Principles of Programming Languages
Lecture 2: Syntax

Andrei Arusoaie[1]

[1] Department of Computer Science

October 10, 2017

Outline

Alphabet. Lexical analysis. Parsing.

Parse Trees

Abstract syntax trees

Sentences in a programming language

- When designing a PL, one question is:

Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Sentences in a programming language

- When designing a PL, one question is:

Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Sentences in a programming language

- When designing a PL, one question is:

  Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute "legal" phrases
  - *Grammar*

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute "legal" phrases
  - *Grammar*

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute “legal” phrases
  - *Grammar*

Alphabet

The Alphabet of C from the Standard has 96 symbols:

- `a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,z`
- `A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z`
- `0,1,2,3,4,5,6,7,8,9`
- `! " # % & ' ( ) * + , - . /`
- `: ; < = > ? [ \ ] ^ _ { | } ~`
- *Separators*: space, horizontal and vertical tab, form feed, newline

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- In practice we use regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`
  - In K: `syntax Int ::= r"[\\+-]?[0-9]+"`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- In practice we use regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`
  - In K: `syntax Int ::= r"[\\+-]?[0-9]+"`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- In practice we use regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`
  - In K: `syntax Int ::= r"[\\+-]?[0-9]+"`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- In practice we use regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`
  - In K: `syntax Int ::= r"[\\+-]?[0-9]+"`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- In practice we use regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`
  - In K: `syntax Int ::= r"[\\+-]?[0-9]+"`

Parsing

Problem: how to we combine the tokens in (valid) language sentences?

- Answer: we define the *grammar* of the language
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to we combine the tokens in (valid) language sentences?

- Answer: we define the *grammar* of the language
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to we combine the tokens in (valid) language sentences?

- Answer: we define the *grammar* of the language
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to we combine the tokens in (valid) language sentences?

- Answer: we define the *grammar* of the language
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to we combine the tokens in (valid) language sentences?

- Answer: we define the *grammar* of the language
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Example II - Grammar

- Language of palindromic strings using symbols a and b
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- Language of palindromic strings using symbols a and b
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- Language of palindromic strings using symbols a and b
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- How do we "mathematically" describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we "mathematically" describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “mathematically” describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “mathematically” describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “mathematically” describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “mathematically” describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we "mathematically" describe palindromic strings?
  - First, observe that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \to \epsilon$
  - $P \to a$
  - $P \to b$
- Recursion:
  - $P \to aPa$
  - $P \to bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (see LFAC course for details)

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions
- How BNF rules look like:
  - `Bool ::= "true" | "false"`
    (we can use lexical rules to define "basic" non-terminals)
  - `Exp ::= Int | Exp "+" Exp | ...`

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions
- How BNF rules look like:
  - `Bool ::= "true" | "false"`
    (we can use lexical rules to define “basic” non-terminals)
  - `Exp ::= Int | Exp "+" Exp | ...`

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions
- How BNF rules look like:
  - `Bool ::= "true" | "false"`
    (we can use lexical rules to define “basic” non-terminals)
  - `Exp ::= Int | Exp "+" Exp | ...`

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions
- How BNF rules look like:
  - `Bool ::= "true" | "false"`
    (we can use lexical rules to define “basic” non-terminals)
  - `Exp ::= Int | Exp "+" Exp | ...`

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions
- How BNF rules look like:
  - `Bool ::= "true" | "false"`
    (we can use lexical rules to define “basic” non-terminals)
  - `Exp ::= Int | Exp "+" Exp | ...`

BNF - example

- Simple expressions language:

```
Int   ::=  [\+-]?[0-9]+
Exp   ::=  Int
        |  Exp "+" Exp
        |  "(" Exp ")"
```

- In K:

```
syntax Int   ::=  r"[\\+-]?[0-9]+"  //builtin
syntax Exp   ::=  Int
               |  Exp "+" Exp
               |  "(" Exp ")"
```

BNF - example

- Simple expressions language:

```
Int  ::=  [\+-]?[0-9]+
Exp  ::=  Int
       |  Exp "+" Exp
       |  "(" Exp ")"
```

- In K:

```
syntax Int  ::=  r"[\\+-]?[0-9]+"  //builtin
syntax Exp  ::=  Int
              |  Exp "+" Exp
              |  "(" Exp ")"
```

