Syntaks-analyse for uttrykk i Simua

Stein Krogdahl

Dette er forslag til recursiv-descent-metoder som analyserer uttrykks-syntaksen for Simula, og som bygger et tilsvarende syntaks-tre. Merk at den vil godkjenne ting som " a && b - c", slik at endelig godkjenning først kan gjøres etter binding og typeanalyse. Den vil derimot *ikke* godkjenne "a IMP b EQV c" siden IMP og EQV bare kan ha to operander, og at denne feilen ikke vil bli oppdaget i semantikksjekken. Den godkjenner heller ikke "a < b < c" selv om dette *ville* bli oppdaget i semantikksjekken (skal også bare ha to operander).

Assignment er ikke med her, men vi må her huske at Simula bruker runde parenteser også til indeksering av arrayer slik at f.eks. "a(5):= 4" må godkjennes i syntaksanalysen, og at om "a" (etter binding) viser seg å være en prosedyre så er dette ulovlig. Siden kall på prosedyrer uten parametere ikke skal ha (tom) parameter-parentes så vil jo tilsvarende gjelde for "a:= b" om a viser seg å være en prosedyre.

For å føle meg tryggere på hva jeg gjør har jeg her tenkt meg at alle presedens-nivåer har sin egen metode, men jeg har altså rasjonalisert litt med metoder som har samme form. Da får man i hvert fall full kontroll over der det bare kan være to operander, og der det kan være flere Det samme gjelser prefiks- og postfiksoperasjoner, selv om jeg må innrømme at jeg stadig måtte tilbake til disse for å diskutere med meg selv om det faktisk ble riktig. Ble de det? Jeg skal ikke mene noe m det er greiest å bruke nivåteller eller flere metoder slik som her.

Det er en generell filosofi for metodene under at, når metoden for denne syntakstiske enheten kalles, inneholder variablen "cur" allerede det første token i den syntaktiske enhet metoden behandler, og at når metoden avsluttes så vil cur inneholde tokenet BAK den tilsvarende syntakstiske enheten. Metoden "moveCur" må kalles mange steder for å leve opp til denne reglen. Generelt er hele dette apparatet et alternativ til Øysteins metoder "expect" og "accept", men jeg mener ikke derved at mitt er bedre. Jeg er bare vandt til å tenke i det.

