ALGORITMEN & DATASTRUCTUREN COLLEGE 5

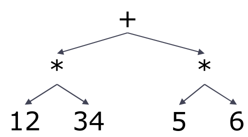
parsing: recognize input & build trees

the goal

2

"12 * 34 + 5 * 6"

↓



the approach

3

- define a grammar for the input

  E → E O E | (E) | **id** | **Number**

  O → **+** | **-** | * | /

- a recursive parse function for each rule in the grammar

```
public class RDparser
{   public RDparser ( String s );
    public BasicNode parseExpr ( );
    public BasicNode parseId ( );
    public BasicNode parseNumber ( );
    …
```

parseerbomen

4

- bomen representeren ook goed complexe invoer
  - door boomstructuur is het eenvoudiger de input te analyseren en manipuleren
- voorbeelden
  - compiler, IDE
  - spreadsheets, rekenmachine
  - bewijssystemen
  - spellingscontrole
  - alle systemen met een dedicated taaltje
  - ..
- het maken van een boom die de invoer representeert heet **parseren**
  - we bekijken hier de principes

herkennen van invoer: parseren

5

- herken invoer volgens gegeven regels
- goede regels zijn het halve werk
- er zijn verschillende formalismen in gebruik om de invoer vast te leggen
  - reguliere expressies
  - contextvrije grammatica
  - ..

zinnen en talen

6

- alfabet Σ:
  - verzameling van symbolen die mogen voorkomen
  - binair: $\Sigma = \{ 0, 1 \}$
  - hexadecimaal: $\Sigma = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F \}$
    - ook te schrijven als [0-9A-F]
- zin: eindig rijtje symbolen
  - ε is de lege zin (heeft lengte 0)
  - lengte van een zin is altijd eindig
- taal: verzameling van zinnen
  - $\varnothing$ lege taal, niet gelijk aan $\{ \varepsilon \}$
  - mag wel oneindig groot zijn

manipulatie van zinnen/talen

7

- concatenatie zinnen: maak zin door er 2 achter elkaar te plakken
  - als s = 12 en t = 34, dan is $s\ t$ gelijk aan 1234
  - als er geen verwarring is schrijven we $s\ t$ als st
  - $s\,\varepsilon = \varepsilon\,s = s$
- concatenatie talen: plak zinnen achter elkaar, uit iedere taal eentje
  - $L\ M = \{ st \mid s \in L \wedge t \in M \}$
  - $L\ M \neq M\ L$
- vereniging: gooi alle zinnen bij elkaar in 1 taal
  - $L \cup M = \{ s \mid s \in L \vee s \in M \}$
  - $L \cup M = M \cup L$

manipulatie van zinnen/talen 2

8

- exponent zin s^i: plak zin i keer achter zichzelf
  - $s^0 = \varepsilon$
  - voor i > 0: $s^i = s^{i-1}s$ (of $s^i = s\,s^{i-1}$)
  - als s = 10 dan is s^3 = 101010
- exponent taal
  - $L^0 = \{ \varepsilon \}$
  - $L^i = L^{i-1}L$
- afsluiting: zinnen uit L zo vaak achter elkaar als je wil
  - $L^* = L^0 \cup LL^*$
- positieve afsluiting: minstens één keer $L^+ = LL^*$

reguliere expressies

9

- ε beschrijft taal { ε }
- ieder a (voor $a \in \Sigma$) beschrijft { a }
- als r de taal L beschrijft dan beschrijft (r) ook L
- (r)|(s) beschrijft $L(r) \cup L(s)$
- (r)(s) beschrijft $L(r)\ L(s)$
- $(r)^*$ beschrijft $(L(r))^*$; $(r)^+$ beschrijft $(L(r))^+$
- r ? is afkorting voor: r | ε
- minder haakjes:
  - voor iedere operator o: r o s o t lezen als (r o s) o t
  - * en + binden het sterkst
  - dan concatenatie
  - dan keuze

reguliere definities

10

- geef namen aan reguliere expressie
  - definitie voor gebruik
  - geen recursie, of alleen aan einde/begin eigen rechterkant
  - namen vet om verwarming met symbolen te voorkomen
- voorbeeld: identifiers in C++/Java

letter	→ [a-z] | [A-Z]
cijfer	→ [0-9]
underscore	→ _
identifier	→ (**letter** | **underscore**) (**letter** | **underscore** | **cijfer**)*

