Parsering og skanning i Java med Antlr

Sven-Jørgen Karlsen,
<svenjok@ifi.uio.no>
gruppelærer i INF5110 våren 2006

Agenda

- Antlr
  - Generelt
    - Syntaks – terminaler
    - Syntaks - inputfil
  - Parsering og skanning
    - Enkelt eksempel
    - Ambisiøs grammatikk med
      - aksjoner
      - Antlrs automatiske bygging av AST
  - Avanserte konfliktløsning med LL(k)
  - Feilhåndtering og feilsøking
- Administrivia
- Spørsmål?

Min bakgrunn

- Kort om min bakgrunn:
  - 7 års erfaring som utvikler/programmerer, i Tansa Systems og Ergo Group. C, C++ og Java.
  - Drøyt halvveis i masterstudiet og skriver oppgave om modelleringsverktøy for SINTEF.
    - Veileder er Arne-Jørgen Berre.
- Tok NF5110 våren 2005

Administrivia

- Obligen er lagt ut på:
  http://www.uio.no/studier/emner/matnat/ifi/INF5110/v06/oblig/oblig1-2006.html
- Grupper på inntil tre personer
  - > 1 anbefales p.g.a. arbeidsmengden (~= 4000 LOC i fjor)
- Frister:
  - Oblig 1 er 30. 3.
  - Oblig 2 blir ca. 3.5
- Veiledning per e-post eller avtale
  Når? Bortsett fra ett par faste avtaler på mandager, og en forelesning på onsdager, er jeg ganske fleksibel gjennom uka.

Bakgrunnen for verktøyvalg i obligen

- Å prøve ut mer moderne verktøy enn `yacc & lex` for kompilatorteknikk, herunder særlig
  - Støtte for moderne, objektorientert programmeringsspråk (yacc & lex er veldig bundet til C)
- Vi har sett på Byacc/J, CUP, JavaCC og Antlr.
  SableCC og Mork ble nedprioritert p.g.a. snever utbredelse, og «kodetaushet» (stillstand i utviklingen).
- Antlr ble valgt p.g.a. modenhet[1], dokumentasjon[1], utbredelse og popularitet.
- Hovedgevinsten med ett av disse Java-baserte verktøyene er å kunne bruke ett mer egnet språk til å implementere aksjoner og semantisk sjekking.

[1]: Ved første øyekast, så disse kriteriene ganske sterke ut, men etter hvert har jeg kommet noe i tvil.

Hva er Antlr?

"Why program by hand in five days what you can spend five years of your life automating."
Terrence Parr, hovedmannen bak Antlr

- «ANother Tool for Language Recognition»
- Verktøy for å generere skannere og parsere (tilsvarende `yacc` og `lex`, men top-down)
  - I flere språk: Java, C++, C# og Python
- Populært: >= 5000 nedlastinger per mnd.
- Lisens: Public Domain / BSD
- Historie:
  - Terrence Parr, prof. ved University of San Francisco, startet i 1989, under navnet PCCTS. Motivasjon: å automatisere håndskrevne recursive-descent skannere og parsere.
  - 1991: Forskning på LL(k > 1) og backtracking gir sterkere teoretisk basis.
  - 2.7.6 utgitt des. 2005
  - 3.0 under arbeid.

Hvordan lage en parser med Antlr?
Utkast til en arbeidsprosess

1) Parsergrammatikk i Antlr-språket: Beskriv språket du ønsker å gjenkjenne i Antlrs EBNF-dialekt.
2) Skannergrammatikk: Beskriv symbolene som skanneren skal gjenkjenne i Antlrs skanner-dialekt.
3) Kjør Antlr på fil(-ene)
4) Gjenta steg 1-3 til alle konflikter i grammatikkene er løst.
5) Kompiler koden som Antlr har generert.
6) Test parseren og skanneren.
7) Legg på aksjonskode eller trebyggingsannotasjoner. Gjenta steg 3-7 til parseren gjør noe fornuftig med gjenkjent inputs.

