Lexing and Parsing

David Raymond Christiansen

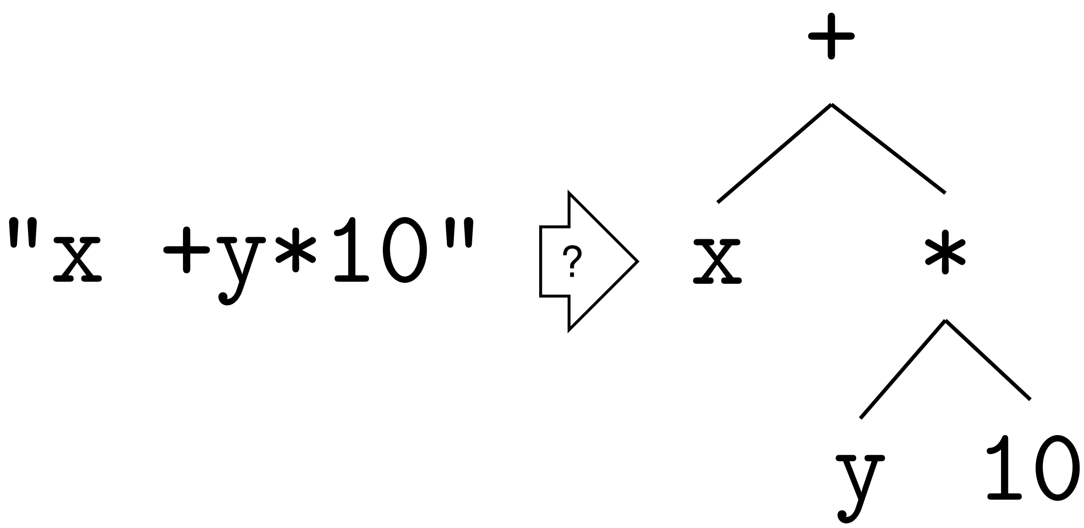
2 September, 2013

These slides have been shortened from the originals available at http://www.itu.dk/courses/BPRD/E2013/