  - beschrijft: [a-zA-Z_] [a-zA-Z_0-9]*

integer	→ [-+]? [0-9]+
natural	→ cijfer **natural** ?

geeft Kleene star

kracht reguliere definities

11

- uitstekende geschikt voor definitie van taalelementen
  - identifiers, getallen, ..
- kan niet de taal beschrijven die alleen netjes geneste paren haakjes bevat
- de reguliere definitie

  haakjes → (*)*

  - bevat ook (((en ( )), ...

bold om aan te geven dat het symbolen zijn

voorkom verwarring door gewone haakjes te verbieden

contextvrije grammatica's

12

- ziet er uit als reguliere definitie, maar
  - mag gebruikt worden voor definitie, en
  - mag recursief zijn
- lost probleem met geneste haakjes op:

  haakjes → (haakjes) haakjes | ε

- we houden vaak de reguliere definities voor zaken als namen en cijfers
  - hoewel die ook met een CFG gaan
  - reguliere definities zijn veel eenvoudiger te parseren dan een CFG

Context Free Grammar

voorbeeld CFG: expressies

13

expr → expr op expr
expr → (expr)
expr → **id**
expr → **getal**
op → **+**
op → **-**
op → *
op → /

- of

E → E O E | (E) | **id** | **getal**
O → **+** | **-** | * | /

- als we niets anders zeggen is de eerste regel de startregel
- layout (spaties etc.) definiëren we nooit, maar mag (bijna) overal

afleiding

14

- hoe maken we met de grammatica een zin
  - zolang er non-terminals (namen van productieregels) in staan moet je daar een invulling voor geven

E → E * E | E **+** E | **num**

- kortste afleiding:

E ⇒ **num**

- iets langer

E ⇒ E + E
⇒ E + E * E
⇒ **num** + E * E
⇒ **num** + **num** * E
⇒ **num** + **num** * **num**

- we zeggen ook

E $\Rightarrow^*$ **num** + **num** * **num**

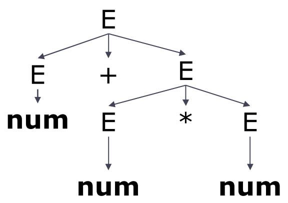
volgorde van regels toepassen maakt vaak niet veel uit

parse tree

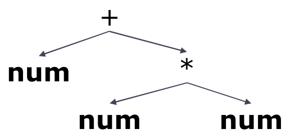
15

- grafische weergave van afleiding

E → E * E | E + E | **num**



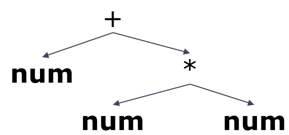
ook vaak getekend als:

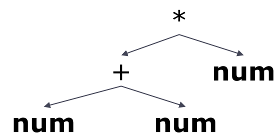


ambiguiteit

16

- ambigue: meer dan een afleidingen voor een zin
- verschillende afleidingen hebben vaak ander interpretatie
- voor E → E * E | E + E | **num**
- en de zin **num** + **num** * **num**
- mogelijke afleidingen:





ambiguiteit 2

17

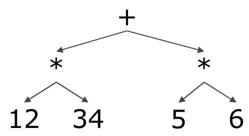
- volgens de regels van de wiskunde is maar een van deze interpretaties juist
  - Meneer van dalen wacht op antwoord
- regel dit door de grammatica aan te passen:

E → T [+ E]

T → F [* T]

