DD1360 Programmeringsparadigm: Artificiella språk och syntaxanalys, föreläsning 3

Karl Palmskog

KTH

20 mars, 2023

Förra föreläsningen

- Mer formellt om automater
  - 5-tupel av mängder: $(Q, \Sigma, q_0, \delta, F)$
  - δ är en funktion (deterministiska automater)
- Reguljära språk
  - klass av språk som kan beskrivas av ändliga automater och reguljära uttryck
  - ändliga automater och reguljära uttryck har samma **uttrycksfullhet**
- Grammatiker
  - mer utrycksfull formalism än reguljära uttryck och automater
  - kan beskriva språket $L = \{a^n b^n \mid n \geq 0\}$

Dagens föreläsning

1. Lexikal analys
2. Härledningar och syntaxträd
3. Parsning genom rekursiv medåkning
4. Eliminering av tvetydighet

Definition av lexikal analys

*Processen som omvandlar en sekvens av **tecken** till en sekvens av **tokens** (delar)*

Mål:

- Ta bort irrelevanta delar av en indatasträng, t ex
  - blanktecken/whitespace (blanksteg, nyrad, tabbar, ...)
  - kodkommentarer (oftast ej en del av kompilerad kod)
- Abstrahera bort detaljer från grammatiken, t ex
  - längsta matchning: `"hello123 45"` ska behandlas som tokensekvensen `Ident(hello123) Num(45)` snarare än `Ident(hello) Num(123) Num(4) Num(5)`

Tumregel: om en del av språket kan beskrivas av ett enkelt reguljärt uttryck så är det bättre att göra den delen till en token än till grammatikregler.

Lexikal analys av siffror och tal

- Idé: Förbehandla indatasträngen så att tal representeras av fullständiga tokens
- Exempel: indatasträngen `"378*232*(582-01) "`
  - indata som teckensekvens: `'3', '7', '8', '*', '2', '3', '2', '*', '(', '5', '8', '2', '-', '0', '1', ')', ' '`
  - tokensekvens: `Num '*' Num '*' '(' Num '-' Num ')'`
- Vissa tokens motsvarar enskilda tecken (som `'*'` och `'('`), medan andra består av delsträngar (som `Num`)
- Tokens kan också innehålla data, som heltalsvärdet för ett tal, t ex `Num(378)`
- Den genererade tokensekvensen blir **indata till parsern** (eller annat verktyg)

Analogi med naturliga språk

Den här meningen består av ord som består av bokstäver (tecken), och skiljetecken (kommatecken och parenteser).

- När vi kontrollerar stavning tittar vi på hur enskilda tecken byggts ihop till ord
- Ur ett stavningsperspektiv är de enskilda bokstäverna "byggstenarna", eller tokens, i texten
- När vi kontrollerar grammatik tittar vi på hur olika typer av ord (substantiv, verb, adjektiv) och skiljetecken byggts ihop
- Ur ett grammatikperspektiv är meningen en sekvens av ordtyper och skiljetecken, dessa är här våra tokens

Exempel: "Ugglan dansar" är sekvensen `<substantiv> <verb>`
(grovt förenklat, olika böjningar av ord har helt ignorerats)

Exempel på lexikal analys

Betrakta en Haskellfunktion:

```
myFunction alpha beta =
 5 * x
 where
 -- compute difference
 x = alpha-beta
```

Vilka tokens finns i funktionen?

- Namn: `myFunction` (funktion), `alpha` (parameter), `beta` (parameter), `x` (variabel)
- Nyckelord: `where`
- Operatorer/symboler: `'='` (lika) `'*'` (gånger), `'-'` (minus)
- Tal: `5` (heltal)

Möjlig tokenisering: `Name Name Name Equal Num Times Name Where Name Equal Name Minus Name`

Blanktecken och kommentarhantering

- För programspråk bortses konventionellt från blanktecken och kommentarer (“äts upp”)
- Blanktecken **kan** dock vara del språket, exempel:
  - indentering i Python
  - nyrad i LaTeX
- Vissa språk stödjer nästlade kommentarer, t ex OCaml

```
(* yttre kommentar börjar
   (* inre kommentar börjar
      inre kommentar slutar *)
   yttre kommentar slutar *)
```