Based on slides by Peter Sestoft

IT University of Copenhagen

From text file to abstract syntax

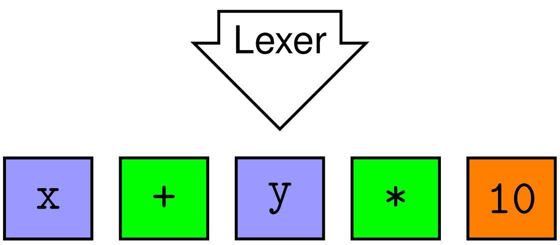


From text file to abstract syntax

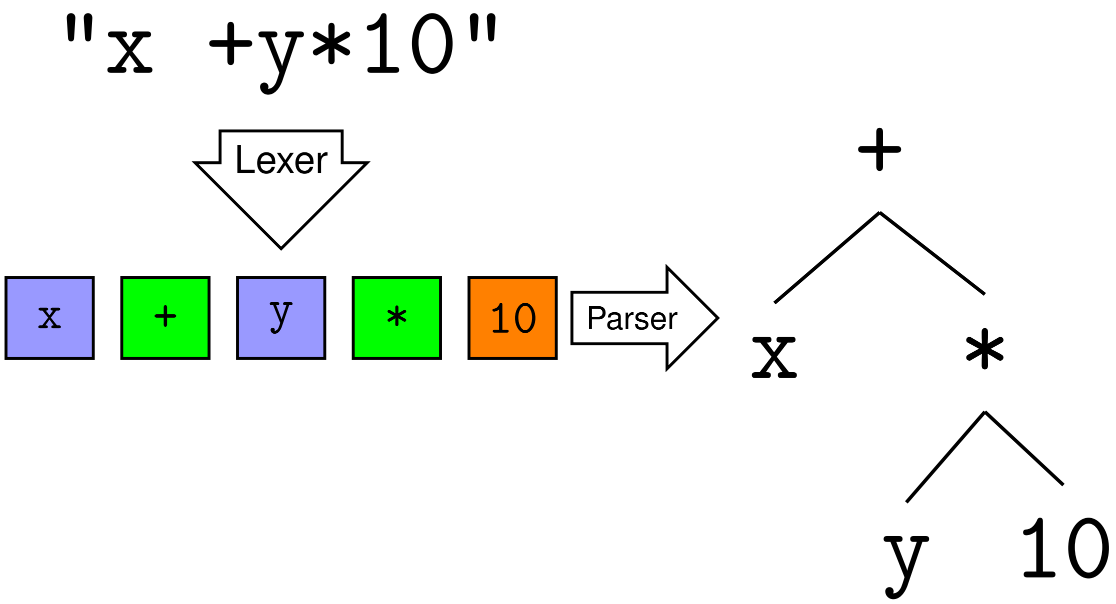
```
"x +y*10"
```


From text file to abstract syntax

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



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

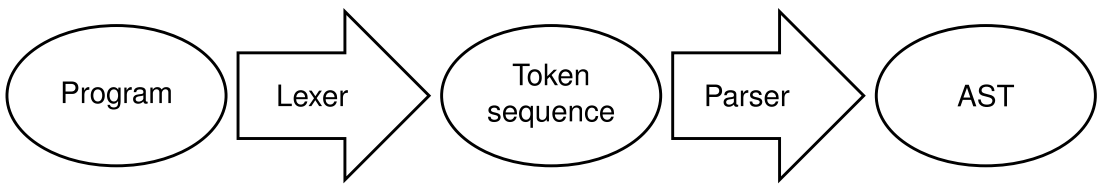
LEXER SPECIFICATIONS
- Regular expressions
- The fslex lexer generation tool
- Automata

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`	{"a"}

Regular expressions

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

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
$\mathtt{a}$	Character $\mathtt{a}$	$\{\text{"a"}\}$
ε	Empty string	$\{\text{""}\}$
$r_1 r_2$	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 $\mathcal{L}(r)$
$\mathtt{a}$	Character $\mathtt{a}$	$\{\text{"a"}\}$
ε	Empty string	$\{\text{""}\}$
$r_1 r_2$	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	$\{s_1 \dots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$

Regular expressions

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

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

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

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

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
$\mathtt{a}$	Character $\mathtt{a}$	$\{"\mathtt{a}"\}$
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	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \| r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

EXAMPLES

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

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

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

EXERCISE

What does $(\mathtt{a}|\mathtt{b})\mathtt{c}^*$ 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

<table>
<tr><th>Abbrev.</th><th>Meaning</th><th>Expansion</th></tr>
<tr><td>[aeiuo]</td><td>Set</td><td>a|e|i|o|u</td></tr>
<tr><td>[0–9]</td><td>Range</td><td>0|1| ... |8|9</td></tr>
</table>

Regular expression abbreviations

<table>
<tr><th>Abbrev.</th><th>Meaning</th><th>Expansion</th></tr>
<tr><td>[aeiuo]</td><td>Set</td><td>a|e|i|o|u</td></tr>
<tr><td>[0–9]</td><td>Range</td><td>0|1| ... |8|9</td></tr>
<tr><td>[0–9a–Z]</td><td>Ranges</td><td></td></tr>
</table>

Regular expression abbreviations

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

Regular expression abbreviations

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

Regular expression abbreviations

Abbrev.	Meaning	Expansion
[aeiuo]	Set	$\mathtt{a} \mid \mathtt{e} \mid \mathtt{i} \mid \mathtt{o} \mid \mathtt{u}$
[0–9]	Range	$0 \mid 1 \mid \ldots \mid 8 \mid 9$
[0–9a–Z]	Ranges	$0 \mid 1 \mid \ldots \mid 8 \mid 9 \mid \mathtt{a} \mid \mathtt{b} \mid \ldots \mid \mathtt{y} \mid \mathtt{z}$
$r?$	Zero or one r	$r \mid \varepsilon$