F → **num**

- 12 * 34 + 5 * 6 heeft nu 1 afleiding: (12 * 34) + (5 * 6)



parseren

18

- parseren: gegeven een zin en een CFG
  - vind een afleiding voor deze zin
    maak een parse tree (een boom!)
  - of beslis dat zin niet tot de taal behoort
- twee aanpakken mogelijk
  - top-down:
    - begin met de start regel van de CFG
    - pas producties toe die nodig zijn om zin te herkennen
  - bottom up:
    - pak terminals, zoek er regels bij tot de hele zin geparseerd is
    - meeste tools werken op die manier
    - we bekijken dat verder niet

recursive descent parsers

19

- parsers volgens de regels van de CFG
  - een parse functie voor elke nonterminal in de CFG
- we gebruiken bijna altijd een aparte ***scanner*** voor het herkennen van ***tokens***
  - tokens zijn beschreven door reguliere definities en dus eenvoudig te herkennen

Left-recursion

20

Top-down parsers cannot handle left-recursion in the grammar parsed

- the parser uses the same rule recursively without consuming any input: **infinite recursion**

- Formally, a grammar is left-recursive if

  $\exists\, A \in V_n$ *such that* $A \Rightarrow^{+} A\alpha$ *for some string* α

- a simple expression grammar can be left-recursive:

  expr → expr + expr

Eliminating left-recursion

21

- To remove left-recursion, we can transform the grammar
- Consider the grammar fragment:

  foo $\rightarrow$ foo α

  | β
- where α and β do not start with foo
- We can rewrite this as:

  foo $\rightarrow$ β bar

  bar $\rightarrow$ α bar

  | ε
- where bar is a new non-terminal
- This new grammar fragment contains no left-recursion

removing empty productions

22

$S \rightarrow X\ X \mid Y$

$X \rightarrow a\ X\ b \mid \varepsilon$

$Y \rightarrow a\ Y\ b \mid Z$

$Z \rightarrow b\ Z\ a \mid \varepsilon$

- all variables are ***nullable***, e.g. $Y \Rightarrow^* \varepsilon$
- duplicate each production with an occurrence of a nullable symbol, remove the nullable symbol in the copy

$S \rightarrow X\ X \mid X \mid Y$

$X \rightarrow a\ X\ b \mid ab$

$Y \rightarrow a\ Y\ b \mid ab \mid Z$

$Z \rightarrow b\ Z\ a \mid ba$

- this can increase the size of the grammar significantly

eliminating cycles ($A \Rightarrow^+ A$)

23

- if there are no ε-productions

- replace each $A \rightarrow B$ where B is cyclic by
  $A \rightarrow \alpha$ s.t. α is not cyclic and
  $C \rightarrow \alpha$ s.t. $B \Rightarrow^* C$
- example:

  $S \rightarrow X \mid Xb \mid SS$

  $X \rightarrow S \mid a$
- we have $S \Rightarrow^+ S$ and $X \Rightarrow^+ X$

  $S \rightarrow a \mid Xb \mid SS$

  $X \rightarrow Xb \mid SS \mid a$

eliminating immediate left recursion

24

- replace

  $A ::= A\alpha_1 \mid .. \mid A\alpha_m \mid \beta_1 \mid .. \mid \beta_n$
- by

  $A \rightarrow \beta_1 A' \mid .. \mid \beta_n A'$

  $A' \rightarrow \alpha_1 A' \mid .. \mid \alpha_m A' \mid \varepsilon$

- example

  $S \rightarrow S\ X \mid S\ S\ b \mid X\ S \mid a$
- replace this by

  $S \rightarrow X\ S\ S' \mid a\ S'$

  $S' \rightarrow X\ S' \mid S\ b\ S' \mid \varepsilon$

Predictive parsing: no backtracking

25

- ***Basic idea:***
  for any two productions $A \to \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand
- for some RHS $\alpha \in G$, define FIRST(α) as the set of tokens that appear *first* in some string derived from α
  - that is, for some $w \in$ FIRST(α) iff $\alpha \Rightarrow^* w\,\gamma$
- ***Key property:***
  Whenever two productions $A \to \alpha$ and $A \to \beta$ *both* appear in the grammar, we would like
  $$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$
- This would allow the parser to make a correct choice with a look ahead of only **one** symbol!