Antlrs inputsyntaks (metaspråk) (4)

Symbol	Scanner-specific?	Description	Example
Whitespace		separators, ignored, as in most prog. languages	`program : (decl)* ;`

Comments		C- og C++-style, Javadoc on rules and classes	`/** This grammar recognizes simple expressions`
			` * @author Terence Parr`
			` */`
			` class ExprParser;`
			` /** Match a factor */`
			` factor : ... ;`

Characters	Y	As in Java	`OTHER : ('a' .. 'z')+;`
			`C : 'c';`
			`OCTAL_CHAR : '\377';`
			`HEX_UNICODE_CHAR: '\uFF00';`

EOF		Automatically available token, can be referred to in the parser.	`program : (decl)* EOF;`

Strings	*Note*: different interpretation	character seqs. in double quotes.
		In the scanner: short notation for appended characters
		In the parser: refs. to tokens.
			`// Grammar rule from an XML example.`
			`attr : NAME "=" "\"" NAME "\""`
			`// Lexer rule from the same example:`
			`OTHER : ({LA(2) == '/'}? "</"`
			`| '<'| '>' | '"' | '=')`
			`;`

Token reference	No, *but* treated as a *rule ref.*	References to tokens, starts with an uppercase letter.	`attr : NAME "=" "\"" NAME "\""`
			`// Be wary of typos like`
			`rule : Decl ...`
			`// instead of`
			`rule : decl ...`

Token defn.	Y	Defn. of a token in the lexer. Same syntax as parser rules.	`NAME: (('a' .. 'z') | ('A' .. 'Z'))+`
Rule reference	NA	Ref. to a grammar rule. Starts with a lowercase letter. *Note*: the typo problem mentioned for token refs.	`doc : elt EOF;`
`( . . . )`		subrule	`program : (decl)* ;`
`( . . . ) *`		closure subrule zero-or-more	`program : (decl)* ;`

Antlrs inputsyntaks (metaspråk) (5)

Symbol	Scanner-specific?	Description	Example
`(...)+`		positive closure subrule one-or-more	`program : (decl)+ ;`
`(...)?`		optional zero-or-one	`if "(" e ")" then s (else s)?`
`{...}`		semantic action	`decl : type ID { System.out.println("Found " + "a decl."); } ;`
`[...]`		rule arguments	`addOpRhs[Exp lhs] returns [Exp e] : ...;`
`{...}?`		semantic predicate	`abOrAc : {LA(2) == LITERAL_b }? "a" "b" | "a" "c" ;`
`(...)=>`		syntactic predicate	`abOrAc : ("a" "b") => "a" "b" | "a" "c" ;`
`|`		alternative operator	`abOrAc : "a" "b" | "a" "c" ;`
`..`	Y	range operator	`WORD : ('a' .. 'z')+`
`~`	Y	not operator	`SL_COMMENT : "//" (~'\n')* '\n';`
`.`	Y	wildcard	`CURLY_BLOCK_SCARF: '{' (.)* '}' ;`
`=`		assignment operator	`op returns [BinaryExp.Operator o] { o = null; } : "+" { o = Exp.Operator.PLUS; } | "-" { o = Exp.Operator.MINUS; } ;`
`<...>`		element option	
`class`		grammar class	`class AmbigiousExpParser extends Parser;`
`extends`		specifies grammar base class	`class Sub extends Super;`

Antlrs inputsyntaks (metaspråk) (6)

Symbol	Scanner-specific?	Description	Example
returns		specifies return type of rule	`op returns [BinaryExp.Operator o]` `  : ... ;`

options		options section	`options {`
			`  buildAST = true;`
			`}`

tokens		tokens section	`tokens {`
			`  NUMBER<AST=NumberExp>;`
			`  ...`
			`}`

header		header section	`header {` `  import antlr.collections.AST; }`

Antlrs inputsyntaks (metaspråk) strukturen til inputfiler

- En inputfil, `grammar.g`, kan bestå av følgende:

  NB: Vær obs. på at Antlrs Java-liknende begreper != Java, før koden er generert.