Regular expression abbreviations

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

Regular expression abbreviations

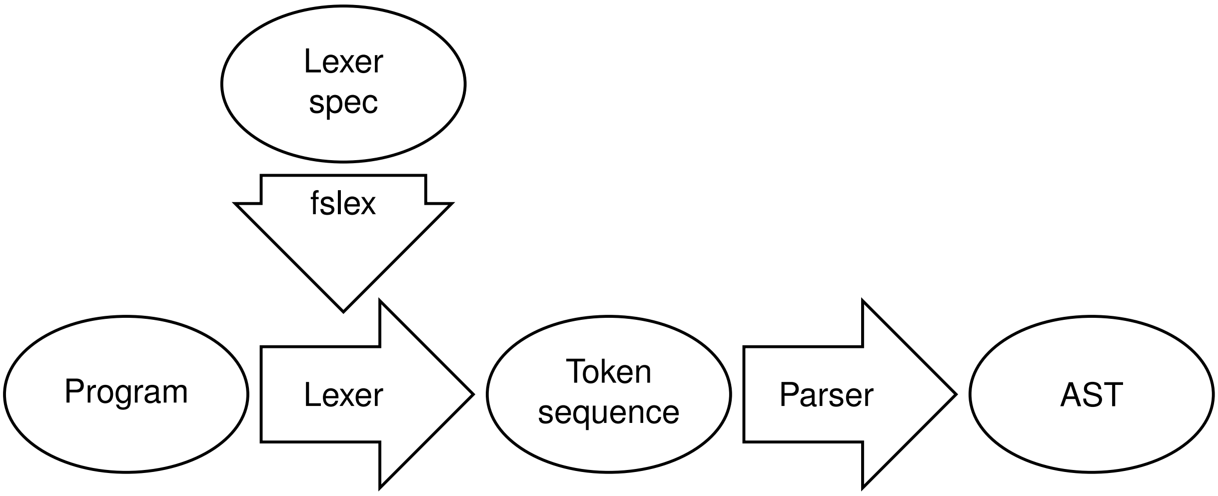
Abbrev.	Meaning	Expansion
[aeiuo]	Set	a|e|i|o|u
[0–9]	Range	0|1|...|8|9
[0–9a–Z]	Ranges	0|1|...|8|9|a|b|...|y|z
$r?$	Zero or one *r*	$r\|\varepsilon$
r^+	One or more *r*	$r\, r^*$

