[Kah93] Stefan Kahrs. Mistakes and Ambiguities in the Definition of Standard ML. Technical report, Laboratory for Foundations of Computer Science, King's Buildings, EH9 3JZ, April 1993.

[Kas80] U. Kastens. Ordered attribute grammars. *ACTA Informatica*, 13(3):229–256, 1980.

[Kas91] U. Kastens. Implementation of visit-oriented attribute evaluators. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 114–139. Springer-Verlag, 1991. LNCS nr. 545.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Inc., 1981.

[Man89] Udi Manber. *Introduction to algorithms*. Addison-Wesley, 1989.

[MRS94] Khalid A. Mughal, John H. Reppy, and Lars Søraas. ML Code Generation for AML specifications. In B. Magnusson, G. Hedin, and S. Minör, editors, *Proceedings of NWPER'94*, pages 295–302, June 1994.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Mass, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Mug88] Khalid A. Mughal. *Generation of Runtime Facilities for Program Editors*. Dr.Scient thesis, University of Bergen, Norway, 1988.

[Nes94] Tor-Inge Nes. *Application and formal specification in the AML-system*. Utkast til hovedfagsoppgave, Universitetet i Bergen, 1994.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[RG86] John H. Reppy and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 218–227, December 1986.

[RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-based Editors*. Springer-Verlag, New York, NY, 1988.

[RTD83] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[TAL90] D. Tarditi, A. Acharya, and P. Lee. No assembly required : Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, November 1990.

[VS91] Harald Vogt and Doaitse Swierstra. Higher order attribute grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 256–296. Springer-Verlag, 1991. LNCS nr. 545.

Litteratur

[Alb91a] Henk Alblas. Attribute evaluation methods. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 48–113. Springer-Verlag, 1991. LNCS nr. 545.

[Alb91b] Henk Alblas. Incremental attribute evaluation. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 215–233. Springer-Verlag, 1991. LNCS nr. 545.

[Alb91c] Henk Alblas. Introduction to attribute grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, pages 1–15. Springer-Verlag, 1991. LNCS nr. 545.

[AMT92] Andrew W. Appel, James S. Mattson, and David R. Tarditi. User's manual for ML-lex, version 1.4. Part of SML/NJ distribution, October 1992.

[ASU86] A. V. Aho, R. Seti, and J. D. Ullmann. *Compilers, Principles, Techniques and Tools*. Adison-Wesley series in Computer Science. Adison-Wesley, 1986.

[AT91] Andrew W. Appel and David R. Tarditi. ML-yacc User's Manual, version 2.1. Part of SML/NJ distribution, March 1991.

[Ell94] Gøran Ellingsen. *Generating pretty-printing facileties for AML specifications*. Hovedfagsoppgave, Universitetet i Bergen, September 1994.

[EMR92] Sofoklis G. Efremidis, Khalid A. Mughal, and John H. Reppy. Attribute grammars in ML. In *Proceedings of the ACM SIGPLAN'92 Workshop on ML and its Applications*, pages 194–200, June 1992.

[EMR93] Sofoklis G. Efremidis, Khalid A. Mughal, and John H. Reppy. AML: Attribute grammars in ML. Technical Report 93-1401, Department of Informatics, University of Bergen, Norway., December 1993.

[GR91] Emden R. Gansner and John H. Reppy. eXene. In *Third International Workshop on Standard ML*, Carnegie Mellon University, September 1991.

[Har93] R. Harper. *Introduction to Standard ML*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, 1993.

[Hoo86] Roger Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, January 1986.

- Utvikling av evaluatorer for *Høyere Ordens Attributtgrammatikker*, HAG [VS91]. HAG er en utvidelse av attributtgrammatikker som gir en mye kraftigere uttrykksform. AML kompilatorens front-end er i stand til å analysere HAG.

- AML kompilatoren genererer ML kode som implementerer attribueringsystemene. Endre kompilatoren til å generere kode for andre språk[TAL90]. For modulene som genereres vil dette i de fleste tilfellene kun kreve at kode-generatorene skrives om. Problemet er å oversette selve evalueringsreglene til målspråket, disse representeres som abstrakt SML syntaks i kompilatorens front-end.

- Konstruksjon av et større oversetter-bibliotek. Back-enden består av en rekke uavhengige oversettere, en for hver modul i run-time systemet som skal genereres. Dette gjør det mulig å konstruere et omfattende oversetter-bibliotek som støtter ulike evalueringsteknikker og lagringsmetoder etc. Brukere av systemet kan da enkelt skreddersy kompilatoren ved å velge passende oversettere fra dette biblioteket.

Strukturmanipulasjon av attribuerte deltrær støttes ved et klippbord paradigme som er utviklet i denne oppgaven, og ved effektive inkrementelle teknikker for klassen OAG. Denne klassen er stor, og omfatter flere vanlige programmeringspråk. Dette gjør at AML systemet allerede nå kan anvendes i skikkelige applikasjoner. Systemet har også vært til utprøving hos Odyssey Research, Ithaca, N.Y., og mottatt positiv omtale. Tor-Inge Nes arbeider i sin hovedoppgave *Application and formal specification in the AML-system*[Nes94] med å anvende AML systemet på sitt eget spesifikasjonspråk. Dette arbeidet vil gi en god indikasjon på hvor anvendelig systemet er.

Språkbaserte strukturediteorer har blitt koblet mot attribueringsystemene via klippbord paradigmet, og det har her vist seg som et svært nyttig rammeverk for å modellere strukturmanipulasjon av attribuerte trær.

Det benyttes to inkrementelle evalueringsteknikker i attribueringsystemene. Den ene, basert på rekursive funksjoner, er utviklet i denne oppgaven. Denne teknikken har vist seg betydelig bedre enn den andre, mer konvensjonelle, teknikken. Jeg kjenner ingen andre arbeider hvor gjensidig rekursive funksjoner er anvendt på denne måten for inkrementell evaluering i fast-plan evaluatorer. En teknisk rapport om teknikken er under utarbeidelse.

Attributtlageret har mekanismer for å assosiere attributtverdier med tre-noder og for lesing/skriving av attributtverdier. Lageret er imidlertid svært enkelt og lite plasseffektivt. Ved spesifikasjon av attributtgrammatikker er det ofte behov for å distribuere attributtverdier rundt i treet, uten at de brukes i noen beregning. I attributtgrammatikken i tabell 3.1 sin Sum produksjon kopieres env attributten nedover i treet kun med det formål å sende den nedover. Det eksisterende attributtlageret implementerer slike *kopi-regler* naivt. Dette fører til at et utall like verdier ligger rundt i treet og tar opp plass. Forbedring av attributtlageret på dette området kan føre til store plass/tid-besparelser.

7.2.1 Forslag til videre arbeid

Arbeidet med denne oppgaven har resultert i en kompilator som genererer fullt brukbare attribueringsystemer for klassen OAG. Nå ligger imidlertid rammeverket der og åpner store muligheter for å følge tråden videre og eksperimentere med attribueringsystemenes ulike aspekt. Det finnes flere interessante grammatikk-klasser, og mye kan gjøres for å effektivisere med hensyn på lagring av attributtverdier. Nedenfor følger en liste med emner som kan være naturlig å fortsette med:

- Utvide de eksisterende inkrementelle evaluatorene til å takle syntaktiske referanser.
- Attributtlageret er enkelt, men ineffektivt. Her er rom for flere forbedringer:
  - Kopi-regler takles naivt. Kompilatorens front-end identifiserer kopi-regler. Dette gjør det mulig å optimere attribueringsystemene med hensyn på slike regler[Hoo86].
  - Strukturmanipulasjon av attribuerte trær krever at "gamle" verdier viskes ut. Dette krever mange kostbare hash-operasjoner.

opp argumentverdiene, anvender dem og lagrer resultatet. Når resultatet skal brukes for å definere en ny attributt må det slåes opp på nytt. Systemet basert på rekursive funksjoner slår opp en verdi en gang i hver partisjon hvor de brukes, mens automatsystemet må slå opp verdien hver gang den brukes.

Automatsystmet har i hvert tilfelle ett oppslag mer i REACTIVATED enn det funksjonsbaserte systemet. Automatsystemet må slå opp i REACTIVATED hver gang kontroll skal overføres, mens det funksjonsbaserte systemet slipper dette når kontroll overføres *mot* startnoden. SMLs run-time system overfører kontroll automatisk tilbake til den kallende funksjonen.

Systemet basert på rekursive funksjoner er langt mer effektivt enn systemet basert på endelig automat.

7.2 Konklusjoner

Den opprinnelige målsettingen med denne oppgaven var å utvikle en back-end for AML kompilatoren. Back-enden skulle generere spesifikasjonsavhengige SML moduler som, kombinert med spesifikasjonsuavhengige SML moduler, skulle gi attribueringsystemer for manipulasjon av attribuerte og ikke-attribuerte syntaks trær. Kompilatoren, og back-enden spesielt, burde støtte gjenbruk av kildekode for å gjøre det enkelt å rette kompilatoren mot ulike attribueringsystemer.

Kompilatoren som er utviklet fungerer etter intensjonene, og støtter gjenbruk av kode. Dette diskuteres også nærmere under punkt 7.1.1, og resultater fra dette arbeidet ble lagt fram på NWPER'94 i Lund i Juni 1994 i artikkelen *ML Code Generation for AML specifications*[MRS94].

I artikkelen *Attribute Grammars in ML*[EMR92], som fastlegger flere av premissene for implementasjon av attribueringsystemene, legges det vekt på følgende punkter:

1. Operasjoner for navigasjon og strukturmanipulasjon av attribuerte trær må støttes.

2. Konvertering mellom attribuerte og ikke-attribuerte trær må være mulig. Attribuerte trær skal representeres som et par av rent syntaks tre og attributtlager for å lette konvertering fra attribuert til ikke-attribuert tre. Konvertering andre veien gjøres av en attributt evaluator.

3. Operasjoner for å assosiere attributter med tre-noder må være med. I tillegg må det være mulig å lese/skrive attributtverdier.

Disse punktene, sammen med Sofoklis G. Efrimidis håndkodede eksempler, dannet utgangspunktet for arbeidet med attribueringsystemene som er utviklet i denne oppgaven. Systemet utviklet av Khalid A. Mughal og John H. Reppy ved AT&T sommeren 1993 kom til under veis, og har i stor grad vært med å fastlegge den videre kursen. Dette systemet er beskrevet i [EMR93].

Navigasjon i attribuerte trær foregår på tilsvarende måte som i ikke-attribuerte trær. Navigasjon foregår i det attribuerte treets syntaks-komponent.

Endelig Automat vs Rekrusive Funksjoner

	Hash-key		Attributt-oppslag		REACTIVATED-oppslag	
Besøkte noder	Automat	Funksjon	Automat	Funksjon	Automat	Funksjon
8	21	7	34	28	0	0
4	16	7	25	20	11	10
5	20	9	32	26	9	8
2	13	4	18	14	6	5

Tabell 7.4: *Antall beregnede hash-nøkler og hash-oppslag for inkrementell evaluering ved endelig automat og rekursive funksjoner.*

vindu på skjermen, og brukeren tillates å merke deltrær i den eksterne representasjonen. Fordi den eksterne representasjonen er isomorf med den interne kan det valgte deltreet mappes til den interne representasjonen, og en *cut*, *paste* eller *copy* operasjon utføres på det abstrakte klippbordet. Endringer som er utført mappes så tilbake til den eksterne representasjonen, og skjermen oppdateres. Figur 7.2 viser syklusen hvor intern og ekstern representasjon vekselvis oppdateres.

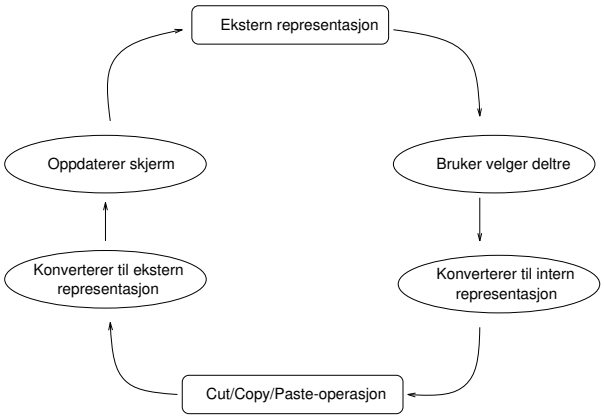
Å modelere strukturendringer av attribuerte trær innenfor rammeverket av et klippbord paradigme viste seg svært nyttig ved denne koblingen. Klippbordet har akkurat de operasjonen editoren ønsker å utføre, slik at grensesnittet mellom ekstern og intern representasjon er svært enkelt.

Utvikling av interaktive grensesnitt og kobling av disse mot attribueringsystemer behandles i [Ell94].

7.1.3 Sammenligning av inkrementelle evalueringsteknikker for klassen OAG

Tabell 7.4 viser en sammenligning av inkrementell evaluering ved endelig automat og ved de rekursive funksjonene som er utviklet i dette hovedfagsarbeidet. Det er gjort en del cut/copy/paste operasjoner, og antall beregnete hash-nøkler og hash-oppslag er tellet opp. Grammatikken som er brukt er AML spesifikasjonen i figur 4.1. Hver rad viser samme operasjon. Kolonne en i tabell 7.4 viser antall besøkte noder, kolonne to og tre sammenligner antall beregnede hash-nøkler, kolonne fire og fem antall hash-oppslag i attributtlageret, og kolonne seks og syv antall hash-oppslag i REACTIVATED. Legg merke til at tabellens øverste rad viser 0 oppslag i REACTIVATED for begge systemer. De inkrementelle systemene har her blitt brukt til batch evaluering. Dette er gjort ved å la alle noder initielt være medlem av REACTIVATED, slik at hash-oppslag i denne mengden blir overflødig.

Systemet basert på rekursive funksjoner er langt mer effektivt når det gjelder antall beregnede hash-nøkler, mens når det gjelder attributt-oppslag er forskjellen mindre. Her skal det bemerkes at med hensyn på hash-oppslag er grammatikken fra figur 4.1 bortimot en verste-tilfelle grammatikk for metoden basert på rekursive funksjoner fordi hver definerte attributt kun brukes i en regel. Når en attributtverdi skal beregnes i automat-systemet kalles en semantisk funksjon. Denne funksjonen slår



Figur 7.2: *Intern/ekstern syklus.*

fra tabell 7.1 er det enkleste systemet. Endelig automat versjonen fra tabell 7.2 må utføre ekstra endrings-forplanting analyse av semantiske regler og støtte attributt modulen med kopi-operasjoner for å støtte struktur-modifisering. Dette forklarer forskjellen i størrelse, både med hensyn på grammatikk-avhengige og run-time avhengige deler av attribueringsystemet. Versjonen for inkrementell evaluering med rekursive funksjoner, tabell 7.3 er størst. Her må kompilatoren partisjonere OAG planene i tillegg til å utføre samme endrings-forplanting analyse som den inkrementelle endelig automat versjonen.

Til tross for forskjellene i de tre systemene er ca. 3/4 deler av kompilatorkoden felles for alle tre. Systemet er fleksibelt og tillater i stor grad gjenbruk av kompilatorkode for ulike målsystemer. Dette er med å redusere tiden det tar å rette systemet mot et nytt attribueringsystem. Systemet egner seg godt til eksperimentering med ulike attribueringsystemer - om det nå er med hensyn på evalueringstrategi eller andre aspekt ved systemet.

Som et eksempel kan det nevnes at det tok under fire uker fra algoritmeutformingen for inkrementell evaluering med rekursive funksjoner startet til kompilatoren beskrevet i tabell 7.2 var modifisert til versjonen beskrevet i tabell 7.3.

7.1.2 Kobling av attribueringsystemer mot språkbasert struktureditor

Vi har koblet attribueringsystemer med abstrakte klippbord, som beskrevet under punkt 6.6, mot språkbaserte struktureditorer implementert i eXene[GR91].

Editoren opererer med en intern og en ekstern representasjon av programmet, eller utsagnet, som editeres. Som intern representasjon brukes attribueringsystemets abstrakte syntaks, mens den eksterne representasjonen er en representasjon som er isomorf med abstrakt syntaks og inneholder informasjon om *hvordan* programmet skal se ut. Dette er informasjon om alt fra hvilken font som skal brukes til hvordan linjene skal brekkes. Et utsnitt av den eksterne representasjonen vises i et

	Linjer program kode		
Komponent	**Klasse avhengig**	**Run-time avhengig**	**Generelle**
Front-end			3,764
Grammatikk analysator	718		
Back-end	*544*	1,166	386
Annet			73
Totalt	1,262	1,166	4,223

Tabell 7.1: *Kompilatorstørrelse, batch evaluering*

	Linjer program kode		
Komponent	**Klasse avhengig**	**Run-time avhengig**	**Generelle**
Front-end			3,764
Grammatikk analysator	718		
Back-end	*1,087*	1,373	386
Annet			73
Totalt	1,805	1,373	4,223

Tabell 7.2: *Kompilatorstørrelse, inkrementell endelig automat evaluering*

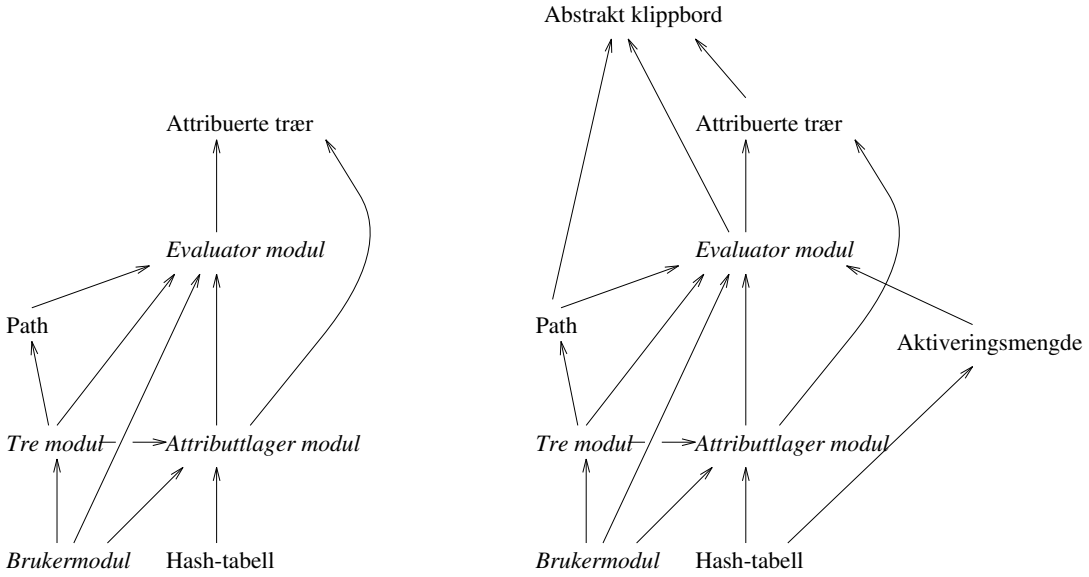
mot inkrementell evaluering ved endelig automat og 7.3 mot inkrementell evaluering ved rekursive funksjoner. Batch evaluering ved endelig automat er ikke tatt med fordi back-endens oversetter for denne evaluator modulen inngår som en del av oversetteren for inkrementell evaluering basert på samme teknikk.

Første kolonne refererer til kode som er grammatikk-klasse avhengig, andre kolonne til kode som er avhengig av attribueringsystemets implementasjonsdetaljer, som representasjon av abstrakt syntaks eller attributtlagringsteknikk, og tredje kolonne refererer til kode som er felles for de ulike versjoner.

Linjeantall skrevet i kursiv i de tre tabellene refererer til kode som er unik for den angjeldende versjon. Tabellene viser at dette kun gjelder klasse-avhengig kode i back-enden. Batch-versjonen

	Linjer program kode		
Komponent	**Klasse avhengig**	**Run-time avhengig**	**Generelle**
Front-end			3,764
Grammatikk analysator	718		
Back-end	*1,168*	1,373	386
Annet			73
Totalt	1,886	1,373	4,223

Tabell 7.3: *Kompilatorstørrelse, inkrementell evaluering ved rekrusive funksjoner.*



a) Attribueringsystem for batch evaluering

b) Attribueringsystem for inkrementell evaluering

Figur 7.1: *Komposisjon av attribueringsystemer.*

eller rekursive funksjoner. Alle andre moduler med samme navn i *a)* og *b)* er identiske. Generering av dynamiske moduler utføres i kompilatorens back-end av et oversetter/ kode-generator par. Å rette kompilatoren mot attribueringsystemer som benytter ulike evalueringsteknikker blir da svært enkelt: Kun oversetter/kode-generator paret som generer evaluator modulen trenger å skiftes ut i back-enden. Resten av de dynamiske modulene er felles for begge systemer, og kan beholdes som de er. Kompilatorens front-end og grammatikk analysator er uavhengig av attribueringsystemet den er rettet mot.

Hvis en ønsker å skifte ut attributtlager modulen med en mer sofistikert implementasjon gjøres dette på tilsvarende måte. Nytt oversetter/kode-generator par utvikles som beskrevet under punkt 5.3.3. Deretter erstattes det aktuelle paret i back-enden. Eventuelle justeringer i grensesnittet mot den nye modulen kan føre til behov for mindre endringer i kode-generator delen for enkelte av de andre modulene.

Kompilatorens front-end er generell, og i stand til å analysere alle lovlige AML spesifikasjoner som beskrevet i [EMR93], som også inkluderer høyere-ordens og sirkulære attributtgrammatikker. Ulike evaluerings-modeller krever ulike grammatikk-analysatorer og endringer av back-endens avhengige deler. Andre endringer i attribueringsystemene krever endringer i kompilatorens run-time avhengige deler. Tabell 7.1, 7.2 og 7.3 gir en oppdeling av kildekodelinjer for AML kompilatorens tre hoved komponenter når den er rettet mot tre ulike evalueringsteknikker som samtlige er beskrevet i kapittel 6. Tabell 7.1 viser AML kompilatoren rettet mot batch evaluering ved visit-funksjoner, tabell 7.2 rettet

Kapittel 7

Konklusjon

Under punkt 7.1 diskuteres en del erfaringer gjort med systemet som er utviklet i dette hovedfagsarbeidet. Under punkt 7.2 trekkes konklusjoner fra arbeidet, og videre arbeid skisseres.

7.1 Diskusjon

Kompilatoren, og back-enden spesielt, bør støtte gjenbruk av kompilatorkode for å gjøre det enklere å rette kompilatoren mot ulike attribueringsystemer. Hvorvidt dette er tilfelle diskuteres nærmere under punkt 7.1.1.

