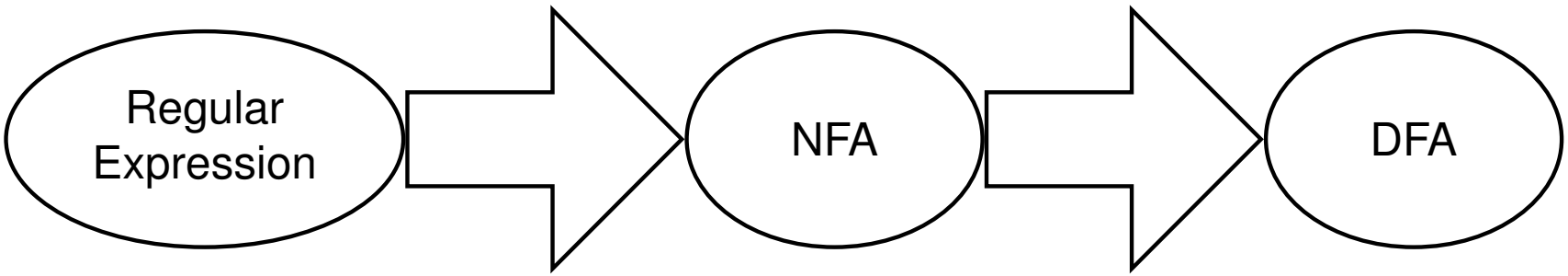
For every regular expression *r*, there exists a deterministic finite automaton that recognizes precisely the strings described by *r*.

Regular expressions and finite automata

For every regular expression *r*, there exists a deterministic finite automaton that recognizes precisely the strings described by *r*.

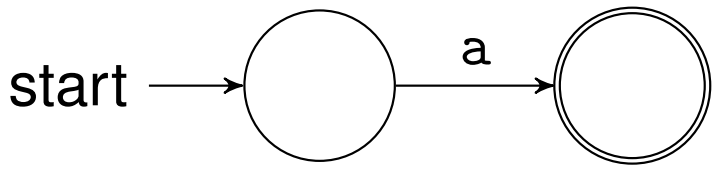
(The converse is also true.)