1. Lexikal analys
2. Härledningar och syntaxträd
3. Parsning genom rekursiv medåkning
4. Eliminering av tvetydighet

Härledningar

Betrakta följande grammatik:

$$\begin{aligned}\text{Expr} \rightarrow \text{Num} \mid \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\ \mid \text{Expr} * \text{Expr} \mid \text{Expr} \,/\, \text{Expr} \mid (\text{Expr})\end{aligned}$$

En **härledning** av en indatasträng är en sekvens av de grammatikregler som genererar strängen.

Som exempel, låt oss härleda `"4*(3+5)"`:

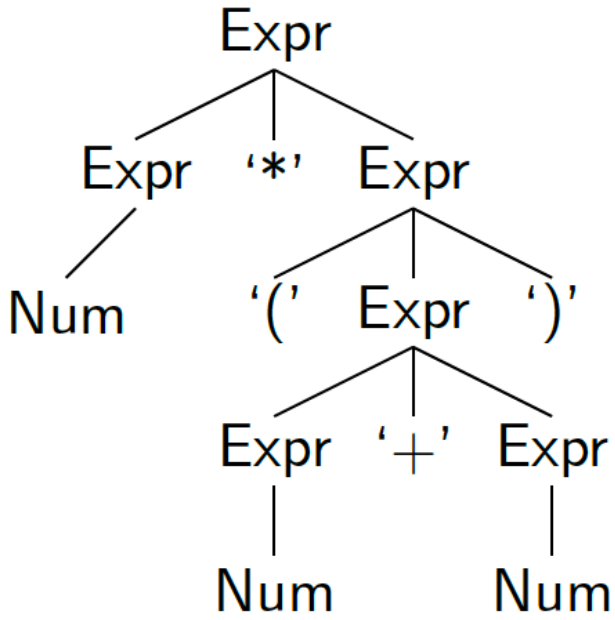
- Börja med **startsymbolen** i grammatiken
- I varje steg: **ersätt precis en ickeslutsymbol** med den högra sidan av en av dess produktionsregler

$$\begin{aligned}\text{Expr} \rightarrow \text{Expr} * \text{Expr} \rightarrow \text{Num} * \text{Expr} \rightarrow \text{Num} * (\text{Expr}) \rightarrow \\ \text{Num} * (\text{Expr} + \text{Expr}) \rightarrow \text{Num} * (\text{Num} + \text{Expr}) \rightarrow \\ \text{Num} * (\text{Num} + \text{Num})\end{aligned}$$

Syntaxträd

Ett **syntaxträd** representerar en (mängd av) härledningar och kapslar in **semantiken** för en indatasträng.

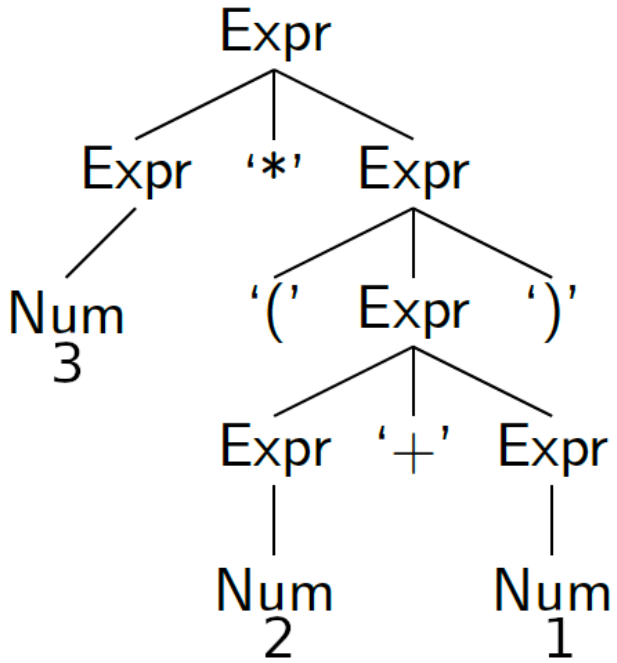
$$\begin{aligned}\underline{\text{Expr}} &\rightarrow \underline{\text{Expr}} * \text{Expr} \rightarrow \text{Num} * \underline{\text{Expr}} \rightarrow \text{Num} * (\underline{\text{Expr}}) \rightarrow \\ &\text{Num} * (\underline{\text{Expr}} + \text{Expr}) \rightarrow \text{Num} * (\text{Num} + \underline{\text{Expr}}) \rightarrow \\ &\text{Num} * (\text{Num} + \text{Num})\end{aligned}$$