Left factoring

26

what if a grammar does not have this property ?

- sometimes, we can *transform* a grammar to become this property
- for each non-terminal A find the longest prefix α common to two or more of its alternatives.
  if $\alpha \neq \varepsilon$ then replace all of the A productions
  $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$
  with $A \to \alpha A'$
  $A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  where A' is a new non-terminal
  - repeat this until no two alternatives for a single non-terminal have a common prefix

Example of left factoring

27

Consider a right-recursive version of the expression grammar:

1	goal	→	expr
2	expr	→	term + expr
3		|	term − expr
4		|	term
5	term	→	factor * term
6		|	factor / term
7		|	factor
8	facto	→	**num**
9		|	**id**

- To choose between productions 5, 6, & 7, the parser must see past the **num** or **id** and look at the **+**, **−**, *****, or **/**.
  FIRST(2) ∩ FIRST(3) ∩ FIRST(4) ≠ ∅
- This grammar *fails* the test.

Example of left factoring

28

There are two non-terminals that must be left factored:

2	expr	→	term + expr
3		|	term − expr
4		|	term
5	term	→	factor * term
6		|	factor / term
7		|	factor

Applying the left factoring transformation gives us:

2	expr	→	term expr'
3	expr'	→	+ expr
4		|	− expr
5		|	ε
6	term	→	factor term'
7	term'	→	* term
8		|	/ term
9		|	ε

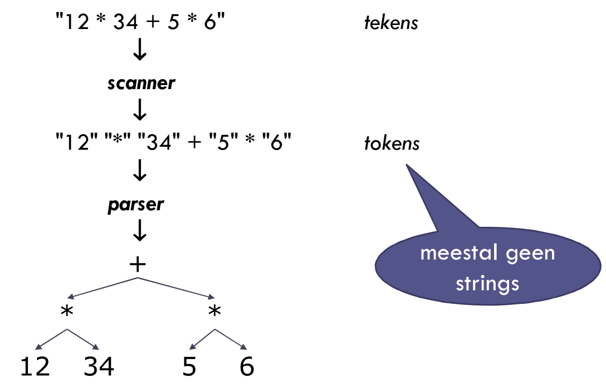
opzet van programma

29

- splits invoer herkennen in 2 delen
  - scanner maakt van rij ascii-tekens (String) een rij tokens
    - reguliere expressies als definitie van de tokens
  - parser maakt van deze tokens een parseerboom
    - contextvrije grammatica als definitie van zinnen
- we willen deterministische scanner en parser met lookahead 1
  - in de scanner we willen aan huidige teken kunnen zien welk token het moet worden
  - in de parser willen we aan het huidige token kunnen zien welke grammatica regel van toepassing is
  - dit voorkomt backtracking
    - er zijn ook parsers waar dit niet nodig is

opzet van programma 2

30



recursive descent parser in Java

31

- any parse error: throw a parse exception

```
public class ParseException extends Exception
{ /**
    * Creates a new instance of ParseException without detail message.
    */
    public ParseException ( ) { }
   /**
    * Constructs an instance of ParseException with the message.
    * @param msg the detail message.
    */
    public ParseException ( String msg )
    {
        super ( msg );
    }
}
```

scanner

32

- we will demonstrate two ways to construct a scanner

1. define a scanner class directly
   - simple and direct
   - somewhat more work
2. build the scanner class upon the Java library to recognize regular expressions
   - the Java library does much of the hard work,
   - but we must know how to use it

scanner

33

- used syntax

  E → E op E | **num**

  op → **+** | **-** | * | /

  num → -? [0-9]$^+$

- we use an enumeration type for the tokens

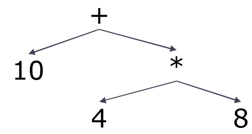
```
public enum Token
{
   Int, Plus, Minus, Times, Divide, Open, Close, EOF, Error;
}
```