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

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)
```

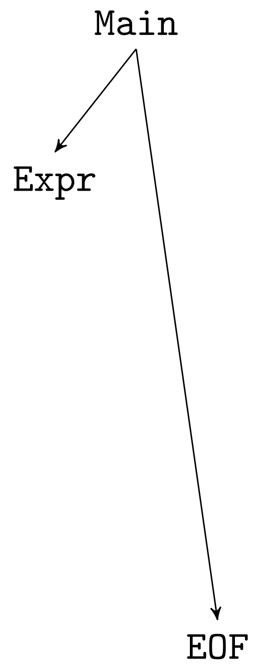
- Nonterminals
- Terminals (from lexer)
- Productions (called A–H)
- Start symbol (the nonterminal Main)

Derivation: grammar as string generator

Main

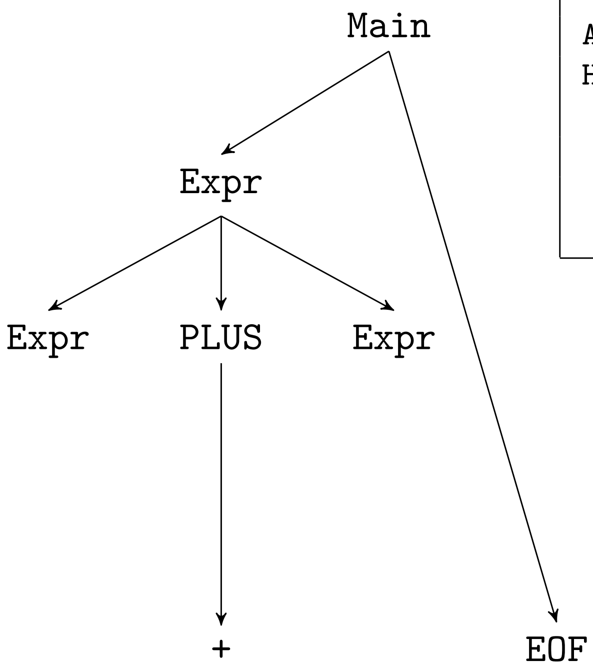
	Main

Derivation: grammar as string generator



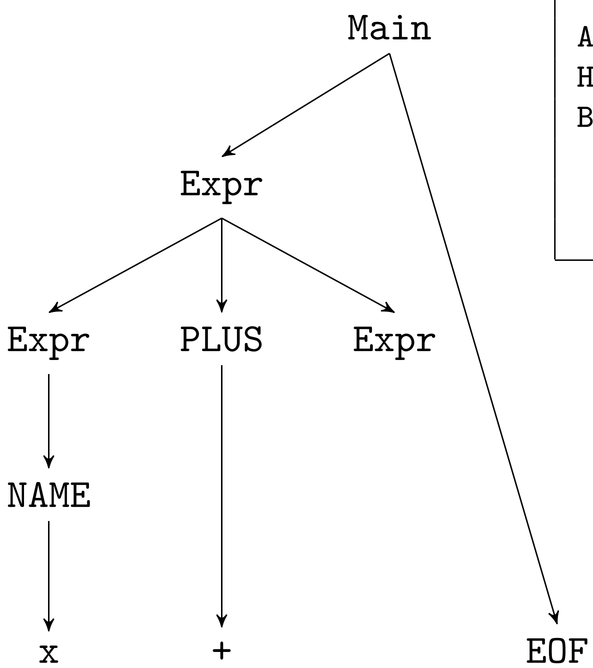
	Main
A	Expr EOF

Derivation: grammar as string generator



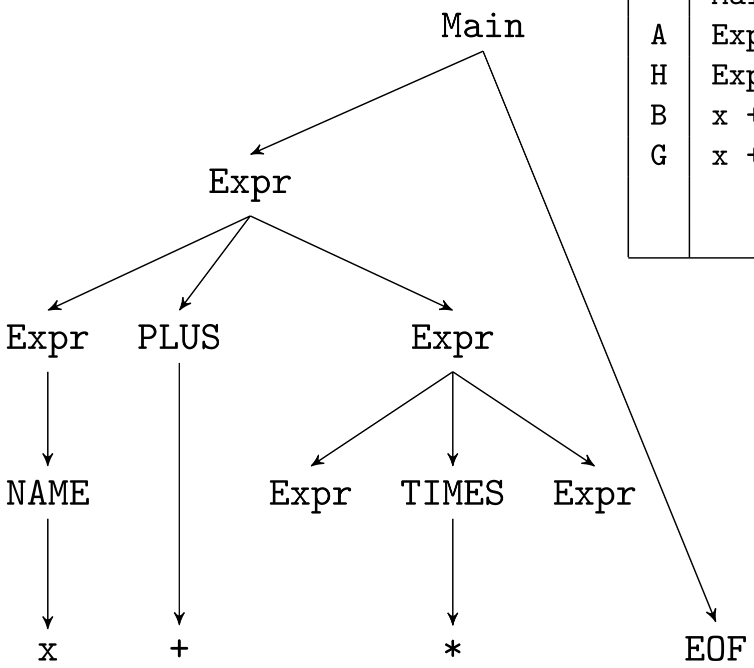
	Main
A	Expr EOF
H	Expr + Expr EOF

Derivation: grammar as string generator



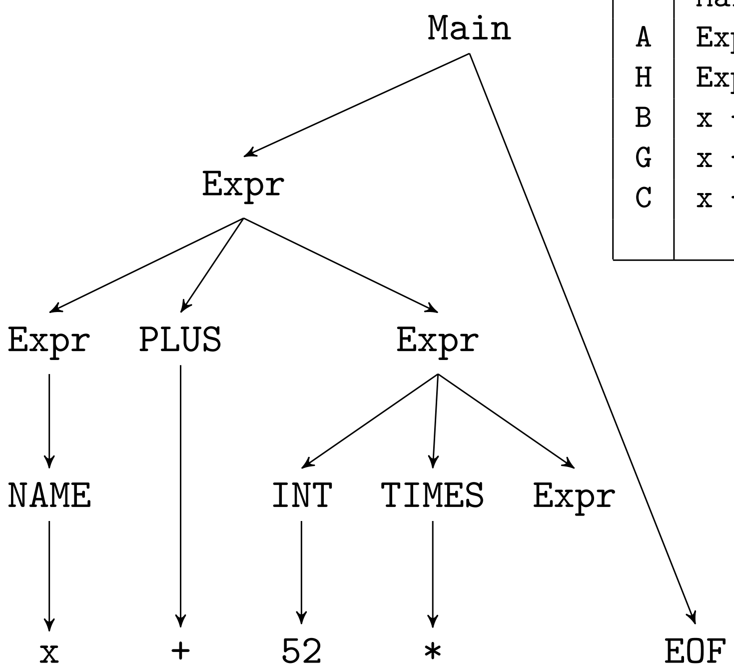
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF

Derivation: grammar as string generator



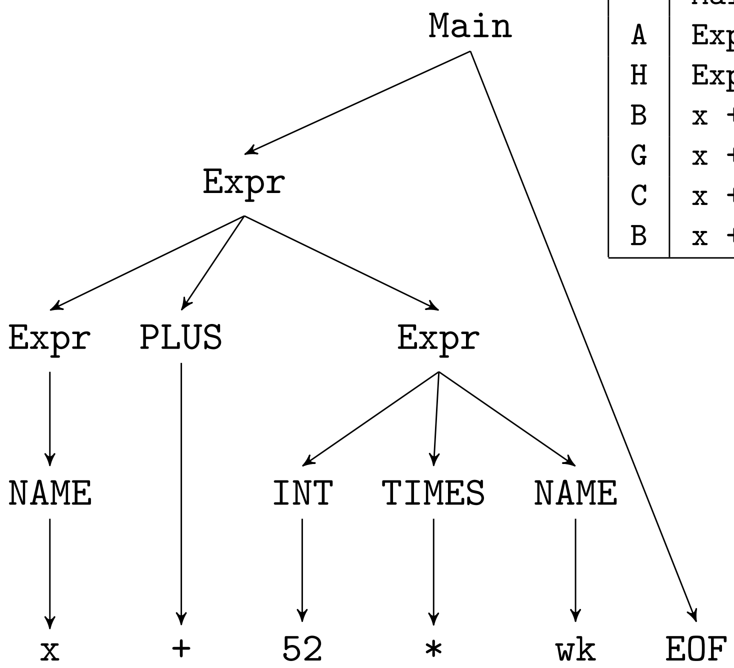