  - **Header-seksjon**: Java import statements.
  - **Parserklassedefinisjon**, høyst én per fil, som
    - kan ha:
      - Opsjons-seksjon: innstillinger, for bl.a. lookahead-dybde, default feilhåndtering og vokabular.
      - Token-seksjon: For avanserte token-deklarasjoner ved multiple vokabularer, og for å deklarere imaginære tokens, spesifikasjon av AST-typer (Antlrs trebygging).
      - Action-blokk (på klassenivå, i motsetning til regel-aksjoner): hvor det kan være praktisk å plassere en main-metode o.a.
    - Må ha (minst én)
      - Parser-regler: CFG i Antlrs EBNF-dialekt.
  - **Skanner-klassedefinisjoner**: Samme struktur som parser-definisjonene, men reglene er i kategorien «lexer rules».
  - **Treparsere**: Samme struktur, men reglene er «tree parser rules»

Anatomien til en .g-fil

```
header {
    import antlr.collections.AST;
}

/** Example of an ambigious expression grammar.*/
class AmbigiousExpParser extends Parser;

options {
    buildAST = true;
}
tokens {
    NUMBER<AST=NumberExp>;
    ...
{
    public static void main(String args[]) throws Exception {...}
}

exp     : term ((PLUS^ | MINUS^) term)* EOF! ;
        ...

class AmbigiousExpWalker extends TreeParser;

exp returns[int r]
{
    int a, b;
    r = 0;
}
        : #(PLUS a=exp b=exp)  { r = a + b; }
        | ... ;
        | i:NUMBER             { r = Integer.parseInt(i.getText()); }
        ;

/** Scanner. */
class AmbigiousExpLexer extends Lexer;

WS      : (' ' | ...)
        ;
NUMBER  : ('0' .. '9')+ ;
OTHER   : ('a' .. 'z')+
    ;
```

Header section

Parser-klasse

Opsjons-seksjon

Token-seksjon

Aksjonsblokk (klassenivå)

Regel

Treparser

Skannerklasse

Enkelt eksempel (1)

Arne Maus' uttrykksgrammatikk i Antlr:

exp → exp op number | number
op → + | -

Problemer:

- Grammatikken er venstrerekursiv

Antlr forteller oss også dette, med følgende regel i inputgrammatikken:

```
exp : exp ("+" | "-") NUMBER | NUMBER ;
```

```
$ java -cp build/classes\;/usr/local/lib/antlr.jar antlr.Tool -o build/src\
        ../../antlr-presentasjon/SimpleExpAmbigious.g
ANTLR Parser Generator   Version 2.7.6 (20060206)   1989-2005
../../antlr-presentasjon/SimpleExpAmbigious.g:11:7: infinite recursion to rule exp from rule exp
../../antlr-presentasjon/SimpleExpAmbigious.g:11:7: infinite recursion to rule exp from rule exp
../../antlr-presentasjon/SimpleExpAmbigious.g:11: warning:nondeterminism between
 alts 1 and 2 of block upon
../../antlr-presentasjon/SimpleExpAmbigious.g:11:     k==1:NUMBER
Exiting due to errors.
```

- Blir venstreassosiattiviteten til operatorene ivaretatt?

Steg 1: Formuler i Antlr-format:

```
exp : exp op NUMBER | NUMBER;
op  : «+» | «-»;
```

Steg 1.a: Skriv om til høyrerekursjon:

```
exp  : NUMBER («+» | «-» NUMBER)*;
```

Enkelt eksempel (2)

Skanner for språket

Vi må definer terminalene som en Antlr-skanner skal kjenne igjen.

Hvilke symboler? NUMBER, '+', '-' (og $/EOF, men den får vi automatisk). I tillegg kan den godt hoppe over blanke tegn.