Constructing automata from regular expressions



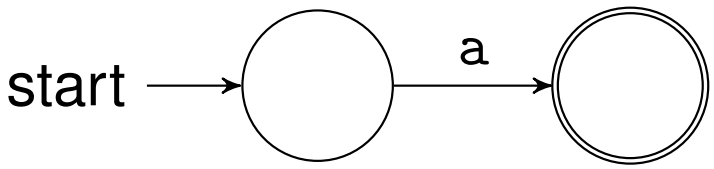
Constructing NFA from RE

a

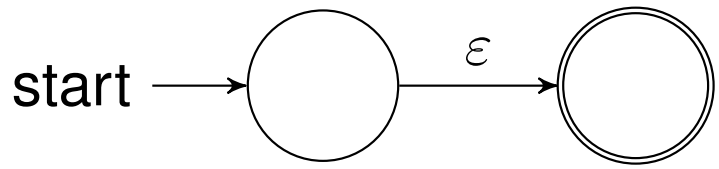


Constructing NFA from RE

a

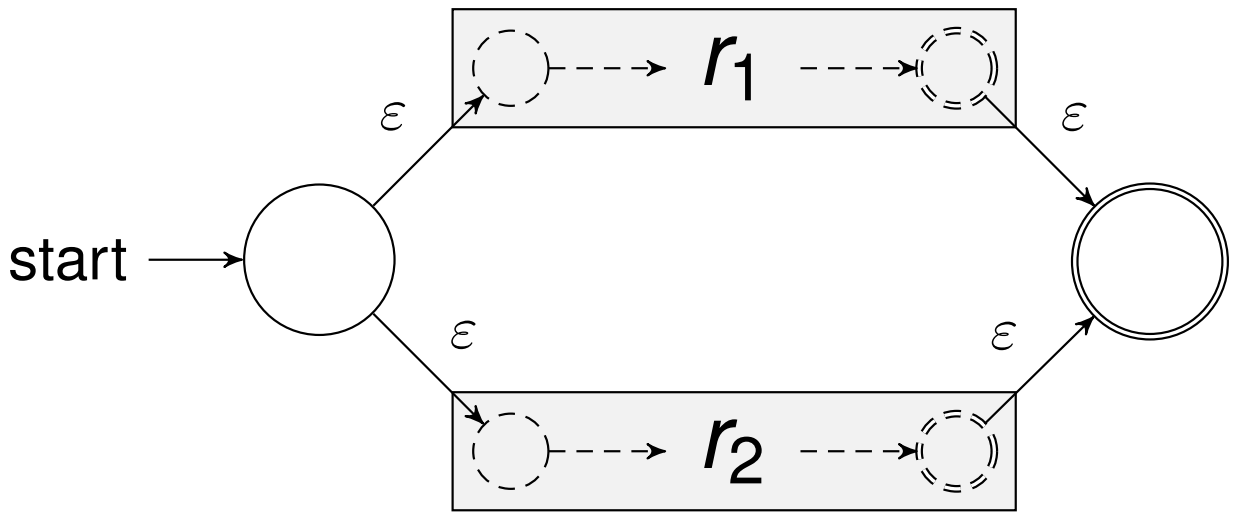


ε



RE to NFA: Alternatives

$$r_1 | r_2$$

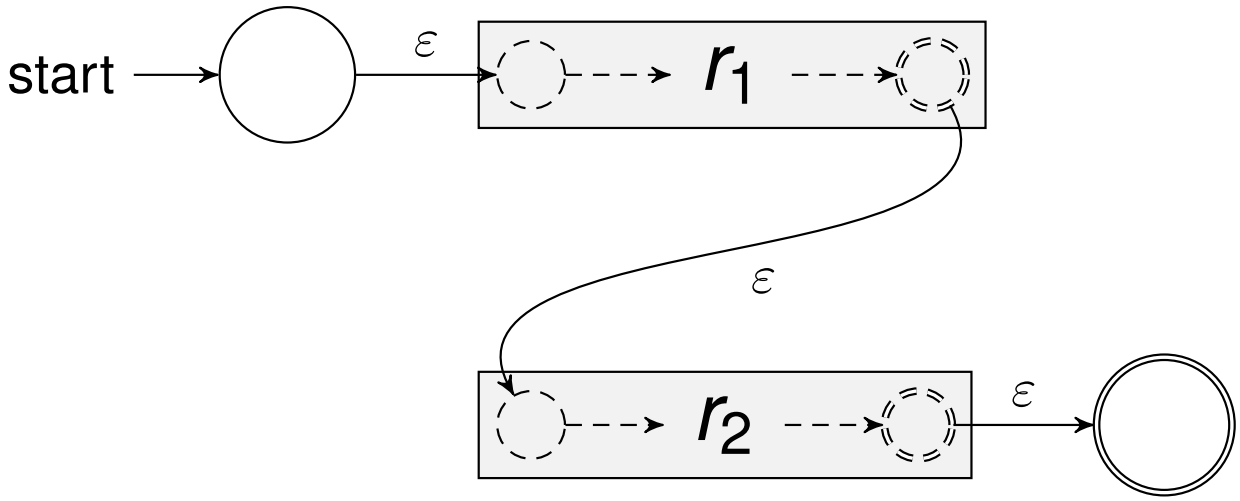


RE to NFA: Sequencing

$r_1 r_2$

RE to NFA: Sequencing

$r_1 r_2$

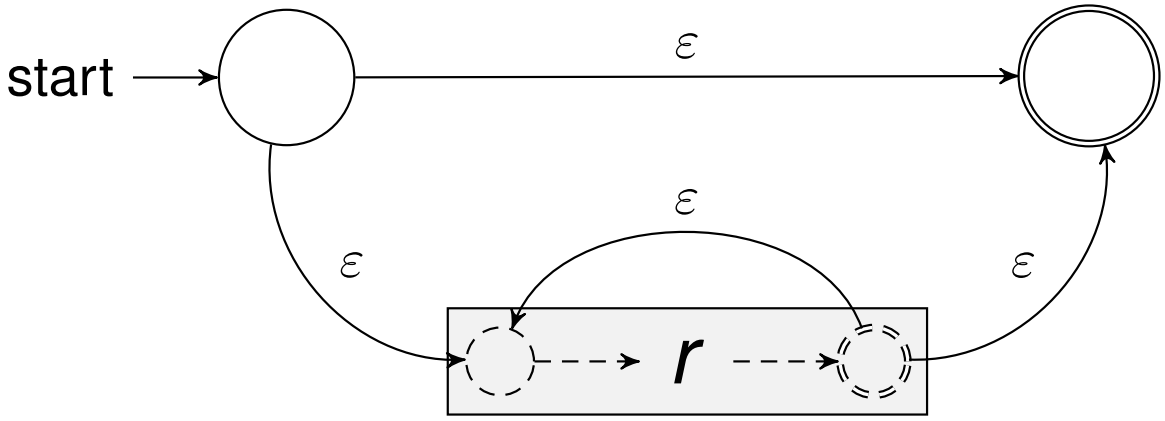


RE to NFA: Repetition

r^*

RE to NFA: Repetition

r^*

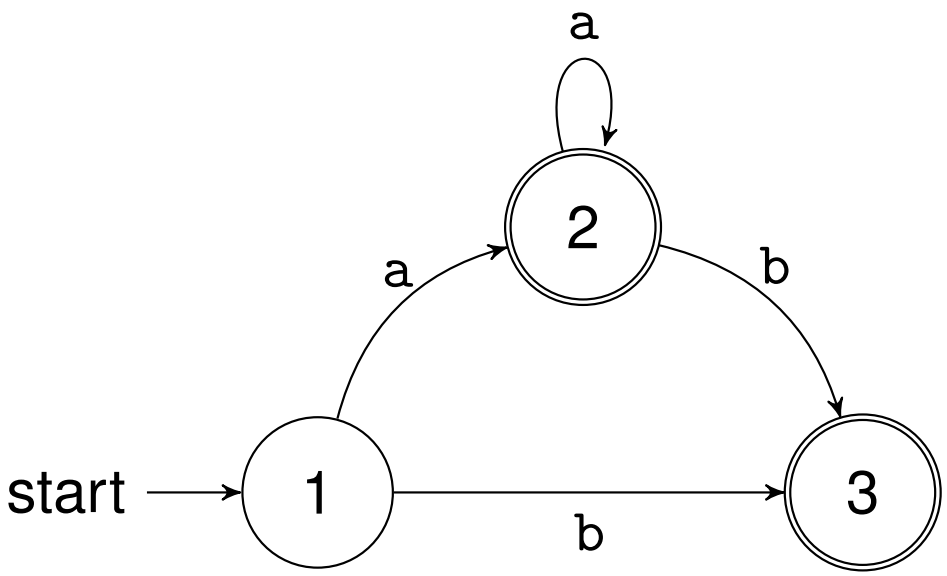


Deterministic Finite Automata

- No ε-transitions
- Distinct transitions from each state
- Multiple accepting states OK

Deterministic Finite Automata

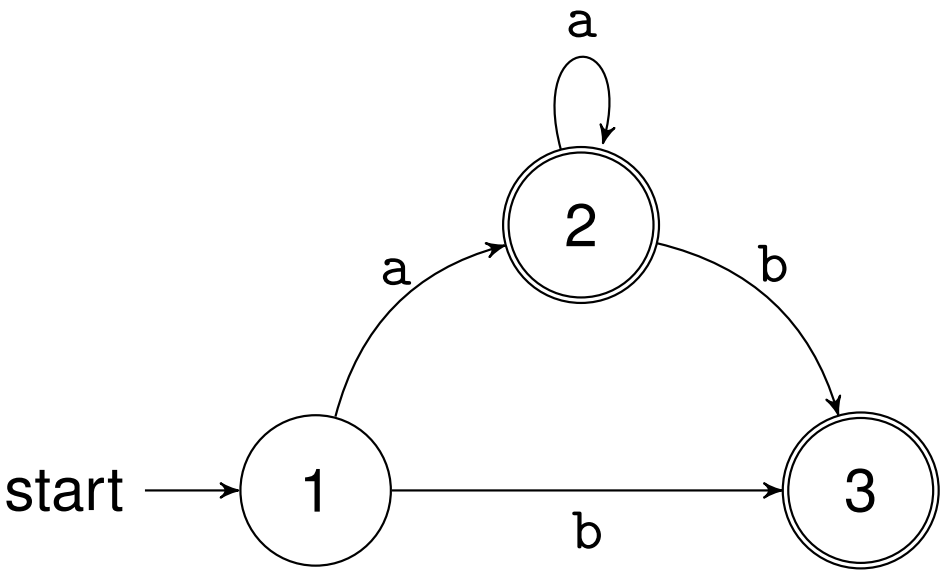
- No ε-transitions
- Distinct transitions from each state
- Multiple accepting states OK



Deterministic Finite Automata

- No ε-transitions
- Distinct transitions from each state
- Multiple accepting states OK

```
state = table[state][input];
```



State	Input	Go to
1	a	2
1	b	3
2	a	2
2	b	3
3	a	fail
3	b	fail

NFA to DFA

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

NFA to DFA

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

Intuition: simulation of parallel execution of NFA.

NFA to DFA

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

Intuition: simulation of parallel execution of NFA.

States

The ε-closed sets of NFA states. They are accepting if they contain an accepting NFA state.

Transitions

For every NFA transition from q_0 to q_1 on $\mathtt{a}$, the DFA has a transition from each state containing q_0 to each state containing q_1 on $\mathtt{a}$.

Whiteboard: NFA to DFA demos

- $(ab)^*$

Whiteboard: NFA to DFA demos

- $(\mathtt{ab})^*$
- $(\mathtt{a}|\mathtt{b})^*$

Whiteboard: NFA to DFA demos

- $(ab)^*$
- $(a|b)^*$
- $a^*(a|b)aa$

(Exercise 1.2 from Mogensen's ICD 2011)

Break

Plan for today

Lexer specifications
- Regular expressions
- The fslex lexer generation tool
- Automata

Parser specifications
- Grammars
- Parsing
- The fsyacc parser generation tool

Parsing Algorithms
- Top-down
- Bottom-up

Languages and automata

Context-free grammars

```
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)
```

- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

Context-free grammars

```
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)
```

- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

Context-free grammars

```
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)
```

- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

Context-free grammars

```
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)
```

- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

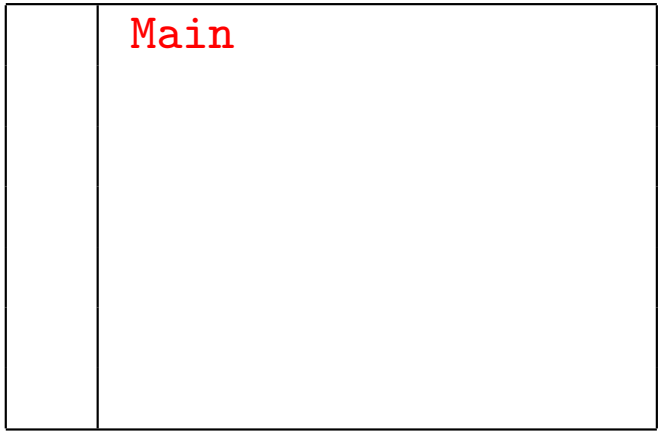
Context-free grammars

```
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)
```

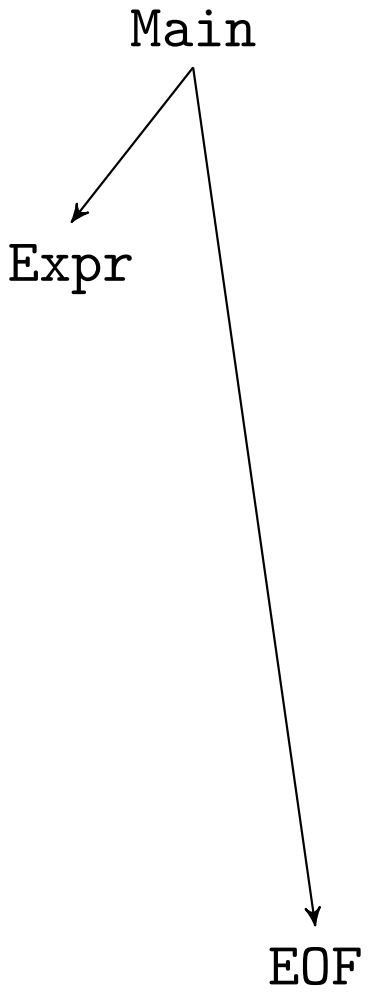
- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