Derivations

Derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (Int + 6)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation	
`Exp`	$\longrightarrow^{(e_2)}$
`Exp + Exp`	$\longrightarrow^{(e_3)}$
`Exp + (Exp)`	$\longrightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\longrightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\longrightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\longrightarrow^{(e_1)}$
`Int + (Int + Int)`	$\longrightarrow^{(i_1)}$
`Int + (Int + 6)`	$\longrightarrow^{(i_1)}$
`Int + (4 + 6)`	$\longrightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (Int + 6)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

`Exp`	$\to^{(e_2)}$
`Exp + Exp`	$\to^{(e_3)}$
`Exp + (Exp)`	$\to^{(e_2)}$
`Exp + (Exp + Exp)`	$\to^{(e_1)}$
`Exp + (Exp + Int)`	$\to^{(e_1)}$
`Exp + (Int + Int)`	$\to^{(e_1)}$
`Int + (Int + Int)`	$\to^{(i_1)}$
`Int + (Int + 6)`	$\to^{(i_1)}$
`Int + (4 + 6)`	$\to^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (Int + 6)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

`Exp`	$\longrightarrow^{(e_2)}$
`Exp + Exp`	$\longrightarrow^{(e_3)}$
`Exp + (Exp)`	$\longrightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\longrightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\longrightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\longrightarrow^{(e_1)}$
`Int + (Int + Int)`	$\longrightarrow^{(i_1)}$
`Int + (Int + 6)`	$\longrightarrow^{(i_1)}$
`Int + (4 + 6)`	$\longrightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

`Exp`	$\longrightarrow^{(e_2)}$
`Exp + Exp`	$\longrightarrow^{(e_3)}$
`Exp + (Exp)`	$\longrightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\longrightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\longrightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\longrightarrow^{(e_1)}$
`Int + (Int + Int)`	$\longrightarrow^{(i_1)}$
`Int + (Int + 6)`	$\longrightarrow^{(i_1)}$
`Int + (4 + 6)`	$\longrightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

```
Exp                      →(e2)
Exp + Exp                →(e3)
Exp + (Exp)              →(e2)
Exp + (Exp + Exp)        →(e1)
Exp + (Exp + Int)        →(e1)
Exp + (Int + Int)        →(e1)
Int + (Int + Int)        →(i1)
Int + (Int + 6)          →(i1)
Int + (4 + 6)            →(i1)
2 + (4 + 6)
```

Grammar

```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Derivations

Derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Exp + Int)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (Int + 6)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

Grammar

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Derivation

```
Exp                          →(e2)
Exp + Exp                    →(e3)
Exp + (Exp)                  →(e2)
Exp + (Exp + Exp)            →(e1)
Exp + (Exp + Int)            →(e1)
Exp + (Int + Int)            →(e1)
Int + (Int + Int)            →(i1)
Int + (Int + 6)              →(i1)
Int + (4 + 6)                →(i1)
2 + (4 + 6)
```

Grammar

```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Another possible derivation

`Exp`	$\rightarrow_{(e_2)}$
`Exp + Exp`	$\rightarrow_{(e_3)}$
`Exp + (Exp)`	$\rightarrow_{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow_{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow_{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow_{(e_1)}$
`Int + (Int + Int)`	$\rightarrow_{(i_1)}$
`Int + (4 + Int)`	$\rightarrow_{(i_1)}$
`Int + (4 + 6)`	$\rightarrow_{(i_1)}$
`2 + (4 + 6)`	

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	|	`Exp "+" Exp`	(e_2)
	|	`"(" Exp ")"`	(e_3)

Derivations

Another possible derivation

```
Exp                        →(e2)
Exp + Exp                  →(e3)
Exp + (Exp)                →(e2)
Exp + (Exp + Exp)          →(e1)
Exp + (Int + Exp)          →(e1)
Exp + (Int + Int)          →(e1)
Int + (Int + Int)          →(i1)
Int + (4 + Int)            →(i1)
Int + (4 + 6)              →(i1)
2 + (4 + 6)
```

```
Int   ::=  [\+-]?[0-9]+    (i1)
Exp   ::=  Int             (e1)
        |  Exp "+" Exp     (e2)
        |  "(" Exp ")"     (e3)
```