  - the value of a number can be obtained separately by a method call from the scanner

parseerboom

34

- we willen een boom bouwen die de structuur van de invoer weergeeft
  - aparte knopen voor operatoren en getallen
- operatoren hebben getallen of expressies als argumenten
  - maak één type knoop
  - maak subtypes voor getallen etc.
  - geef een operator pointers naar z'n argumenten



knopen en subtypes

35

- één basis type Knoop voor alle knopen in de parse tree
  - we kunnen ieder subtype als argument gebruiken

```
abstract public class BasicNode
{
   abstract public String toString ( );
   abstract public int value ( ) throws
  Exception;
}
```

- in Clean gebruiken we iets als

:: Expr = Int Int | Operator Expr Op Expr

Integer Knoop

36

```
public class IntNode extends BasicNode
{  private int i;
   public IntNode ( int n )
   {   i = n;
   }
   public String toString ( )
   {   return Integer.toString ( i );
   }
   public int value ( )
   {    return i;
   }
}
```

operator knoop

37

```
public class OpNode extends BasicNode
{  public BasicNode l, r;
   public Token op;
   public OpNode ( ) { }
   public OpNode ( Token t )
   {   op = t;
   }
   public OpNode ( BasicNode n1, Token t, BasicNode n2 )
   {  l  = n1;
      op = t;
      r  = n2;
   }
   public String toString ( )
   {
      return ( "("+ l + " " + op + " " + r + ")" );
   }....
```

operator knoop 2

38

```
public int value ( ) throws Exception
{  int x = l.value ( );
   int y = r.value ( );
   switch ( op )
   {  case Plus:    return x + y;
      case Minus:  return x - y;
      case Times:  return x * y;
      case Divide:
         if ( y != 0 )
            return x / y;
         else
            throw new Exception( "Opnode: try to devide by zero" );
      default:
         throw new Exception( "Opnode: Unknow operator: " + op );
   }
  }
}
```

scanning

39

- the current character tells which token is currently in the input
  - no backtracking necessary in the tokenizer
- there is one exception:
  - if you do not know the context you cannot tell whether the input -123 is
    1. a single token with value -123, or
    2. two tokens: a minus operator and an integer token
  - solution: alway use the two token option and correct this in the parser if this is necessary

using the Java scanner?

40

- the Java libraries provide a handy scanner class
- this works excellent if we know what to expect
- here it is less obvious what we can expect
  - e.g. is -123 a single token, or two tokens,
  - or do we expect a number of an expression
- hence we have to use something different

input

41

- in our examples we use a String as input type
- in real parser we often use a file
- in Java we can handle both cases uniformly with a scanner

- here we use a special iterator class for characters in a string: StringCharacterIterator

input by StringCharacterIterator

42

- we use a special iterator for characters in a string
- constructor:
  StringCharacterIterator(String text)
- methods used:
  char current ( )
  char next ( )
- checking for more input not by hasNext ( ), but
  input.current ( ) == CharacterIterator.DONE

TokenizerDirect

43

```
public class TokenizerDirect
{  StringCharacterIterator input;
   private Token tok;
   private int value = -1;
   /**
    * tokens are regular expressions with [0-9]+ | '+' | '-' | '*' | '/' | ')' | '('
    * @param s the sring to parse
    * @throws parsers.ParseException
    */
   public TokenizerDirect ( String s ) throws ParseException
   {  input = new StringCharacterIterator( s );
      scan ( );
   }
   public Token currentToken ( )
   {     return tok;
   }
```

TokenizerDirect 2

44

```
public int intValue   ( ) throws ParseException
{  if ( tok == Token.Int )
      return value;
   else
      throw new ParseException( "intValue: not available for " + tok );
}
public Token nextToken ( ) throws ParseException
{  scan ( );
   return currentToken ( );
}
private int parseInt( )
{  int i = 0;
   for  ( char c = input.current ( )
        ; c != CharacterIterator.DONE && Character.isDigit ( c )
        ; c = input.next ( )
        )
      i = 10 * i +  Character.digit ( c, 10 );
   return i;
}
```