Derivation: grammar as string generator

Main

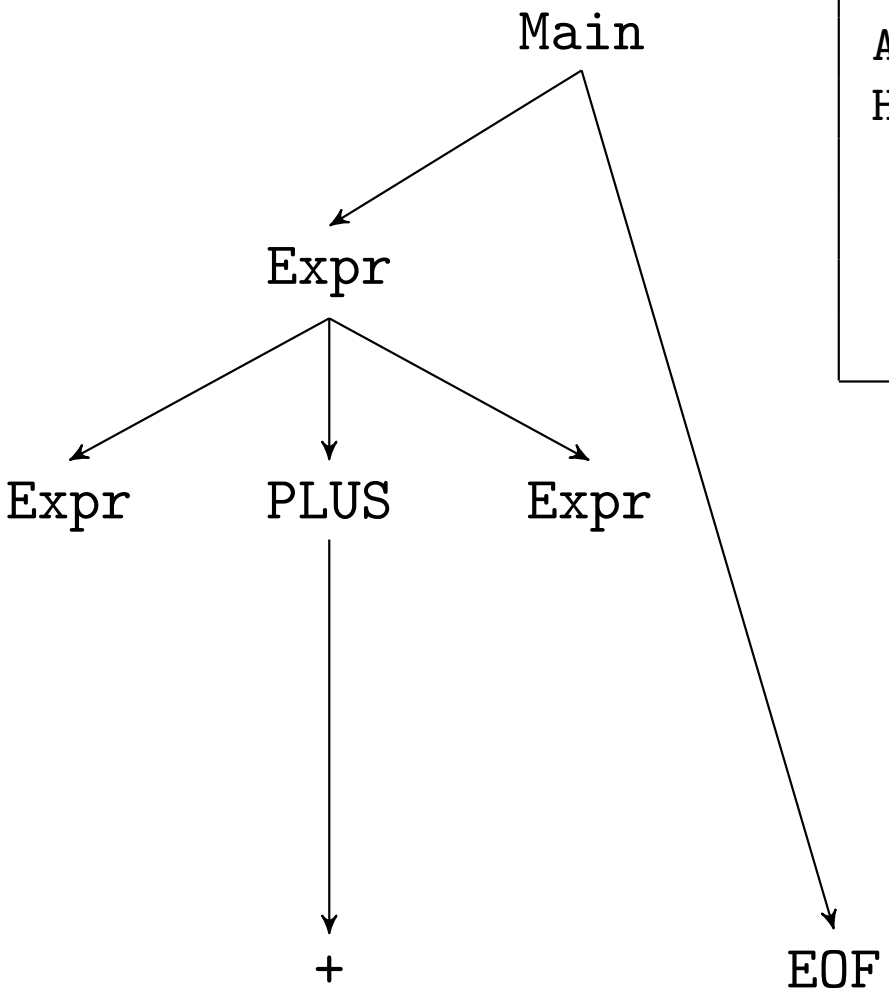


Derivation: grammar as string generator



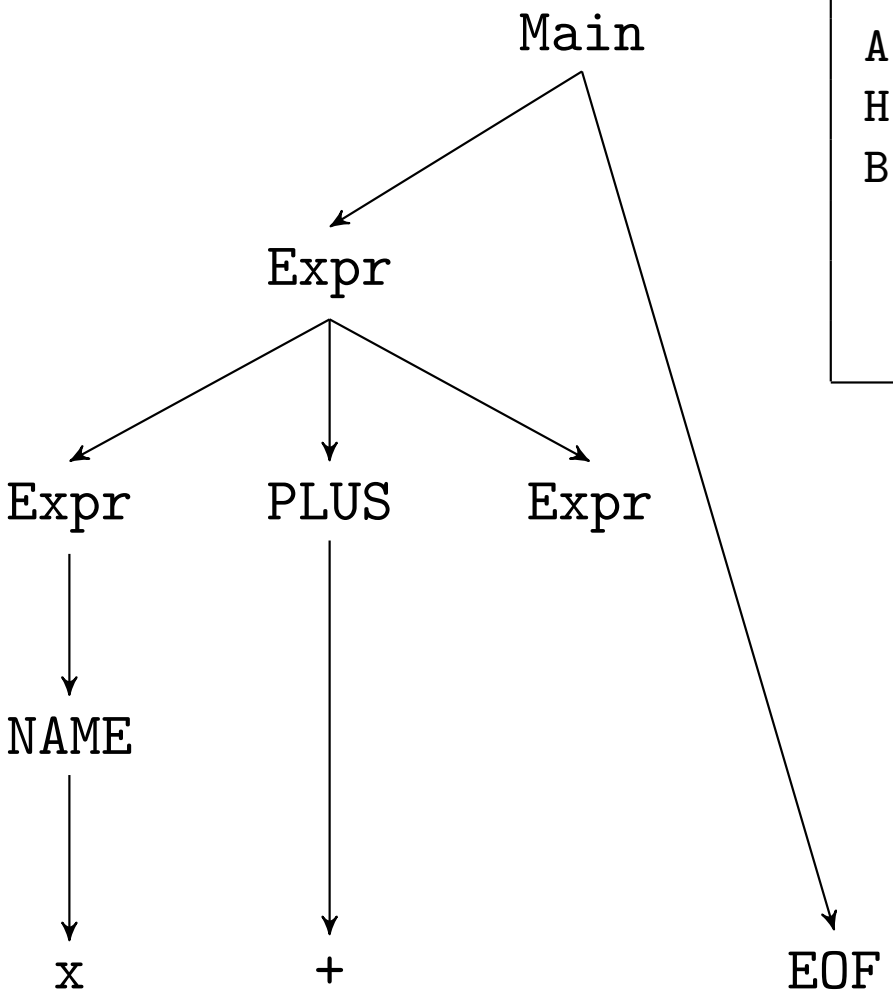
	Main
A	Expr EOF

Derivation: grammar as string generator



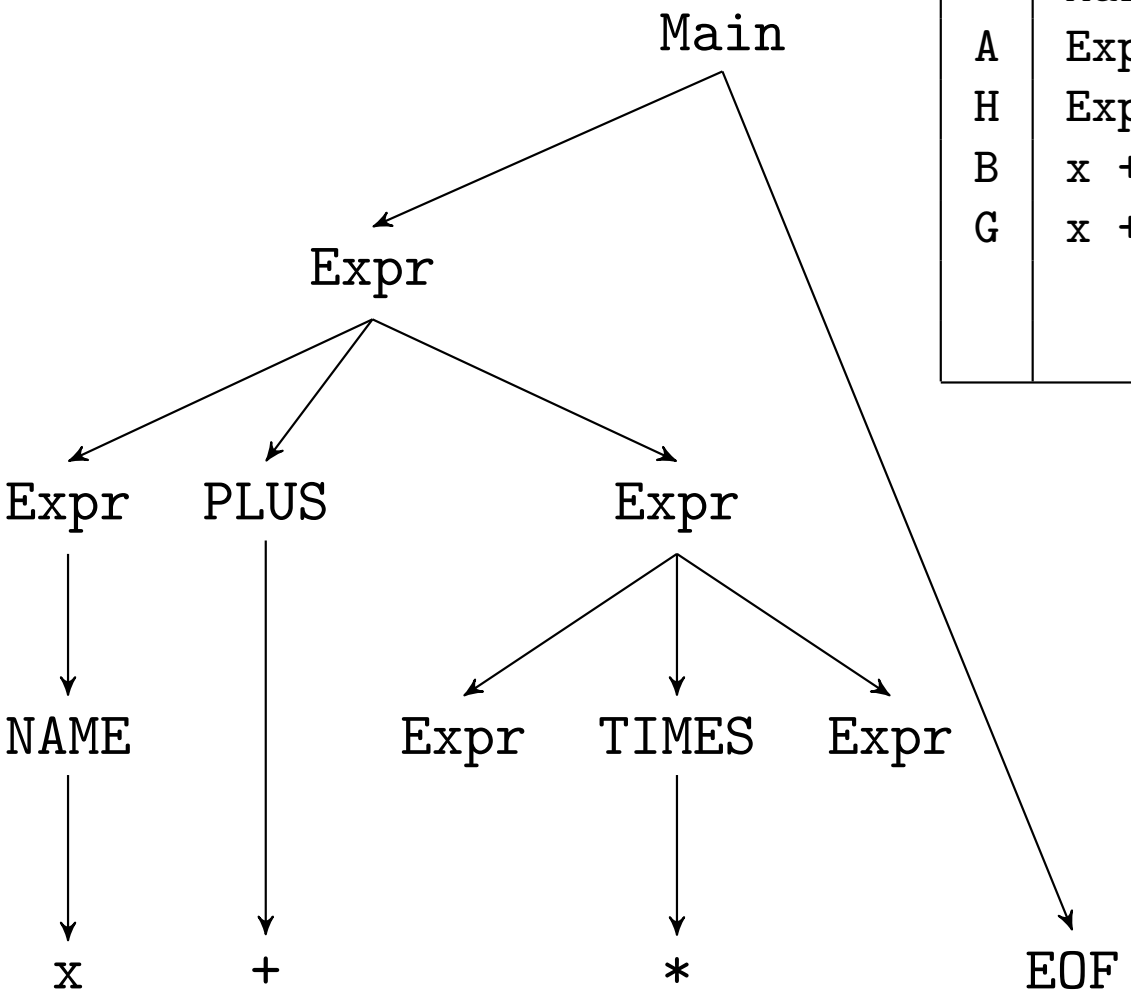
	Main
A	Expr EOF