Antlr-skanner, Lexer-klasse:

```
/** Scanner. */
class ExpLexer extends Lexer;

WS      : (' '
          | '\t'
          | ('\n') { newline(); })+
        {$setType(Token.SKIP);}
    ;
PLUS    : "+";
MINUS   :  "-";
NUMBER  : ('0' .. '9')+
        ;
```

Enkelt eksempel (3.a)

Generering og kompilering

1. Generering av skanner og parser:

```
$ java -cp build/classes\;/usr/local/lib/antlr.jar  antlr.Tool \
      -o build/src src/SimpleExp.g
ANTLR Parser Generator   Version 2.7.6 (20060206)   1989-2005
```

2. Kompilering

```
$ javac -cp /usr/local/lib/antlr.jar -sourcepath build/src \
       -d build/classes build/src/SimpleExp*.java
Note: build/src/SimpleExpLexer.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

1 + 2 = Automatisering:

Disse to stegene har jeg automatisert i Ant-byggeskript, `build.xml`, som legges ut sammen eksempelkode på kurssidene.

Byggingen blir da bare (*NB*: lokasjonen til `antlr.jar` må settes opp i byggeskriptet)

```
$ ant
```

Som bygger målet compile, som avhenger av og derfor også kjører antlr, via en ant-task (som følger med std.distribusjonen til Ant (1.6.5))

Enkelt eksempel (3.b)

Kjøre Antlr på grammatikkene

3. Testing:

- Suksess-scenario:

```
$ echo 1 + 1 -2 | java -cp build/classes\;/usr/local/lib/antlr.jar \
                        SimpleExpParser;
 (=> Ingen output (no news is good news)).
```

- Skannerfeil:

```
$ echo 1 + 1 - foobar | java -cp build/classes\;/usr/local/lib/antlr.jar  SimpleExpParser
Exception in thread "main" line 1:9: unexpected char: 'f'
    at SimpleExpLexer.nextToken(SimpleExpLexer.java:97)
    at antlr.TokenBuffer.fill(TokenBuffer.java:69)
    at antlr.TokenBuffer.LA(TokenBuffer.java:80)
    at antlr.LLkParser.LA(LLkParser.java:52)
    at antlr.Parser.match(Parser.java:210)
    at SimpleExpParser.exp(SimpleExpParser.java:78)
    at SimpleExpParser.main(SimpleExpParser.java:24)
```

- Parsefeil:

```
$ echo 1 1  | java -cp build/classes\;/usr/local/lib/antlr.jar \
                    SimpleExpParser
```

- `line 1:3: expecting EOF, found '1'`

Enkelt eksempel (4)

Utvidelse for å skrive ut syntakstreet

Vi kan bruke Antlrs innebygde støtte for AST-bygging for lettvint utskrift.

1. SimpleExp.g:

```
class SimpleExpParser extends Parser;

{
    public static void main(String args[]) throws Exception {
        final SimpleExpParser parser =
            new SimpleExpParser(new SimpleExpLexer(System.in));
        parser.exp();
    }
}

exp : NUMBER ((("+" | "-") NUMBER))*
      EOF!
    ;
```

Enkelt eksempel (5)

2. `SimpleExpWithASTParser`

```
class SimpleExpWithASTParser extends Parser;

options {
    importVocab = SimpleExpLexer;
    buildAST = true;
}

{
    public static void main(String args[]) throws Exception {
        final SimpleExpWithASTParser parser =
            new SimpleExpWithASTParser(new SimpleExpLexer(System.in));
        parser.exp();
        final AST t = parser.getAST();
        System.out.println(t.toStringTree());
    }
}

exp : NUMBER^ ((("+"^ | "-"^) NUMBER))*
            EOF!
    ;
```

Kjøreeksempel:

```
$ echo 1 + 1 -  2 -3 +4  | java -cp build/classes\;/usr/local/lib/antlr.jar  SimpleExpWithASTParser
 ( + ( - ( - ( + 1 1 ) 2 ) 3 ) 4 )
$
```

Resultatet av kodegenerering (1)

- Hva blir generert?
  - `SimpleExpParser.java:` Parser-implementasjonen
  - `SimpleExpLexer.java:` Scanner-implementasjonen
  - `SimpleExpLexerTokenTypes.java:` Java-grensesnitt med tallkonstanter for terminalene, Fra skanneren, i dette tilfellet. Mer komplisert utveksling av token-vokabular fins, se opsjonen importVocab og exportVocab.
  - `SimpleExpLexerTokenTypes.txt`