	Main
A	Expr EOF
H	Expr + Expr EOF
B	x + Expr EOF
G	x + Expr * Expr EOF

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

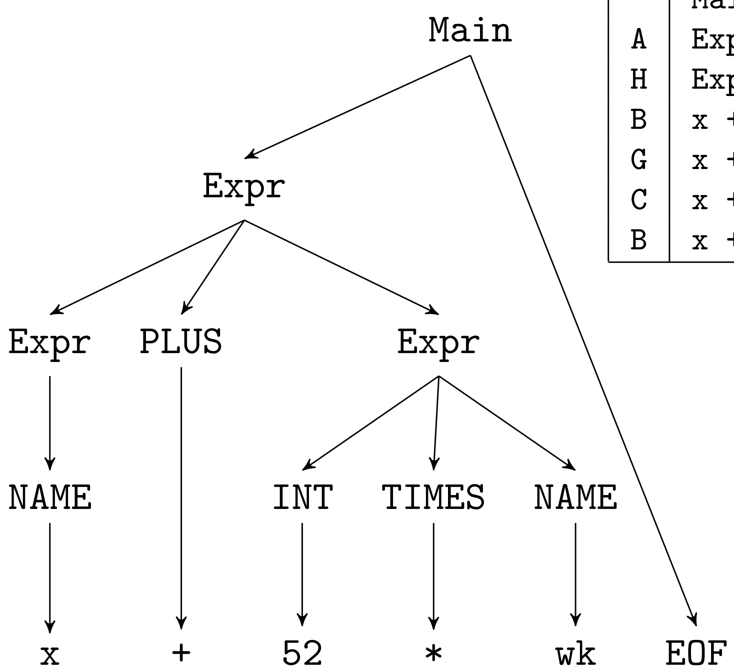
Derivation: grammar as string generator



	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

IT UNIVERSITY OF COPENHAGEN

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

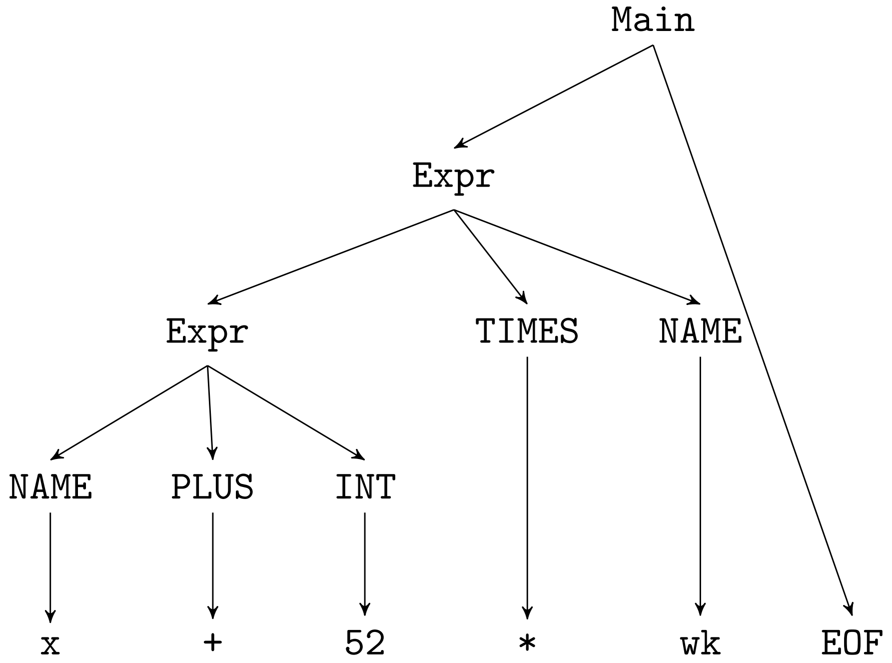
IT UNIVERSITY OF COPENHAGEN

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.

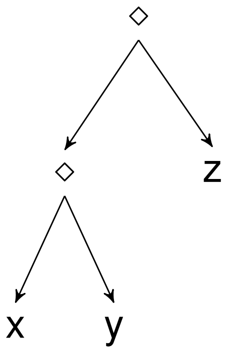
Associativity

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

Associativity

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

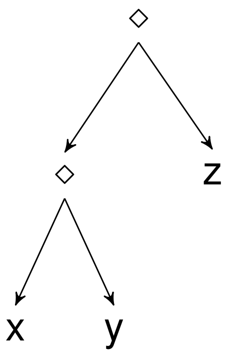
$\diamond$ is left-associative



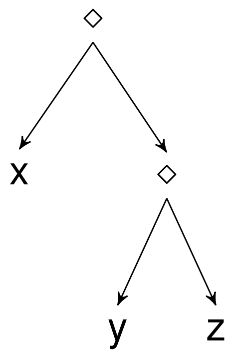
Associativity