H	Expr + Expr EOF

Derivation: grammar as string generator



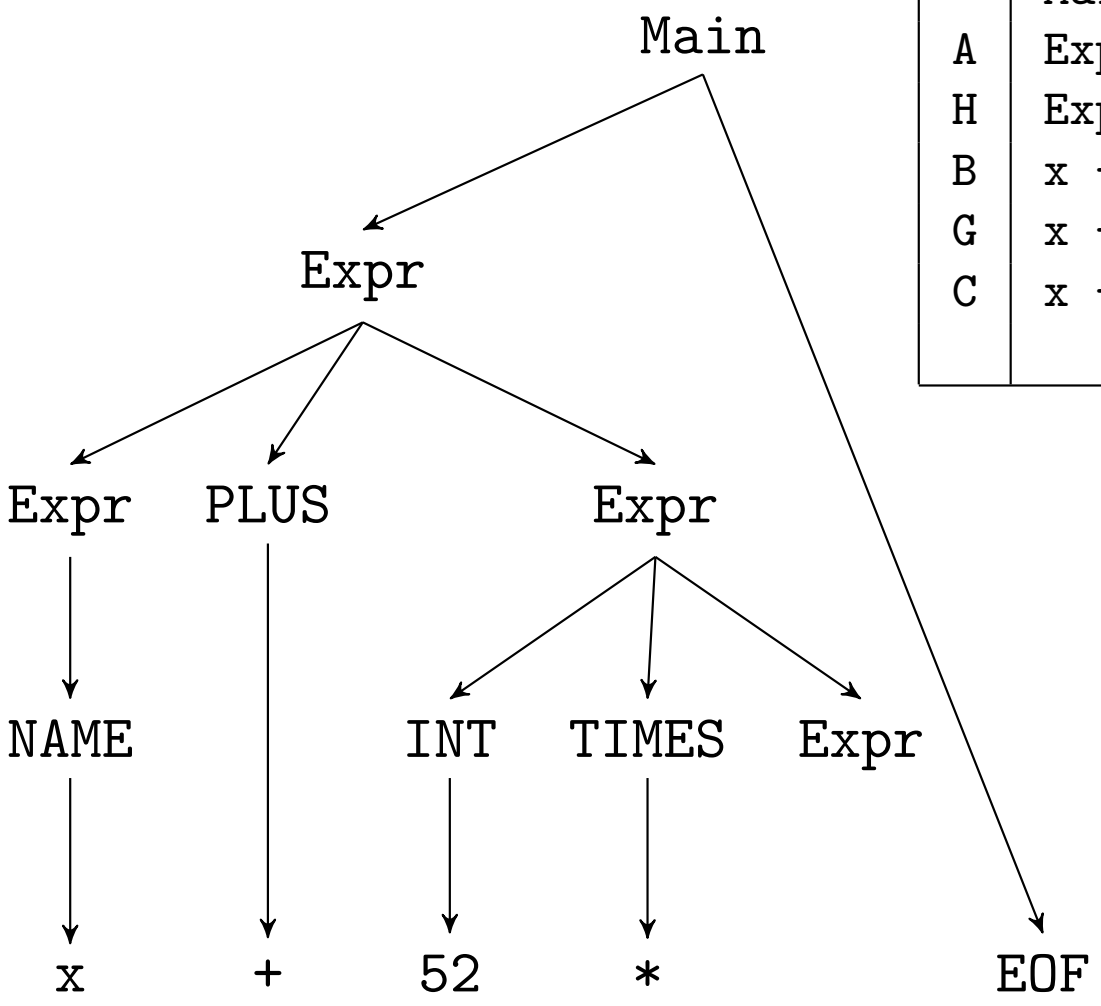
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF

Derivation: grammar as string generator



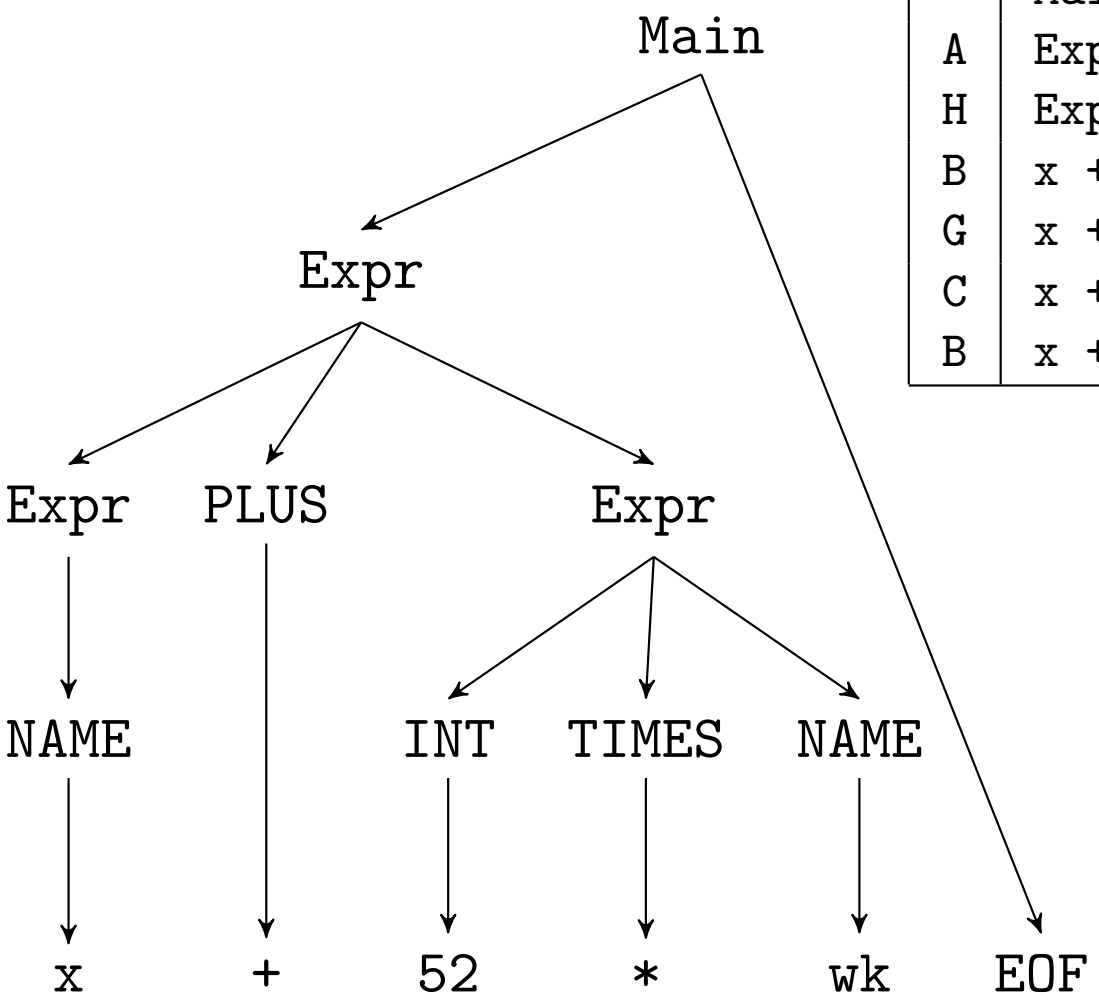
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF

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

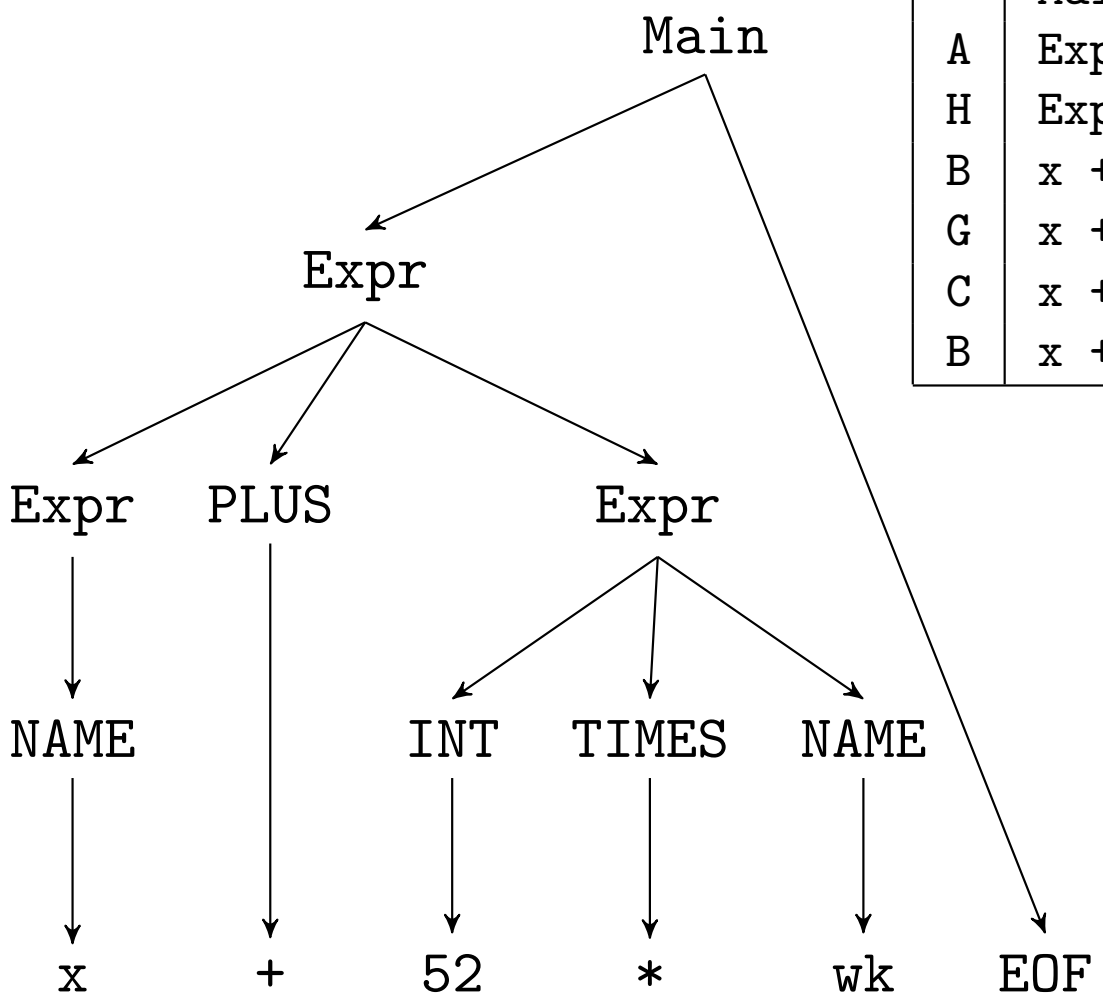
Derivation: grammar as string generator



	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

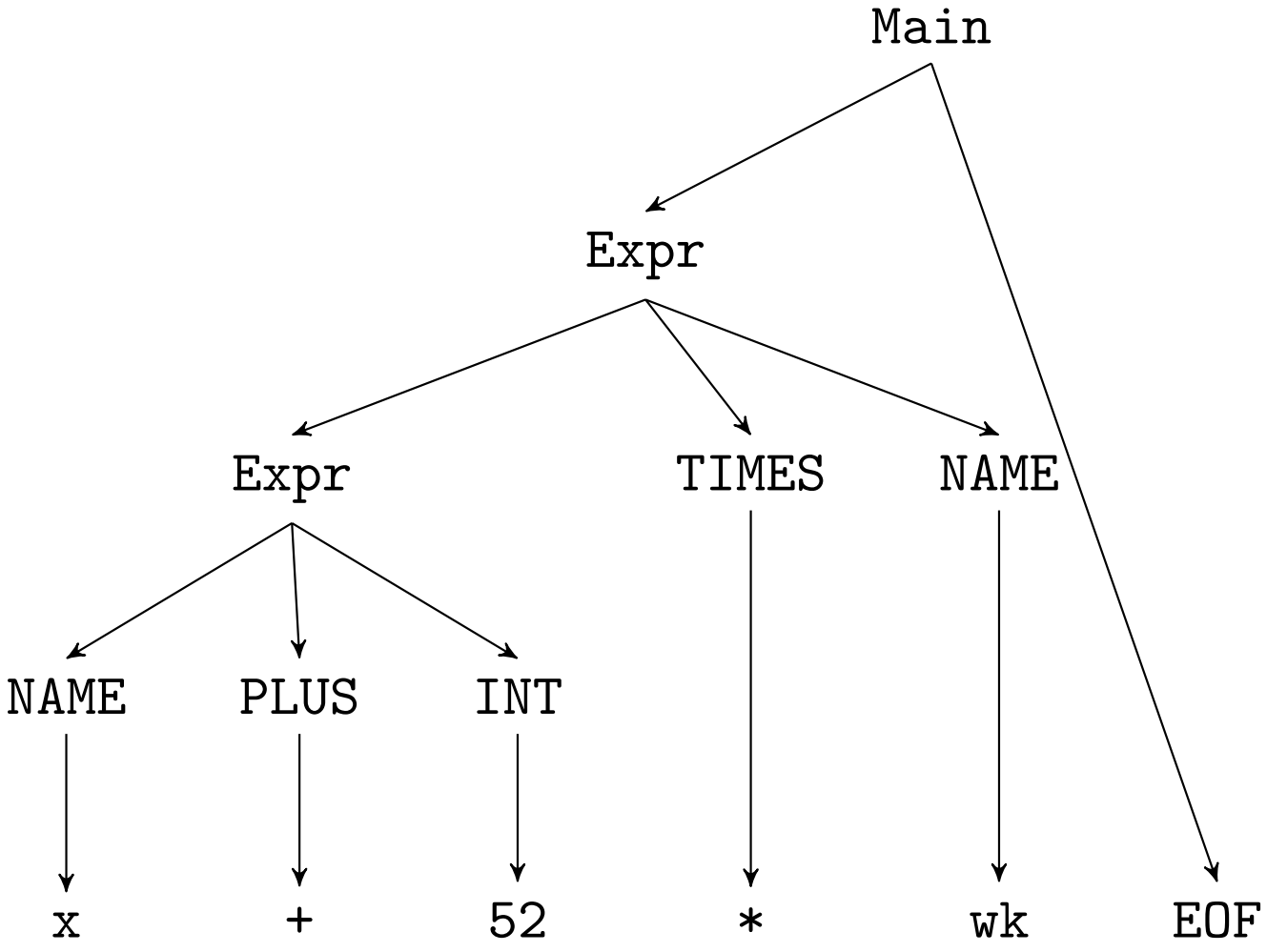


Grammar ambiguity

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

Grammar ambiguity

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



Leftmost and rightmost derivations

Leftmost derivation

Always expand the leftmost nonterminal.
See first example.

Rightmost derivation

Always expand the rightmost nonterminal.
See second example.