```
/ $ANTLR 2.7.6 (20060206): "SimpleExp.g" -> "SimpleExpParser.java"$

 import antlr.TokenBuffer;
...
/** Parser. */
public class SimpleExpParser extends antlr.LLkParserimplements SimpleExpLexerTokenTypes
{

    public static void main(String args[]) throws Exception {
        final SimpleExpParser parser =
            new SimpleExpParser(new SimpleExpLexer(System.in));
        parser.exp();
    }
    ...
    public SimpleExpParser(TokenStream lexer) {
        this(lexer,1);
    }
    ...
```

Resultatet av kodegenerering (2)

```
public final void exp() throws RecognitionException, TokenStreamException {
    try {      // for error handling
        match(NUMBER);
        {
            _loop14:
            do {
                if ((LA(1)==8||LA(1)==9)) {
                    {
                        {
                            switch ( LA(1)) {
                            case 8:
                                {
                                    match(8);
                                    break;
                                }
                            case 9:
                                {
                                    match(9);
                                    break;
                                }
                            default:
                                {
                                    throw new NoViableAltException(LT(1), getFilename());
                                }
                            }
                        }
                        match(NUMBER);
                    }
                }
                else {
                    break _loop14;
                }

            } while (true);
        }
        match(Token.EOF_TYPE);
    }
    catch (RecognitionException ex) {
        reportError(ex);
        recover(ex,_tokenSet_0);
    }
  }
}
```

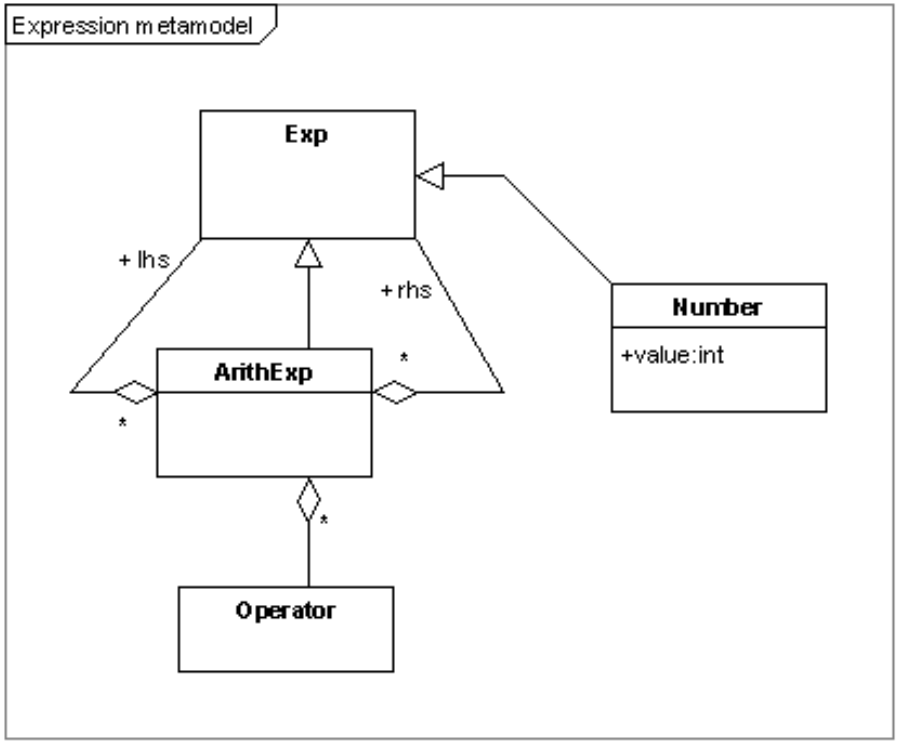
Anonyme konstanter for «+» og «-».
Hadde disse vært navngitt i parsergrammatikken, hadde det stått f.eks. PLUS og MINUS her i stedet.

Default feilhåndtering.
Kan skrus av med opsjonen defaultErrorHandler, på Parser-klassen

Ambisiøs grammatikk med trebygging

- Parser for språket:

```
exp     -> exp (+ | -) |
           exp (* | /) |
           NUMBER
```



Created with Poseidon for UML Community Edition. Not for Commercial Use.

- **Presedens**:
  + and - har lavere presedens enn * og /.
- **Assosiativitet**:
  +, -, * og / er venstreassosiative.

Ambisiøs grammatikk med trebygging (2)

«The hard way»

- Omskrevet grammatikk:

```
exp : NUMBER (op NUMBER)* EOF
```

  - **NB:** Denne grammatikken har minst ett problem i forhold til spek. på foregående side!

- Antlr-grammatikk med aksjoner som bygger AST:

```
// Manual tree construction:
exp returns[Exp e]
{
    BinaryExp.Operator op;
    e = null;
}
        : lhs:NUMBER {e = new NumberExp(Integer.valueOf(lhs.getText())); }
          ((op=op rhs:NUMBER)
           { e=new BinaryExp(op, e,
                             new NumberExp(Integer.valueOf(rhs.getText()))); }
            )*
          EOF!
    ;
op returns [BinaryExp.Operator o]
{ o = null; }
        : "+" {o = Exp.Operator.PLUS; }
        | "*" {o = Exp.Operator.MULT; }
        ;
```

Label

Aksjon, i Java

Tilordning

Ambisiøs grammatikk med trebygging (3.a)
Syntakstreklassene (1)

- `Exp.java`
  - TBD: Operatortypen kan godt plasseres i en egen fil.

```
public interface Exp {

    public enum Operator {
        PLUS("+"), MINUS("-"), MULT("*"), DIV("/");

        private final String strValue;

        Operator(String value) {
            this.strValue = value;
        }

        public String toString() {
            return strValue;
        }

        public static Operator fromString(String strValue) {
            for (Operator op : Operator.values()) {
                if (op.strValue.equals(strValue)) {
                    return op;
                }
            }
            return null;
        }
    }

}
```

Ambisiøs grammatikk med trebygging (3.b)
Syntakstreklassene (2)

- NumberExp.java

```
public final class NumberExp extends BaseASTAdapter implements Exp  {

    private int value;

    public NumberExp(int value) {
        super();
        this.value = value;
    }

    public NumberExp(Token token) {
        super();
        this.value = Integer.parseInt(token.getText());;
    }

    public String toString() {
        return "(" + value + ")";
    }
    // Cumbersome Antlr callback to have this class usable in a tree
    // grammar.

    /** Get the token text for this node */
    public String getText() {
        return toString();
    }

    /**
     * Get the token type for this node
     */
    public int getType() {
        return AmbigiousExpParserTokenTypes.NUMBER;
    }
}
```

Antle-spesifikke tillegg for å bruke klassen til automatisk bygging av heterogent AST.

Ambisiøs grammatikk med trebygging (3.c)
Syntakstreklassene (3)

- `BinaryExp.java`

```
public final class BinaryExp extends BaseASTAdapter implements Exp  {

    private final Operator op;
    // These should actually be final, but AST construction with Antlr seems
    // to happen before the children are parsed.
    private Exp lhs;
    private Exp rhs;

    public BinaryExp(Operator op, Exp lhs, Exp rhs) {
        super();
        this.op = op;
        this.lhs = lhs;
        this.rhs = rhs;
    }

    /**
     * Special ctor version for use by Antlr's tree construction.
     */
    public BinaryExp(Token op) {
        super();
        this.op = Exp.Operator.fromString(op.getText());
    }

    public String toString() {
        return "(" + op + " " + lhs + " " + rhs + ")";
    }

    // Cumbersome Antlr callback to have this class usable in a tree
    // grammar.
    ...
```

Ambisiøs grammatikk med trebygging (4)
Automatisk trebygging (1)

```
class AmbigiousExpASTParser extends Parser;

options {
    buildAST = true;
}

tokens {
    NUMBER<AST=NumberExp>;
    PLUS<AST=BinaryExp>;
    MINUS<AST=BinaryExp>;
    MULT<AST=BinaryExp>;
    DIV<AST=BinaryExp>;
}