Semantik för syntaxträd

Exempel: syntaxträdet kapslar in semantiken för ett uttryck (ett program).

- Syntaxträd möjliggör att **evaluera ett uttryck**: utför beräkningen och erhåll **värdet**
- Behöver veta heltalsvärdena för `Num`-tokens: lägg till som data
- Evaluering utförs som **trädtraversering**



Härledningar och träd

$$\text{Expr} \rightarrow \text{Num} \mid \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr}$$
$$\mid \text{Expr} * \text{Expr} \mid \text{Expr} / \text{Expr} \mid (\text{Expr})$$

Betrakta följande uttryck: `3 + 5 * 4`

- Går det att hitta två olika **härledningar**?
- Går det att hitta två olika **syntaxträd**?

$$\underline{\text{Expr}} \rightarrow \underline{\text{Expr}} + \text{Expr} \rightarrow \text{Num} + \underline{\text{Expr}} \rightarrow \text{Num} + \underline{\text{Expr}} * \text{Expr} \rightarrow$$
$$\text{Num} + \text{Num} * \underline{\text{Expr}} \rightarrow \text{Num} + \text{Num} * \text{Num}$$

$$\underline{\text{Expr}} \rightarrow \text{Expr} + \underline{\text{Expr}} \rightarrow \text{Expr} + \text{Expr} * \underline{\text{Expr}} \rightarrow$$
$$\text{Expr} + \underline{\text{Expr}} * \text{Num} \rightarrow \underline{\text{Expr}} + \text{Num} * \text{Num} \rightarrow \text{Num} + \text{Num} * \text{Num}$$

1. Lexikal analys
2. Härledningar och syntaxträd
3. Parsning genom rekursiv medåkning
4. Eliminering av tvetydighet

Parsning genom rekursiv medåkning

Metod för att konstruera en effektiv parser för en given grammatik:

- En indatasträng parsas enligt produktionsreglerna som krävs för dess härledning
- När en sträng för en ickeslutsymbol ska parsas, **titta framåt** på nästa token för att välja motsvarande produktionsregel
- För varje ickeslutsymbol, skapa en **rekursiv funktion** som parsar den ickeslutsymbolen
- Parsa en produktionsregel på följande sätt:
  - För varje slutsymbol: kolla om den matchar nästa token
  - För varje ickeslutsymbol: anropa funktionen för motsvarande ickeslutsymbol

Grammatik för binära träd

```
<BinTree> ::= Leaf LPar Num RPar
  | Branch LPar <BinTree> Comma <BinTree> RPar
```

Slutsymboler:

- Leaf: `"leaf"`
- Branch: `"branch"`
- Num: `[0-9]+`
- LPar: `'('`, RPar: `')'`, Comma: `','`

Exempel:

```
branch(branch(leaf(17),leaf(42)),leaf(5))
```

Resultat från lexikal analys:

```
Branch LPar Branch LPar Leaf LPar Num RPar Comma Leaf
LPar Num RPar RPar Comma Leaf LPar Num RPar RPar
```