Klippbord paradigmet beskrevet under punkt 5.1.1 har vist seg som et nyttig rammeverk for manipulasjon av attribuerte trær. Vi har koblet språkbaserte strukturediteror mot attribueringsystemene via klippbord som beskrevet under punkt 6.6. Dette diskuteres mer detaljert under punkt 7.1.2.

Det benyttes to inkrementelle evalueringsteknikker for klassen OAG i attribueringsystemene. Den ene er utviklet i denne oppgaven, og har vist seg betydelig bedre enn den andre mer konvensjonelle teknikken. Disse teknikkene sammenlignes under punkt 7.1.3.

7.1.1 Gjenbruk av kompilatorkode

Et av formålene med AML systemet er å eksperimentere med ulike evalueringstrategier og attribueringsystemer. AML kompilatoren støtter gjenbruk av kompilatorkode slik at det ikke er nødvendig med endringer i andre deler av kompilatoren enn de som har direkte med de delene av attribueringsystemet en ønsker å eksperimentere med.

Figur 7.1 viser komposisjon av de ulike attribueringsystemene beskrevet i kapittel 6. *a)* viser komposisjon av attribueringsystem for batch evaluering, og *b)* for inkrementell evaluering. Modulene benevnt med kursive bokstaver er dynamiske, og inneholder spesifikasjonsavhengig informasjon. Modulene benevnt med rette bokstaver er statiske, og inneholder spesifikasjonsuavhengig informasjon. For hvert av de to systemene er det to alternative evaluator moduler, basert på enten endelig automat

```
fun suspend_e_0 (lhs_key,path,tbl,reactivated,T.PK_e_Sum,2) = let
    val t = P.treeOf path
    val lhs_pk = T.prodKind t
    val [T.T_e c1,T.T_e c2] = T.children t
    val c1_key = down (lhs_key, 1)
    val c1_pk = T.prodKind (T.nthChild( t, 1))
    val c1_path = P.down (path,1)
    val _ = if AC.isActivated (reactivated,c1_key)
        then visit_e_0 (c1_key,c1_path,tbl,reactivated,c1_pk) else ()
    val SOME c1_value = A.get_e_value(tbl, c1_key, c1_pk)
    val c2_pk = T.prodKind (T.nthChild( t, 2))
    val c2_key = down (lhs_key, 2)
    val SOME c2_value = A.get_e_value(tbl, c2_key, c2_pk)
    val parent_key = up lhs_key
    val parent_pk = T.prodKind (P.treeOf (P.up path))
    val parent_pos = P.childOrd path
    val save_lhs_value = A.attrRef_e_value(tbl, lhs_key, lhs_pk)
    val (lhs_value) = r_0001 (c1_value, c2_value)
    val _ = if (!save_lhs_value) <> (SOME lhs_value) then let
        val _ = save_lhs_value := (SOME lhs_value)
        in if applies (parent_pk,parent_pos,0)
         then AC.activate (reactivated,parent_key)
         else () end else ()
    val parent_path = P.up path
    val _ = if AC.isActivated (reactivated,parent_key)
        then
         suspend_e_0
          (parent_key,parent_path,tbl,reactivated,parent_pk,parent_pos)
         else ()
    in
    ()
    end
```

Figur 6.25: `suspend`-*funksjon for* `Sum` *som mottar kontroll fra barn 2.*

```
fun start (lhs_key,path,tbl,reactivated,T.PK_e_Sum) = let
    val t = P.treeOf path
    val lhs_pk = T.prodKind t
    val [T.T_e c1,T.T_e c2] = T.children t
    val SOME lhs_env = A.get_e_env(tbl, lhs_key, lhs_pk)
    val c1_pk = T.prodKind (T.nthChild( t, 1))
    val c1_key = down (lhs_key, 1)
    val save_c1_env = A.attrRef_e_env(tbl, c1_key, c1_pk)
    val c2_pk = T.prodKind (T.nthChild( t, 2))
    val c2_key = down (lhs_key, 2)
    val save_c2_env = A.attrRef_e_env(tbl, c2_key, c2_pk)
    val (c1_env, c2_env) = r_0001 (lhs_env)
    val _ = if (!save_c2_env) <> (SOME c2_env) then let
        val _ = save_c2_env := (SOME c2_env)
        in if applies (c2_pk,0,1)
         then AC.activate (reactivated,c2_key)
         else () end else ()
    val _ = if (!save_c1_env) <> (SOME c1_env) then let
        val _ = save_c1_env := (SOME c1_env)
        in if applies (c1_pk,0,1)
         then AC.activate (reactivated,c1_key)
         else () end else ()
    val c2_path = P.down (path,2)
    val _ = if AC.isActivated (reactivated,c2_key)
        then visit_e_0 (c2_key,c2_path,tbl,reactivated,c2_pk) else ()
    val c1_path = P.down (path,1)
    val _ = if AC.isActivated (reactivated,c1_key)
        then visit_e_0 (c1_key,c1_path,tbl,reactivated,c1_pk) else ()
    val SOME c1_value = A.get_e_value(tbl, c1_key, c1_pk)
    val SOME c2_value = A.get_e_value(tbl, c2_key, c2_pk)
    val parent_key = up lhs_key
    val parent_pk = T.prodKind (P.treeOf (P.up path))
    val parent_pos = P.childOrd path
    val save_lhs_value = A.attrRef_e_value(tbl, lhs_key, lhs_pk)
    val (lhs_value) = r_0002 (c1_value, c2_value)
    val _ = if (!save_lhs_value) <> (SOME lhs_value) then let
        val _ = save_lhs_value := (SOME lhs_value)
        in if applies (parent_pk,parent_pos,0)
         then AC.activate (reactivated,parent_key)
         else () end else ()
    val parent_path = P.up path
    val _ = if AC.isActivated (reactivated,parent_key)
        then
         suspend_e_0
          (parent_key,parent_path,tbl,reactivated,parent_pk,parent_pos)
         else ()
    in
    ()
    end
```

Figur 6.24: *Startfunksjon for produksjonen* `Sum`.

```
val start :
    H.key * P.path * A.attr_tbl * AC.active * T.prod_kind -> unit
val suspend_nt_i :
    H.key * P.path * A.attr_tbl * AC.active * T.prod_kind * int -> unit
val visit_nt_i :
    H.key * P.path * A.attr_tbl * AC.active * T.prod_kind -> unit
```

Figur 6.23: *Rekursive funksjoner.*

mønstergjenkjenning på denne slik at riktig plan utføres. Figur 6.24 viser `start`-funksjonen for produksjonen `Sum` fra figur 4.1. Legg merke til hvordan hver verdi som brukes kun beregnes *en* gang, og at kontroll sendes videre ved kall til `visit-` eller `suspend`-funksjoner.

`visit-` og `suspend`-funksjonene tar samme parametre som `start`-funksjonen, men den merkede noden og korresponderende hash-nøkkel identifiserer noden som mottar kontroll. `suspend`-funksjonen tar et heltall i tillegg til de andre parameterne. Dette heltallet identifiserer hvilket barn noden mottar kontroll fra, slik at mapV-tabellen kan kodes inn i mønsteret som avgjør hvilken partisjon som skal utføres. Figur 6.25 viser `suspend`-funksjonen som kalles når barn nummer 2 til en instans av produksjonen `Sum` sender kontroll oppover ved rekursivt kall for 0-te gang.

Partisjoner for ikke-sluttsymbol e

1	2	3	4	5	6
			Mottar kontroll fra		
Produksjon	Partisjon	Forelder	Barn 1	Barn 2	Barn 3
Root	1	Eval $e.env$ Visit $e, 0$ Eval $r.value$	Eval $r.value$ Suspend 0		
Let	1	Eval $e_1.env$ Visit $e_1, 0$ Eval e_2, env Visit $e_2, 0$ Eval $e_0.value$		Eval $e_2.env$ Visit $e_2, 0$ Eval $e_0.value$ Suspend 0	Eval $e_0.value$ Suspend 0
Sum	1	Eval $e_1.env$ Eval $e_2.env$ Visit $e_2, 0$ Visit $e_1, 0$ Eval $e_0.value$	Eval $e_0.value$ Suspend 0	Visit $e_1, 0$ Eval $e_0.value$ Suspend 0	
Const	1	Eval $e.value$			
Use	1	Eval $e.value$			

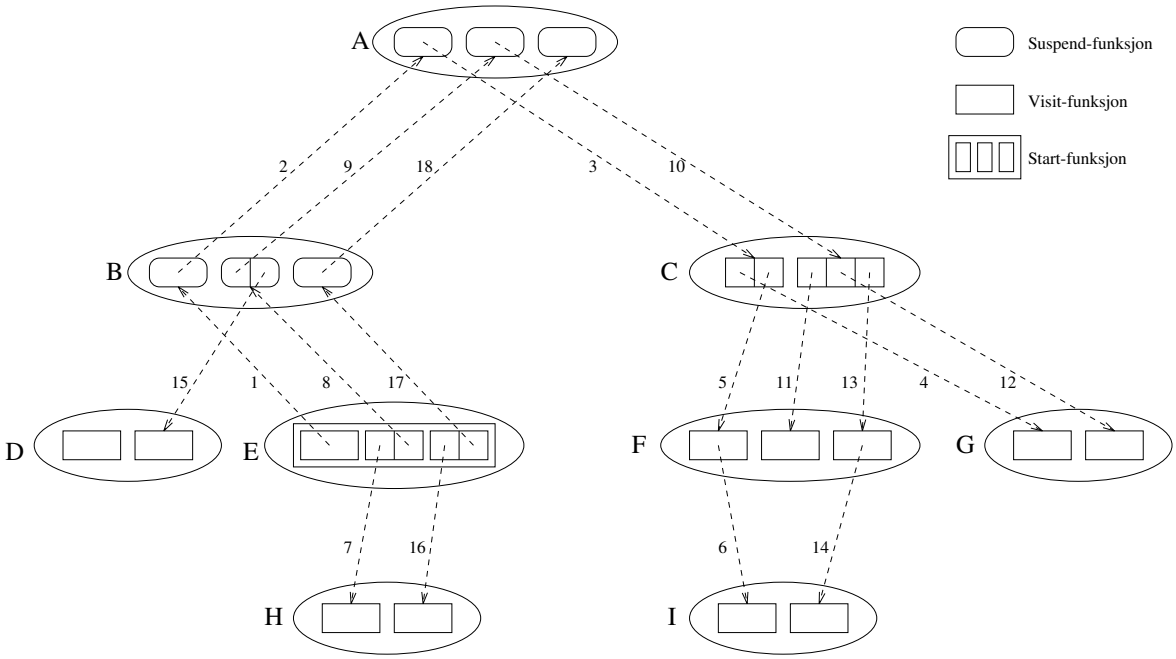
Tabell 6.1: *OAG planer partisjonert for inkrementell evaluering ved rekursive funksjoner.*

angir partisjonene som skal utføres når kontroll mottaes ved rekursivt kall over høyresidens første, andre eller tredje symbol. Legg merke til at **Suspend** instruksjonene er fjernet fra forelder-kolonnen. Dette er instruksjonen som sender kontroll tilbake til kaller, og simuleres ved at den rekursive (visit-) funksjonen som utfører sekvensen terminerer.

Tabellen viser partisjonene som inngår i funksjonene visit_e_0 og suspend_e_0. Visit-funksjonen bygges opp av partisjonene i tredje kolonne og velger partisjon som skal utføres på grunnlag av produksjonene i første kolonne, mens suspend-funksjonen bygges opp av partisjonene i kolonne fire, fem og seks. Riktig partisjon velges på grunnlag av produksjonene i første kolonne og barnet kontroll overføres fra, angitt ved kolonne fire, fem og seks. Partisjonsnummeret, gitt i kolonne 2, reflekteres i funksjonsnavnene som i (i = partisjonsnummer $-$ 1).

Implementasjon

Figur 6.23 viser signaturen til funksjonene som utgjør den rekursive evaluatoren i AMLs attribueringsystemer. `start`-funksjonen tar fem argument; en hash-nøkkel, et syntaks tre med en merket node, et attributtlager, mengden REACTIVATED og en produksjonstype. Den merkede noden svarer til noden hvor reevalueringen skal starte, og hash-nøkkelen svarer til denne noden. Reevalueringen skal starte i noden som svarer til *parent startNode* fra figur 6.17, og REACTIVATED initialiseres også som definert her. Produksjonstypen svarer til produksjonen som utledes av start-noden, og funksjonen foretar en

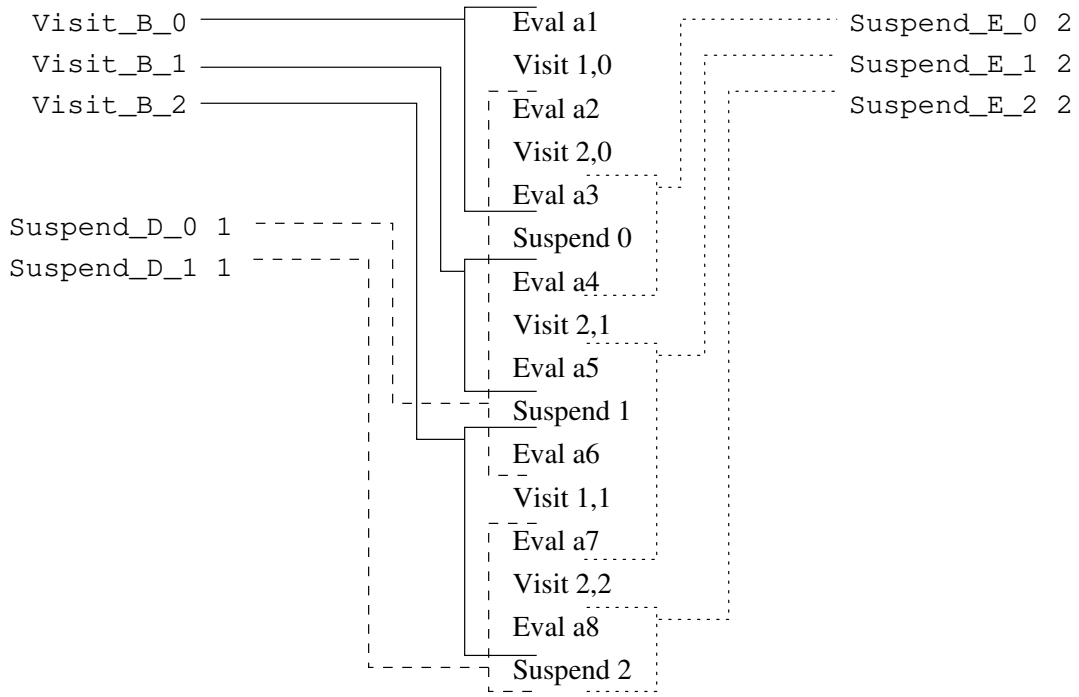


Figur 6.22: *Kontrollflyt ved inkrementell evaluering ved rekurisve funksjoner.*

seg mellom to **Suspend** instruksjoner (visit_nt_$(i+1)$: **Suspend** $i \cdots$ **Suspend** $i+1$). Tilsvarende strekker `suspend`-funksjonene seg mellom to **Visit** instruksjoner (suspend_nt_i j : **Visit** $i, j \cdots$ **Visit** $i+1, j$). Figur 6.21 viser en OAG plan som passer til produksjonen utledet av noden merket B, $B \rightarrow DE$, i figur 6.20, og hvilke partisjoner av instruksjoner som inngår i de ulike visit- og supend-funksjonene konstruert på grunnlag av denne planen. `start`-funksjonen for produksjonen er ikke vist, men inkluderer samtlige instruksjoner.

Figur 6.22 viser kontroll-flyt ved rekursiv reevaluering av treet gitt i figur 6.20, og med start i noden merket E. De ovale boksene representerer `suspend`-funksjoner. `start`-funksjoner representeres ved store rektangulære bokser som omslutter sekvenser av mindre rektangulære bokser. Disse boksene viser hvilke partisjoner av `start`-funksjonen som svarer til `visit`-funksjonene fra figur 6.20. De stiplete linjene representerer kall, og kallene er nummerert i den rekkefølgen de utføres. `suspend`-funksjoner benyttes kun når kontroll skal sendes oppover ved rekursive kall, altså kun på stien fra og med start-noden til roten av treet. Legg merke til at D nodens første `visit`-funksjon ikke kalles.

Tabell 6.1 viser hvordan OAG planene gitt i tabell 3.2 for grammatikken i tabell 3.1 partisjoneres. Første kolonne viser produksjonen som mottar kontroll mens andre kolonne angir besøksnummeret for partisjonene som står lenger til høyre. Første gang en produksjon mottar kontroll skal partisjon 1 utføres, deretter 2 etc. Grammatikken er en *en-besøks* grammatikk; hver produksjon mottar kontroll kun en gang over hvert ikke-sluttsymbol. Kolonne 3 angir partisjonen som skal utføres når kontroll mottas ved rekursivt kall over produksjonens venstre-side symbol mens kolonne fire, fem og seks



Figur 6.21: `visit-` *og* `suspend`*-funksjoner for OAG plan.*

suspend_nt_j (t, i) : Rekursiv funksjon som sender kontroll oppover i t for j-te gang fra en node merket med ikke-sluttsymbolet nt. Noden kontroll sendes fra er sin forelders i-te barn. Noden som mottar kontroll sender kontroll videre oppover ved kall til andre `suspend`-funksjoner og nedover ved `visit`-funksjoner. Unntaket er når kontroll skal sendes nedover til barnet som gav fra seg kontroll ved kall til `suspend`-funksjonen. I dette tilfellet sendes kontroll nedover ved at den kalte `suspend`-funksjonen terminerer. På samme måte som mapV-tabellen kodes i `visit`-funksjonene, kodes den i `suspend`-funksjonene (parametrene nt, i og j er tilstrekkelig) slik at eksplisitt oppslag i en mapV-tabell blir unødvendig.

Ved inkrementell evaluering praktiserer `visit-`, `start-` og `suspend`-funksjoner betinget adgang til nodene ved å vedlikeholde REACTIVATED på tilsvarende måte som beskrevet i 6.6.3.

Utvidelsen med start- og supend-funksjonene er en generalisering av `visit`-funksjonene. Batch-evaluering foregår ved et enkelt kall til en `visit`-funksjon som utfører en hel evalueringsplan for en rot-node; tilsvarende tillater `start`-funksjonene å utføre en hel plan for en hvilken som helst node i treet. `suspend`-funksjonene tillater å sende kontroll oppover i treet ved rekursive kall, på samme måte som `visit`-funksjonene tillater å sende kontroll nedover.

Ved kall utfører `suspend`-funksjonene en partisjon av en OAG plan, på tilsvarende måte som ved kall til `visit`-funksjoner. Partisjonene `visit`-funksjonene er implementasjoner av strekker

Inkrementell evaluering ved rekursive funksjoner

Batch-evaluering ved `visit`-funksjoner baserer seg på følgende prinsipper;

- Evaluering foretaes ved ett enkelt kall til rot-nodens assosierte `visit`-funksjon. En egenskap ved evalueringsplaner for OAG er at evalueringsplanene for rot-symbolet kun har en **Suspend**-instruksjon[Kas80], slik at en enkelt `visit`-funksjon er tilstrekkelig for å utføre hele planen, og derved hele evalueringen.
- Kontroll sendes nedover i treet ved rekursive kall til ulike `visit`-funksjoner.
- Kontroll sendes oppover i treet ved at `visit`-funksjonene terminerer.

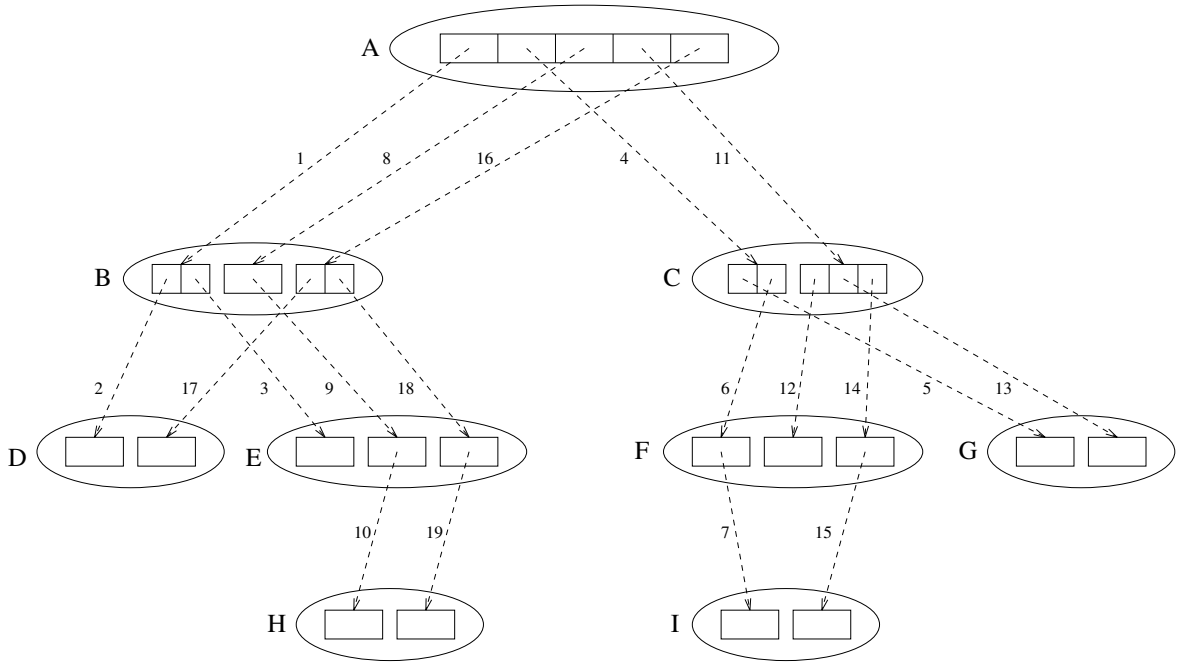
Denne evalueringsteknikken har følgende gode egenskaper som det er ønskelig å bevare ved inkrementell evaluering.

- Evaluering startes ved *ett enkelt* kall til en funksjon assosiert med start-noden.
- Kontroll sendes *fra* start-noden ved rekursive kall.
- Kontroll sendes *mot* start-noden ved at rekursivt kalte funksjoner terminerer.

OAG planene til rot-symboler har den egenskap at de kun har *en* **Suspend** instruksjon, slik at det er tilstrekkelig med ett enkelt kall til en `visit`-funksjon for batch-evaluering, mens planene for andre symboler vil vanligvis ha flere **Suspend**-instruksjoner. Evaluering av attributtene til disse nodene må utføres med like mange kall til ulike `visit`-funksjoner som den aktuelle nodens assosierte plan har **Suspend**-instruksjoner. Derfor er det ikke mulig å få utført en hel plan til en node inne i treet ved å kalle opp dens første `visit`-funksjon. Når en `visit`-funksjon terminerer vil kontroll returneres til kaller, og ikke nødvendigvis oppover i treet. `visit`-funksjoner alene er derfor ikke tilstrekkelig for inkrementell evaluering ved rekursive funksjoner.