TokenizerDirect 3

45

```
private void scan ( ) throws ParseException
{  if (input.current ( ) == CharacterIterator.DONE )
       tok = Token.EOF;
   else
   {  char c = input.current ( );
      if ( Character.isWhitespace( c ) )
      {  input.next ( );
         scan ( );
      } else
         switch ( c )
         {  case '+': tok = Token.Plus;    break;
            case '-':  tok = Token.Minus;  break;
            case '*': tok = Token.Times;   break;
            case '/':  tok = Token.Divide; break;
            case '(':  tok = Token.Open;   break;
            case ')':  tok = Token.Close;  break;
            default:
               if ( Character.isDigit ( c ) )
               {  tok  = Token.Int;
                  value = parseInt ( c );
               } else throw new ParseException ( "unknown token: " + c );
}  }   }
```

using the Java library

46

- Java has a library to recognize regular expressions using this library we can easily create a tokenizer
- using this library we can specify regular expressions and obtain substrings matching this syntax in an iterator like fashion

- we will use only the basic features
- for a complete description see http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

character patterns

47

construct	matches
x	The character *x*
\\	The backslash character
\0*n*	The character with octal value 0*n* (0 <= *n* <= 7)
\0*nn*	The character with octal value 0*nn* (0 <= *n* <= 7)
\0*mnn*	The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
\x*hh*	The character with hexadecimal value 0x*hh*
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\c*x*	The control character corresponding to *x*

Character classes in patterns

48

construct	matches
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Predefined character classes in patterns

49

construct	matches
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

quantifiers in patterns

50

construct	matches
X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

there are many more quantifiers giving fine control over matching

our regular expression

51

- we have
  [0-9]+ | '+' | '-' | '*' | '/' | ')' | '('
- as pattern in Java this is
  \d+ | \+ | \- | * | \/ | \) | \(
  - since many of our special characters are special characters we need an \ to escape them
  - since \ is special in Java String we need to escape the escape character:
    "\\d+|\\+|\\-|*|\\/|\\)|\\(|"
  - we add a match any token to produce error messages

the tokenizer

52

```
public class Tokenizer
{  private Pattern pattern;
   private Matcher matcher;
   private Token tok;
   private int value = -1;
   /**
    * Constructor. Tokens are regular expressions with
    * [0-9]+ | '+' | '-' | '*' | '/' | ')' | '('
    * @param s the sring to parse
    * @throws parsers.ParseException
    */
   public Tokenizer (String s) throws ParseException
   {  pattern
         = Pattern.compile( "\\d+|\\+|\\-|\\*|\\/|\\)|\\(|\\S" );
      matcher = pattern.matcher( s );
      scan ( );
   }
```

the tokenizer 2 (nothing different)

53

```
public Token currentToken ( )
{   return tok;
}
public int intValue ( ) throws ParseException
{  if ( tok == Token.Int )
        return value;
    else
        throw new ParseException
                    ("intValue: not available for " + tok);
}
public Token nextToken ( ) throws ParseException
{  scan ( );
    return currentToken ( );
}
```

the tokenizer 3: transform strings to tokens

54

ugly layout to put it all on one slide

```
private void scan ( ) throws ParseException
  {  if (matcher.find ( ))
     {   String token = matcher.group ( );
         if ( token.equals("+") )        tok = Token.Plus;
         else if ( token.equals("-") )   tok = Token.Minus;
         else if ( token.equals("*") )   tok = Token.Times;
         else if ( token.equals("/") )   tok = Token.Divide;
         else if ( token.equals("(") )   tok = Token.Open;
         else if ( token.equals(")") )   tok = Token.Close;
         else if ( Character.isDigit( token.charAt( 0 )))
         {   tok   = Token.Int;
             value = Integer.parseInt(token);
         } else throw new
                 ParseException("unknown token: " + token);
     } else tok = Token.EOF;
  }
```