Parser för binära träd i Java

```
<BinTree> ::= Leaf LPar Num RPar
  | Branch LPar <BinTree> Comma <BinTree> RPar

ParseTree BinTree() throws SyntaxError {
  Token t = lexer.peekToken();
  if (t.getType() == TokenType.Leaf) {
    lexer.nextToken();
    expect(TokenType.LPar);
    Token param = expect(TokenType.Num);
    expect(TokenType.RPar);
    return new LeafNode((Integer) param.getData());
  } else if (t.getType() == TokenType.Branch) {
    lexer.nextToken();
    expect(TokenType.LPar);
    ParseTree left = BinTree();
    expect(TokenType.Comma);
    ParseTree right = BinTree();
    expect(TokenType.RPar);
    return new BranchNode(left, right);
  } else { throw new SyntaxError(); }
}
```

Katter och hundar

```
<Start> ::= 'c' <Cat> | 'd' <Dog>
<Cat> ::= 'p' | 'z' | 'f' <Dog> | <Start> <Start>
<Dog> ::= 'p' | 'z' | 'b' <Cat> | <Start>
```

```
function Cat():
  char next = peekChar();
  if (c == 'p') // Cat, regel 1
    eatChar(); // gå förbi 'p'
  else if (c == 'z') // Cat, regel 2
    eatChar(); // gå förbi 'z'
  else if (c == 'f') // Cat, regel 3
    eatChar(); // gå förbi 'f'
    Dog(); // ska nu komma en Dog
  else // måste vara Cat, regel 4
    Start(); // första Start
    Start(); // andra Start
```

1. Lexikal analys
2. Härledningar och syntaxträd
3. Parsning genom rekursiv medåkning
4. Eliminering av tvetydighet

Tvetydig uttrycksgrammatik

$$\text{Expr} \rightarrow \text{IntLiteral} \mid \text{Ident} \mid \text{Expr} + \text{Expr} \mid \text{Expr} \,/\, \text{Expr}$$

```
foo + 42 / bar + arg
```

- Varje nod i syntaxträdet ges av ett grammatikalternativ
- Indatasträngen ovan har två syntaxträd

Använd prioritetslager i grammatiken

$$\text{Expr} \rightarrow \text{Ident} \mid \text{Expr} - \text{Expr} \mid \text{Expr} \,\widehat{}\, \text{Expr} \mid (\text{Expr})$$

omvandlas till

$$\begin{aligned} \text{Expr} &\rightarrow \text{Term}\,(-\,\text{Term})^* \\ \text{Term} &\rightarrow \text{Factor}\,(\widehat{}\,\text{Factor})^* \\ \text{Factor} &\rightarrow \text{Ident} \mid (\text{Expr}) \end{aligned}$$

Hantera högerassociativ operator

Använd rekursion (eller loopa och omvänd en lista)

$$x\hat{}y\hat{}z \rightarrow x\hat{}(y\hat{}z)$$

```
Exp(Var("x"), Exp(Var("y"), Var("z")))
```

```
Expr term() {
  Expr e = factor();
  if (lexer.token == ExpToken) {
    lexer.next();
    return new Exp(e, term());
  } else {
    return e;
  }
}
```

Hantera vänsterassociativ operator

$$x - y - z \rightarrow (x - y) - z$$

```
Minus(Minus(Var("x"), Var("y")), Var("z"))
```

```
Expr expr() {
  Expr e = term();
  while (lexer.token == MinusToken) {
    lexer.next();
    e = new Minus(e, term());
  }
  return e;
}
```

Grammatiker och tvetydighet

- Om vi kan hitta en sträng med **två olika syntaxträd** är grammatiken **tvetydig**
- Tvetydighet påverkar inte vilket språk grammatiken definierar
- Det kan vara svårt att bestämma om en grammatik är tvetydig eller ej
  - problemet är **oavgörbart**: det finns ingen generell algoritm
- Hur man gör grammatiker entydiga:
  - säkerställ att det bara finns ett syntaxträd per korrekt sträng
  - beakta associativitet och andra egenskaper vid syntaxträdskonstruktion
- Tvetydiga grammatiker kan fortfarande vara användbara, t. ex. för teori om språk

Manuell konstruktion av en parser

- Applicera föregående transformationer för att få en mer parsebar grammatik
- Skriv sedan en parser med rekursiv medåkning som avgör om indata är korrekt
- Ornamentera sedan parsern för att konstruera träd, och ta då hänsyn till associativet och prioritet för operatorer