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

`Int`	`::=`	`[\+-]?[0-9]+`	(i_1)
`Exp`	`::=`	`Int`	(e_1)
	`|`	`Exp "+" Exp`	(e_2)
	`|`	`"(" Exp ")"`	(e_3)

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Derivations

Another possible derivation

`Exp`	$\rightarrow^{(e_2)}$
`Exp + Exp`	$\rightarrow^{(e_3)}$
`Exp + (Exp)`	$\rightarrow^{(e_2)}$
`Exp + (Exp + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Exp)`	$\rightarrow^{(e_1)}$
`Exp + (Int + Int)`	$\rightarrow^{(e_1)}$
`Int + (Int + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + Int)`	$\rightarrow^{(i_1)}$
`Int + (4 + 6)`	$\rightarrow^{(i_1)}$
`2 + (4 + 6)`	

```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Derivations

Another possible derivation

$$
\begin{array}{ll}
\texttt{Exp} & \rightarrow^{(e_2)} \\
\texttt{Exp + Exp} & \rightarrow^{(e_3)} \\
\texttt{Exp + (Exp)} & \rightarrow^{(e_2)} \\
\texttt{Exp + (Exp + Exp)} & \rightarrow^{(e_1)} \\
\texttt{Exp + (Int + Exp)} & \rightarrow^{(e_1)} \\
\texttt{Exp + (Int + Int)} & \rightarrow^{(e_1)} \\
\texttt{Int + (Int + Int)} & \rightarrow^{(i_1)} \\
\texttt{Int + (4 + Int)} & \rightarrow^{(i_1)} \\
\texttt{Int + (4 + 6)} & \rightarrow^{(i_1)} \\
\texttt{2 + (4 + 6)} &
\end{array}
$$

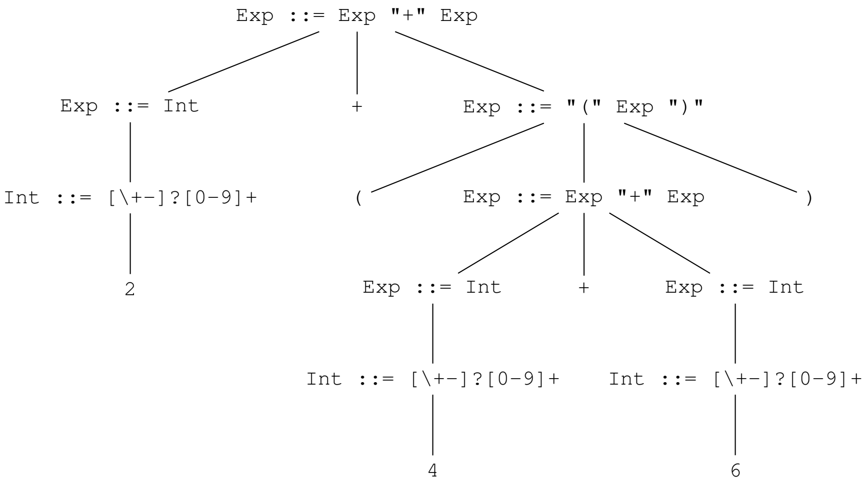
```
Int  ::=  [\+-]?[0-9]+     (i1)
Exp  ::=  Int              (e1)
       |  Exp "+" Exp      (e2)
       |  "(" Exp ")"      (e3)
```

Parse trees

Parse tree for `2 + (4 + 6)`:

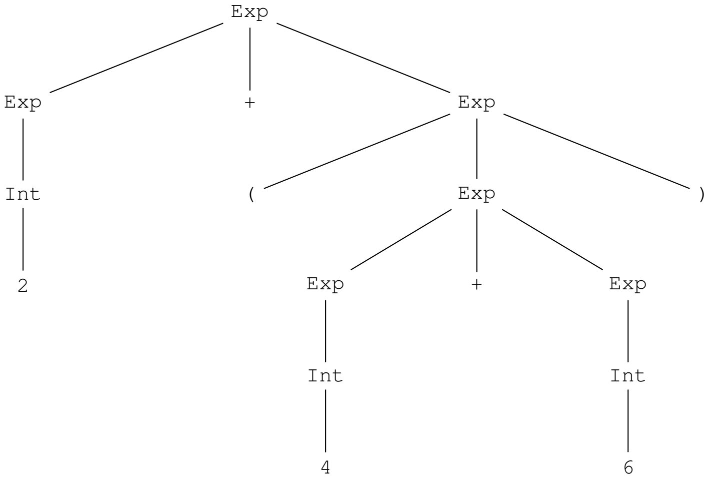
Parse trees

Parse tree for `2 + (4 + 6)`:



Parse trees: simplified

Parse tree for `2 + (4 + 6)`:



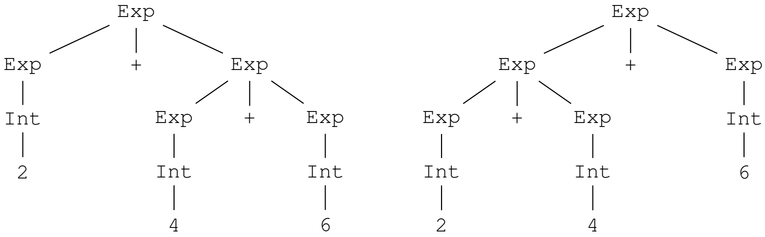
Multiple parses available

Possible parse trees for `2 + 4 + 6`:

Experiment with K!

Multiple parses available

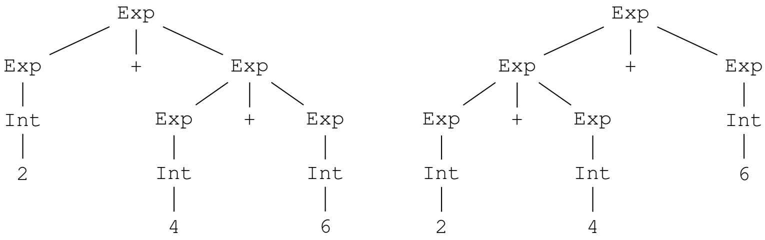
Possible parse trees for `2 + 4 + 6`:



Experiment with K!

Multiple parses available

Possible parse trees for `2 + 4 + 6`:



Experiment with K!

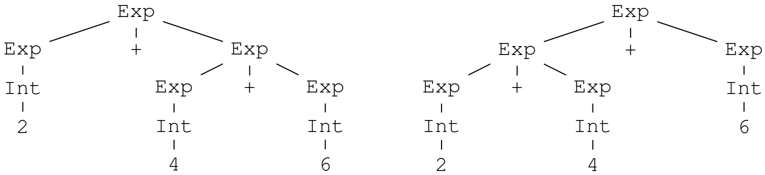
Ambiguities

Possible parse trees for `2 + 4 + 6`:

- Solutions?
  - Use the parentheses defined in the syntax: ‘`(`’ and ‘`)`’
  - Use [bracket] attribute
  - Use associativity attributes: [left] or [right]

Ambiguities

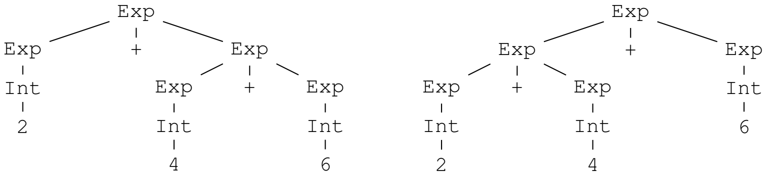
Possible parse trees for `2 + 4 + 6`:



- Solutions?
  - Use the parentheses defined in the syntax: '(' and ')'
  - Use [bracket] attribute
  - Use associativity attributes: [left] or [right]

Ambiguities

Possible parse trees for `2 + 4 + 6`:



- Solutions?
  - Use the parentheses defined in the syntax: '`(`' and '`)`'
  - Use [bracket] attribute
  - Use associativity attributes: [left] or [right]

Extend the syntax of expressions

- Append division:

```
syntax Exp  ::=  Int
               |  Exp "/" Exp  [left]
               |  Exp "+" Exp  [left]
               |  "(" Exp ")"  [bracket]
```

- Question: what's the parse tree of 4 / 2 + 6?

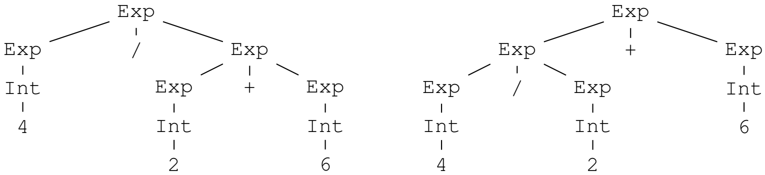
Priorities

Possible parse trees for `4 / 2 + 6`:

- Is this what we want?

Priorities

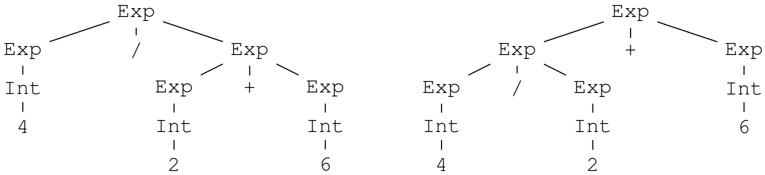
Possible parse trees for `4 / 2 + 6`:



- Is this what we want?

Priorities

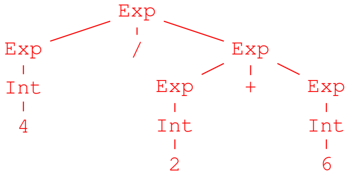
Possible parse trees for `4 / 2 + 6`:



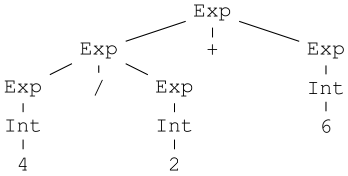
- Is this what we want?

Priorities

NO

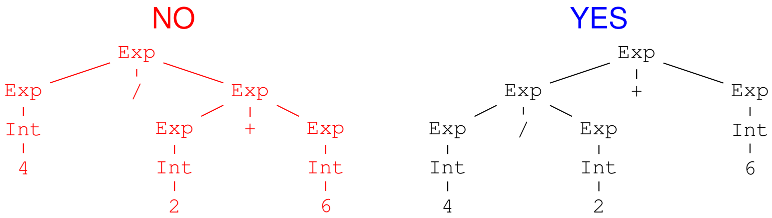


YES



- Experiment with K!

Priorities



- Experiment with K!

Extended BNF

- Extended BNF: “>”
- Solution:

```
syntax Exp  ::=  Int
              |  Exp "/" Exp   [left]
              >  Exp "+" Exp   [left]
              |  "(" Exp ")"   [bracket]
```

- Expected result:

- Experiment with K!

Extended BNF

- Extended BNF: “>”
- Solution:

```
syntax Exp  ::=  Int
              |  Exp "/" Exp  [left]
              >  Exp "+" Exp  [left]
              |  "(" Exp ")"  [bracket]
```

- Expected result:

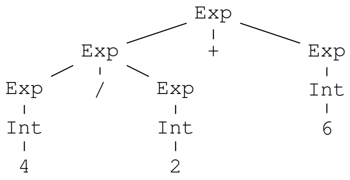
- Experiment with K!

Extended BNF

- Extended BNF: “>”
- Solution:

```
syntax Exp  ::=  Int
              |  Exp "/" Exp  [left]
              >  Exp "+" Exp  [left]
              |  "(" Exp ")"  [bracket]
```

- Expected result:



- Experiment with K!

Extend the syntax of expressions

- Append minus:

```
syntax Exp  ::=  Int
              |  Exp "/" Exp  [left]
              >  Exp "+" Exp  [left]
              |  Exp "-" Exp  [left]
              |  "(" Exp ")"  [bracket]
```

- Question: what's the parse tree of 4 - 2 + 6?

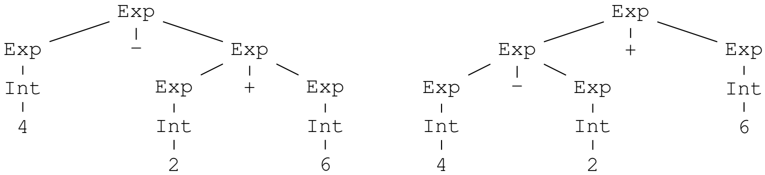
Operation associativity

Possible parse trees for `4 - 2 + 6`:

- How can we avoid this?

Operation associativity

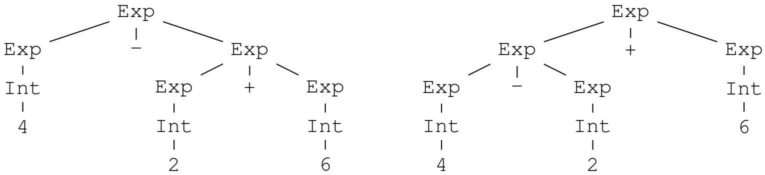
Possible parse trees for `4 - 2 + 6`:



- How can we avoid this?

Operation associativity

Possible parse trees for `4 - 2 + 6`:



- How can we avoid this?

Associativity

- Solution: use left or right