`visit`-funksjonene tillater altså å sende kontroll nedover i treet, men kan ikke utføre planen til en vilkårlig node ved ett enkelt kall eller sende kontroll oppover i treet. For å løse disse problemene innføres to nye funksjonsklasser:

start t : Tar et tre med en merket node som parameter. Denne funksjonen utfører hele evalueringsplanen til den merkede noden. Kontroll sendes nedover i treet ved kall til `visit`-funksjoner, og oppover ved kall til *suspend*-funksjoner. Funksjonen terminerer når alle instruksjonene i den merkede nodes plan er utført.



Figur 6.20: *Kontrollflyt ved attribuering av syntaks tre ved bruk av* `visit`*-funksjoner*

kall, mellom funksjonene, og kall mot venstre i en funksjon utføres før kall lenger mot høyre. Kallene er nummerert i den rekkefølgen de utføres.

En `visit`-funksjon som terminerer tilsvarer en endelig automat som når en **Suspend**-instruksjon; kontroll sendes oppover i treet, og maskinen over startes opp der den stoppet ved visit til noden som nå gir fra seg kontroll. `visit`-funksjonen som sender kontroll oppover har ikke behov for å slå opp i en mapV-tabell, run-time systemet tar seg implisitt av denne oppgaven.

Inkrementell evaluering ved visit-funksjoner

En vanlig teknikk for inkrementell evaluering med rekursive `visit`-funksjoner beskrives i [Alb91b]. Teknikken går kort fortalt ut på, i tillegg til initialisering og vedlikehold av mengden REACTIVATED som tidligere beskrevet, å innlemme alle noder på stien fra og med roten av syntaks-treet til noden hvor reevalueringen skal starte i en mengde *subTreeAffected*. Noder med medlemskap i REACTIVATED besøkes og attributtene reevalueres, mens noder med medlemskap kun i *subTreeAffected* besøkes uten at attributter reevalueres. På denne måten vil, ved kall av rotens `visit`-funksjon, kontroll nå regionen som trenger reevaluering og nødvendig oppdatering utføres. Svakheten med denne metoden er at kontroll må gå fra roten av treet, slik at mange unødvendige attributt-oppslag, og eventuelle andre beregninger, må utføres.

```
fun  do_Use_error_value_0 (tbl,path,lhs_key) = let
     val p as T.T_exp(Use(c1)) = P.treeOf path
     val lhs_pk = T.prodKind p
     val SOME lhs_env = A.get_exp_env(tbl, lhs_key, lhs_pk)
     val save_local_error = A.attrRef_Use_error(tbl, lhs_key)
     val parent_key = up lhs_key
     val parent_pk = T.prodKind (P.treeOf (P.up path))
     val parent_pos = P.childOrd path
     val save_lhs_value = A.attrRef_exp_value(tbl, lhs_key, lhs_pk)
     val (local_error, lhs_value) = r_000a (c1, lhs_env)
     val affected = []
     val affected =
               if (!save_lhs_value) <> (SOME lhs_value) then let
               val _ = save_lhs_value := (SOME lhs_value)
               in if applies (parent_pk,parent_pos,0)
                then (parent_key::affected)
                else affected
               end
                else affected
     val _ = if (!save_local_error) <> (SOME local_error)
               then let
               val _ = save_local_error := (SOME local_error)
               in () end else ()
     in
     affected
     end
```

Figur 6.19: *Semantisk funksjon for beregning av* `value` *og* `error` *attributter i inkrementellt system.*

```
datatype action
 = Eval of attr_tbl * path * key -> key list
 | Visit of int * int
 | Suspend of int
 | EndOfPlan
```

Figur 6.18: *Intruksjoner for endelig automat for inkrementell evaluering av OAG.*

Automaten tolker et instruksjonsett definert som i figur 6.18. `Eval` instruksjonen bærer en semantisk funksjon som returnerer en liste med hash-nøkler som identifiserer produksjonsinstansene som er påvirket av endringer i de reevaluerte attributtinstansene. Automaten kan da enkelt oppdatere REACTIVATED samtidig som attributtinstanser reevalueres;

```
(app (fn key => AC.activate (reactivated,key)) (semantic (tbl,p,key))
```

Figur 6.19 viser den semantiske funksjonen konstruert på grunnlag av evalueringsregelen for `Use` sine `value` og `error` attributter definert i kapittel 4. Funksjonen bygger opp en liste, `affected`, med produksjonsinstanser som er påvirket av endringer i attributtverdier, og som returneres av funksjonen. Legg merke til at endringer i verdien til `error` ikke kan påvirke attributtene i andre produksjonsinstanser - den er en lokal attributt, og at predikatet `applies` brukes for å undersøke om `value` brukes i sin øvre kontekst. Legg og merke til at operatoren <> brukes for å teste på ulikhet. For at dette skal fungere korrekt kreves det at typen til attributten er en `eqtype`, at kun syntaktisk like verdier betraktes som like. Brukes for eksempel en liste som representasjon av en mengde vil denne testen si at mengdene representert ved listene `1::2::nil` og `2::1::nil` er ulike, noe som er feil. I de fleste tilfeller er dette likevel en hensiktsmessig test.

6.6.5 Inkrementell evaluering av OAG med rekursive funksjoner

Evaluering ved `visit`-funksjoner har en rekke fortrinn framfor evaluering ved endelig automat; det er blant annet mulig å hurtigbuffre attributtverdier og hash-nøkler over flere semantiske funksjoner, og kontrollflyten i treet er kodet implisitt i funksjonene. I dette avsnittet utvikles en inkrementell evalueringsteknikk basert på rekursive funksjoner. Teknikken baserer seg på algoritmen for inkrementell evaluering av OAG som beskrevet under punkt 6.6.3, og har samme gode egenskaper som batch teknikken basert på `visit`-funksjoner. Til slutt beskrives hvordan denne evalueringsteknikken implementeres i AMLs attribueringsystemer.

Figur 6.20 viser kontrollflyt ved evaluering av et syntaks tre ved bruk av `visit`-funksjoner. De ovale boksene A til I representerer noder, og de rektangulære er `visit`-funksjoner assosiert med nodene. `visit`-funksjonene assosiert med en node er ordnet slik at funksjonen lengst mot venstre kalles ved første visit, nestlengst mot venstre ved andre etc. De stiplete linjene viser kontrollflyt, eller

```
fun reevaluate startNode = let
    fun propagate (currentNode,planIndex,reactivated) =
        case plan (prodKind currentNode,planIndex)
         of (Eval attribute) => let
              val reactivated = let
                    val changed = reevaluate attribute
                    val neighborNode = neighbor (currentNode,attribute)
                    in
                    case changed andalso (appliesAt (attribute, neighborNode))
                     of true => add (neighborNode,reactivated)
                      | false => reactivated
                    end
                in
                propagate (currentNode,succ planIndex,reactivated)
                end
          | (Visit (i,j)) =>
              case member (child (currentNode,i),reactivated)
               of true => let
                        val childNode = child (currentNode,n)
                        val childIndex = mapV (prodKind childNode,0,j)
                        in
                        propagate (childNode,childIndex,reactivated)
                        end
                | false => propagate (currentNode,succ planIndex,reactivated)
          | (Suspend j) =>
              case isRoot currentNode
               of false =>
                    case member (parent currentNode,reactivated)
                     of true => let
                                val parentNode = parent currentNode
                                val parentIndex =
                                      mapV (prodKind parentNode,childOrd currentNode,j)
                                in
                                propagate (parentNode,parentIndex,reactivated)
                                end
                      | false => propagate (currentNode,succ planIndex,reactivated)
                | true => ()
          | (EndOfPlan) => ()
    val reactivated = add (startNode, add ( parent startNode, empty))
    in
    propagate (parent startNode,startIndex,reactivated)
    end
```

Figur 6.17: *Visit-sekvensbasert endringsforplantingsalgoritme*

AFFECTED. Denne teknikken er kjent som *endringsforplantning* (eng: *change-propagation*), fordi endringene forplanter seg ut i treet, og er asymptotisk optimal fordi | REACTIVATED | er proporsjonal med | AFFECTED |.

Figur 6.17 viser en algoritme for inkrementell evaluering basert på prinsippene beskrevet over. Algoritmen er basert på en tilsvarende algoritme presentert i [RT88], kapittel 12. *reevaluate* tar et attribuert tre med en merket node, *startNode*, som argument. *startNode* identifiserer det nye deltreets rot. Informasjon mellom det nye deltreet og resten av treet flyter gjennom *startNode*s attributter, og for at endringene skal få forplante seg videre ut i treet må alle attributtforekomster i *startNode*s øvre og nedre kontekst reevalueres. Dette sikres ved å gi begge disse produksjonsinstansene medlemskap i REACTIVATED og starte maskinen til øvre kontekst.

Algoritmens indre løkke, *propagate*, svarer til den indre løkken i algoritmen gitt i figur 6.10, men med endringslogging ved evaluering og adgangskontroll ved besøk til nabo-noder. Det er innført en **EndOfPlan** instruksjon. Ved batch evaluering terminerer prosessen etter at den første (og eneste) **Suspend** instruksjonen i roten av treet er utført. Ved inkrementell evaluering kan prosessen terminere et vilkårlig sted i treet. **EndOfPlan** innføres som siste instruksjon i alle planer og signaliserer at reevalueringen er fullført og prosessen kan terminere.

empty, *add* og *member* er mengdeoperasjoner; *empty* er den tomme mengden, *add* innlemmer et element i en mengde og *member* er et predikat som forteller om et element er medlem i en mengde. *reevaluate* evaluerer en attributtinstans og forteller om den skiftet verdi, og *neigbor* beregner produksjonsinstansen hvor en attributtforekomst brukes. *appliesAt* er et predikat som forteller om en attributt virkelig brukes i en produksjon.

6.6.4 Endelig automat-basert inkrementell OAG-evaluator

I dette avsnittet beskrives implementasjon av en endelig automat for inkrementell OAG evaluering av AML spesifikasjoner. Automaten er en videreutvikling av automaten beskrevet i 6.5.2 men som nå også inkluderer prinsippene beskrevet i 6.6.

Algoritmen gitt i figur 6.17 er basert på dynamisk oppdatering av og oppslag i mengden REACTIVATED. En modul for vedlikehold av mengden REACTIVATED bygges oppå den generelle hash-tabellen beskrevet i figur 6.6, og tilbyr følgende operasjoner;

mkActive $() \rightarrow active$: Lager en ny, tom, mengde av aktive produksjonsinstanser.

isActivated $(active, key) \rightarrow bool$: Predikat som avgjør om en produksjonsinstans identifisert ved key er medlem av mengden $active$.

activate $(active, key) \rightarrow ()$: Legger produksjonsinstansen identifisert ved key inn i mengden av reaktiverte produksjoner, $active$.

attributtene. Delmengden av treets inkonsistente attributtinstanser benevnes AFFECTED.

La T være et strukturmodifisert, ikke-konsistent tre, og T' et tre med samme struktur men konsistent. AFFECTED *er mengden av attributtinstanser med ulike verdier i T og T' [RT88].*

Dette definerer arbeidet ved å oppdatere T til T' til $O(|\mathsf{AFFECTED}|)$. En inkrementell evaluator er *asymptotisk optimal* hvis kostnaden ved reevaluering er $O(|\mathsf{AFFECTED}|)$. Dette kostnadsestimatet ignorerer kostnadene ved beregning av semantiske funksjoner, kun antall semantiske funksjoner som må beregnes og kostnader for beregning av evalueringsorden er tatt med. Kostnadsestimatetet er derfor underestimert.

6.6.3 Inkrementell reevaluering ved endringsforplantning og faste planer

De faste OAG planene for en attributtgrammatikk gir en total evalueringsorden for attributtinstansene i ethvert tre, T, utledet av grammatikkens underliggende kontekstfrie grammatikk. Ordnen respekterer den partielle ordningen gitt ved $D(T)$, og planene for hver produksjon p gir også en akseptabel reevalueringsorden for attributtforekomstene i enhver instans av p.

Ved å benytte de samme faste planene som ved ikke-inkrementell evaluering ved inkrementell evaluering sikres det at ingen attributtinstans vurderes reevaluert før attributtinstanser den er avhengig av er vurdert reevaluert, og at ingen instans tilordnes en verdi som ikke er dens endelige. Under bestemte forhold vil det være mulig å ignorere **Visit** og **Suspend** instruksjoner, slik at det med en enkelt operasjon er mulig å unngå reevaluering av en stor mengde attributtinstanser som er tilordnet endelige konsistente verdier.

For å unngå unødige nabo-besøk, skal kun produksjonsinstanser med attributtforekomster som er direkte avhengig av attributtforekomster som er reevaluert og tilordnet en ny verdi besøkes. En måte å holde rede på dette er å vedlikeholde en mengde, REACTIVATED, av reaktiverte produksjonsinstanser under reevalueringen. Kun medlemmer av denne mengden tillates besøk ved **Visit** og **Suspend** instruksjoner.

Ved reevalueringens start er medlemmene av AFFECTED ukjent, men attributtinstansene som er direkte påvirket av modifikasjonen, og alle instanser som, med hensyn på $D^*(T)$[2], er avhengig av disse, er potensielle medlemmer. En produksjonsinstans, p_T, oppnår medlemskap i REACTIVATED når minst en av p_T's *brukte* attributtforekomster er identifisert som medlem av AFFECTED. Reevalueringen initieres ved at produksjonsinstansene hvor de direkte påvirkede attributtinstansene defineres innlemmes i REACTIVATED, og evaluatoren startes med den noden i REACTIVATED som er nærmest roten og kalles *currentNode*. De potensielle medlemmene reevalueres, og for hver instans, a, som endrer verdi og identifiseres som medlem av AFFECTED, innlemmes produksjonsinstansen, q_T, hvor a er en brukt attributtforekomst i REACTIVATED. Effekten av dette er at alle q_T sine definerte attributtforekomster som defineres ved senere besøk til q_T reevalueres og vurderes som medlemmer av

[2] $D^*(T)$ er den transitivt lukkede $D(T)$. Hvis kantene (a, b) og (b, c) er med i $D(T)$ er kantene (a, b), (b, c) og (a, c) med i $D^*(T)$.

```
signature ABS_CLIPBOARD =
  sig
  structure AT : ATTR_TERMS
  type clipboard
  val cut : AT.attributed_term -> AT.attributed_term * clipboard
  val copy : AT.attributed_term  -> clipboard
  val paste : AT.attributed_term * clipboard -> AT.attributed_term
  end
```

Figur 6.16: *Abstrakt klippbord.*

6.6 Abstrakt klippbord

Klippbord paradigmet, beskrevet under punkt 5.1.1, beskriver strukturmodifikasjon av attribuerte trær, hvor kun hele *velformede* deltrær kan klippes og limes. Med velformet deltre menes at det eksisterer et konsistent tre som deltreet er en del av.

Modifikasjon av attribuerte trær i AML følger klippbord paradigmet. Klippbordet som benyttes kalles et *abstrakt klippbord* for å understreke at trærene som manipuleres er attribuerte *abstrakte* syntaks trær. I de påfølgende avsnitt beskrives implementasjonen av klippbordets operasjoner, og inkrementell OAG evaluering for vedlikehold av modifiserte trær.

6.6.1 Cut, Copy og Paste

Klippbordet har den abstrakte spesifikasjonen gitt i figur 6.16. `cut`,`copy` og `paste` operasjonene oppfører seg som tidligere beskrevet. Deltreet som skal manipuleres angies med en `path` til deltreets rotnode. Typen `clipboard` som bærer de utklippede deltrærene tilsvarer typen `attributed_term` og er et par av et deltre og en datastruktur som bærer deltreets attributtverdier.

`cut` og `paste` må viske ut attributtverdier i trærene som manipuleres, mens `copy` og `paste` må kopiere attributtverdier mellom klippbordet og treet. Operasjonene bruker attributtlagerets underliggende hash-tabells `reset` operasjon for å viske ut attributtverdier som skal fjernes fra treet, og attributtlagerets typesikre `copy` operasjon for å kopiere attributtverdier. Dette kombinert med det typesikre grensesnittet mot abstrakt syntaks definert ved `TREE` og `PATH` signaturene under punkt 6.2, gjør at disse operasjonene kan implementeres som statisk, spesifikasjonsuavhengig, kode.

6.6.2 Klippbordparadigmet og inkrementell evaluering

Etter en strukturmodifikasjon i et attribuert tre utført ved en `cut` eller `paste` operasjon vil en delmengde av treets attributter ikke lenger være konsistent, og treet må reevalueres. Til dette er det er mulig å benytte *inkrementelle* teknikker som forsøker å begrense reevalueringen til de inkonsistente

```
fun visit_e_0 (tbl, lhs_key, U.Sum(c1, c2)) = let
    (* EVAL Sum[1]$env *)
      val lhs_env = A.get_e_env(tbl, lhs_key)
      val (c1_env) = (lhs_env)
      val c1_key = down (lhs_key, 1)
      val _ = A.put_e_env(tbl, c1_key, c1_env)
    (* EVAL Sum[2]$env *)
      val (c2_env) = (lhs_env)
      val c2_key = down (lhs_key, 2)
      val _ = A.put_e_env(tbl, c2_key, c2_env)
    (* VISIT (e@2, 0) *)
      val _ = visit_e_0 (tbl, c2_key, c2)
    (* VISIT (e@1, 0) *)
      val _ = visit_e_0 (tbl, c1_key, c1)
    (* EVAL Sum[0]$value *)
      val c1_value = A.get_e_value(tbl, c1_key)
      val c2_value = A.get_e_value(tbl, c2_key)
      val (lhs_value) = r_0001 (c1_value, c2_value)
      val _ = A.put_e_value(tbl, lhs_key, lhs_value)
    (* SUSP 0 *)
      in () end
```

Figur 6.15: *Visit-funksjon for produksjon* `Sum`.

nøkler og produksjonstyper beregnes. Ved kall til andre `visit`-funksjoner er det også nødvendig å beregne hash-nøkelen for den aktuelle noden. Beregning av slike verdier optimeres på samme måte som i de semantiske funksjonene beskrevet i 6.5.2; ved at hver verdi kun beregnes *en* gang.

Evaluering ved `visit`-funksjoner gir imidlertid en betydelig effektivitetsgevinst sammenlignet med evaluering ved endelig automat. Hver `visit`-funksjon er direkte implementasjon en *partisjon*[1] av en OAG plan, og ikke kun en enkelt **Eval** instruksjon som tilfellet er med semantiske funksjoner for endelig automat. Dette gjør at verdier kan beregnes en enkelt gang for hver partisjon, og ikke for hver enkelt instruksjon som tilfellet er for evaluering ved endelig automat. Dette illustreres ved følgende eksempel: Figur 6.15 viser `visit`-funksjonen for 0-te visit til produksjonen `Sum` fra eksempelet i figur 4.1. Å utføre den samme sekvensen, som denne funksjonen er en implementasjon av, ved hjelp av evaluatoren beskrevet i 6.5.2 ville for eksempel ført til at hash-nøklene `c1_key` og `c2_key` ble beregnet 3 ganger og at attributtverdien `lhs_env` ble slått opp 1 gang, i motsetning til beregnet 1 gang og ikke slått opp i det heletatt.

[1] Med partisjon menes en delsekvens.

```
fun  semantic_Use_value_0_error (tbl,path,lhs_key) = let
      val p as T.T_e(Use(c1)) = P.treeOf path
      val lhs_pk = T.prodKind p
      val SOME lhs_env = A.get_e_env(tbl, lhs_key, lhs_pk)
      val (lhs_value, local_error) = r_0002 (lhs_env, c1)
      val _ = A.putLocal_Use_error(tbl, lhs_key, local_error)
      val _ = A.put_e_value(tbl, lhs_key, lhs_value, lhs_pk)
      in
      ()
      end
```

Figur 6.14: *Semantisk funksjon for beregning av* `value` *og* `error` *attributter.*

denne noden. Det er tilstrekkelig med oppslagsnøkkelen for å identifisere attributtinstansene som skal brukes i beregningen, men for å muliggjøre syntaktiske referanser må treet med den merkede noden også være med.

Evaluering av en attributt foregår slik; først hentes argumentene til evalueringsfunksjonen, deretter beregnes verdien ved å anvende argumentene på denne, og til sist lagres resultatet i attributtlageret. Henting og lagring av en nodes attributtverdier impliserer beregning av nodens hash-nøkkel og produksjonstype. De semantiske funksjonene tar seg av disse operasjonene. Hver semantisk funksjon er konstruert slik at operasjoner som beregning av hash-nøkkel eller produksjonstype ikke foretaes mer enn en gang, selv om den aktuelle verdien brukes flere ganger i funksjonen. Figur 6.14 viser den semantiske funksjonen konstruert på grunnlag av evalueringsregelen for `Use` sin `value` og `error` attributt fra figur 4.1. Selve evalueringsregelen er oversatt til et SML funksjonsuttrykk som er bundet til et unikt navn, `r_0002`, og tar attributter og syntaktiske referanser som argument. På denne måten kan flere semantiske funksjoner dele evalueringsfunksjon, jmf. faktorisert attribuering.

6.5.3 Implementasjon av OAG evaluator basert på rekursive funksjoner

En annen måte å implementere OAG evaluatorer på, og som blant annet er foreslått i [Kas80], er ved bruk av gjensidige rekursive funksjoner. I dette avsnittet beskrives en slik implementasjon for AML spesifikasjoner.

For hver **Visit** i til et ikke-sluttsymbol X er det en funksjon `visit_`X`_`i. Funksjonen tar tre argument; et attributtlager, noden som skal besøkes og hash-nøkkelen for denne noden. `visit`-funksjonen foretar mønstergjenkjenning på produksjonen som utleder noden som besøkes, slik at *mapV*-tabellen blir overflødig. **Suspend** simuleres ved at funksjonen terminerer og kontroll returneres til kaller, altså noden over.