Grammatik kontra rekursiv medåkning

$$\text{Expr} \rightarrow \text{Term}\ \text{TermList} \qquad \text{Term} \rightarrow \text{Factor}\ \text{FactorList}$$

$$\text{TermList} \rightarrow + \text{Term}\ \text{TermList} \mid - \text{Term}\ \text{TermList} \mid \epsilon$$

$$\text{FactorList} \rightarrow * \text{Factor}\ \text{FactorList} \mid / \text{Factor}\ \text{FactorList} \mid \epsilon$$

$$\text{Factor} \rightarrow \text{Ident} \mid (\text{Expr})$$

```
def expr = { term; termList }
def termList =
  if (token == PLUS) {
    skip(PLUS); term; termList
  } else if (token == MINUS)
    skip(MINUS); term; termList
  }
def term = { factor; factorList }
...
def factor =
  if (token == IDENT) name
  else if (token == LPAR) {
    skip(LPAR); expr; skip(RPAR)
  } else { error("expected ident or (") }
```

Sammanfattning av rekursiv medåkning

- En av de **mest använda** metoderna för att parsa kod i kompilatorer för generalistiska programspråk, exempel:
  - GCC frontend för C/C++
  - Javas referenskompilator
  - Scalas referenskompilator
- **Effektiv** (linjär) i tokensekvensens längd
- **Rättfram att implementera manuellt** baserat på en grammatik
  - finns även **parsergeneratorer** som ger källkod som utdata
- Nära överensstämmelse mellan grammatik och kod
  - vanligt att citera grammatiken i kommentarer

Parsergenerering utan medåkning

$$\text{Prop} \to \bot \mid \neg\text{Prop} \mid p \mid \text{Prop} \wedge \text{Prop} \mid \text{Prop} \vee \text{Prop} \mid \text{Prop} \Rightarrow \text{Prop} \mid (\text{Prop})$$

```
%token <string> IDENT
%token COMMA LPAREN RPAREN CONT NOT AND OR IMPLIES

%right IMPLIES
%left AND
%left OR
%right NOT

prop:
  CONT {} | NOT prop {} | IDENT {} | prop AND prop {}
| prop OR prop {} | prop IMPLIES prop {}
| LPAREN prop RPAREN {}
;
```

Talseriegrammatik

Betrakta följande grammatik:

```
<Nums> ::= Num <NumsAux> | NoNums
<NumsAux> ::= <NumsAux> SemiColon Num | Epsilon
```

- Startsymbol: `<Nums>`
- `Num`: `[0-9]+`
- `SemiColon`: `';'`
- `NoNums`: `'_'`
- `Epsilon`: tomma strängen
- Exempelsträng: `1;2;3`

Kan vi parsa den här grammatiken med rekursiv medåkning?

Talseriegrammatik, fortsättning

```
<Nums> ::= Num <NumsAux> | NoNums
<NumsAux> ::= <NumsAux> SemiColon Num | Epsilon
```

- Rekursiv medåkning måste terminera på varje (ändlig) indatasträng för att parsa korrekt
- Implementationer av `NumsAux` regel 1 måste göra anrop till `NumsAux` och når därmed aldrig basfall
- Givna grammatiken kan **inte** parsas med rekursiv medåkning (ty den är **vänsterrekursiv**)
- Med **högerrekursiv** grammatik för samma språk fungerar rekursiv medåkning

```
<Nums> ::= Num <NumsAux> | NoNums
<NumsAux> ::= SemiColon Num <NumsAux> | Epsilon
```

Vilka uttryck är tvetydiga?