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

$\diamond$ is left-associative



$\diamond$ is right-associative



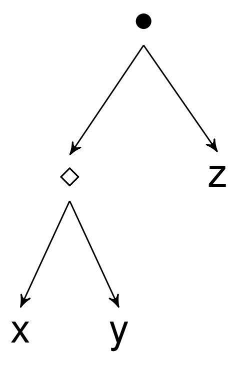
Precedence

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

Precedence

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

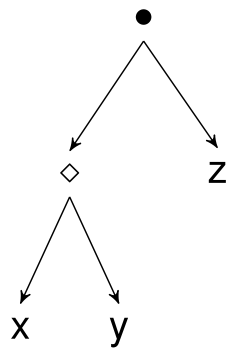
$\diamond$ has higher precedence



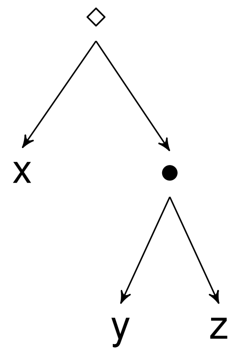
Precedence

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

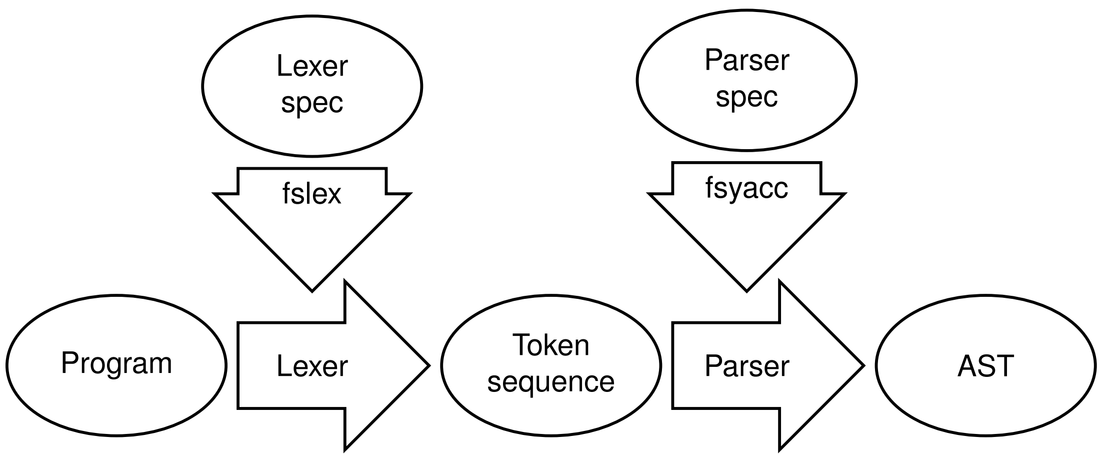
Parsing is inverse derivation

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 &lt;int&gt; CSTINT
%token &lt;string&gt; 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)`?

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)

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