Evaluering av hver enkelt attributt foregår som ved evaluering med endelig automat; argumenter må hentes fra attributtlageret, verdien beregnes og til slutt lagres i attributtlageret. Underveis må hash-

```
datatype action
 = Eval of attr_tbl * path * key -> unit
 | Visit of int * int
 | Suspend of int
```

Figur 6.12: *Instruksjoner for endelig automat*

```
fun plan ((T.PK_e_Let), 0) = Eval S.semantic_Let_env_2
  | plan ((T.PK_e_Let), 1) = Visit (2,0)
  | plan ((T.PK_e_Let), 2) = Eval S.semantic_Let_env_3
  | plan ((T.PK_e_Let), 3) = Visit (3,0)
  | plan ((T.PK_e_Let), 4) = Eval S.semantic_Let_value_0
  | plan ((T.PK_e_Let), 5) = Suspend (0)
  | plan ((T.PK_e_Const), 0) = Eval S.semantic_Const_value_0
  | plan ((T.PK_e_Const), 1) = Suspend (0)
fun mapV ((T.PK_e_Let),0,0) = 0
  | mapV ((T.PK_e_Let),2,0) = 2
  | mapV ((T.PK_e_Let),3,0) = 4
  | mapV ((T.PK_e_Const),0,0) = 0
```

Figur 6.13: *Utsnitt av* `plan-` *og* `mapV-`*tabeller*

som holder evalueringstrategien.

Automaten er i stand til å tolke og utføre instruksjoner som definert i figur 6.12. `Visit` og `Suspend` instruksjonene er bærere av informasjon som tidligere beskrevet under punkt 6.5.1, mens `Eval` instruksjonen angir attributtforekomstene som skal evalueres. Dette gjør den ved å gi en semantisk funksjon som i enhver kontekst, gitt korrekte argumenter, evaluerer de riktige attributtinstansene. I AML er det tillatt å la en enkelt attribueringsregel definere flere attributter (jmf. multippel attribuering), og på denne måten 'slipper' automaten å ta stilling til hvilke og hvor mange attributtinstanser som skal evalueres ved hver enkelt `Eval` instruksjon.

Instruksjoner hentes suksessivt fra en *plan*-tabell og når kontroll skal overføres til en nabo-node, beregnes ny plan-indeks ved oppslag i en *mapV*-tabell. Figur 6.13 viser utsnitt av plan- og mapV-tabellene for AML spesifikasjonen i figur 4.1. Legg merke til navnkonvensjonen for funksjonene båret av `Eval` instruksjonene. Navnet `semantic_Let_env_2` forteller at dette er funksjonen som evaluerer `env`-attributten til andre symbol på høyre-side av produksjonen `Let`.

De semantiske funksjonene båret av `Eval` instruksjonene tar et attributtminne, et syntaks tre med en merket node og en oppslagsnøkkel for attributtminnet som argument. Den merkede noden svarer til *currentNode* i algoritmen gitt i figur 6.12, og oppslagsnøkkelen er nøkkelen som korresponderer til

```
fun attribute t = let
    fun eval (tbl,p,key,ind) = (
        case PL.plan ((T.prodKind (P.treeOf p)),ind)
         of (PL.Eval semantic) => (
               semantic (tbl,p,key); eval (tbl,p,key,(ind+1)))
          | PL.Visit (i,j) => let
               val childInd =
                     PL.mapV (T.prodKind (P.treeOf(P.down (p,i))),0,j)
               in
               eval (tbl,P.down (p,i),A.down (key,i),childInd)
               end
          | PL.Suspend j => (case P.isBase p
               of true => tbl
                | false => let
                    val parInd = PL.mapV (T.prodKind(P.treeOf(P.up p)),
                                          P.childOrd p,j)
                    in
                    eval (tbl,P.up p,A.up key,parInd)
                  end))
    val tbl = A.mkTable ()
    val p = P.path t
    in
    AT.AttributedTerm { tbl = eval (tbl, t, A.rootKey, 0), path = p }
    end
```

Figur 6.11: *Implementasjon av endelig automat for evaluering av OAG.*

plan (p, i) : Gir i-te instruksjon for produksjon p.

mapV (p, n, i) : Gir planindeks ved vekking av maskin når kontroll overføres for i-te gang fra n-te ikke-sluttsymbol forekomst i produksjon p.

Funksjonene *parent*, *child*, *isRoot* og *rootOf* svarer til operasjonene `up`, `down`, `isBase` og `path` fra signaturen i figur 6.5. *startIndex* er indeks til første instruksjon i alle planer, og *succ* gir indeks til neste instruksjon.

6.5.2 Implementasjon av endelig automat-basert OAG evaluator

I dette avsnittet beskrives implementasjonen av en endelig automat for OAG evaluering av AML spesifikasjoner.

Kjernen i evaluatoren er en implementasjon av en endelig automat, gitt i figur 6.11. Implementasjonen er basert på, og svært lik, algoritmen gitt i figur 6.10. Enkelte av identifikatorene har kvalifiserte navn som angir hvilken SML struktur de er definert i. `A`, `P` og `T` betegner henholdsvis attributtlageret beskrevet i 6.3 og syntaks trær med og uten merkede noder som beskrevet i 6.2. `PL` angir strukturen

på produksjonens venstre side. For begge instruksjoner er det en parameter, j, som angir for hvilken gang kontroll overføres over den gitte ikke-sluttsymbol forekomsten. Denne parameteren, kallt *visit/suspend-nummer*, brukes til å beregne oppstart-tilstand for nabo-maskinen som skal startes.

```
fun Evaluate tree = let
    fun iterate (currentNode,planIndex) =
        case plan (prodKind currentNode,planIndex)
          of (Eval attribute) => let
                val _ = evaluate attribute
                in
                iterate (currentNode,succ planIndex)
                end
           | (Visit (i,j)) => let
                val childNode = child (currentNode,i)
                val childIndex = mapV (prodKind childNode,startIndex,j)
                in
                iterate (childNode,childIndex)
                end
           | (Suspend j) => case isRoot currentNode
                of false => let
                      val parentNode = parent currentNode
                      val parentIndex = mapV (prodKind parentNode,childOrd currentNode,j)
                      in
                      iterate ( parentNode, parentIndex )
                      end
                 | true => ()
    in
    iterate (rootOf tree,startIndex)
    end
```

Figur 6.10: *Algoritme for attributtevaluering av OAG.*

Algoritmen i figur 6.10 er basert på tilsvarende algoritmer gitt i [RT88, Kas80] og beskriver hvordan en *endelig automat* evaluerer et tre ved hjelp av faste planer for OAG. En endelig automat er en "maskin" som alltid befinner seg i en av et endelig antall tilstander. Automaten leser en fast sekvens av instruksjoner. Avhengig av hvilken tilstand automaten er i og hvilken instruksjon som sist ble lest havner automaten i en ny tilstand. Automaten stanser når den når en såkalt slutt tilstand [LP81].

Evaluatoren benytter to invarianter, *currentNode* og *planIndex*, til å holde rede på hvilken beregnings-tilstand treet til enhver tid er i. *currentNode* angir noden med aktiv maskin, og *planIndex* indekserer hvilken instruksjon som er den neste som skal utføres i *currentNode*'s plan. Hver gang en instruksjon skal utføres slår evaluatoren opp i en tabell, *plan*, og henter instruksjonen. Hvis kontroll skal overføres til en ny node beregnes *planIndex* for den nye noden ved oppslag i en tabell, *mapV*, og nodens maskin starter opp i denne tilstanden. De to tabellene *plan* og *mapV* er definert slik:

attributter og overfører kontroll til x, mens x sin maskin beregner en delmengde av x sine syntetiserte attributter og overfører kontroll tilbake til x sin far. x sine attributter blir beregnet i alternerende grupper av arvede og syntetiserte attributter. Evaluering av attributtene til andre symbolinstanser og overførsel av kontroll til andre noder kan forekomme mellom disse operasjonene, men her er vi kun interessert i handlinger som direkte påvirker x sine attributter.

Samarbeidsmønstrene beskrevet over må holde for alle mulige par av nabomaskiner. Ulike instanser av produksjonen `Sum` kan grense mot instanser av alle produksjoner som har forekomster av ikke-sluttsymbolet e, men planen for `Sum` kan ikke variere med hensyn på hvilken produksjonsinstans den grenser mot. Klassen OAG har den egenskapen at det er mulig å konstruere en partiell orden for et hvert symbols assosierte attributter slik at de tilsvarende attributtinstanser kan evalueres i henhold til denne ordenen i enhver kontekst symbolet måtte forekomme. Den partielle ordenen brukes til å partisjonere mengden av hver enkelt ikke-sluttsymbols attributter i en sekvens av disjunkte delmengder hvor delmengdene alternerer mellom delmengder med kun arvede attributter og delmengder med kun syntetiserte attributter. Partisjonene for hvert ikke-sluttsymbol overholder evalueringsordenen på den måten at attributtene i en partisjon, A, som kommer før en annen partisjon, B, kan evalueres før attributtene i B. Planene for hver produksjon konstrueres ved å sammenholde partisjonene for hvert symbol i produksjonen med produksjonens lokale attributtavhengigheter. Flere detaljer om denne prosessen finnes i [Kas80].

Fast-plan evaluatorer karakteriseres slik:

- For hver produksjon, $X_0 \rightarrow X_1 \cdots X_n$, er det en fast sekvens av instruksjoner $I_1, \cdots I_j$.
- Hver instruksjon beregner enten en av X_0's syntetiserte attributter, en av X_i, hvor $1 <= i <= n$, sine arvete attributter eller overfører kontroll til en av X_0 sine naboer. Det er tre slags instruksjoner:
  1. En **Eval** $X_i.a$ instruksjon som evaluerer attributten $X_i.a$. For $i = 0$ må a være syntetisert, ellers arvet.
  2. En **Visit** i, j instruksjon. Overfører kontroll til X_0's i-te barn, X_i, for j-te gang.
  3. En **Suspend** j som overfører kontroll til X_0's forelder for j-te gang. Alle planer avsluttes med en **Suspend** instruksjon.
- For et hvert tre T opererer plan-instansene assosiert med T's produksjonsinstanser som en mengde samarbeidene maskiner.
- Evaluering starter ved å starte opp maskinen assosiert med T's rot, og stanser når denne maskinen når en **Suspend** instruksjon.

Når en **Visit** instruksjon utføres, overføres kontroll til en av høyresidens ikke-sluttsymbol forekomster, og tilsvarende for **Suspend** instruksjoner; kontroll overføres over ikke-sluttsymbol forekomsten

6.4.3 Strukturmodifikasjon etter klippbord paradigmet

Applikasjoner som arbeider med attribuerte trær har ofte behov for å foreta endringer i treet. Hvis for eksempel brukeren av en språkbasert programeditor sletter en linje i et program tilsvarer dette at deler av det attribuerte treet som representerer programmet endres, og må reevalueres. For å foreta reevalueringen kan det benyttes effektive inkrementelle algoritmer som bruker mest mulig av informasjonen som allerede finnes i treet for å minimalisere arbeidet med reevalueringen. Direkte bruk av evalueringsystemer basert på slike algoritmer kan være tungvint og krever en del kunnskaper om algoritmene som brukes og hvordan de er implementert. For å skjerme brukeren av attribueringsystemet mot slike detaljer modeleres strukturendringer av attribuerte trær etter klippbord paradigmet beskrevet i 5.1.1. Punkt 6.6 beskriver inkrementell OAG evaluering av attribuerte trær i AML som følger klippbord paradigmet.

6.5 Evaluering av OAG ved fast-plan evaluatorer

I dette avsnittet beskrives attributtevaluering av klassen OAG ved fast-plan evaluatorer. Først beskrives algoritmen som benyttes. Beskrivelsen av algoritmen er basert på tilsvarende algoritmer beskrevet av Kastens i [Kas80] og av Reps og Teitelbaum i [RT88]. Deretter presenteres en OAG evaluator for AML spesifikasjoner som ligger tett opptil den beskrevede algoritmen. Til slutt, under punkt 6.5.3, beskrives en mer effektiv evaluator basert på såkalte *visit*-funksjoner. Denne evaluatoren er implementert for AML systemet av John H. Reppy ved AT&T, og er beskrevet i [EMR93].

6.5.1 Fast-plan evaluatorer for OAG

En måte å se på evalueringsprosessen ved bruk av fast-plan evaluatorer er å se prosessen som en mengde samarbeidende *maskiner*, en for hver node i treet som skal evalueres. Hver maskin utfører instruksjonene i planen til noden den tilhører, og kontroll overføres til nabonoder når maskinen kommer til en **Suspend** eller **Visit** instruksjon. Når kontroll overføres til en node, x, starter x sin maskin opp og begynner å utføre instruksjoner fra dit den var kommet i x sin plan.

Kontroll kan overføres til en node flere ganger. Hvis x sin maskin overfører kontroll til en nabonode ved instruksjon k, vil maskinen starte opp ved instruksjon $k + 1$ neste gang kontroll overføres til x. Kontroll overføres til barn (**Visit**) for å beregne syntetiserte attributter, og til far (**Suspend**) for å beregne arvede, eller når alle attributtforekomster definert ved noden x er beregnet.

Nabomaskiner må koordineres. Abstrakt syntaksen `Let(_,Sum _, _))` representerer to naboproduksjoner; `Sum _` er barn til `Let _`. La noden som er rot til deltreet `Sum _` være x. Da forekommer x i produksjonen `Let` sin høyre side og `Sum` sin venstre side. x sine syntetiserte attributter beregnes av planen for `Let` mens de arvede beregnes av planen for `Sum`. Generelt oppfører x sin maskin og maskinen til x sin far seg slik: Maskinen til x sin far beregner en delmengde av x sine arvede

Støtte for manipulasjon av attribuerte trær

For å støtte manipulasjon av attribuerte trær er det en egen *copy* funksjon som kopierer attributtposten assosiert med en node. Funksjonen allokerer ny plass til attributtverdiene og kopierer disse. Figur 6.9 viser denne funksjonen for spesifikasjonen fra figur 4.1 utvidet med lokal attributt for `Use`. Applikasjoner som vil benytte *copy* funksjonen kan bruke hash-tabellens `fetch` funksjon for å hente node-verdier som skal kopieres. Noder som skal slettes fra attributt-minnet slettes med hash-tabellens `reset` operasjon.

6.4 Manipulering av attribuerte trær

Attribueringsystemet tilbyr operasjoner for manipulasjon av attribuerte trær, mye på samme måte som `Tree` strukturen beskrevet under punkt 6.2 har operasjoner på rene syntaks trær. De ulike formene for manipulasjon av attribuerte trær beskrives i de påfølgende avsnitt.

6.4.1 Konvertering mellom attribuerte og ikke-attribuerte trær

Den enkleste form for slik manipulasjon er konvertering mellom attribuerte og ikke attribuerte trær. Attribuerte trær representeres som et par av et attributtlager og et syntaks-tre hvor det er tillatt å merke noder. Trærene representeres med datatypen `attributed_term`;

```
datatype attributed_term
    = AttributedTerm of { attr_tbl : attr_tbl, path : path }
```

Navnet på typen og datakonstruktøren henspiller på at det attribuerte treet er et utsagn, eller en *term*, i språket definert av attributtgrammatikken. `attr_tbl` er attributt-minne som beskrevet i 6.3, og defineres slik;

```
type attr_tbl = attr_types hash_tbl
```

Konvertering fra attribuert til ikke-attribuert tre gjøres ved å hente ut tre-komponenten fra det attribuerte treet, mens konvertering andre veien gjøres ved å foreta attributtevaluering av treet. Punkt 6.5 beskriver OAG evaluering i AML for å konvertere syntaks-trær til attribuerte trær.

6.4.2 Navigasjon

Navigasjon i attribuerte trær foregår ved å navigere i treets syntaks komponent ved hjelp av et begrenset utvalg av navigasjonsoperasjonene definert under punkt 6.2. Hash-nøkler for oppslag av attributtverdier kan, om nødvendig, vedlikeholdes inkrementelt ved hjelp av attributtlagerets `up` og `down` operasjoner.

```
val attrRef_e_env : attr_tbl * key * prod_kind -> int environment attr
val attrRef_e_value : attr_tbl * key * prod_kind -> int attr
val attrRef_r_value : attr_tbl * key * prod_kind -> int attr
val attrRef_Use_error : attr_tbl * key -> string option attr

val get_e_env : attr_tbl * key * prod_kind -> int environment option
val get_e_value : attr_tbl * key * prod_kind -> int option
val getLocal_Use_error : attr_tbl * key -> string option
val get_r_value : attr_tbl * key * prod_kind -> int option

val put_e_env : attr_tbl * key * int environment * prod_kind -> unit
val put_e_value : attr_tbl * key * int * prod_kind -> unit
val putLocal_Use_error : attr_tbl * key * string option -> unit
val put_r_value : attr_tbl * key * int * prod_kind -> unit
```

Figur 6.8: *Attributt funksjoner.*

```
fun copy_LOCAL_e (LOCAL_Use{error})=LOCAL_Use{ error = ref (!error)}
  | copy_LOCAL_e _ = LOCALS_e_VOID

fun copy (ATTR_e{value,env,locals})= ATTR_e {
      value = ref (!value), env = ref (!env),
      locals = ref (copy_LOCAL_e (!locals))}
  | copy (ATTR_r{value})= ATTR_r {
      value = ref (!value)}
```

Figur 6.9: *Kopi funksjon for attributter.*

```
datatype attr_types
  = ATTR_e of {
      value : NUMERAL attr,
      env : int environment attr,
      locals : locals_of_e ref
    }
  | ATTR_r of {
      value : int attr}
and locals_of_e
  = LOCAL_Use of {
      error : string option attr
      }
  | LOCALS_e_VOID
```

Figur 6.7: *Disjunkt union for attributtverdier*

En post har et slikt felt hvis det assosierte ikke-sluttsymbolet opptrer på venstre side av produksjoner med lokale attributter.

For hver attributt er det en *put*, en *get* og en *attrRef* operasjon som tar et attributtlager, en oppslagsnøkkel og en produksjonstype som argument. *attrRef* operasjonen gir referansen til en bestemt attributt, og *put* og *get* operasjonene benytter seg av denne operasjonen slik at de henholdsvis kan endre eller dereferere verdien i lokasjonen det refereres til. *attrRef* funksjonen kan brukes direkte for å få ned antall oppslag, for eksempel ved inkrementell evaluering hvor det er behov for å kontrollere som en attributtverdi endrer seg ved reevaluering.

Ved oppslag bruker *attrRef* den underliggende datastrukturens `lookup` funksjon sammen med en standardverdi for noden oppslaget er på, slik at noden initialiseres første gang den slåes opp. For noder med et assosiert `locals` felt er systemet avhengig av å kjenne produksjonen som utledes fra noden slik at `locals` feltet initialiseres med riktig verdi. For eksempel vil første oppslag til en `e` node, med felt som definert i figur 6.7, initialisere `locals` feltet til `LOCALS_e_VOID` for alle produskjoner untatt `Use`.

Figur 6.8 viser *put*, *get* og *attrRef* funksjonene for AML spesifikasjonen i figur 4.1 utvidet med lokal attributt for `Use`. Legg merke til funksjonene for den lokale attributten `error`. Her er det ikke nødvendig å oppgi produksjonen som utledes fra ikke-sluttsymbolet, fordi denne informasjonen er implisitt. Verdiene som returneres av *get* funksjonene har typen `'a option` fordi det i enkelte tilfeller kan være nødvendig å slå opp ikke-definerte attributtinstanser, for eksempel under inkrementell evaluering.

```
signature HASH =
  sig
    type 'a hash_tbl
    type key
    exception UpKey
    val mkTable : unit -> '1a hash_tbl
    val lookup : 'a hash_tbl * key * 'a -> 'a
    val fetch : 'a hash_tbl * key -> 'a option
    val reset : 'a hash_tbl * key -> unit
    val rootKey : key
    val up : key -> key
    val down : key * int -> key
  end
```

Figur 6.6: *Signatur for attributtlagerets underliggende datastruktur.*

reset $(sto, key) \rightarrow$ `()` Nullstiller verdi assosiert med key i sto.

rootKey key Gir en oppslagsnøkkel, key.

up $key_1 \rightarrow key_2$ Gir key_1 sin foreldrenøkkel key_2.

down $(key_1, i) \rightarrow key_2$ Gir nøkkelen til key_1 sitt i-te barn, key_2.

Følgende gjelder for nøklene; `up rootKey` $\equiv \perp$ (reiser `UpKey`), og `up (down (key,`i`))` $\equiv$ `key` for alle i. `rootKey` gir oppslagsnøkkel til roten av et syntaks tre.

Oppslagsnøklene representeres av et heltall og stien som går fra roten til noden. Fra en nøkkel kan nøkkelen til hver nabonode (far eller barn) beregnes i konstant tid ved hjelp av operasjonene `up` og `down`. Oppslagsnøklene kan beregnes inkrementelt ved tregjennomgang.

6.3.2 Grensesnitt mot attributtlager

Verdiene som lagres i datastrukturen er medlemmer av en spesifikasjonsavhengig disjunkt union. Figur 6.7 viser hvordan denne typen er definert for spesifikasjonen i figur 4.1 utvidet med lokal attributt for `Use`.

Hvert ikke-sluttsymbol har en konstruktør for en post med et felt for hver av symbolets attributter, og hvert felt refererer til en oppdaterbar minne-lokasjon hvor verdien enten er definert, `SOME` v, eller udefinert, `NONE`. Konstruktøren for feltene, `attr`, er defineres slik;

```
type 'a attr = 'a option ref
```

Posten for ikke-sluttsybolet `e` har et ekstra felt, `locals`, som skal brukes til å lagre lokale attributter.

treeOf $p \rightarrow t$ t er deltreet som har rot i p's merkede node.

up $p_1 \rightarrow p_2$ Den merkede noden i p_2 er den merkede noden i p_1's forelder.

6.3 Lagring av attributtverdier

Den vanligste måten å lagre attributtverdier er, som tidligere nevnt i under punkt 5.1.1, i syntaks treet. Et av konseptene i AML er at attributtverdier skal lagres i en egen datastruktur, adskilt fra syntaks treet, slik at nodenes representasjon er uavhengig av attributtene, og attribuerte trær enkelt kan konverteres til rene syntaks-trær. Stien fra roten av syntaks-treet brukes som oppslagsnøkkel.

Det er to mulige grensesnitt mot datastrukturen som holder attributtverdiene, et *spesifikasjonsuavhengig* og et *spesifikasjonsavhengig*. I det første må alle attributter ha samme type, og det må være en funksjon for å hente og en for å endre attributtverdiene for en gitt node. I koden for evalueringsreglene må attributtverdiene injiseres og projiseres fra attributt typen. I det spesifikasjonsavhengige må det være en egen funksjon for å sette og hente hver enkelt attributt for hvert ikke-sluttsymbol. Det siste grensesnittet er mer effektivt enn det første fordi det drar nytte av kunnskap om attributten som skal slåes opp ved kompilering.