the tokenizer 3 better layout

55

```
private void scan ( ) throws ParseException
  {  if (matcher.find ( ))
     {  String token = matcher.group ( );
        if (token.equals( "+" ))
          tok = Token.Plus;
        else if (token.equals( "-" ))
          tok = Token.Minus;
        else if (token.equals( "*" ))
          tok = Token.Times;
        else if (token.equals( "/" ))
          tok = Token.Divide;
        else if (token.equals( "(" ))
          tok = Token.Open;
        else if (token.equals( ")" ))
          tok = Token.Close;
        else if ( Character.isDigit( token.charAt ( 0 )))
        {   tok = Token.Int;
            value = Integer.parseInt( token );
        } else throw new ParseException( "unknown token: " + token );
     } else tok = Token.EOF;
  }
```

recursive descent parser

56

works also with other tokenizer

```
public class RDparser
{  private Tokenizer tok;
   public RDparser ( String s ) throws ParseException
   {   tok = new Tokenizer( s );
   }
   /**
    * starts parsing: E EOF
    * @return the parse tree if the whole input can be consumed
    * @throws parsers.ParseException
    */
   public BasicNode parse ( ) throws ParseException
   {  BasicNode n = parseExpr ( );
      if ( tok.currentToken ( ) == Token.EOF )
         return n;
      else
         throw new ParseException( "end of input expected" );
   } ..
```

parsing expressions

57

```
/**
 * Expr = Term (('+'|'-') Expr)?
 * @return parse tree
 * @throws parsers.ParseException
 */
public BasicNode parseExpr ( ) throws ParseException
{  BasicNode l = parseTerm ( );
   Token token = tok.currentToken ( );
   switch ( token)
   {  case Plus:
      case Minus:
         tok.nextToken ( );
         BasicNode r = parseExpr ( );
         return new OpNode( l, token, r );
      default: return l;
   }
}
```

consume operator

parsing terms

58

```
/**
 * Term = Factor (('*'|'/') Term)?
 * @return parse tree
 * @throws parsers.ParseException
 */
public BasicNode parseTerm ( ) throws ParseException
{  BasicNode l = parseFactor ( );
   Token token = tok.currentToken ( );
   switch (token)
   {  case Times:
      case Divide:
         tok.nextToken ( );
         BasicNode r = parseTerm ( );
         return new OpNode( l, token, r );
      default: return l;
   }
}
```

parsing factor

59

```
/** Factor = ['-'] Nat | '(' Exp ')'
 * @return parse tree
 * @throws parsers.ParseException
 */
public BasicNode parseFactor ( ) throws ParseException
{    switch ( tok.currentToken ( ) )
   { case Int:
         IntNode node = new IntNode( tok.intValue ( ) );
         tok.nextToken ( );
         return node;
      case Minus:
         if ( tok.nextToken ( ) == Token.Int )
         {  IntNode node = new IntNode( tok.intValue ( ) * -1 );
            tok.nextToken ( );
            return node;
         }
         else throw new ParseException ( "number expected instead of " +
                                         tok.currentToken ( ) );
      case Open: ...
```

parsing factor 2

60

```
      case Open:
         tok.nextToken ( );
         BasicNode tree = parseExpr ( );
         if ( tok.currentToken ( ) == Token.Close )
         {
            tok.nextToken ( );
            return tree;
         }
         else
            throw new ParseException ( ") expected" );
      default:
         throw new ParseException( "factor expected" ));
   }
}
```

using these parsers

61

```
public class Main
{  public static void main(String[] args)    { new Main  ( )  }
   public Main  ( )
   {  Scanner in = new Scanner (System.in);
      while ( true )
      {   try
         {  System.out.print( "Enter input string to parse: " );
            String regel = in.nextLine ( );
            if ( regel.length ( ) == 0 ) break;
            TRparser parser = new TRparser( regel );
            BasicNode node  = parser.parseExpr ( );
            System.out.print( "parse result: " + node );
            System.out.println( " with value " + node.value  ( ));
         }  catch ( ParseException pe )
         {   System.out.println( "Parse error: " + pe.getMessage  ( ) );
         }   catch ( Exception e )
         {    System.out.println( "Error: " + e.getMessage  ( ) );
} }  }
```