```
syntax Exp  ::=  Int
              |  Exp "/" Exp
              >  left:
                 Exp "+" Exp
              |  Exp "-" Exp
              |  "(" Exp ")"
```

- Variants in K: left , right, non-assoc
- Experiment with K!

Extend the syntax again

- Define boolean expressions and statements

```
 syntax Exp   ::=   Id | Int
                 |   Exp "/" Exp                  [left]
                 >   left:
                     Exp "+" Exp                  [left]
                 |   Exp "-" Exp                  [left]
                 |   "(" Exp ")"                  [bracket]
syntax BExp   ::=   Exp "<=" Exp
                 |   "(" BExp ")"                 [bracket]
syntax Stmt   ::=   Id "=" Exp ";"
                 |   "if" BExp Stmt "else" Stmt
                 |   "if" BExp Stmt
```

Dangling `else` problem

- Consider the following program:

```
if (x <= 0)
  if (y <= 0)
    y = y + 1;
   else x = x + 1;
```

- The `else` belongs to which `if` statement?
- Experiment with K!

Dangling `else` problem

- Consider the following program:

```
if (x <= 0)
  if (y <= 0)
    y = y + 1;
   else x = x + 1;
```

- The `else` belongs to which `if` statement?
- Experiment with K!

Dangling `else` - Solution

- Solution:

```
syntax Stmt   ::=   Id "=" Exp ";"
                |   "if" BExp Stmt "else" Stmt   [avoid]
                |   "if" BExp Stmt
```

- Experiment with K!

Lists in K

- EBNF notation: `Ids ::= (Id, ",")*`
- In K: `syntax Ids ::= List{Id, ","}`
- Experiment with K!

Abstract Syntax Trees - AST

- Parse trees are *concrete* representations of the programs
- Abstract Syntax Trees are *abstract* representations of programs
- Some advantages of ASTs compared to parse trees:
  - ASTs are “smaller” in size
  - ASTs do not contain useless details (e.g., bracket)
  - Takes less time to process them

Abstract Syntax Trees - AST

- Parse trees are *concrete* representations of the programs
- Abstract Syntax Trees are *abstract* representations of programs
- Some advantages of ASTs compared to parse trees:
  - ASTs are “smaller” in size
  - ASTs do not contain useless details (e.g., bracket)
  - Takes less time to process them

AST - Example

- Append *labels*:

```
syntax Exp   ::=  Int
                |  Exp "/" Exp   [klabel(div), left]
                |  Exp "+" Exp   [klabel(plus), left]
                |  "(" Exp ")"   [bracket]
```

AST - Example

Recall: `4 - 2 + 6`

Parse Tree

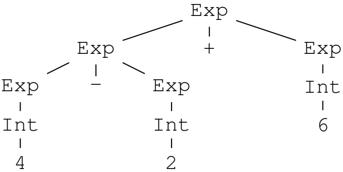
Abstract Syntax Tree

- Typical ASCII representation: `plus(minus(4, 2), 6)`
- Experiment with K!

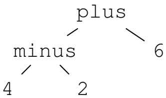
AST - Example

Recall: `4 - 2 + 6`

Parse Tree



Abstract Syntax Tree

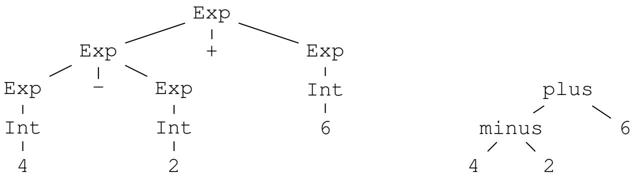


- Typical ASCII representation: `plus(minus(4, 2), 6)`
- Experiment with K!

AST - Example

Recall: `4 - 2 + 6`

Parse Tree | Abstract Syntax Tree



- Typical ASCII representation: `plus(minus(4, 2), 6)`
- Experiment with K!

Bibliography

- Sections 2.1-2.4 from the [Gabbrielli&Martini 2010].

Lab - this week

- Defining the syntax of an imperative programming language