Sofoklis G. Efremidis opprinnelige håndkodede prototype benyttet et spesifikasjonsuavhengig grensesnitt mot attributtlageret. Dette systemet videreutviklet jeg til en første prototype for attributtlagring i AML systemet. Sommeren 1993 utviklet Khalid A. Mughal og John H. Reppy et attributtlager med et spesifikasjonsavhengig grensesnitt, beskrevet i [EMR93]. Dette systemet har jeg senere foretatt noen mindre endringer i med hensyn på måten lokale attributter takles, og utvidet for å støtte manipulasjon av attribuerte trær. Det er dette systemet for attributtlagring som brukes i de eksisterende attribueringsystemene, og som beskrives i de følgende avsnitt. Først beskrives et grensesnitt mot en datastruktur som brukes som et underliggende lager for attributtverdier, og deretter oppbygningen av det spesifikasjonsavhengige grensesnittet for henting og lagring av attributtverdier.

6.3.1 Underliggende datastruktur

En datastruktur med den abstrakte signaturen gitt i figur 6.6 brukes som lager for attributtverdiene. I den eksisterende implementasjonen er denne strukturen en hash-tabell. `'a hash_tbl` er hash-tabellens type, hvor verdiene som lagres har typen `'a`. Typen `key` er typen til oppslagsnøkklene.

mkTable `()` $\rightarrow sto$ Lager en ny, tom datastruktur sto.

lookup $(sto, key, v_1) \rightarrow v_2$ Slår opp verdi, v_2, assosiert med key i sto. Hvis denne ikke finnes, initialiseres den til v_1, og v_2 settes lik v_1.

fetch $(sto, key) \rightarrow v$ `option` Slår opp verdi v assosiert med key i sto. Hvis denne ikke finnes returneres `NONE`.

```
signature PATH =
  sig
    structure T : TREE
    type path
    exception Down
    exception Up
    val childOrd : path -> int
    val down : path * int -> path
    val eqPath : path -> path -> bool
    val isBase : path -> bool
    val isLeaf : path -> bool
    val left : path -> path
    val leftMostLeaf : path -> path
    val path : T.tree -> path
    val replace : path * T.tree -> path
    val right : path -> path
    val rightMostLeaf : path -> path
    val rootOfPath : path -> T.tree
    val treeOf : path -> T.tree
    val up : path -> path
  end
```

Figur 6.5: *Navigasjon i syntaks-trær med merkede noder.*

defaultProd $s \rightarrow p$ Returnerer en kompletthetsproduksjon, p, utledet fra symbolet s. Denne operasjonen er for bruk i *Cut* operasjoner, som beskrevet i 5.1.1.

Feiltilstanden `Child` reises hvis `nthChild` eller `replaceNth` referer til et barn som ikke finnes. Feiltilstanden `DefaultProd` reises hvis `defaultProd` anvendes på et ikke-sluttsymbol som ikke forekommer i en kompletthetsproduksjon eller et sluttsymbol.

Navigasjon i syntaks trær foregår ved å håndtere en *merket* node. Den abstrakte signaturen i figur 6.5 viser operasjoner for slik navigasjon, og er signaturen til en funktor som tar en struktur med signaturen `TREE` som argument. Som signaturens navn, `PATH`, antyder, er det stien fra roten av treet til noden som er merket som identifiserer den enkelt node - eller gir den sitt unike navn. De viktigste operasjonene er definert slik:

childOrd $p \rightarrow i$ i enumererer hvilket barn nummer den merkede noden i p er til sin forelder.

down $(p_1, i) \rightarrow p_2$ Den merkede noden i p_2 er den merkede noden i p_1's i-te barn.

isBase $p \rightarrow b$ Predikat som forteller om den merkede noden i p er rot-noden.

path $t \rightarrow p$ p er t med merket rot.

replace $(p_1, t) \rightarrow p_2$ p_2 tilsvarer p_1 hvor deltreet med rot i p_1's merkede node er erstattet med t.

```
signature TREE =
  sig
    exception DefaultProd
    exception Child
    type tree
     and prod_kind
     and symb_kind
    val prodKind : tree -> prod_kind
    val symbKind : tree -> symb_kind
    val sameProd : tree * tree -> bool
    val sameSymb : tree * tree -> bool
    val isRoot   : tree -> bool
    val isTerminal : symb_kind -> bool
    val isLeaf   : tree -> bool
    val nChildren : tree -> int
    val children : tree -> tree list
    val nthChild : tree * int -> tree
    val replaceNth : tree * int * tree -> tree
    val defaultProd : symb_kind -> tree
  end
```

Figur 6.4: *Signaturen til strukturen* `Tree`.

sjoner. Operasjonene i signaturen defineres slik:

symbKind $t \rightarrow sb$ Returnerer symbolet, sb, som rot-noden av treet t er merket med.

prodKind $t \rightarrow p$ Returnerer produksjonen, p, utledet fra rot-noden av treet t.

sameProd $(t_1 * t_2) \rightarrow bool$ Predikat som avgjør om samme produksjon utleder trærene t_1 og t_2.

sameSymb $(t_1, t_2) \rightarrow bool$ Predikat som avgjør om samme symbol finnes i roten av to trær, t_1 og t_2.

isRoot $t \rightarrow bool$ Predikat som avgjør om roten i et tre er et AML `root` symbol.

isTerminal $s \rightarrow bool$ Predikat som avgjør om et symbol s er et sluttsymbol.

isLeaf $t \rightarrow bool$ Predikat som avgjør om roten av treet t er et løv.

nChildren $t \rightarrow i$ Gir antall barn, i, utledet fra rot-noden av treet t.

children $t \rightarrow ts$ Gir en liste med barn, ts, utledet av rot-noden av treet t.

nthChild $(t_1, i) \rightarrow t_2$ Gir det i'te barnet, t_2, utledet fra roten av et tre t_1.

replaceNth $(t_1, i, t_2) \rightarrow t_3$ Gir et tre, t_3, hvor t_1's i'te barn er erstattet med t_2.

```
type ID = string
datatype e = Sum of e * e
           | Use of ID
```

Figur 6.2: `type` *og* `datatype` *representasjon*

```
Sum ( Sum ( Use "x", Use "y" ), Use "z" )
```

Figur 6.3: *Syntaks tre*

det spesifiserte språket konstrueres ved hjelp av disse typene. Figur 6.3 viser et gyldig syntaks tre for språket definert i figur 6.1. Merking av nodene blir implisitt; typen forteller hvilket symbol som representeres og datakonstruktøren hvilken produksjon som utledes av noden.

Deklarasjonene som utgjør *Usertypes* modulen har samme innbyrdes rekkefølge som i spesifkasjonen. Dette gjelder også for `datatyper` og `typer` som svarer til `prodtype` og `termtype` definisjoner.

6.2 Manipulasjon av abstrakt syntaks

I trær dannet av datakonstruktører fra `datatyper` og `typer` som svarer til definisjonen av abstrakt syntaks i den opprinnelige AML spesifikasjonen, vil noder som er instanser av ulike symboler ha ulik type. Operasjoner på abstrakte syntaks trær, som navigasjon og manipulasjon, krever et uniformt grensesnitt mot treet slik at alle noder kan behandles på samme måte.

Strukturen `Tree` gir et slikt uniformt, typesikkert grensesnitt mot abstrakt syntaks. Denne strukturen er utviklet for AML systemet av Khalid A. Mughal og John H. Reppy ved AT&T sommeren 1993, og er beskrevet i [EMR93]. Jeg har senere utvidet den med en enkelt operasjon for å gi kompletthetsproduksjoner. Opprinnelig utviklet jeg en slik struktur for AML systemet selv, basert på Sofoklis E. Efremidis håndkodete prototype, men denne strukturen hadde store mangler og er ikke lenger i bruk. Khalid A. Mughal og John H. Reppy hadde likevel en viss nytte av dette arbeidet under utviklingen av sin nye struktur.

Strukturen passer til signaturen gitt i figur 6.4. Typen `tree` spenner over unionen av grammatikksymboler. For AML spesifikasjonen i figur 4.1 defineres `tree` slik;

```
datatype tree = T_e of e
              | T_r of r
              | T_NUMERAL of NUMERAL
              | T_ID of ID
```

Typene `symb_kind` og `prod_kind` enumererer spesifikasjonens grammatikk-symboler og produk-

Kapittel 6

Attribueringsystemer for AML

I dette kapittelet beskrives attribueringsystemene som støttes av AML kompilatoren. Kompilatoren støtter i alt fire ulike attribueringsystemer, alle for klassen OAG. To av systemene er såkalte batch systemer som kun er i stand til å attribuere et ikke-attribuert tre, mens de to andre er inkrementelle systemer som kan brukes til å oppdatere attribuerte trær hvor det er foretatt endringer.

Kompilatoren genererer fire moduler, alle kort beskrevet under punkt 5.2.3. Brukermodulen beskrives under punkt 6.1, tre modulen under punkt 6.2 og attributtlager modulen under punkt 6.3. Disse tre modulene er like for både batch og inkrementell evaluering. Punkt 6.4 beskriver generelt hvordan attribuerte trær skal manipuleres, og fastlegger rammene for operasjonene som skal tilbyes av de ulike evaluator modulene. Systemene for batch evaluering beskrives under punkt 6.5, og systemene for inkrementell evaluering under punkt 6.6. De inkrementelle evalueringsystemene er bygget inn i et abstrakt klippbord som følger klippbord paradigmet beskrevet under punkt 5.1.1.

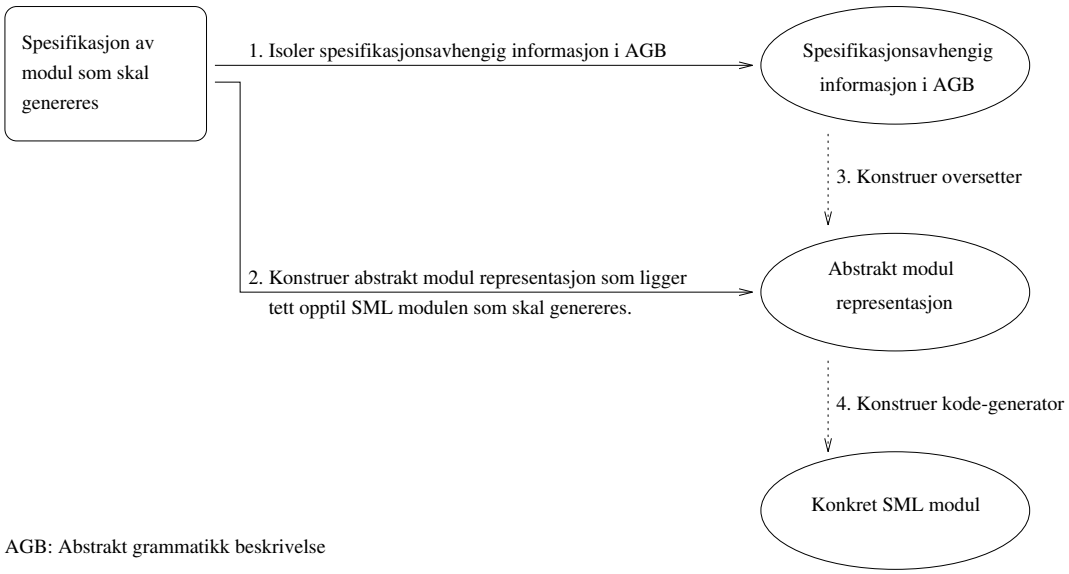
6.1 Brukerdefinisjoner og abstrakt syntaks

En AML spesifikasjon kan inneholde vanlige SML deklarasjoner, som samles i en egen modul kalt *Usertypes*.

Som beskrevet under punkt 5.1.1 representeres attribueringsystemets abstrakte syntaks, gitt ved `prodtype` og `termtype` deklarasjoner i spesifikasjonen, som `datatype` og `type` definisjoner. Disse definisjonene inkluderes også i *Usertypes* modulen. `termtype` og `prodtype` deklarasjonene gitt i figur 6.1 vil oversettes til `type` og `datatype` deklarasjonene gitt i figur 6.2. Syntaks trær for

```
termtype ID = string
prodtype e = Sum of e * e
           | Use of ID
```

Figur 6.1: `termtype` *og* `prodtype` *deklarasjoner*



Figur 5.8: *Konstruksjon av oversettere og kode-generatorer.*

Systemet består av tre slags funksjoner; *suspend*, *visit* og *start*. *Suspend*- og *visit*-funksjonene simulerer henholdsvis **Suspend** og **Visit** instruksjoner, mens *start*-funksjonene skal benyttes for å starte evalueringen. Når de ulike sekvensene er beregnet, konstrueres abstrakte representasjoner av systemets ulike `suspend`-, `visit`- og `start`-funksjoner. Hver `suspend`-funksjon består av alle instruksjoner som skal utføres når kontroll mottas nedenifra fra et bestemt barn (ikke-sluttsymbol) for n-te gang, mens hver `visit`-funksjon består av alle instruksjoner som skal utføres når et bestemt barn (ikke-sluttsymbol) mottar kontroll fra foreldrenoden for n-te gang. `start`-funksjonene bygges opp av alle sekvenser som definerer alle beregninger i hele planen til hvert enkelt ikke-sluttsymbol. Figur 5.7 viser funksjonenes abstrakte representasjon.

Når de abstrakte representasjonene er klare, sendes de videre til evaluator modulens kode-generator som skriver ut sekvensene med `step` som SML kode pakket inn i SML representasjoner av sine respektive `rec_fun` funksjonsrepresentasjoner.

5.3.3 Konstruksjon av oversettere og kode-generatorer

Utgangspunktet for konstruksjon av et par av oversettere og kode-generatorer er SML modulen som skal genereres av det nye paret. Modulen betraktes som spesifikasjonsavhengig informasjon pakket inn i SML syntaks.

Modulens spesifikasjonsavhengige informasjon isoleres, og tilsvarende informasjon identifiseres i den abstrakte grammatikk-representasjonen. For evaluator moduler må eventuell informasjon om evalueringsorden identifiseres i grammatikk-analysatoren. Deretter velges en intern representasjon for denne informasjonen, som ligger tett opptil måten den representeres på i den konkrete SML modulen som skal genereres, men uten å binde representasjonen opp mot SMLs syntaks. Denne representasjonen kalles *modulens abstrakte representasjon*. Datatypene `step` (figur 5.6) og `rec_fun` (figur 5.7) er eksempler på abstrakt representasjon. Deretter konstrueres oversetteren som transformerer informasjon fra spesifikasjonens abstrakte representasjon til modulens abstrakte representasjon, og kode-generatoren som genererer SML kode på grunnlag av denne representasjonen. Figur 5.8 viser punktvis framgangsmåten for konstruksjon av nye oversettere og kode-generatorer.

Et bibliotek av slike par med oversettere og kode-generatorer kan bygges opp, og brukeren av systemet kan 'plugge' ut og inn av back-enden de parene som genererer de spesifikasjonsavhengige modulene i det ønskede attribueringsystem.

```
datatype step
 = EvalRule of {rule:GD.rule,outputs:var list,inputs:var list}
 | ComputeKey of {neighbor:int,key_id:var}
 | ComputeProdKind of {neighbor:int,key_id:var}
 | FetchAttr of {attr:attr,attr_id:var,key_id:var,pk_id:var option}
 | PutAttr of {attr:attr,key_id:var,attr_id:var,pk_id:var option}
 | PosNonterm of {pos_id:var,pos:int}
 | SaveAttr of {attr:attr,key_id:var,attr_id:var,pk_id:var option}
 | Compare of {
    comps:{saveAttr_id:var,attr_id:var,affected_neighbor:var option,
      pos_id:var,put_step:step,pk_id:var,attr:int}list,
      resVal:var}
 | Result of {value:var}
 | VisitChild of {child:(GD.nonterm*int),
    visit:int,key_id:var,pk_id:var,path_id:var}
 | Suspend of {suspend:int,key_id:var,pos_id:var,path_id:var,pk_id:var}
 | Path of {path_id:var,neighbor:int}
```

Figur 5.6: *Abstrakt representasjon av operasjoner for attributtevaluering.*

```
datatype rec_fun
  = SuspendFun of
      {child:GD.nonterm,visit:int,prods:(GD.prod*step list*int) list}
  | Visit of
      {nonterm:GD.nonterm,visit:int,prods:(GD.prod*step list) list}
  | Start of
      {nonterm:GD.nonterm,prods:(GD.prod*step list) list} list
```

Figur 5.7: *Abstrakt representasjon av rekursive funksjoner.*

sentasjon inneholder ikke implementasjonsdetaljer, disse bestemmes av kode-generatoren, slik at det i prinsippet er mulig å skifte ut målspråket for modulene ved å skifte ut kode-generatoren. Evalueringsregler og SML definisjoner representert som abstrakt SML syntaks i den abstrakte grammatikkbeskrivelsen vil være en del av den abstrakte representasjonen av moduler som inneholder slike element. En egen pretty-printer for abstrakt SML kode deles av kode-generatorene for slike moduler. Figur 5.5 viser skjematisk hvordan en spesifikasjonsavhengig modul genereres av back-enden.

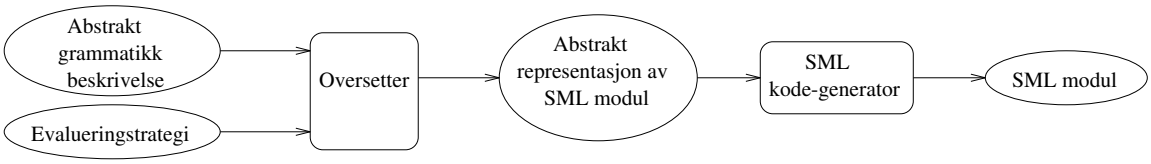
5.3.2 Generering av Evaluator modul

For å illustrere hvordan AML kompilatorens back-end genererer SML moduler beskriver dette avsnittet generering av en evaluator-modul som implementerer gjensidig rekrusive funksjoner for inkrementell evaluering ved hjelp av faste OAG planer. Dette er en evalueringsteknikk som beskrives i detalj i avsnitt 6.6.5 og er kjernen i den mest avanserte OAG evaluatoren den eksisterende AML kompilatoren støtter. Det er ikke nødvendig å kjenne detaljene ved denne teknikken her, kun å være klar over at evaluatoren består av en rekke gjensidig rekursive funksjoner som hver implementerer ulike partisjoner av de forskjellige OAG planene. Hvilke partisjoner som skal inngå i de ulike funksjonene avgjøres etter bestemte regler.

Funksjonene bygges opp som sekvenser av betingede kall til andre rekursive funksjoner og av attributtevalueringer. **Visit** og **Suspend** instruksjoner simuleres ved betingede kall til andre rekursive funksjoner, og **Eval** instruksjoner simuleres ved å slå opp argumentverdiene til evalueringsregelen i attributtlageret, anvende regelen på dem og til sist lagre resultatet. I tillegg oppdateres en intern datastruktur som holder rede på endrede attributtverdier. For å utføre disse operasjonene er det en rekke verdier som hash-nøkkler og produksjonstyper som må beregnes, og funksjonene kan optimeres slik at ingen verdi beregnes mer enn en gang i hver funksjon.

Back-endens første fase, oversetteren, henter ut alle faste planer for alle produksjoner fra grammatikkanalysatoren. Hver plan partisjoneres deretter med hensyn på hver enkelt nabo den kan motta kontroll fra - for eksempel må en plan for en produksjon med 3 ikke-sluttsymbol partisjoneres på tre måter. Hver partisjon oversettes til en sekvens av operasjoner definert ved datatypen `step` i figur 5.6. Sekvensen beskriver hvordan partisjonen skal utføres ved runtime.

De fleste `step`-operasjonene har flere felt av typen `var`. Slike felt holder identifikatorer som enten er bundet til aktuelle parametre til operasjonen, eller skal bindes til operasjonens resultat. Identifikatorer som er bundet til aktuelle parametre ble bundet til resultatet av en operasjon tidligere i sekvensen. `EvalRule` definerer beregning av attributter, og har tre felt; `rule`, `inputs` og `outputs`. `rule` er selve evalueringsregelen, mens `inputs` er identifikatorer som er bundet til regelens aktuelle parametre og er beregnet ved `FetchAttr` steg tidligere i sekvensen. Resulatet av funksjonsanvendelsen bindes til identifikatorene i `outputs` og lagring defineres ved `PutAttr` operasjoner senere i sekvensen. AML tillater multippel attribuering, og dette reflekteres i at `outputs` feltet er en `var` liste.



Figur 5.5: *Generering av SML modul.*

5.3 Back-end design

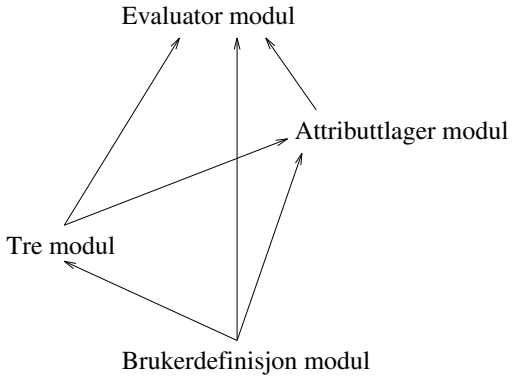
Back-enden er bygget opp av *oversettere* og *kode-generatorer*, et par for hver spesifikasjonsavhengig modul som skal genereres. Oversetteren trekker ut spesifikasjonsavhengig informasjon fra den abstrakte grammatikkbeskrivelsen og kode-generatoren genererer deretter den konkrete SML modulen. Disse fasene beskrives i detalj under punkt 5.3.1. Under punkt 5.3.2 eksemplifiseres dette ved å vise generering av en evaluator modul. Brukere av systemet kan "plugge" ut og inn oversetter og kode-generator par etter behov for å rette back-enden mot ulike attribueringsystemer. Brukere av systemet kan også konstruere slike par selv. Under punkt 5.3.3 beskrives framgangsmåte for å konstruere slike par.

5.3.1 Generering av spesifikasjonsavhengige moduler

Generering av en modul foregår i to faser. I første fase trekker oversetteren ut nødvendige spesifikasjonsavhengig informasjon fra grammatikk beskrivelsen og bearbeider denne til en *abstrakt modul representasjon*. Denne representasjonen sendes videre til modulens kode-generator som genererer den bestemte SML modulen.

I mange tilfeller er oversetterens oppgave triviell fordi front-enden allerede har gjort storparten av arbeidet. Den abstrakte grammatikk-beskrivelsen inneholder blandt annet detaljert informasjon om abstrakt syntaks, som symboler og produksjoner, slik at videreutvikling av abstrakt syntaks til abstrakte representasjoner av attribueringsystemets moduler er trivielt. På den annen side finnes det element som krever omfattende analyse under generering av abstrakte representasjoner. Et eksempel er generering av kode evaluator modulen; når en attributt skal evalueres må argumentverdiene hentes fra en lagermekanisme, anvendes på evalueringsfunksjonen og resultatet lagres i attributtlageret. Henting og lagring av attributtverdier kan involvere beregning av flere hash-nøkler, en evalueringsregel kan, ved faktorisering, være definert over flere produksjoner og inkrementelle evalueringsteknikker kan kreve beregning av hvordan informasjon som endrer seg forplanter seg i treet. Oversetteren må avgjøre hvilke operasjoner som er nødvendige og hvilken rekkefølge de skal utføres i, og den kan forsøke å optimere evalueringskoden for eksempel ved hurtigbuffring av beregnete verdier. Generering av denne modulen studeres i detalj under punkt 5.3.2.