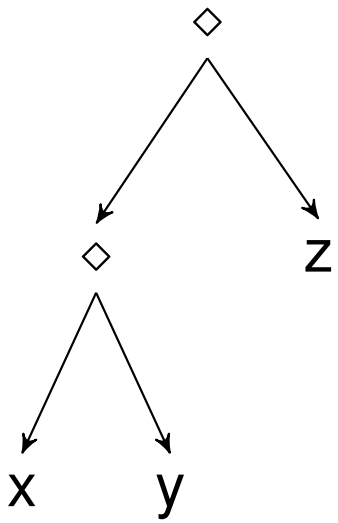
Associativity

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

Associativity

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

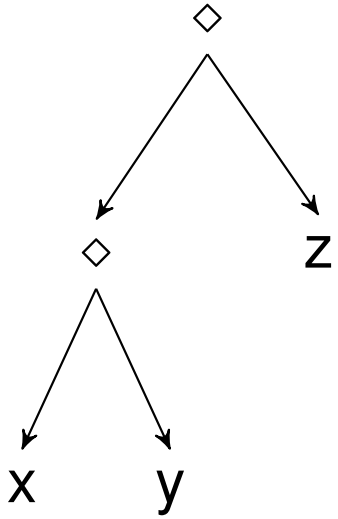
$\diamond$ is left-associative



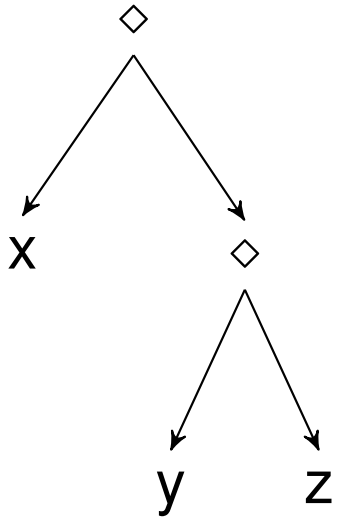
Associativity

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

$\diamond$ is left-associative



$\diamond$ is right-associative



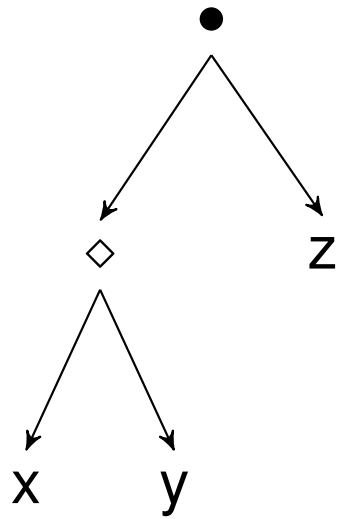
Precedence

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

Precedence

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

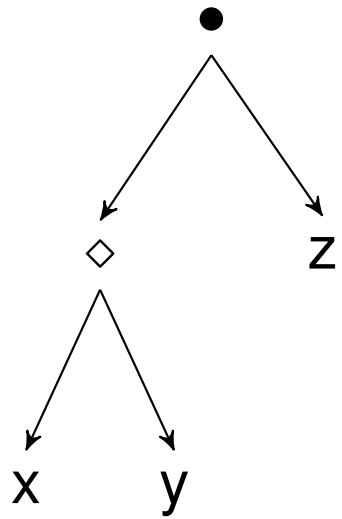
- $\diamond$ has higher precedence



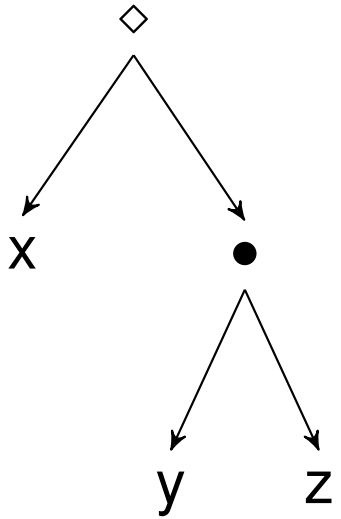
Precedence

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

$\diamond$ has higher precedence



$\bullet$ has higher precedence



Exercise

What Java or C# operators

- are left-associative?
- are right-associative?
- have higher or lower precedence than others?

Java operator precedence

`() [] .`	Left
`x++ x--`	Right
`++x --x +x -x !x ~x (T)x`	Right
`* / %`	Left
`+ -`	Left
`<< >>`	Left
`< <= > >= instanceof`	Left
`== !=`	Left
`&`	Left
`^`	Left
`|`	Left
`&&`	Left
`||`	Left
`b ? tt : ff`	Right
`= += -= *= /= %= &= ^= |= <<= >>= >>>=`	Right

Parsing is inverse derivation

PARSING

Reconstruct the derivation for a string, if possible

Parsing is inverse derivation

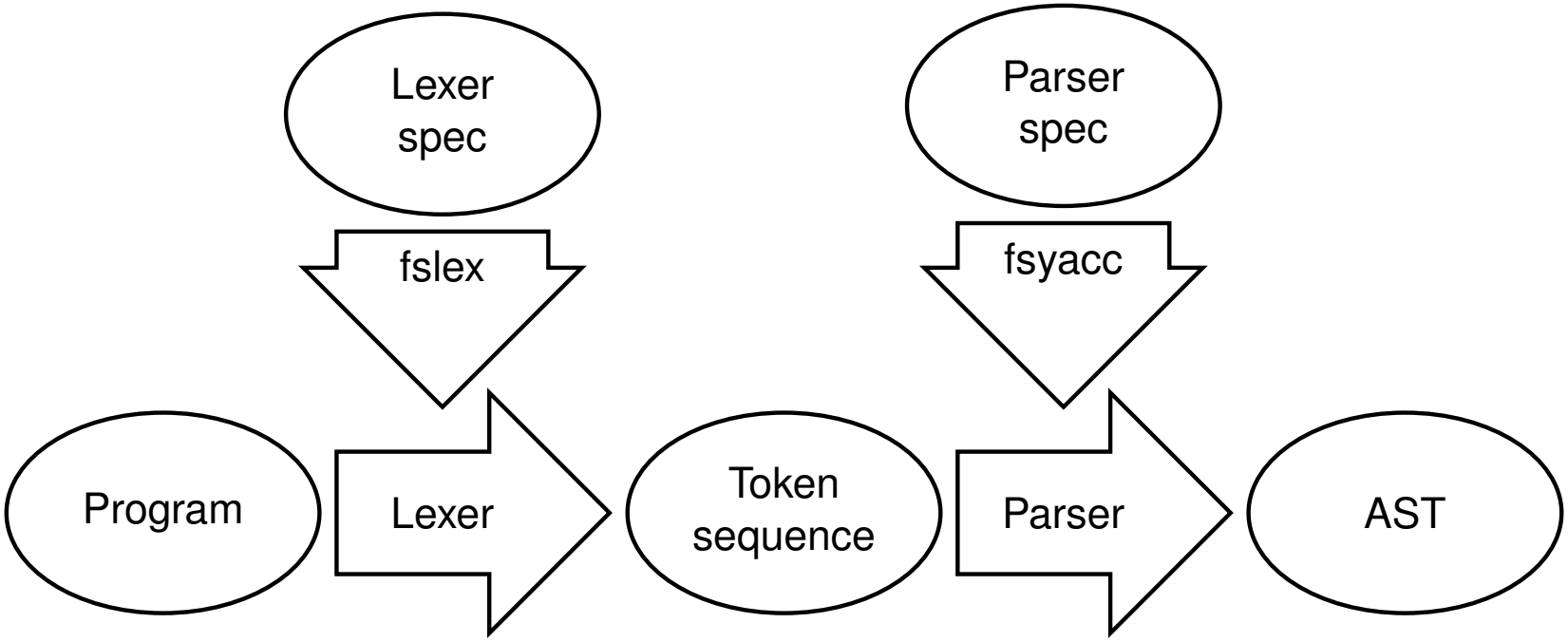
Parsing

Reconstruct the derivation for a string, if possible

Methods

- Top-down: parser structured like grammar. Example next week.
- Generated bottom-up: parser generated using tool.

Parser specification and generator



Tokens, associativity and precedence in fsyacc

```
%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, associativity and precedence in fsyacc

```
%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 - expanded to a datatype

Tokens, associativity and precedence in fsyacc

```
%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

Tokens, associativity and precedence in fsyacc

```
%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

Tokens, associativity and precedence in fsyacc

```
%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

Parser specification

```
%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
```

Parser specification

```
%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

Parser specification

```
%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

Parser specification

```
%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

Parser specification

```
%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

Parser specification

```
%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

Parser specification

```
%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

Parser specification

```
%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

Putting together lexer and parser

From 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 ... "
```

Putting together lexer and parser

From 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 ... "
```

Entry point in parser

Putting together lexer and parser

From 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 ... "
```

Entry point in lexer

Invoking fslex and fsyacc

- Build the lexer and parser vs 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";;
```

Whiteboard

How do we change the lexer and/or parser to

- accept brackets `[ ]` in addition to parens `( )`?

Whiteboard

How do we change the lexer and/or parser to

- accept brackets `[ ]` in addition to parens `( )`?
- accept the division operator (`/`) also?

Whiteboard

How do we change the lexer and/or parser to

- accept brackets `[ ]` in addition to parens `( )`?
- accept the division operator (`/`) also?
- accept the syntax
  `{ x <- 2 in x * 3 }`
  instead of
  `let x = 2 in x * 3 end`
  ?

Whiteboard

How do we change the lexer and/or parser to

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

Break

Plan for today

Lexer specifications
- Regular expressions
- The fslex lexer generation tool
- Automata

Parser specifications
- Grammars
- Parsing
- The fsyacc parser generation tool

Parsing Algorithms
- Top-down
- Bottom-up

Languages and automata

Parsing Algorithms

LR

Read *Left* to right, make derivations from *Rightmost* nonterminal.