building parse trees: associativity

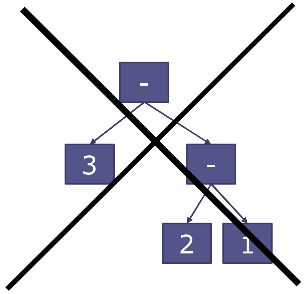
62

- for operators like + and * we can put the parenthesis in any way we want
  - e.g. 1 + 2 + 3 = (1 + 2) + 3 = 1 + (2 + 3)
- for other operators this does not hold e.g.
  - (3 - 2) - 1 = 1 - 1 = 0
  - 3 - (2 - 1) = 3 - 1 = 2
  - (3 - 2) - 1 ≠ 3 - (2 - 1)
  - default way of parsing: 3 - 2 - 1 = (3 - 2) - 1
  - similar 8 / 2 / 2 should have value 2 (instead of 8)

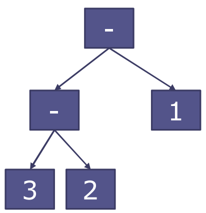
associativity: parse trees

63

- parse trees for 3 - 2 - 1:
- 3 - 2 - 1 = (3 - 2) - 1 = 0



value: 2



value: 0

associativity

64

- by changing
  $E \to E - T \mid T$
  to (in order to eliminate left recursion)
  $E \to T - E \mid T$
  we also introduce right associativity for a RD-parser
- simply applying rotates does not solve the problem
- solution: use
  $E \to T\ (- T)^*$
  - as soon as we recognized $E \to T_1 - T_2\ (- T)^*$ we replace this by $E \to T_3\ (- T)^*$ where $T_3 = T_1 - T_2$

a tail recursive parser

tail recursive parser

65

```
/**
 * Expr = Term (('+'|'-') Term)*
 * @return parse tree
 * @throws parsers.ParseException
 */
public BasicNode parseExpr ( ) throws ParseException
{  BasicNode t = parseTerm ( );
   Token token = tok.currentToken ( );
   while
     (isElement( new Token [] {Token.Plus, Token.Minus}, token))
   {  tok.nextToken ( );
      BasicNode r = parseTerm ( );
      t = new OpNode( t, token, r );
      token = tok.currentToken ( );
   }
   return t;
}
```

similar for other grammar rules

haakjes

66

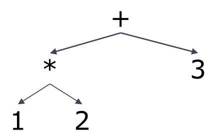
- zijn zeker van belang, maar geen knoop in de boom

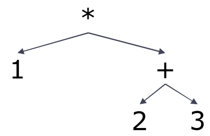
E $\rightarrow$ T (+ T)*

T $\rightarrow$ F (* F)*

F $\rightarrow$ **num** | (E)

- voorbeeldjes
- 1 * 2 + 3 en 1 * (2 + 3)





conclusie recursive descent parser

67

- het geheim is het maken van een geschikte grammatica
  - indien we verder dan 1 teken vooruit moeten kijken hebben we soms backtracking nodig
    - met de juiste syntax en grammatica hoeft dat niet
  - einde van de zin staat meestal niet in grammatica, moet je dus iets extra's voor verzinnen
- aanpak: splits in deelproblemen
  - invoer, scanner, één parser per syntax regel
- hier globale objecten voor invoer en scanner
  - kun je natuurlijk ook doorgeven, is wat onhandig
  - de parsers gebruiken alleen de scanner

wat hebben we gedaan

68

- grammatica's
  - maak grammatica zodat je handig kunt parseren
    - geen linksrecursie
    - prioriteit van operatoren in grammatica
    - look ahead 1 d.m.v. left factoring
- parseren
  - recursive descent
    - handig als je met de hand een simpele parser maakt
    - volg de regeltjes van de grammatica direct
  - bindingsrichting operatoren
    - simpele verplaatsing van de recursie naar rechts maakt operatoren rechts associatief
    - een tail recursive parser lost dit probleem netjes op