I andre fase genereres konkret SML kode for den bestemte modulen. Modulens abstrakte repre-



Figur 5.4: *Avhengigheter mellom spesifikasjonsavhengige moduler.*

sjonsavhengige SML moduler. Ved run-time kombineres disse med spesifikasjonsuavhengige moduler til et komplett attribueringsystem.

De eksisterende utgavene av back-enden genererer fire moduler:

- *Brukerdefinisjoner.* Denne modulen inneholder alle vanlige SML deklarasjoner fra AML spesifikasjonen samt `type` og `datatype` representasjoner av `termtype` og `prodtype` deklarasjoner. Denne modulen er beskrevet under punkt 6.1.
- *Tre modul.* Modulen definerer et typesikkert grensesnitt mot abstrakt syntaks for språket spesifisert i AML spesifikasjonen. Modulen beskrives i detalj under punkt 6.2 og har signaturen `TREE` definert under samme punkt.
- *Attributtlager modul*. Modulen er beskrevet under punkt 6.3, og definerer et lager for attributtverdier.
- *Evaluator modul.* De eksisterende versjonene av denne modulen støtter klassen OAG. Ulike versjoner for vanlig attributtevaluering er beskrevet under punkt 6.5.2 og 6.5.3, mens versjonene for inkrementell evaluering er beskrevet under punkt 6.6.4 og 6.6.5.

Figur 5.4 viser avhengighetene mellom attribueringsystemets ulike spesifikasjonsavhengige moduler. Modulen med brukerdefinisjoner er uavhengig av de andre modulene, mens det typesikre grensesnittet definert i tre modulen er avhengig av abstrakt syntaks definert i modulen med brukerdefinisjoner. Kjernen i attributtlager modulen er en imperativ datastruktur for lagring av attributtverdier. Verdiene lagres i `ref`-celler og verdier assosiert med samme node lagres sammen. `ref`-cellenes monomorfiske typer hentes fra modulen med bruker-definisjoner, og node-informasjon via det typesikre grensesnittet definert i tre modulen. Evaluator modulen benytter tre modulen for manipulasjon av abstrakt syntaks, attributtlager modulen for lagring av attributtverdier og det semantiske domene definert i modulen med brukerdefinisjoner.

```
attribution e0 : e
    = (Use id)
      with
        rule (error,e0$value) = (case (fetch (id,e0$env))
              of (SOME v) => (NONE,v)
               | _ => (SOME "(* undeclared identifier *)",bogus_value))
        end
```

til evalueringsfunksjonen

```
(fn (s__id__1 : string, a__e0__env__22 : int environment ) =>
      let
        val (l__error__2, a__e0__value__21) =
          ((case (fetch (s__id__1, a__e0__env__22))
            of (SOME v) => (NONE, v) | NONE =>
                (SOME "(* undeclared identifier *)", bogus_value)))
        in (l__error__2, a__e0__value__21) end)
```

Den syntaktiske referansen `id` og attributt referansen `env` parametriseres, mens identifikatorene `fetch` og `bogus_value`, som er bundet til vanlige SML verdier, er frie i funksjonen.

5.2.2 Grammatikk-analysator

Grammatikk-analysatorens oppgave er å analysere en abstrakt grammatikk beskrivelse produsert av front-enden, og generere en statisk evalueringstrategi for denne. Faste planer for OAG, beskrevet i avsnitt 3.2, er et eksempel på en statisk evalueringstrategi. Behovet for en grammatikk-analysator bortfaller hvis målet er et evalueringsystem basert på dynamiske evalueringsteknikker, for eksempel topologisk sortering.

Den eksisterende grammatikkanalysatoren støtter klassen OAG. Analysatoren benytter Kastens 5-stegs algoritme [Kas80] for analyse, og genererer en fast plan for hver produksjon. På tilsvarende måte som front-enden lar etterfølgende faser hente informasjon om spesifikasjonen, har OAG analysatoren operasjoner for å trekke ut informasjon om en grammatikks OAG planer. *analyze* og *planOfProd* er de to viktigste operasjonene;

analyze $g \rightarrow oag$: Gir faste OAG planer, oag, for alle produksjoner i grammatikken g. g er generert av front-enden.

planOfProd $(oag, p) \rightarrow steps$: Gir en OAG plan for produksjonen p. Planen er bygget opp som en sekvens av steg, $steps$, hvor hvert steg er **Eval** rle, **Visit** $(child, i)$ eller **Suspend** i. rle er en attribueringsregel, $child$ er en symbolforekomst i p og i er visit/suspend nummeret.

5.2.3 Back-end

Back-enden oversetter informasjon om grammatikken fra den front-endens abstrakte grammatikk-beskrivelse og analyse-informasjon fra grammatikk-analysatoren til attribueringsystemets spesifika-

5.2.1 Front-end

Oppgaven til front-endens er å oversette en AML spesifikasjon til en bearbeidet utgave av spesifikasjonen, en *abstrakt grammatikk beskrivelse*. En rekke operasjoner tilbyes for å la senere faser i kompilatoren hente ut informasjon om spesifikasjonen fra grammatikk beskrivelsen. De følgende operasjoner viser et representativt utvalg;

prodsOf $g \rightarrow pds$: Gir alle produksjoner, pds, i en abstrakt grammatikk beskrivelse g. g er generert av kompilatorens front-end.

rulesOfProd $p \rightarrow rls$: Gir alle attribuerings regler, rls, for en produksjon p.

lhsOfProd $p \rightarrow nt$: Gir ikke-sluttsymbolet på venstre side av en produksjon p.

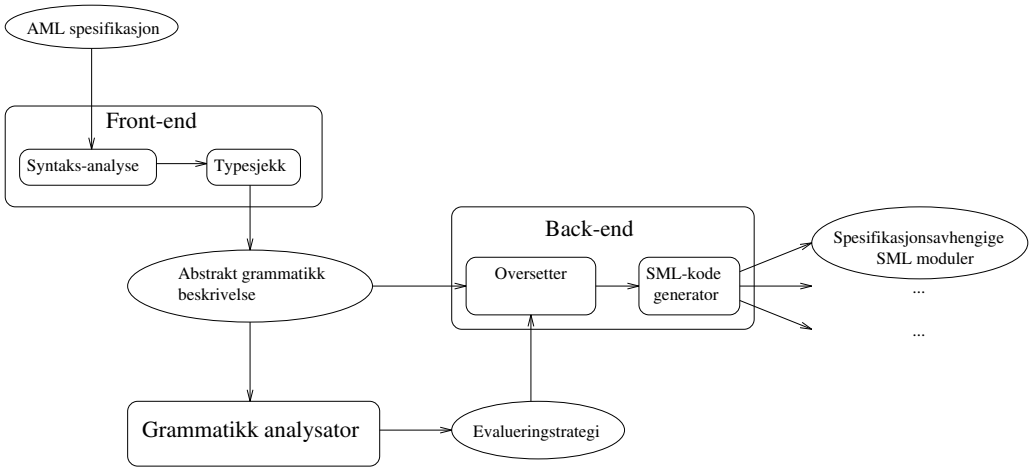
rhsOfProd $p \rightarrow symbs$: Gir høyre side symbolene til en produksjon p. Symbolene kan være både sluttsymbol og ikke-sluttsymbol, og opptrer i samme rekkefølge som i p.

prodsOfNonterm $nt \rightarrow pds$: Gir alle produksjoner, pds, utledet av et ikke-sluttsymbol nt.

Front-endens analyse av spesifikasjonen utføres i to pass. Første pass foretar syntaks-analyse, og består av leksikalsk avsøking og parsing av spesifikasjonen samtidig som det konstrueres et abstrakt syntaks tre. Syntaks treet inneholder abstrakt syntaks for SML-kode fra spesifikasjonen, og definisjonen av denne ligger tett opptil syntaksen for SML beskrevet på sidene 69-73 i [MTH90].

Avsøkeren er konstruert ved help av *ML-lex*[AMT92] og syntaksanalysatoren ved hjelp av*ML-yacc*[AT91]. ML-lex genererer en funktor som er parametrisert på en spesifikasjon av sluttsymboler, generert av ML-yacc. Deler av syntaksen til SML krever at man kan se et ubestemt antall tegn framover, så SML er ikke LALR(1). Fordi AML front-enden bruker ML-yacc til syntaksanalyse, noe som krever en LALR(1) grammatikk, og fordi grammatikken beskrevet i [MTH90] er tvetydig på enkelte steder [MT91, Kah93], aksepterer syntaksanalysatoren en del SML uttrykk med syntaksfeil, og videre syntaktisk analyse er nødvendig. AML kompilatoren foretar ikke slik videre analyse, fordi hovedhensikten med å inkludere abstrakt SML kode er å kunne reprodusere utrykkene som konkret SML kode i en senere fase.

Andre pass foretar typesjekk av syntakstreet, og genererer en abstrakt grammatikkbeskrivelse. Deklarerte symboler, produksjoner og attributter samles, og evalueringsregler analyseres. Under analysen av evalueringsreglene må det holdes rede på bindinger og gyldighetsområder for identifikatorer. En identifikator som ikke referer til en bestemt attributt ved operatoren `$` i en evalueringsregel kan være bundet til en lokal attributt, en syntaktisk referanse eller en ordinær SML konstruktør eller verdi. Fordi verdiene bundet til syntaktiske referanser eller attributter er kontekst-avhengig skrives det abstrakte SML-treet, som representerer en attribueringsregel, om til et SML-funksjonsutrykk hvor referanser til attributter og syntaks er parametrisert. For eksempel oversettes regelen i attribueringsuttrykket



Figur 5.3: *Oppbygning av AML-kompilator.*

Cut $t_1 \rightarrow (t_2, cb)$ **:** Klipper deltre fra t_1 til et klippbord cb. t_2 er det strukturmodifiserte treet.

Copy $t \rightarrow cb$ **:** Kopierer et deltre fra t til et klippbord cb.

Paste $(t_1, cb) \rightarrow t_2$ **:** Kopierer innholdet i et klippbord cb inn som et deltre i t_1. t_2 er det strukturmodifiserte treet.

Deltrærene i operasjonene ovenfor, angies ved å merke roten av det aktuelle deltreet i treet operasjonen skal utføres på. En node merkes ved å angi stien fra roten av treet til noden som markerer roten av det aktuelle deltreet. Figur 5.2 viser skjematisk hvordan et deltre markeres valgt.

Klippene på klippbordet representeres på tilsvarende måte som et attribuert tre; som et par av et ikke-attribuert deltre med en tilhørende datastruktur for attributtverdier. Ved en *Cut* operasjon erstattes deltreet med et tomt deltre, noe som krever at det er definert en kompletthetsproduksjon for hvert symbol i grammatikken.

5.2 AML kompilatoren

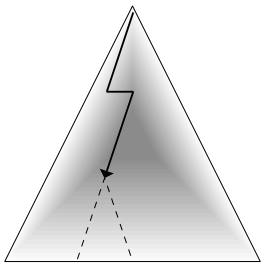
AML kompilatoren er implementert i SML. Kompilatoren er bygget opp modulært for å støtte eksprimentering med ulike grammatikk-klasser og evalueringsteknikker. Den består i hovedsak av tre element; *front-end*, *grammatikk-analysator* og *back-end*. Figur 5.3 viser AML kompilatorens oppbygning. Pilene viser informasjonsflyt, de rektangulære boksene er generatorer og de ovale informasjonstilstander. Front-enden er generell for alle grammatikk-klasser, mens resten av kompilatoren er avhengig av valg av evalueringstrategi og implementasjon av attribueringsystem. De tre delene beskrives i de påfølgende avsnitt.

Konvertering fra ikke-attribuert til attribuert tre gjøres ved å foreta en evaluering av treet, mens konvertering andre veien krever at treets attributtinstanser fjernes. En vanlig måte å representere trær på, og som Synthesizer Generator benytter, er å la nodene i syntaks treet ha felt som holder assosiert informasjon, som attributtverdier og pekere til nabonoder. Denne teknikken gir hurtig tilgang til attributtverdier og gjør det enkelt å navigere i treet eller foreta strukturmodifikasjoner. Nodenes representasjon er imidlertid avhengig av informasjonen som skal lagres i dem, og det er problematisk å konvertere fra attribuerte til ikke-attribuerte trær - rene syntaks trær vil ha en annen representasjon enn attribuerte trær.

Figur 5.1 *a)* viser et attribuert tre, mens figur 5.1 *b)* viser hvordan det er tenkt avbildet i AMLs attribueringsystemer; representert som et par av et rent abstrakt syntaks tre og en egen datastruktur som holder informasjon assosiert med de ulike nodene i treet. Stien fra treets rot til hver enkelt node brukes som oppslagsnøkkel i datastrukturen. Det abstrakte syntaks treet representeres ved hjelp av SML `datatyper` som svarer direkte til `prodtyper` deklarert i den tilhørende AML spesifikasjonen. Abstrakt syntaks representeres på samme måte for attribuerte og ikke-attribuerte trær, og konvertering fra attribuert til ikke-attribuert tre gjøres ved å plukke ut tre-komponenten fra paret som representerer det attribuerte treet.

Strukturmodifikasjon av ikke-attribuerte syntaks trær blir trivielt fordi trærene konstrueres direkte ved SML `datatyper`.

Klippbord paradigmet



Figur 5.2: *Deltre.*

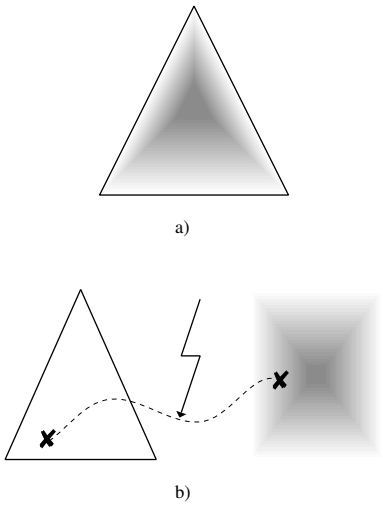
Ved strukturmodifikasjon av attribuerte trær er det hensiktsmessig å følge et *klippbord paradigme*. Dette gjør det mulig å benytte seg av attributtverdiene som finnes i trærene og anvende effektive inkrementelle algoritmer ved oppdatering. Klippbord paradigmet modelleres som operasjoner for å *klippe* og *lime* i attribuerte trær, og beskrives med følgende operasjoner:

1. Attribueringsystemet må være i stand til å beregne en evalueringsorden som respekterer avhengighetene gitt ved grammatikken.

2. Attribueringsystemet må være istand til å lagre attributtverdier som defineres i en beregning, og brukes i en annen.

Mange ulike attribueringsystem kan konstrueres på grunnlag av en enkelt attributtgrammatikk. Forskjellene kan ligge i alt fra hvilken evalueringsteknikk de implementerer og hvordan den er implementert til hvordan minnet håndteres. Videre vil et attribueringsystem som takler en klasse attributtgrammatikker være ute av stand til å takle en annen. Attribueringsystemer inngår ofte i en større applikasjon, for eksempel språkbaserte editorer [RT88], hvor attribuerte og ikke-attribuerte trær manipuleres og inspiseres av andre deler av applikasjonen.

5.1.1 Attribueringsystemer for AML



Figur 5.1: *Representasjon av attribuert tre.*

Et *attribueringsystem for AML* er et system for attributtevaluering i SML konstruert på grunnlag av en attributtgrammatikk formulert som en AML spesifikasjon. For at et attribueringsystem for en AML spesifikasjon skal kunne inngå i en applikasjon som beskrevet under punkt 5.1 må systemet tilby følgende mekanismer[EMR92];

- Mekanismer for konvertering mellom attribuerte og ikke-attribuerte syntaks trær.
- Mekanismer for å assosiere informasjon, som attributtverdier, med tre-noder. Muligheter for å inspisere og endre assosiert informasjon.
- Mekanismer for navigasjon og strukturmodifikasjon av både attribuerte og ikke-attribuerte trær.

Kapittel 5

Generering av attribueringsystemer for AML

AML kompilatoren oversetter en AML spesifikasjon av en attributtgrammatikk til en rekke spesifikasjonsavhengige SML moduler. Ved å kombinere de spesifikasjonsavhengige modulene med spesifikasjonsuavhengige moduler får man attribueringsystemer for attributtevaluering og term-manipulasjon. Kompilatoren er konstruert slik at back-enden enkelt kan rettes mot ulike attribueringsystemer med hensyn på evalueringstrategier, attributtlagringsteknikker og navigasjonsfasiliteter. Kompilatorens konstruksjon støtter gjenbruk av kode for ulike attribueringsystemer.

Dette kapittelet tar for seg generering av attribueringsystemer for AML. Første del av kapittelet gir en generell beskrivelse av hva et attribueringsystem er, og hvordan det relateres til en spesifikasjon av en attributtgrammatikk. Deretter gjøres det rede for hva attribueringsystemer for AML spesifikasjoner er, og en del grunnleggende valg som er gjort med hensyn på implementasjon og grensesnitt mot disse. En del av disse valgene er gjort på forhånd og gjort rede for i [EMR92], og en del av valgene er mine egne. AML kompilatoren er bygget opp modulært for å støtte ulike grammatikkklasser og evalueringsteknikker. Andre del av kapittelet gjør rede for AML kompilatorens ulike komponenter og hvordan disse er satt sammen. Kompilatorens back-end er ansvarlig for å generere attribueringsystemenes spesifikasjonsavhengige kode. Tredje del av kapittelet behandler back-enden spesielt, og legger vekt på å beskrive hvordan den er konstruert for å støtte ulike attribueringsystemer med høy grad av gjenbruk av kildekode.

5.1 Attribueringsystemer

Helt enkelt er et attribueringsystem en implementasjon av et system som kan foreta en attributtevaluering av et gitt tre i henhold til spesifikasjonene i en attributtgrammatikk.

Ved konstruksjon av attribueringsystemer er det to problem som må løses [Kas91];

```
attribution e0 : e
    = (Sum(e1,_) | Let(_,e1,_))
      with
        rule e1$env = e0$env
      end
```

Identifikatorene som bindes på høyre side må være identiske, med hensyn på navn og symbol, for hver produksjon som inngår i den faktoriserte `attribution` deklarasjonen. Kun lokale attributter definert for samtlige produksjoner i faktoriseringsatsen kan refereres.

Syntaktiske referanser og lokale attributter refereres ved navn i semantiske regler. Her vises hvordan den nydeklarerte lokale attributten `error` kan brukes til registrering av feilmeldinger.

```
attribution e0 : e
    = (Use id)
      with
        rule (error,e0$value) = (case (fetch (id,e0$env))
              of (SOME v) => (NONE,v)
               | _ => (SOME "(* undeclared identifier *)",bogus_value)
        end
```

Dette er og et eksempel på *multippel attribuering*, flere attributter defineres ved en enkelt semantisk regel. Alle SML mønstre tillates brukt på venstre side av en regel så lenge identifikatorene er bundet til attributter, lokale attributter eller syntaktiske referanser.

Lokale attributter, attributter som kun er gyldig for enkelte produksjoner, tillates også deklarert. Her

```
attribute Let, Use
    with
      local error : string option
    end
```

defineres den lokale attributten `error` for produksjonene `Let` og `Use`. Legg merke til bruken av faktorisering.

Deklarasjon av abstrakt syntaks og attributter kan kombineres:

```
prodtype ikke-slutt with attrib end
         = Prod of rhs
           with local-attrib end
```

tilsvarer deklarasjonene:

```
prodtype ikke-slutt = Prod of rhs

attribute ikke-slutt with attrib end

attribute ikke-slutt with lokale-attrib end
```

4.3 Semantiske regler

Semantiske regler spesifiseres med nøkkelordet `attribution`. En regeldeklarasjon angir produksjoner og produksjonsymbol ved SML mønster. Attributtforekomster spesifiseres med operatoren `$`.

```
attribution e0 : e
  = (Sum(e1,e2))
    with
      rule e1$env = e0$env
      rule e2$env = e0$env
      rule e0$value = e1$value + e2$value
    end
```

Her bindes identifikatoren `e0` til ikke-sluttsymbolet *e*. I produksjonen *Sum* som utledes fra *e* bindes identifikatorene `e1` og `e2` til symbolene på produksjonens høyre side.

På lignende måte som `attribute` deklarasjoner kan gjelde flere produksjoner, kan semantiske regler også deklareres for flere produksjoner. Slik deling er en praktisk måte for konstruksjon av kompakte spesifikasjoner. For å uttrykke dette benyttes såkalte *or*-mønstre, noe som er en ikke-standard utvidelse av ML og støttes av SML/NJ versjon 0.94 og senere.

4.1 Abstrakt syntaks

En spesifikasjons underliggende abstrakte syntaks utgjør spesifikasjonens kjerne. Denne defineres ved nøkkelordene `termtype`, `prodtype` og `root`. AML vektlegger at det er abstrakt syntaks, slik at kun sluttsymbol som er bærere av semantisk informasjon tillates. Sluttsymbol for som f.eks. vanlig semikolon i Pascal og C utelates. Ved deklarasjon bindes sluttsymbolet til den *monomorfiske* typen som skal være bærer av informasjonen. I eksempelet i figur 4.1 bindes sluttsymbolene `ID` og `NUMERAL` til SML typene `identifier` og `int`. Disse typen er *frie* i spesifikasjonen. Dette betyr at de er definert et annet sted, men fordi alle vanlige SML deklarasjoner tillates kan slike typer godt spesifiseres i selve spesifikasjone.

Ikke-sluttsymbol og produksjoner deklareres ved nøkkelordet `prodtype`. Gjensidig rekursive produksjoner tillates, og må da deklareres med nøkkleordet `and` på samme måte som gjensidig rekursive `datatype` deklarasjoner i SML. For å tillate lister av uttrykk i spesifikasjonen i figur 4.1 kan spesifikasjonen utvides på følgende måte:

```
prodtype e_list
    = EListCons of e * e_list
    | EListNil
and e
    = List e_list
    ...
```

Ikke-sluttsymbol deklarert som `root` er start-symbol, og kan utlede selvstendige syntaks trær. Deklarasjon av `root`-symbol har ingen konsekvenser for andre deler av spesifikasjonen, men er informasjon som er nyttig i andre applikasjoner. Attributteringsystemene evaluerer trær med `root`-symbol i roten.