- Build parse tree from bottom to top
- Very difficult to get right, but excellent tools (like `fsyacc`)
- Easy encoding of precedence and associativity

Parsing Algorithms

LR

Read *Left* to right, make derivations from *Rightmost* nonterminal.

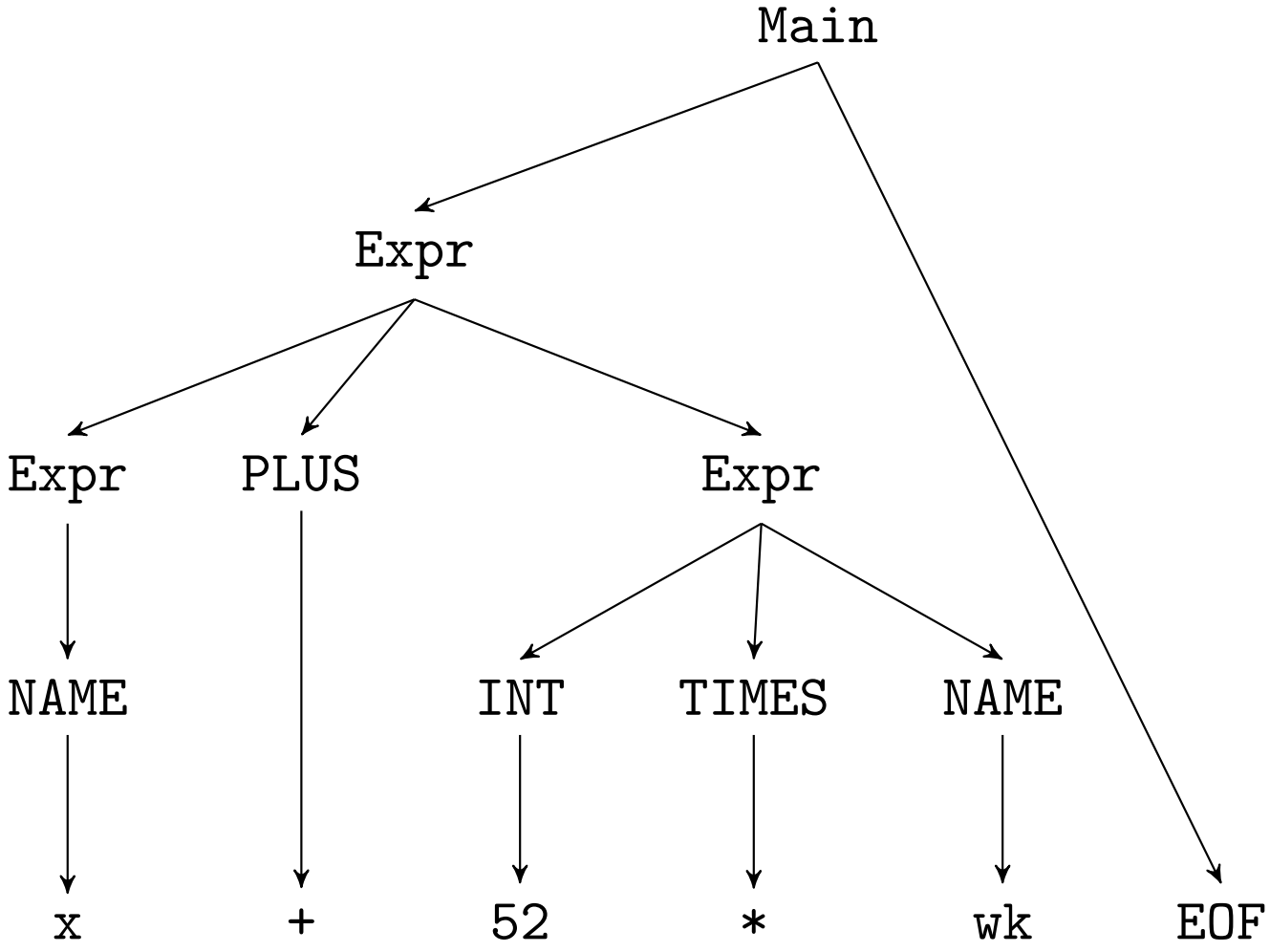
- Build parse tree from bottom to top
- Very difficult to get right, but excellent tools (like `fsyacc`)
- Easy encoding of precedence and associativity

LL

Read *Left* to right, make derivations from *Leftmost* nonterminal.

- Build tree from top to bottom
- Parsing follows structure of grammar — easy to read and write
- Precedence and associativity require modifying grammar

Our derivation — let's parse!



The (fsyacc, LR) parser automaton

(File `ExprPar.fsyacc.output` from `fsyacc -v ExprPar.fsy`)

```
state 19:
     items:
       Expr -> Expr . 'TIMES' Expr
       Expr -> Expr . 'PLUS' Expr
       Expr -> Expr 'PLUS' Expr .
       Expr -> Expr . 'MINUS' Expr
     actions:
       action 'EOF':   reduce Expr --> Expr 'PLUS' Expr
       action 'LPAR':   reduce Expr --> Expr 'PLUS' Expr
       action 'RPAR':   reduce Expr --> Expr 'PLUS' Expr
       action 'END':   reduce Expr --> Expr 'PLUS' Expr
       action 'IN':   reduce Expr --> Expr 'PLUS' Expr
       action 'LET':   reduce Expr --> Expr 'PLUS' Expr
       action 'PLUS':   reduce Expr --> Expr 'PLUS' Expr
       action 'MINUS':   reduce Expr --> Expr 'PLUS' Expr
       action 'TIMES':   shift 21
       action 'EQ':   reduce Expr --> Expr 'PLUS' Expr
       action 'NAME':   reduce Expr --> Expr 'PLUS' Expr
       action 'CSTINT':   reduce Expr --> Expr 'PLUS' Expr
       action 'error':   reduce Expr --> Expr 'PLUS' Expr
       action '#':   reduce Expr --> Expr 'PLUS' Expr
       action '$$':   reduce Expr --> Expr 'PLUS' Expr
     immediate action: <none>
    gotos:
```

- Parser consists of states, transitions, and stack
- Dots in items show current positions
- Actions determine what happens based on input token

Parser stack example

```
Input        Parse stack (top on right)                    Action
-------------------------------------------------------------------
x+52*wk EOF  #0                                            shift #4
```

Parser stack example

```
Input         Parse stack (top on right)                        Action
-------------------------------------------------------------------------
x+52*wk EOF   #0                                                shift #4
+52*wk EOF    #0 x #4                                           reduce by B
```

Parser stack example

```
Input        Parse stack (top on right)                    Action
-------------------------------------------------------------------
x+52*wk EOF  #0                                            shift #4
+52*wk EOF   #0 x #4                                       reduce by B
+52*wk EOF   #0 Expr                                       goto #2
```

Parser stack example

```
Input        Parse stack (top on right)                    Action
-------------------------------------------------------------------
x+52*wk EOF  #0                                            shift #4
+52*wk EOF   #0 x #4                                       reduce by B
+52*wk EOF   #0 Expr                                       goto #2
+52*wk EOF   #0 Expr #2                                    shift #22
```

Parser stack example

```
Input        Parse stack (top on right)                    Action
-------------------------------------------------------------------
x+52*wk EOF  #0                                            shift #4
+52*wk EOF   #0 x #4                                       reduce by B
+52*wk EOF   #0 Expr                                       goto #2
+52*wk EOF   #0 Expr #2                                    shift #22
52*wk EOF    #0 Expr #2 + #22                              shift #5
```

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C

Parser stack example

```
Input        Parse stack (top on right)                    Action
-------------------------------------------------------------------
x+52*wk EOF  #0                                            shift #4
+52*wk EOF   #0 x #4                                       reduce by B
+52*wk EOF   #0 Expr                                       goto #2
+52*wk EOF   #0 Expr #2                                    shift #22
52*wk EOF    #0 Expr #2 + #22                              shift #5
*wk EOF      #0 Expr #2 + #22 52 #5                        reduce by C
*wk EOF      #0 Expr #2 + #22 Expr                         goto #19
```

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H
EOF	#0 Expr	goto #2

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H
EOF	#0 Expr	goto #2
EOF	#0 Expr #2	shift 3

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H
EOF	#0 Expr	goto #2
EOF	#0 Expr #2	shift 3
	#0 Expr #2 EOF #3	reduce by A

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H
EOF	#0 Expr	goto #2
EOF	#0 Expr #2	shift 3
	#0 Expr #2 EOF #3	reduce by A
	#0 Main	goto #1