Betrakta följande grammatik:

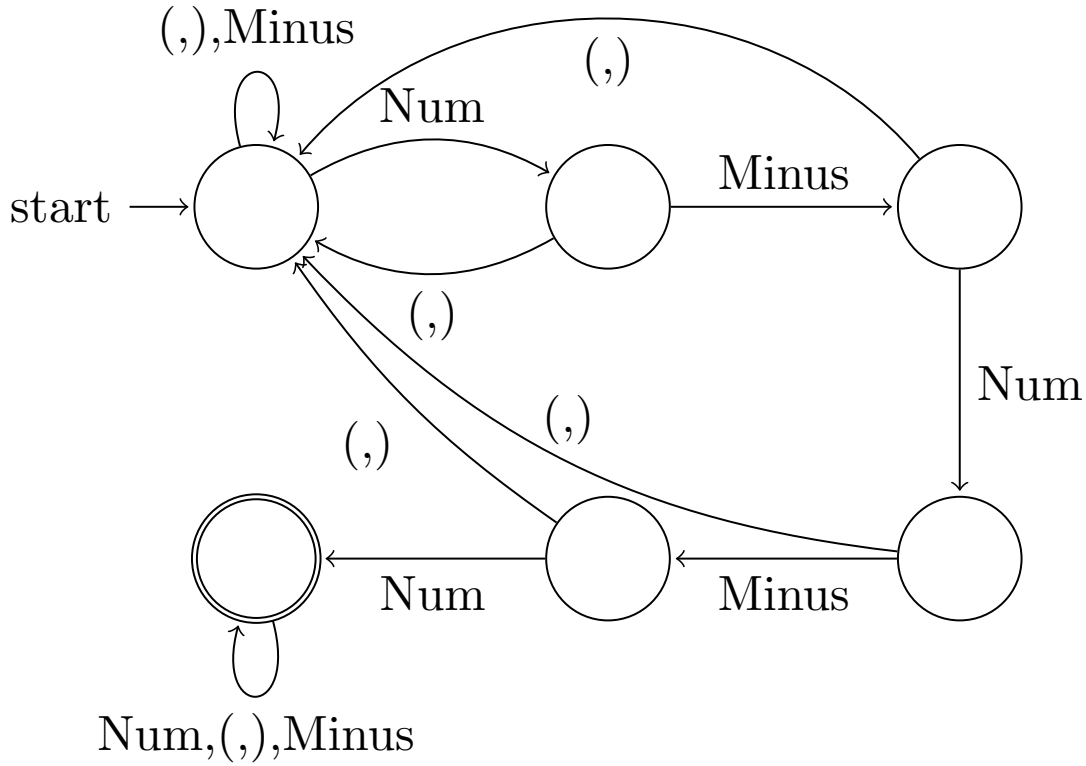
```
<Exp> ::= <Exp> Minus <Exp> | Num | LPar <Exp> RPar
```

- Startsymbol:
- `Minus`: `'-'`
- `Num`: `[0-9]+`
- `LPar`: `'('`
- `RPar`: `')'`

Hur kan vi känna igen tvetydiga uttryck som genereras av grammatiken? (Inte nödvändigtvis alla.)

Att bygga en automat för tvetydighet

- Kärnan är att minusoperatorn kan associera på flera sätt
  - `1 - 4 - 5` kan tolkas som `(1 - 4) - 5`
  - `1 - 4 - 5` kan tolkas som `1 - (4 - 5)` också
- Men: parenteser kan göra uttryck entydiga
- Vi antar att vi bara får tokensekvenser enligt grammatiken



Utökningar av BNF

- För att göra det lättare att skriva BNF-grammatiker kan vi införa **utökningar av BNF**
- En möjlig utökning är `?` för "noll eller en gång" (som för reguljära uttryck)
- Men: grammatiker med utökningar ska kunna skrivas om i vanliga BNF

```
<Stm> ::= If <Exp> Then <Stms> [ Else <Stms> ]? End
```

Hur kan vi skriva om denna deklaration i vanlig BNF? Hur generaliserar vi det till godtyckliga användningar av ??

Omskriven utökad grammatik

```
<Stm> ::= If <Exp> Then <Stms> [ Else <Stms> ]? End
```

- För varje förekomst av ? i ett högerled inför vi två stycken regler istället för den gamla regeln.
- I första regel utelämnar vi innehållet inom hakparenteser
- I andra regeln tar vi med innehållet inom hakparenteser

```
<Stm> ::= If <Exp> Then <Stms> End
| If <Exp> Then <Stms> End <Stms> End
```