4.2 Attributter

Attributter kan deklareres for ethvert deklararert ikke-sluttsymbol. Attributtene er bærere av semantiske verdier i syntaks treet og kan ha en hvilken som helst monomorfisk type, og deklareres med nøkkelordet `attribute`. For ikke-sluttsymbol er det to varianter; *arvet* og *syntetisert*. Arvete attributter sender informasjon nedover i syntaks treet, og syntetiserte oppover. I figur 4.1 deklareres det to attributter for ikke-sluttsymbolet `e` og ett for `r`. Her brukes det en `attribute` deklarasjon for hvert ikke-sluttsymbol, men det er mulig å deklarere en attributt for flere ulike ikke-sluttsymbol med en *faktorisert* attributtdeklarasjon. Faktorisert deklarasjon av attributten `value` fra tabell 3.1 kan foretaes slik;

```
attribute r, e
    with
      synth value : int
    end
```

```
grammar Calc =
  struct
  termtype ID = identifier
       and NUMERAL = int
  prodtype r = Root of e
       and e = Let of ID * e * e
                | Const of NUMERAL
                | Use of ID
                | Sum of e * e
  root r
  attribute e
    with
      synth value : int
      inher env : int environment
    end
  and r
    with
      synth value : int
    end
  attribution e0 : e
    = (Sum(e1,e2))
      with
        rule e1$env = e0$env
        rule e2$env = e0$env
        rule e0$value = e1$value + e2$value
      end
    | (Use ID)
      with
        rule e0$value = fetch (e0$env, ID)
      end
    | (Const i)
      with
        rule e0$value = i
      end
    | (Let(ID,e1,e2))
      with
        rule e1$env = e0$env
        rule e2$env = bind (e0$env,(ID,e1$value))
        rule e0$value = e2$value
      end
  and r : r
    = (Root e)
      with
        rule r$value = e$value
        rule e$env = empty
      end
  end
```

Figur 4.1: *Enkel attributtgrammatikk spesifisert i AML.*

Kapittel 4

AML: Attribution in ML

Attribution in ML, eller AML, systemet har et spesifikasjonspråk for spesifikasjon av attributtgrammatikker innenfor et SML-rammeverk. Spesifikasjonspråket utvider SML med konstruksjoner for deklarasjon av kontekstfri grammatikk, deklarasjon av attributter assosiert med grammatikksymboler og produksjoner, og til sist deklarasjon av semantiske regler over attributtforekomster i produksjoner. Vanlige SML deklarasjoner tillates også, og brukes til spesifikasjon av hjelpefunksjoner og grammatikkens semantiske domene.

En AML spesifikasjon ligner på mange måter en SML struktur-definisjon:

`grammar` *navn* `=` `struct` *deklarasjoner* `end`

navn er den spesifiserte grammatikkens navn, og *deklarasjoner* er en sekvens av følgende deklarasjoner:

`termtype` : Deklarasjon av sluttsymbol.

`prodtype` : Deklarasjon av ikke-sluttsymbol.

`root` : Deklarasjon av startsymbol.

`attribute` : Deklarasjon av attributt.

`attribution` : Deklarasjon av semantiske regler.

I tillegg kommer eventuelle SML deklarasjoner. Nøkkelordet `and` tillates brukt på samme måte som i SML for deklarasjon i sekvenser. Figur 4.1 viser attributtgrammatikken definert i tabell 3.1 som AML spesifikasjon. Grammatikken har fått navnte `Calc` fordi den definerer et språk for enkle beregninger, en kalkulator.

ikke treet lenger være konsistent før en delmengde av treets attributtinstanser har blitt tilordnet nye verdier. Reevaluering foregår ved bruk av inkrementelle algoritmer for å sikre hurtig oppdatering, og flere slike algoritmer er beskrevet i bl.a. [RTD83, RT88, Alb91b]. Disse algoritmene er basert på *endringsforplantning*, som vil si at de starter med å reevaluere en mengde antatt inkonsistente attributtinstanser og lar endringer forplante seg til andre attributtinstanser som så må reevalueres.

Anvendelse av språkbasert editering modelert som strukturendringer i syntaks trær inkluderer systemer som interaktive språkbaserte editorer for programmeringspråk [RT88], inkrementelle kodegeneratorer [Mug88] og optimerende kompilatorer [Alb91b].

Produksjoner				
Root: $r \rightarrow e$	*Let:* $e_0 \rightarrow$ *ID* $e_1\ e_2$	*Const:* $e_0 \rightarrow$ *NUMERAL*	*Use:* $e_0 \rightarrow$ *ID*	*Sum:* $e_0 \rightarrow e_1\ e_2$
Eval $e.env$	Eval $e_1.env$	Eval $e_0.value$	Eval $e_0.value$	Eval $e_1.env$
Visit $e, 1$	Visit $e_1, 1$	Suspend	Suspend	Eval $e_2.env$
Eval $r.value$	Eval $e_2.env$			Visit $e_2, 1$
Suspend	Visit $e_2, 1$			Visit $e_1, 1$
	Eval $e_0.value$			Eval $e_0.value$
	Suspend			Suspend

Tabell 3.2: *Evalueringsplaner for den enkle attributtgrammatikken.*

Eval a : Evaluer attributten a.

Visit i, j : Besøk barn i for j-te gang.

Suspend j : Overfør kontroll til forelder for j-te gang.

Visit og **Suspend** instruksjonene overfører kontroll til nabonoder i syntaks treet, mens **Eval** instruksjonen indikerer evaluering av en attributtinstans. Evalueringsordenen gitt ved de faste evalueringsplanene for et gitt syntaks tre T er en total orden som respekterer den partielle ordenen gitt ved $D(T)$.

Kastens har gitt en 5-stegs algoritme for å konstruere slike faste planer[Kas80]. Steg 1 og 2 bestemmer transitive avhengigheter mellom attributtene. Steg 3 partisjonerer hvert ikke-sluttsymbols attributter i sekvenser av alternende mengder av syntetiserte og arvede attributter. Mengdene ordnes slik at ingen attributt er i samme mengde som en attributt den er avhengig av, og en mengde med attributter som, transitivt, definerer andre attributter, kommer før mengdene med avhengige attributter. Steg 4 og 5 kombinerer sekvensene generert i steg 3 med transitive avhengigheter mellom attributtforekomstene i produksjonene, og genererer de faste planene. **Eval** instruksjoner korresponderer med definisjon av attributtforekomster i DO, mens **Visit** og **Suspend** instruksjoner korresponderer med definisjon av attributtforekomster i UO. Kastens gir også flere implementasjonsforslag for run-time evaluering med slike planer.

Grammatikken vist i tabell 3.1 tilhører klassen OAG, e sine attributter har ordningen $env \succ value$. Legg merke til at barna til *Sum* i dette tilfellet besøkes fra høyre mot venstre, men at venstre mot høyre er like riktig. Evalueringen termineres ved første **Suspend** instruksjonen i roten.

3.3 Språkbasert editering og inkrementell evaluering

Attributtgrammatikker har vist seg velegnet til definisjon av språkbaserte editorer [RTD83]. Slike system bruker attribuerte syntaks trær som intern representasjon av kildekoden som editeres, og behandler endringer i koden som strukturendringer i syntaks treets grener. Etter en slik endring vil

3.1.1 Evalueringsteknikker

Det finnes mange ulike evalueringstekinkker for å foreta en attributtevaluering av et syntaks tre. Disse evalueringsteknikkene kan deles i to kategorier; *dynamisk* og *statisk* [Alb91a]. En dynamisk evaluator vil, gitt et syntaks tre T, konstruere $D(T)$ og ut fra denne avgjøre en evalueringsorden for attributtinstansene assosiert med T - for eksempel ved topologisk sortering [Man89]. Statiske evalueringsteknikker baserer seg derimot på statiske avhengigheter mellom attributtforekomstene i hver enkelt produksjon, noe som medfører at en statisk evaluator kan evaluere alle syntaks trær utledet fra samme attributtgrammatikk uten å analysere hver enkelt $D(T)$. Dynamiske evaluatorer avgjør evalueringsordenen under selve attributt evalueringen, mens for statiske er denne avgjort ved konstruksjon av evaluatoren.

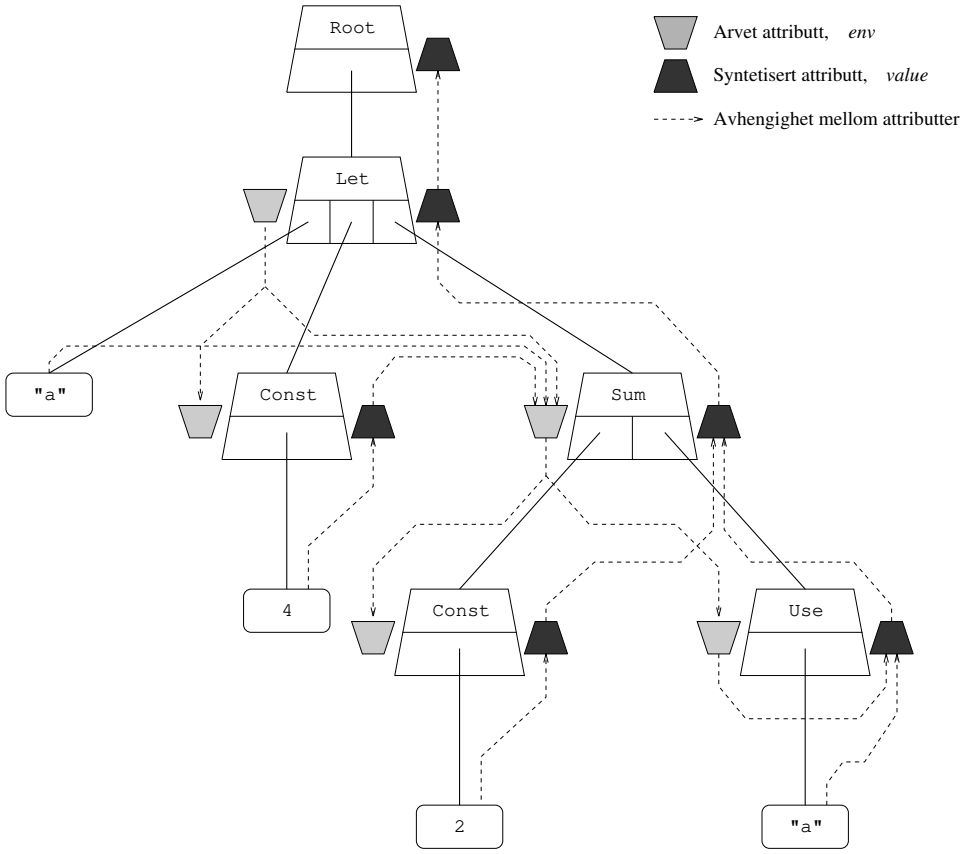
Det går og et skille mellom *batch* og *inkrementelle* evalueringsteknikker. Ved batch evaluering attribueres et ikke-attribuert tre, mens ved inkrementell evaluering reevalueres et ikke-konsistent attribuert tre slik at det blir konsistent. Optimalt gjøres dette på en slik måte at arbeidet er proposjonalt med antall inkonsistente attributtinstanser. Inkrementelle evalueringsteknikker utgjør en viktig del av optimerende kompilatorer hvor optimeringen foregår ved omskriving av syntaks treet [Alb91b], og ved konstruksjon av intergrerte programutviklingsystemer hvor endringer i programmer svarer til endringer av det attribuerte treet [RT88].

Fordi noen evalueringsteknikker er bedre enn andre er det er vanlig å klassifisere en attributtgrammatikk etter den evalueringsteknikken den er best egnet for [Alb91a]. Evaluering ved topologisk sortering kan vanligvis benyttes, men ofte er raskere og bedre metoder tilgjengelig. Syntaks-analysator generatoren *yacc* kan for eksempel benyttes til implementasjon av effektive statiske attributtevaluatorer for attributtgrammatikker med kun syntetiserte attributter som kan evalueres i et enkelt bunn-opp pass [ASU86].

3.2 OAG

Ordnete Attributtgrammatikker, OAG, er en viktig klasse attributtgrammatikker [Kas80]. Denne klassen er så stor at den omfatter mange virkelige programmeringsspråk, samtidig som det er mulig å konstruere meget effektive statiske evaluatorer for den.

OAG karakteriseres ved at det for hvert symbol kan defineres en partiell orden over symbolets assosierte attributter, slik at de tilsvarende attributtinstanser kan evalueres ihenhold denne ordenen i enhver kontekst en instans av symbolet måtte forekomme. Denne egenskapen tillater konstruksjon av *fast-plan* evaluatorer hvor de faste planene består av statisk beregnete evalueringsplaner for hver enkelt produksjon. Planene bygges opp som sekvenser av de følgende tre instruksjoner;



Figur 3.1: *Attribuert tre for uttrykket* `"let a = 4 in 2 + a end".`

representeres ved en *rettet graf* [1] kallt en *attributtavhengighetsgraf*, betegnet $D(T)$, og definert som følger [RT88];

1. For hver attributtinstans b, har grafen en node b'.

2. Forekommer en attributtinstans b på høyre side av en evalueringsregel som definerer en attributtinstans c, har grafen en kant (b', c') rettet fra b' til c'.

Figur 3.1 viser det attribuerte syntaks-treet for uttrykket

```
let a = 4 in 2 + a end
```

utledet fra grammatikken gitt i figur 3.1. De stiplete linjene viser attributtavhengighetsgrafen for uttrykket, legg merke til at referanser til sluttsymbolene er inkludert.

[1] En *rettet graf* $G = (V, E)$ er en mengde *noder* V og en mengde *kanter* E, hvor $E \subseteq V \times V$. Hver kant $(a, b) \in E$ er rettet fra a til b

LHS	Produksjoner				Evalueringsregler		
r	*Root*	r	$\rightarrow$	e	$r.value$	$=$	$e.value$
					$e.env$	$=$	$empty$
e	*Let*	e_0	$\rightarrow$	*ID* e_1 e_2	$e_1.env$	$=$	$e_0.env$
					$e_2.env$	$=$	$bind\ (e_0.env, (ID, e_1.value))$
					$e_0.value$	$=$	$e_2.value$
	Const	e_0	$\rightarrow$	*NUMERAL*	$e_0.value$	$=$	*NUMERAL*
	Use	e_0	$\rightarrow$	*ID*	$e_0.value$	$=$	$fetch\ (e_0.env$,*ID*$)$
	Sum	e_0	$\rightarrow$	e_1 e_2	$e_1.env$	$=$	$e_0.env$
					$e_2.env$	$=$	$e_0.env$
					$e_0.value$	$=$	$e_1.value + e_2.value$

Tabell 3.1: *En enkel attributtgrammatikk.*

3.1 Attributtevaluering

Prosessen å analysere et syntaks tre med hensyn på en attributtgrammatikk kalles *attributtevaluering*. Analysen går ut på å beregne så mange som mulig av treets assosierte *attributtinstanser* ved hjelp av evalueringsfunksjoner som svarer til attributtgrammatikkens spesifiserte evalueringsregler. Evalueringsreglene definerer en mengde begrensinger over attributtinstansene, og analysen kan sees på som å løse et begrensingsproblem; kan attributtinstansene beregnes i henhold til attributtevalueringsreglene definert i attributtgrammatikken?

En node merket X i et syntaks tre har en assosiert mengde attributtinstanser som svarer til attributtene til symbolet X. Et *attribuert tre* er et syntaks tre sammen med nodenes tilhørende attributtinstanser hvor attributtinstansene enten er *definert* ved at de er tilordnet en verdi innenfor den assosierte attributtens domene, eller er *udefinert* og ikke tilordnet noen verdi.

Et *fullt attribuert tre* er et attribuert tre hvor alle tilhørende attributtinstanser er definert. Når alle argumenter til en udefinert attributtinstans er definert er den *klar for evaluering*. Med argumenter til en attributtinstans b menes argumentene til evalueringsfunksjonen som definerer b.

En attributtinstans b er *konsistent* hvis

1. Alle argumenter til b er definert.

2. Verdien tilordnet b er lik evalueringsfunksjonen som definerer b anvendt på b sine argument.

Hvis disse kravene ikke er tilfredstillt er b *inkonsistent*. Et attribuert tre er konsistent hvis det er fullt attribuert og alle attributtinstanser er konsistente [RTD83].

Avhengighetene mellom attributtinstansene i et attribuert tre T, gitt ved evalueringsreglene, kan

En attributt a tilhørende et symbol X betegnes $X.a$. For en attributt a er $TP(\mathrm{a}) \in TS$ mengden av alle mulige verdier for a, altså typen til a.

A er mengden av alle attributter. $A = \cup_{X \in V} A(X)$.

I er mengden av alle arvete attributter. $I = \cup_{X \in V} I(X)$.

S er mengden av alle syntetiserte attributter. $S = \cup_{X \in V} S(X)$.

R er en endelig mengde attributt evalueringsregler assosiert med hver produksjon $p \in P$.

$p : X_{p0} \rightarrow X_{p1} \cdots X_{pn_p}$ sies å ha en *attributtforekomst* (a, p, k) hvis $a \in A(X_{pk})$. Mengden av p sine attributtforekomster betegnes $AO(p)$, og kan deles i to disjunkte delmengder av definerte (*ut*) attributtforekomster og brukte attributtforekomster. $DO(p)$ og $UO(p)$ defineres som følgende;

$$DO(p) = \{(s, p, 0) | s \in S(X_{p0})\} \cup \{(i, p, k) | i \in I(X_{pk}) \wedge 1 <= k <= n_p\}$$

$$UO(p) = \{(s, p, 0) | i \in I(X_{p0})\} \cup \{(i, p, k) | i \in S(X_{pk}) \wedge 1 <= k <= n_p\}$$

Evalueringsreglene i $R(p)$ spesifiserer hvordan attributtforekomstene i $DO(p)$ skal beregnes som funksjoner av andre attributtforekomster i $AO(p)$. Evalueringsregelen som definerer attributtforekomsten (a, p, k) har formen

$$(a, p, k) = f((a_1, p, k_1), \cdots (a_n, p, k_n))$$

hvor $(a, p, k) \in DO(p)$, $f : TP(a_1) \times \cdots TP(a_n) \rightarrow TP(a)$, $f \in FS$
og $(a_i, p, k_i) \in AO(p)$ for $1 <= i <= n$. (a, p, k) sies å være *avhengig av* (a_i, p, k_i) for $1 <= i <= n$. En attributtgrammatikk er på normal form hvis betingelsen $(a_i, p, k_i) \in UO(p)$, $1 <= i <= n$ holder for alle evalueringsregler.

$C(p)$ er en endelig mengde semantiske betingelser assosiert med produksjonen p. Disse er predikat på formen

$$f((a_1, p, k_1), \cdots (a_n, p, k_n))$$

hvor $f : TP(a_1) \times \cdots TP(a_n) \rightarrow \{true, false\}$,
$f \in FS \wedge (a_i, p, k_i) \in AO(p)$ for $1 <= i <= n$.□

Vanlige utvidelser av attributtgrammatikker er *lokale* attributter og *syntaktiske referanser*i attributtevalueringsregler. En lokal attributt er en attributt som er assosiert med en bestemt produksjon, og en syntaktisk referanse er en referanse til et symbol i en produksjon.

Tabell 3.1 gir et uformelt eksempel på en enkel attributtgrammatikk. Det er to ikke-sluttsymbol (e og r), hvorav et startsymbol (r), to sluttsymbol (*ID* og *NUMERAL*) og fem produksjoner (*Root, Let, Const, Use* og *Sum*). Ikke-sluttsymbolet e har to attributter; en syntetisert, $value$, og en arvet, env, mens r bare har en syntetisert, $value$. Grammatikken definerer et enkelt språk for addisjon av heltall og bruk av variabler. Omgivelsen med variabel-deklarasjoner sendes nedover via env attributten, og beregnete verdier sendes opp via $value$ attributten. Legg merke til bruken av syntaktiske referanser til sluttsymbolene og at grammatikken er på normal form.

Kapittel 3

Attributtgrammatikker

En attributtgrammatikk er en kontekstfri grammatikk hvor hvert ikke-sluttsymbol har en mengde assosierte *attributter*. Hver attributt er enten *syntetisert* eller *arvet*. For hver produksjon $p : X_0 \rightarrow X_1...X_n$, er det en endelig mengde assosierte *attributtevalueringsregler* som *definerer* X_0 sine syntetiserte attributter og $X_1...X_n$ sine arvede attributter med hensyn på andre attributter. Attributtene som defineres i p kalles for p sine ut-attributter. Hver evalueringsfunksjon definerer en ut-attributt med hensyn på produksjonens andre attributter.

Enhver attributtgrammatikk kan skrives på *normal form*; evalueringsreglene assosiert med hver produksjon definerer kun ut-attributter, og ingen regel bruker en ut-attributt for å definere en annen ut-attributt. Nedenfor følger en mer formell beskrivelse av attributtgrammatikker. Beskrivelsen er hentet fra [Alb91c].

En attributtgrammatikk er et fem-tuppel $AG = (G, SD, AD, R, C)$

$G = (V, \Sigma, P, S)$ er attributtgrammatikkens underliggende kontekstfrie grammatikk.

- V er et alfabet av symboler.
- Σ er mengden av sluttsymbol. $\Sigma \subset V$.
- P er den endelige mengden av produksjoner. $(V - \Sigma) \times V^*$.
- S er startsymbol. $S \in V - \Sigma$.

$SD = (TS, FS)$ er et semantisk domene.

- TS er en mengde av typer, en endelig mengde av mengder av verdier.
- FS er en endelig mengde av totale funksjoner med domene $type_1 \times \cdots type_n \rightarrow type_0$, hvor $n >= 0$ og $type_i \in TS$, $(0 <= i <= n)$.

$AD = (A, I, S, TP)$ er en beskrivelse av attributter. Hvert grammatikk-symbol $X \in V$ har en mengde, $A(X)$, av attributter som kan deles i to disjunkte delmengder, $I(X)$ og $S(X)$, av arvete og syntetiserte attributter. Attributter fra ulike symbol er ulike, $X \neq Y \Rightarrow A(X) \cap A(Y) = \emptyset$.

På samme måte som strukturer kan bindes til identfikatorer kan signaturer bindes til identifikatorer ved hjelp av *signaturbindinger* - signaturen over er bundet til identifikatoren `SIG`. Signaturen `SIG` angir at `t` er bundet til en `eqtype`, en type som tillater likhetstesting med operatoren =, `x` til en verdi av typen `t` og `f` til en funksjon fra `t` til `t`.

En signatur spesifiserer et grensesnitt mot en struktur. Signaturene kan brukes for kontroll ved deklarasjon av strukturer. Deklarasjonen av `S` kan skrives slik:

```
structure S : SIG = ...
```

SML kompilatoren vil da kontrollere at identifikatorene i signaturen, bindes til verdier av den spesifiserte type i strukturen. En signatur behøver ikke ha spesifikasjoner for alle identifikatorene som bindes i strukturen.