NB: I programmet under tenkes det definert en klasse "Expression", og et antall subklasser av denne som har null, én, to eller tre sub-pekere, samt variable. Når man skal aksessere disse gjennom en Expression-peker til objektet må man derfor bruke et typekast, men det er ikke gjort i teksten under. Videre er det noen unøyaktigheter mht. typene når den aktuelle operasjonen legges inn i noden. Og selvfølgelig: programmet er ikke testet (eller i det hele tatt kjørt), så det er sikkert mange tullefeil.

```
// ---------------------------------- Globale deklarasjoner: ----------------------------------
Token cur;  // Felles-variabel.  Har inneværende Token.  Initiert med første token
void moveCur();  // Henter neste token inn i variablen cur

// --------------------- Selve metodene som analyserer og bygger tre: -------------------
Expression parseGENEXPR() {
   Expression retExpr;
   If (cur == IF) {
      moveCur();   // Betyr altså: cur = next Token
      retExpr = new IFTHENELSEnode( );  // ingen parameter, bare denne klassen
      retExpr.cond = parseGENEXPR();
      If (cur != THEN) { ERROR ... }    //  Feil i betingelse eller mangler "then".
      moveCur();
```

```
        retExpr.if = parseORELSE();  // Tilsvarer "simple expression"
        If (cur != ELSE) { ERROR ... }    //  Feil i  then-gren eller mangler else.
        moveCur();
        retExpr.else = parseGENEXPR();
    } else {
        retExpr = parseORELSE
    }
}

// Metoden under tilsvarer altså "simple expression".  Bør det puttes inn en faktisk metode
"parseSIMPLEEXPR" mellom her? Hva skal den eventuelt gjøre?

Expression parseORELSE() {   // Metode-form 9: Kan ha en eller flere operander, tolkes venstreassos.
    Expression retExpr, expr;
    retExpr = parseANDTHEN();
    while (cur == ORELSE)  {
        expr  = new BinaryNode(cur)  ;    // Bygger tre med ny node (rot) på toppen
        moveCur();
        expr.left = retExpr;                        // Gammel rot blir ny rot sitt venstre subtre
        expr.rigtht = parseANDTHEN();          // Nytt høyre subtre.
        retExpr = expr;
    }
    return retExp;
}

<  Metode parseANDTHEN med metode-form 9, kall EQV >

Expression parseEQV() {   // Metode-form 2: Kan ha EN eller TO operander
    Expression retExpr, expr;
    retExpr = parseIMP();
    if (cur == IMP) {
        moveCur();
        expr  = new BinaryNode("EQV")  ;
        expr.left = retExpr;
        retExpr.rigtht = parseIMP();
    }
    return retExpr;
}

<  Metode parseIMP med Metode-form 2, kall OR >

<  Metode parseOR med metode-form 9, kall AND >

<  Metode parseAND med metode-form 9, kall NOT >

Expression  parseNOT() {   // Metode-form 1:  Unær prefiks-operasjon, EN operand
    Expression retExpr;
    if (cur == NOT) {
        retExpr  = new UnaryNode(cur);    // Form 1 skal også brukes for unær +/-
        moveCur();
        retExpr.sub = ParseREL();  //  Flere not etter hverandre godtas ikke
    } else {
```

```
        retExpr = ParseREL();
    }
    return retExpr;
}

Expression parseREL() {   // Metode-form
    Expression retExpr, expr;
    retexpr = parseADDSUB();
    if (cur == REL)                    // Om det er en av relasjons-operasjonene
        expr  = new BinaryNode(cur)  ;
        moveCur();
        expr.left = retExpr;
        expr.rigtht = parseADDSUB();
        retExpr = expr;
    }
    return retExpr;
}

<  Metode parseADDSUB med Form 9, kall UNIADDSUB >

< Metode parseUNIADDSUB med form 1 (som NOT), kall MULDIV >

<  Metode parseMULDIV med Form 9, kall EXPON  >

<  Metode parseEXPON med Form 9, kall BASICEXPR >  // EXPON er altså venstre-ass. i Simula!?

Expression parseBASICEXPR() {      // Dette er vel kanskje det samme som “primary”?
// Merk: Alt som kan stå foran et postfix (DOT, IS, IN og QUA) må være et BASICEXPR
    Expression retExpr, expr;
    if (cur == LEFTPAR) {
        moveCur();
        retExpr = ParseGENEXPR();
        if (cur != LEFTPAR) { ERROR … }   // Venstreparentes mangler,  eller feil I uttrykk.
        moveCur();
    } else If (cur == THIS) {
        retExpr = parseTHIS();  // Eller gjøre det ferdig her?
    } else If (cur == NEW) {
        retExpr = parseNEW();  // Eller gjøre det ferdig her?
    } else If (cur == CONSTANT) {
        retExpr = new ZeroNode(CONST);
        retExpr.value = cur.value;
        moveCur();
    } else If (cur == NAME) {
        retExpr = parseVARARRCALL();    // see below
    } else { ERROR …}  // Uttrykk forventes her eller det starter galt

   // Så kan det komme en sekvens av postfikser, som bygger tre “oppover mot høyre”
    while (cur == DOT || cur== IS || cur == IN || cur == QUA  ) {
    if (cur == DOT ) {
            expr = new BinaryNode(DOT) {
            moveCur();
            expr.left = retExpr;
```

```
            expr.right = parseVARARRCALL();
            retExpr = expr;
        } else {  // Vet at cur == IS or cur == IN or cur == QUA.  Alle skal ha et klassenavn etter seg
            expr = new ISINQUAnode(cur);  //  Kunne kanskje brukt samme som ved Not og Unary +/-
            moveCur();
            expr.sub = retExpr
            if ( cur != NAME ) then { ERROR … }  // Etter IS, IN eller QUA skal det være et (klasse)navn
            expr.className = cur.name;  // Klassenavnet lagres altså ikke i en sub-node, men i noden
            moveCur();
            retExpr = expr;
        }
    }

    return retExpr
}

Expression parseVARARRCALL() {
    // Et navn med valgfri argument-parentes etter.  Er også det som kan stå etter DOT
    // Altså: Enkelt-variabel, array-aksess eller prosedyre-kall.
}
```