Parser stack example

Input	Parse stack (top on right)	Action
x+52*wk EOF	#0	shift #4
+52*wk EOF	#0 x #4	reduce by B
+52*wk EOF	#0 Expr	goto #2
+52*wk EOF	#0 Expr #2	shift #22
52*wk EOF	#0 Expr #2 + #22	shift #5
*wk EOF	#0 Expr #2 + #22 52 #5	reduce by C
*wk EOF	#0 Expr #2 + #22 Expr	goto #19
*wk EOF	#0 Expr #2 + #22 Expr #19	shift #21
wk EOF	#0 Expr #2 + #22 Expr #19 * #21	shift #4
EOF	#0 Expr #2 + #22 Expr #19 * #21 wk #4	reduce by B
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr	goto #18
EOF	#0 Expr #2 + #22 Expr #19 * #21 Expr #18	reduce by G
EOF	#0 Expr #2 + #22 Expr	goto #19
EOF	#0 Expr #2 + #22 Expr #19	reduce by H
EOF	#0 Expr	goto #2
EOF	#0 Expr #2	shift 3
	#0 Expr #2 EOF #3	reduce by A
	#0 Main	goto #1
	#0 Main #1	accept

Parser state and actions

Parser state

Stack of symbols and states

Parser actions

- **Shift** read a symbol from input onto the stack, and go to new state
- **Reduce** take grammar rule RHS symbols off the stack and replace them by its LHS nonterminal, and evaluate a semantic action
- **Goto** go to a new parser state (after reduce)

Shift/reduce conflicts

- Sometimes the parser generator does not know whether to shift or to reduce
  (especially if the grammar is ambiguous)
- Warnings are issued by `fsyacc`

Shift/reduce conflicts

- Sometimes the parser generator does not know whether to shift or to reduce
  (especially if the grammar is ambiguous)
- Warnings are issued by `fsyacc`

- To resolve shift/reduce conflicts, change the parser specification
- To understand how, study the parser automaton in `ExprPar.fsyacc.output`

LL parsing: recursive descent

Scheme terms (S-expressions)

Symbols like `foo`, `bar`, `b52`, `+`, `*`

Numbers like `117`, `-4`

Lists in parens: `(foo (+ n 1))`

LL parsing: recursive descent

Scheme terms (S-expressions)

Symbols like `foo`, `bar`, `b52`, `+`, `*`

Numbers like `117`, `-4`

Lists in parens: `(foo (+ n 1))`

Grammar

```
Sexp ::= SYMBOL
       | NUMBER
       | ( Sexp* )
```

Handwritten lexer and parser in C#

```
interface IToken { }
class Lpar : IToken { ... }
class Rpar : IToken { ... }
class Symbol : IToken {
  public readonly String name;
  ...
}
class NumberCst : IToken {
  public readonly int val;
  ...
}
```

Handwritten lexer and parser in C#

```
public static IEnumerator<IToken> Tokenize(TextReader rd) {
  for (;;) {
    int raw = rd.Read();
    char ch = (char)raw;
    if (raw == -1)
      yield break;
    else if (Char.IsWhiteSpace(ch)) { }
    else if (Char.IsDigit(ch))
      yield return new NumberCst(ScanNumber(ch, rd));
    else switch (ch) {
      case '(':
        yield return Lpar.LPAR; break;
      case ')':
        yield return Rpar.RPAR; break;
      case '-': // negative number, or symbol
        ...
      default:
        yield return ScanSymbol(ch, rd);
        break;
} } }
```

Parsing S-expressions top-down

```
Sexp ::= SYMBOL
       | NUMBER
       | ( Sexp* )
```

- If next token is symbol, success
- If next token is number, success
- If next token is left paren, parse repeated S-expressions while next token not right paren

Handwritten recursive descent parser

```
public static void ParseSexp(IEnumerator<IToken> ts) {
  if (ts.Current is Symbol) {
    Console.WriteLine("Parsed symbol " + ts.Current);
  } else if (ts.Current is NumberCst) {
    Console.WriteLine("Parsed number " + ts.Current);
  }
```

Handwritten recursive descent parser

```
public static void ParseSexp(IEnumerator<IToken> ts) {
  if (ts.Current is Symbol) {
    Console.WriteLine("Parsed symbol " + ts.Current);
  } else if (ts.Current is NumberCst) {
    Console.WriteLine("Parsed number " + ts.Current);
  } else if (ts.Current is Lpar) {
    Console.WriteLine("Started parsing list");
    Advance(ts);
    while (!(ts.Current is Rpar)) {
      ParseSexp(ts);
      Advance(ts);
    }
    Console.WriteLine("Ended parsing list");
  }
```

Handwritten recursive descent parser

```
public static void ParseSexp(IEnumerator<IToken> ts) {
  if (ts.Current is Symbol) {
    Console.WriteLine("Parsed symbol " + ts.Current);
  } else if (ts.Current is NumberCst) {
    Console.WriteLine("Parsed number " + ts.Current);
  } else if (ts.Current is Lpar) {
    Console.WriteLine("Started parsing list");
    Advance(ts);
    while (!(ts.Current is Rpar)) {
      ParseSexp(ts);
      Advance(ts);
    }
    Console.WriteLine("Ended parsing list");
  } else
    throw new ArgumentException("Parse error);
}
```

Handwritten recursive descent parser

```
public static void ParseSexp(IEnumerator<IToken> ts) {
  if (ts.Current is Symbol) {
    Console.WriteLine("Parsed symbol " + ts.Current);
  } else if (ts.Current is NumberCst) {
    Console.WriteLine("Parsed number " + ts.Current);
  } else if (ts.Current is Lpar) {
    Console.WriteLine("Started parsing list");
    Advance(ts);
    while (!(ts.Current is Rpar)) {
      ParseSexp(ts);
      Advance(ts);
    }
    Console.WriteLine("Ended parsing list");
  } else
    throw new ArgumentException("Parse error);
}
...
private static void Advance(IEnumerator<IToken> ts) {
  if (!ts.MoveNext())
    throw new ArgumentException("Unxpected eof");
}
```

Plan for today

Lexer specifications
- Regular expressions
- The fslex lexer generation tool
- Automata

Parser specifications
- Grammars
- Parsing
- The fsyacc parser generation tool

Parsing Algorithms
- Top-down
- Bottom-up

Languages and automata

The Chomsky Hierarchy (1958)

The Chomsky Hierarchy (1958)

Type 3: regular grammars

Same expressiveness as regular expressions.

$$A \to cB \qquad A \to B \qquad A \to c \qquad A \to \varepsilon$$

The Chomsky Hierarchy (1958)

Type 3: regular grammars

Same expressiveness as regular expressions.

$$A \to cB \qquad A \to B \qquad A \to c \qquad A \to \varepsilon$$

Type 2: Context-free grammars

$$A \to cBd$$

The Chomsky Hierarchy (1958)

Type 3: regular grammars

Same expressiveness as regular expressions.

$$A \to cB \qquad A \to B \qquad A \to c \qquad A \to \varepsilon$$

Type 2: Context-free grammars

$$A \to cBd$$

Type 1: Context-sensitive grammars

Non-abbreviating rules.

$$aAb \to acAdb$$

The Chomsky Hierarchy (1958)

Type 3: regular grammars

Same expressiveness as regular expressions.

$$A \to cB \qquad A \to B \qquad A \to c \qquad A \to \varepsilon$$

Type 2: Context-free grammars

$$A \to cBd$$

Type 1: Context-sensitive grammars

Non-abbreviating rules.

$$aAb \to acAdb$$

Type 0: unrestricted grammars

Same as term-rewrite systems.

$$0Ay \to 0$$

Chomsky hierarchy and computation

Grammar	Languages	Automaton
Type 3	Regular	Finite automata
Type 2	Context-free	Pushdown automata (finite + stack)
Type 1	Context-sensitive	Bounded Turing machines
Type 0	Recursively enumerable	Turing machines

Reading

This week

- *Programming Languages Concepts*, Chapter 3
- Mogensen's *Introduction to Compiler Design*, Sections 1.1–1.8 and 2.1–2.5
- Tutorial on regular expressions and automata from course website

Next week

- *Programming Languages Concepts*, Chapter 4
- Mogensen's *Introduction to Compiler Design*, Sections 2.11, 2.12, 2.16

Exercises

- Exercises for week 35 due Wednesday:
  1.1, 1.2, 1.4, 2.1, 2.2, 2.3, and optionally 2.6

- Exercises for week 36 due 11 September:
  3.2, 3.3, 3.4, 3.5, 3.6, 3.7