{
    public static void main(String args[]) throws Exception {
        final AmbigiousExpASTParser parser =
            new AmbigiousExpASTParser(new AmbigiousExpLexer(System.in));
        parser.exp();
        final AST t = parser.getAST();
        System.out.println(t.toStringList());
        AmbigiousExpWalker w = new AmbigiousExpWalker();
        System.out.println("=> " + w.exp(t));
    }
}
```

Ambisiøs grammatikk med trebygging (5)
Automatisk trebygging (2)

- Grammatikk med nodebyggingsoperatorer, og en treparser, som regner ut verdien av treet:

```
exp     : term ((PLUS^ | MINUS^) term)*
        EOF!
        ;
term    : factor ((MULT^ | DIV^) factor)*
        ;
factor  : NUMBER
        ;


class AmbigiousExpWalker extends TreeParser;

exp returns[int r]
{
    int a, b;
    r = 0;
}
        : #(PLUS a=exp b=exp)  { r = a + b; }
        | #(MINUS a=exp b=exp) { r = a - b; }
        | #(MULT a=exp b=exp)  { r = a * b; }
        | #(DIV a=exp b=exp)   { r = a / b; }
        | i:NUMBER             { r = Integer.parseInt(i.getText()); }
        ;
```

Automatisk bygging av AST-er?
Førsteinntrykk

- Pro
  - Besnærende enkel måte å besvare store deler av oblig 1
  - Antlr-grammatikken blir mye ryddigere enn ved tradisjonelle aksjonsblokker strødd utover.
- Kontra:
  - Læringskurve: Ny syntaks og abstraksjoner bak.
  - Tuklete å integrere heteregene nodetyper.
    F.eks. ved gjenbruk av egne AST-impl. fra andre parsere og verktøy.
    - Må implementere grensesnittet `AST`
    - `Tokens` i nye konstruktører, og minst to nye entry-points for tregrammatikker (spesifikasjon av traversering).
    - Hvor kan ikke typeinformasjonen til nodene brukes i stedet?
  - Rammeverket tvinger deg til å beholde tokens i AST-et (nødvendig for at tregrammatikker skal funke).
    => Merkelig blanding av konkret og abstrakt syntaks.

LL(k)
Lookahead > 1

- Hva når man trenger mer lookahead enn ett symbol?
  Generelt, fire løsninger:

  1) Venstrefaktorisering

  2) Økt konstant lookahead
     Opsjonen k = n, (øker lookahead «globalt») på Parser eller Lexer-klassen.
     Eks: options { k = 2; }

  3) Semantiske predikater (oftest «disambiguating» / eksplisitt syntaktisk lookahead): Run-time betingelse som må møtes for å velge alternativ.
     Syntaks: `{ expression }?` (uttr. er stort sett plain Java, som i aksjoner).

  4) Syntaktiske predikater: Parseren prøver ut en «forutsigelse», og backtracker via exceptions (i Java) o.l. hvis den ikke matcher.
     Syntaks: (prediction block) => production

- De tre første kan brukes på problemer når k er kjent, mens den siste også kan velge ved vilkårlig lookahead (grammatikker som ikke er LL(k) for noen k).

LL(k)
LL(2)-eksempel

Følgende LL(2)-grammatikk:

```
g       : abOrAc EOF ;
abOrAc  : ("a" "b"
        | "a" "c")
        EOF
        ;
```

kan håndteres ved

- Venstrefaktorisering:

```
// Solution 2: Left factorization.
g   : "a" ("b" | "c") EOF
    ;
```

- Økt lookahead

```
class LL2Parser extends Parser;
options {
    k = 2;
}
```

- Semantisk predikat:

```
// Solution 3: Semantic predicate
// (explicit syntactic predicate):
abOrAc : {LA(2) == LITERAL_b }? "a" "b"
       | "a" "c"
       ;
```

- Syntaktisk predikat:

```
// Solution 4: Syntactic predicate on an
//             LL(2)-production.
// abOrAc  : ("a" "b") => "a" "b"
//         | "a" "c"
//         ;
```

LL(k)
Non-LL(k)-eksempel

- Variant av ifelse-problemet (hvor »:» er i både First- og Follow-mengden til else-greina som er utnullbar. Problem:
  EBNF:

```
qcAlt       : condition "?" stmt (":" stmt)?
```

  BNF:

```
qcAlt       : condition "?" stmt elseBranch ;
elsebranch  : ":" stmt | // Empty (epsilon) ;
```

```
exp     : qcAlt EOF
        ;
// Another example, variant of the question-colon operator with the
// dangling-else problem. The problem is that the ":" (aka "else") is both
// in the first set of (":" stmt) and its follow set.
// Solutions:
// 1. Distuingish the two branches by a syntactic predicate.
// 2. Checking if the generated parser choses the preferred alternative (most
//    closely nested rule), and in that case, turn off the ambiguity warning.
//    To match the first alternative is the default in Antlr, and the warning
//    can be turned off by
//      (options {greedy=true;} : ":" stmt)*

condition   : "true"
            | "false"
            ;
qcAlt       : condition "?" stmt ((":" stmt) => ":" stmt)?
            ;
stmt        : qcAlt
            ;
```

Feilhåndtering i parseren

- **Linjetelling** i skanneren:
  Standard feilhåndtering, med utskrift av feilmeldinger og lokasjon er ganske grei, hvis man også sørger for å telle linjer i skanneren. Dette ordnes av `CharScanner.newline()` (baseklasse til den genererte skanneren):

```
class SimpleExpLexer extends Lexer;

WS      : (' '
          | '\t'
          | ('\n') { newline(); })+
          {$setType(Token.SKIP);}
      ;
```

- **Holde rede på parsingstatus** (dette *må* gjøres i obligen):
  Antlr holder ikke styr på status over feilforekomstene, og man vet fremdeles *ikke* om parsinga var vellykket eller ikke!
  En løsning er å overkjøre Antlrs rammeverkmetode for feilhåndtering, noe á la følgende, i aksjonsblokka til Parser-klassen:

```
private boolean parseFailure = false;

/**
 * Notes whether parsing failures has occured, after Antlr's
 * default error reporting.
 */
public void reportError(RecognitionException ex) {
    super.reportError(ex);
    parseFailure = true;
}
```

- **Mer avansert:** Skru av `defaultErrorHandler` og legg på blokker med unntakshåndtering.

Feilsøking og debugging

- 1. bud: Klar (for deg sjøl) og entydig (for Antlr) grammatikk.

  Min erfaring er at når grammatikken er:

  1) uambisiøs (alle konflikter er forstått og håndtert), og
  2) Leselig og forståelig,

  så er det ikke så mye å debugge. Det forutsetter at man bruker hodet når man bruker predikater (sjekk manualen for bivirkninger på generert kode), eller skrur av advarsler.
  Tips:
  - Start feilsøkinga bottom-up med minst mulig input, d.v.s. ett symbol eller linje.
    - Test skanneren separat om nødvendig.
      Eksempel på dette legges ut på kurssidene.
  - «Println»-debugging i aksjonskoden, eller
  - Alminnelig debugging kan brukes

- Antlr har to kommandolinjeopsjoner:
  - `-trace{parser|Lexer}: Gir en dump av den rekursive parseringen. Ikke så veldig nyttig, i forhold til hva man allerede vet fra grammatikken sin.`
  - `-diagnostic:` rapporterer om hva som blir generert, pseudokode og LA-sett. Ikke så veldig nyttig dette heller, i forhold til å lese den genererte koden.

Fallgruver

- Antlr gir som regel brukeren 3-4 forskjellige måter å løse ett problem på, hvorav bare 1-2 er riktige!

Referanser

- http://www.antlr.org/
- FAQ: http://www.jguru.com/faq/ANTLR
- 7-8 tutorials:
  http://www.antlr.org/doc/getting-started.html
  Dessverre gir disse stort sett bare en falsk følelse av trygghet, all den vanskelige gørra er i referansemanualen!

Spørsmål

- Se på eksempler?