For å angi identifikatorer bundet i strukturer benyttes *kvalifiserte navn*. Et kvalifisert navn til en identifikator er et navn som prefikses med navnet på strukturen den er bundet i. For å angi fakultetsfunksjonen fra `S` brukes det kvalifiserte navnet `S.f`.

2.2.2 Funktorer

En funktor lager strukturer fra andre strukturer. De er funksjoner fra strukturer til strukturer. Kroppen til en funktor definerer en struktur ved hjelp av en formell parameter, spesifisert ved hjelp av en signatur. Applikasjon av en funktor substituerer en aktuell struktur inn i funktorkroppen. Funktorer er til en viss grad polymorfe siden en funktor kan benyttes på ulike strukturer så lenge de passer til signaturen til funktorens formelle parameter. Funktorer kan ikke referere til frie typer og verdier.

Funktorer referer bare til strukturer som formelle parametre. Funktoren vet ikke annet om parameteret enn det som er spesifisert i signaturen. Funktoren

```
functor FUN ( FP : SIG ) : SIG =
  struct
   type t = FP.t * FP.t
   val x = (FP.x,FP.x)
   fun f (x1,x2) = (FP.f x1, FP.f x2)
  end
```

definerer en funktor som tar en struktur, `FP`, med signaturen `SIG` som parameter. Strukturen definert ved funktoren har også signaturen `SIG`. Den definerer en type `t` som er produkt-typen til typen `FP.t` og de tilsvarende operasjonene `x` og `f` til å operere på denne produkt-typen.

En struktur kan gies en *signatur*. På samme måte som verdier har typer, kan en signatur sees på som typen til en struktur. Hvert element i signaturen er en type-spesifikasjon. En struktur sies å passe til en signatur hvis den tilfredstiller spesifikasjonene i signaturen.

En *funktor* beskriver en parametrisert struktur. Strukturen som beskrives i funktorens kropp beskriver en struktur med hensyn på en formell parameter som er abstrakt spesifisert ved en signatur. Ved anvendelse substitueres den aktuelle strukturen med den formelle i funktorens kropp. Funktorer tillater separat kompilering og uttrykker generelle definisjoner.

2.2.1 Strukturer og Signaturer

En struktur er en omgivelse betraktet som et manipulerbart objekt. Strukturer beskrives ved å inkapsle sekvenser av deklarasjoner med nøkkelordene `struct` og `end`. Ett eksempel

```
struct
 type t = int
 val x = 3
 fun f x = if x=0 then 0 else x * f (x-1)
end
```

Denne innkapslede sekvensen er en struktur hvor `t` er bundet til typen `int`, `x` er bundet til verdien `3` og `f` er bundet til fakultetsfunksjonen. Strukturer bindes til identifikatorer ved hjelp av *strukturbindinger*. Strukturen over kan bindes til identifikatoren `S` slik:

```
structure S =
  struct
    type t = int
    val x = 3
    fun f x = if x=0 then 0 else x * f (x-1)
   end
```

Strukturbindinger kan opptre på toppnivå, eller inne i andre strukturer. Typer og verdier som ikke bindes i strukturen sies å være *frie* i strukturen.

Alle strukturer har en signatur. En signatur er en sekvens av spesifikasjoner som angir typen til verdiene identifikatorene er bundet til. Hvis en struktur ikke har gitt en signatur vil SML kompilatoren interferere en signatur som inneholder maksimal informasjon om strukturen. `S` har den maksimale signaturen

```
signature SIG =
 sig
  eqtype t
  val x : t
  val f : t -> t
 end
```

`'a` verdier. Operasjonen `ref : 'a -> 'a ref` allokerer plass til argumentet og returnerer en peker til den allokerte lokasjonen. Operasjonen `! : 'a ref -> 'a` derefererer verdien det pekes på, og operasjonen `:= : 'a ref * 'a -> unit` tilordner lokasjonen en ny verdi.

Svak type variabel

Identitetsfunksjonen er en polymorfisk funksjon. Vi oppretter en referanse til identitet:

```
val fp = ref identitet
```

som gir `fp : ( 'a -> 'a ) ref`. Fordi den er polymorfisk må den kunne anvendes på en hvilken som helst type. Videre tillates tilordning til refererte lokasjoner, og `fp := not` burde være lov. Fordi `fp` referer til en polymorfisk funksjon, må `!fp 42` fremdeles være tillatt, men det er det ikke.

Problemet med `fp` er at innholdet varierer over tid. For å unngå slike problemer, og sikre at typen i den lokasjon det refereres til ikke kan endres, innføres *svake type variable* som navngies i`a`. De svake type variablene spenner over de svake polymorfiske typer, som er alle typer uten sterke type variable. Alle monotyper er svakt polymorfiske.

Operasjonen `ref` får den svakt polymorfiske typen `'1a -> '1a ref`, mens de andre operasjonene beholder sine sterke polymorfiske typer. `ref` spenner over typer som `int -> int ref` og `( '1a * bool ) -> ( '1a * bool ) ref`, og dette må spesifiseres ved bruk. Referanse til identitets funksjonen kan nå deklarareres slik:

```
val fp : ( int -> int ) ref = ref identitet
```

Referanseobjektet begrenses til en monotype ved deklarasjon.

2.1.7 Exceptions

Nøkkelordet `exception` brukes til å deklarere feiltilstander. Et program går inn i en feiltilstand ved at feiltilstanden *reises*. Programmet forblir i feiltilstanden til en egen error-handler rydder opp og lar programmet fortsette som normalt.

2.2 Modularisering av SML programmer

SMLs modulsystem er en mekanisme for å bryte SML programmer ned i mindre mer håndterlige, men relaterte, programenheter. I SML kalles en slik programenhet en *structure*, eller struktur, som er forkortelse for *environment structure*, fordi en struktur bygges opp som en sekvens av deklarasjoner. Deklarasjonene kan inkludere typer, verdier og andre strukturer. Strukturer kan grupperes i større strukturer, og programmer kan organiseres som hirarki av strukturer.

```
val konv : ('a -> 'b) -> 'a tre -> 'b tre
```

Det første tilfellet sier at hvis treet er en intern node, skal verdien konverteres og de to deltrærene konverteres. Det andre tilfellet sier at hvis treet er en `Tupp` skal verdien konverteres, og det tredje og siste at hvis treet er `Tomt` skal ingenting gjøres. _ står for en vilkårlig verdi. `tre1` kan for eksempel konverteres til et `string tre` slik:

```
val tre2 = konv makestring tre1
```

`makestring` er en funksjon med spesifikasjonen `int -> string`.

Lister

En av de viktigste rekursive typene er den polymorfiske liste typen, definert som følger:

```
datatype 'a list = nil | :: of ( 'a * 'a list )
infix ::
```

En liste er enten tom, konstruert ved `nil`, eller et element og en liste, konstruert ved `::`. `infix` deklarasjonen deklarerer `::` som en infix, høyre assosiativ operator. Fordi lister er så vanlige, er det tilatt med følgende alternative liste-notasjon: `[1,2,3]`

Det er definert mange standard liste-operasjoner. Her er noen av de mest brukte:

```
val map    : ('a -> 'b) -> 'a list -> 'b list
val app    : ('a -> 'b) -> 'a list -> unit
```

`map` anvender en funksjon på hvert hvert element av en liste og gir en liste med resultatene, mens `app` kun anvender en funksjon på hvert element av en liste.

Option

En annen nyttig type er typen `'a option`, definert som følger:

```
datatype 'a option = SOME 'a | NONE
```

En verdi av denne typen bærer enten en verdi av typen `'a` eller ingen verdi. Denne typen brukes ofte til å bære pekere som skal endres en eller flere ganger i løpet av en kjøring. `NONE` tilsvarer da en vanlig null-peker.

2.1.6 Referanser

SML støtter imperativene referering og tilordning. Referering er en type-sikker peker, og tilordning er en måte å endre verdien pekeren refererer til (peker på). Typen `'a ref` er typen til pekere til

Overlasting er en *ad hoc* metode for å få operatorer til å behandle ulike typer på ulik måte. Et godt eksempel på dette er addisjonsfunksjonen `+`. Summen av de to heltallene `3` og `2` skrives `3 + 2`, og summen av de to reelle tallene `3.0` og `2.0` skrives `3.0 + 2.0`. Dette ser uskyldig ut, men algoritmene for å utføre de to operasjonene er forskjellig. + symbolet betegner med andre ord ikke en polymorfisk funksjon, men to forskjellige funksjoner hvor valget av hvilken som brukes bestemmes av argumentets type.

2.1.5 Datatyper og mønstergjennkjenning

I tillegg til de grunnleggende typene, er det mulig å definere egne datatyper med egne verdier. Strukturerte verdier kan dekomponeres vha *mønster gjennkjenning* notasjon.

En `datatype` deklarasjon innfører en ny, mulig rekursiv, type. Generelle binære trær kan defineres slik:

```
datatype 'a tre
  = Tomt
  | Tupp of 'a
  | Node of ( 'a * 'a tre * 'a tre )
```

Et tre er enten *tomt*, en *tupp* med en verdi av treets type, eller en node med en verdi og to undertrær. `Tomt`, `Tupp` og `Node` kalles *type konstruktører* og brukes til å konstruere treverdier. Et `int tre` kan for eksempel konstrueres slik

```
val tre1 = Node (1, Tomt, Node (2,Tomt,Tupp 3))
```

og vil få spesifikasjonen

```
val tre1 : int tre
```

`type` deklarasjoner brukes for å gi eksisterende typer synonyme navn. Typen `int` kan for eksempel døpes om til `heltall` slik:

```
type heltall = int
```

På samme måte som det er mulig å konstruere verdier vha. type konstruktørene, er det mulig å dekomponere dem. En funksjon som konverterer et `'a tre` til et `'b tre` kan defineres slik:

```
fun konv f (Node(v,t1,t2)) = Node (f v, konv f t1, konv f t2)
  | konv f (Tupp v) = Tupp (f v)
  | konv _ Tomt = Tomt
```

Funksjonen tar to parameter; en funksjon fra `'a` til `'b` som skal brukes til å konvertere hver enkelt verdi i treet, og et `'a tre`. Funksjonen får spesifikasjonen

eksempel defineres slik:

```
val person = { navn = "Lars", telefon = 55232658 }
```

Hvor `person` får typen { `navn : string, telefon : int` }. De navngitte feltenes rekkefølge er likegyldig. Navngitte felt velges vha. operatoren `#`. Navnet fra `person` kan velges slik:

```
val nn = #navn person
```

2.1.4 Funksjoner og Polymorfisme

Funksjoner deklareres ved nøkkleordet `fun`. Faktoriseringsfunksjonen kan uttrykkes slik:

```
fun fakt n = let
    fun loop (i,resultat) = case i
         of 0 => resultat
          | _ => loop (i-1,i*resultat)
    in
    loop (n,1)
    end
```

I SML uttrykkes iterasjon ved halerekursjon. Fordi det ikke finnes variabler i vanlig forstand, kun verdibindinger, må samtlige invarianter gies eksplisitt som parametre til hver iterasjon. Legg og merke til hvordan lokale bindinger opprettes ved `let` konstruksjonen; `loop` er kun gyldig fra der den deklarares til `end`.

Polymorfisme og overlasting

Enkelte funksjoner er *polymorfe*; de kan anvendes på verdier av flere typer. Eksempel på en slik funksjon er identitetsfunksjonen.

```
fun identitet x = x
```

Denne kan ta et argument av en hvilken som helst type, og returnere det samme elementet. Den har den *polymorfiske* typen $\forall a \cdot a \to a$, hvor a er en typevariabel med domene alle typer. Typevariabler betegnes med en apostrof foran navnet, `identitet` får derfor spesifikasjonen

```
val identitet : 'a -> 'a
```

hvor $\forall$ er implisitt. Typer som har typevariable kalles for *polytyper* og typer som ikke har det, kalles *monotyper*.

Det er viktig å skille polymorfisme fra overlasting. Polymorfiske funksjoner er funksjoner som behandler den parametriserte typen på en uniform måte, funksjonen bryr seg ikke om verdiens type.

Type	Verdier
`unit`	`()`
`int`	`1, 42, 26, ~1`
`bool`	`true, false`
`string`	`"verdi", "test"`

Tabell 2.1: *Vanlige SML typer og noen verdier*

2.1 Kjernespråket

I SML er alt verdier, og verdier med felles egenskaper utgjør en type. Funksjoner er første-ordens dataobjekt, de er verdier på lik linje med alle andre verdier, og kan manipuleres på tilsvarende måte; som argument, resultat eller lagres i variabler. Navn kan bindes til verdier, og det er mulig å definere sine egne typer med egne verdier.

2.1.1 Grunnleggende typer

Figur 2.1 viser en rekke av SMLs grunnleggende typer med noen verdier. Typen `unit` har kun en verdi, `()`, og brukes ofte som resultat-type på funksjoner og uttrykk som beregnes kun for side-effekter eller argument-type når en funksjon ikke har argument. Negative tall prefikses med `~` som også er negasjonsoperator.

2.1.2 Bindinger og Omgivelser

Verdideklarasjoner brukes for å etablere bindinger mellom identifikatorer og verdi. Nøkkelordet for verdideklarasjon er `val`.

```
val x = "Dette er strengen bundet til x" and fire = 4
```

oppretter dynamiske bindinger mellom identifikatorene `x` og `fire` og verdiene `"Dette er strengen bundet til x"` og `4`. Deklarasjonen oppretter samtidig statiske bindinger mellom identifikatoren `x` og typen `string` og identifikatoren `fire` og typen `int`. De to type-bindingene utgjør tilsammen en statisk type-omgivelse, og kan oppsumeres ved spesifikasjonen

```
val x    : string
and fire : int
```

2.1.3 Tupler og Poster

I tillegg til de nevnte typer har SML tupler og poster. Uttrykket `(true, 42)` har produkt-typen `bool * int`. Poster er tupler med navngitte felt. En post for å holde personopplysninger kan for

Kapittel 2

Standard ML

I løpet av de siste ti femten årene har det vært en økende interresse for funksjonell programmering. Det finnes en rekke forskjellige funksjonelle programeringsspråk, og felles for disse er at de baserer seg på konstante deklarasjoner og funksjonell syntaks for deklarasjoner. Et funksjonsuttrykk har den egenskapen at det står for en enkelt verdi. Denne verdien kan ikke endres, bare formen på uttrykket. I imperative programmerinspråk, hvor programmer utføres ved å bevege seg i et tilstandsrom, tillates globale variable og destruktive tilordninger, slik at en funksjon kan endre sin verdi fra det ene kallet til det andre. Dette kalles sideeffekter. Funksjonelle program lages ved funksjonsapplikasjon og funksjonssammensetning. Funksjonelle språk har ikke løkker, men benytter rekursjon isteden, og variable kan ikke oppdateres. Et programmeringspråk som ikke har sideeffekter kalles et *rent funksjonellt språk*.

Etterhvert har det kommet effektive implementasjoner av funksjonelle språk, slik som for eksempel *Standard ML of New Jersey*, SML/NJ, er for språket Standard ML, SML, ([MTH90]). Dette har ført til at funksjonelle språk har blitt mer akseptert som utfordrere til konvensjonelle imperative språk som Pascal og C.

SML er sterkt typet; et hvert uttrykk må ha en entydig bestemt type. SML har et avansert typesystem som gjør at kompilatoren vanligvis er i stand til å dedusere typen til et uttrykk på egenhånd. Typesystemet er polymorfisk, slik at typen som deduseres er mest mulig generell, og kan ofte ta parametre.

I tillegg støttes språket med et avansert modulsystem som gjør at store komplekse program kan deles opp i mindre, mer håndterlige deler.

Dette kapittelet gir en kort introduksjon til SMLs kjernespråk og modulsystem med tanke på SML eksempler i etterfølgende kapittler. Det finnes flere gode introduksjoner til SML. Paulsons bok, *ML for the working programmer* [Pau91], og Harpers *Introduction to Standard ML* [Har93] anbefales. Formell definisjon av SML finnes i [MTH90, MT91].

arbeid skisseres.

1.3 AML Gruppen

Personene som er involvert i AML prosjektet, AML gruppen, er John H. Reppy ved AT&T Bell Laboratories, Sofoklis G. Efremidis ved Cornell University og førsteamanuensis Khalid A. Mughal, Gøran Ellingsen, Tor-Inge Nes og meg selv ved Universitetet i Bergen. Gøran og Tor-Inge er hovedfagstudenter som meg selv, og arbeider med henholdsvis utvikling av brukergrensesnitt for AML spesifikasjoner og realistisk anvendelse av AML systemet.

SML dialekten som benyttes i prosjektet er *Standard ML of New Jersey*, SML/NJ, som er utvilklet ved AT&T. AT&T utvikler kompilatorer for denne dialekten, og på grunn av prosjektets tilknytning til AT&T, har vi alltid vært godt oppdatert med de siste nye versjoner av kompilatoren og tilhørende verktøy, ofte beta-versjoner. Jeg har vært ansvarlig for vedlikehold og oppdatering av denne programvaren i Bergen.

1.1 Målsetting

Utgangspunktet for denne oppgaven var et eksisterende spesifikasjonspråk for AML, en front-end for en kompilator for spesifikasjonspråket og en del enkle håndkodede eksempler på attribueringssystemer for AML. I artikkelen *Attribute Grammars in ML*[EMR92] stilles det i tillegg en del krav med hensyn på implementasjon av attribueringsystemene. Kompilatorens front-end tar seg av syntaks-analyse og typesjekk av spesifikasjonene, og er utviklet av John H. Reppy ved AT&T Bell Laboratories. De håndkodede attribueringsystemene er utviklet av Sofoklis G. Efremidis ved Cornell University. Utførlig status for prosjektet i denne fasen er beskrevet i [EMR92].

Målet med oppgaven er å utvikle en back-end for den eksisterende kompilatoren, og videreutvikle attribueringsystemene for å gi dem hensiktsmessige operasjoner for manipulasjon av det spesifiserte språk. Back-endens oppgave er å generere de spesifikasjonsavhengige delene av attribueringsystemene for spesifikasjonene front-enden har analysert. AML systemet er et eksperimentelt system, slik at back-enden bør tilrettelegges for å gjøre det enkelt å eksperimentere med ulike attribueringsystemer.

1.2 Oppbyggning

Første del av oppgaven gir nødvendig bakgrunnsteori. Programmeringspråket SML er sentralt for denne oppgaven. I tillegg til at spesifikasjonspråket bygger på SML, og at kompilatoren og attribueringsystemene implementeres i dette språket, har jeg valgt å bruket det som algoritme-språk. For å gi nødvendig bakgrunn for å forstå alle eksempler og algoritmer tar kapittel 2 for seg programmeringspråket SML. Kapittel 3 gir bakgrunn om attributtgrammatikker og attributtevaluering. Delen om attributtevaluering er tatt med spesielt med hensyn på attribueringsystemene som behandles senere i oppgaven, mens delen om attributtgrammatikker er av mer generell art. Kapittel 4 tar for seg spesifikasjonspråket for AML, og er et møtepunkt for attributtgrammatikker og SML.

Kapittel 5 tar for seg generering av attribueringsystemer for AML. Første del av kapittelet gir en generell beskrivelse av hva et attribueringsystem er, og hvordan det relateres til en spesifikasjon av en attributtgrammatikk. Deretter gjøres det rede for hva attribueringsystemer for AML spesifikasjoner er, og en del grunnleggende valg som er gjort med hensyn på implementasjon og grensesnitt mot disse. En del av disse valgene er gjort på forhånd og gjort rede for i [EMR92], og en del av valgene er mine egne. Andre del av kapittelet gjør rede for hvordan AML kompilatoren er bygget opp. Tredje og siste del av kapittelet behandler konstruksjon og virkemåte av kompilatorens back-end spesielt.

Kapittel 6 behandler implementasjon av attribueringsystemer for AML spesifikasjoner. Prinsippene beskrevet i første del av kapittel 5 legges til grunn for implementasjonene. Det legges vekt på bruk av effektive inkrementelle algoritmer.

Kapittel 7 oppsummerer en del erfaringer gjort med både kompilatoren og anvendelse av de genererte attribueringsystemene. Til slutt trekkes konklusjoner fra arbeidet med oppgaven og videre

Etter disse generelle bemerkingene om programutvikling vil jeg ta for meg utvikling av programmeringsverktøy ved hjelp av verktøyet AML, eller *Attribution in ML*. AML er et verktøy for spesifikasjon og anvendelse av *attributtgrammatikker* i språkbaserte omgivelser for programutvikling skrevet i programmeringspråket SML (Standard ML). Attributtgrammatikker har vist seg som en velegnet formalisme for spesifikasjon av semantikk og kontekstavhengig syntaks for programmeringspråk, og for implementasjon av språkbaserte omgivelser for programutvikling.

UiB deltar i et samarbeidsprosjekt med AT&T Bell Laboratories for utvikling av AML. AML prosjektet bygger på erfaringer fra Synthesizer Generator[RT88] utviklet ved Cornell University og Pegasus-prosjektet ved AT&T Bell Laboartories[RG86].

Synthesizer Generator er en generator for språkbaserte omgivelser for programutvikling. På samme måte som en syntaks-analysator generator, som f.eks. *yacc*, genererer en syntaks-analysator på grunnlag av en spesifikasjon av et språks konkrete syntaks, vil Synthesizer Generator genrere en språkbasert programeditor på grunnlag av en spesifikasjon av et språks syntaks og semantikk. Synthesizer Generator prosjektet la stor vekt på bruk av effektive inkrementelle algoritmer i de genererte systemene. En inkrementell algoritme er en algoritme som forsøker å gjøre nytte av gamle data ved beregning av nye data.

Pegasus-prosjektet forente høynivå brukergrensesnitt i interaktive systemer med generator-verktøy for å forenkle konstruksjon av språkbaserte omgivelser for programutvikling. Parallellitet kombinert med høynivå konstruksjoner fra SML ble brukt for å modelere samhandling mellom systemets ulike komponenter. En viktig konklusjon trukket på grunnlag av erfaringer med dette prosjektet var at spesifikasjonspråket og målspråket for generator-baserte systemer bør ha et felles base-språk.

SML er et funksjonellt programmeringspråk. Funksjonelle språk kjennetegnes ved at de baserer seg på konstante deklarasjoner og funksjonell syntaks for deklarasjoner. Et funksjonsuttrykk har en enkelt verdi, og denne kan ikke endres - kun formen på uttrykket. Program lages ved funksjonsapplikasjon og funksjonssammensetning, og funksjoner er første ordens dataobjekter - de kan behandles på samme måte som alle andre verdier. Den funksjonsorienterte uttrykksformen er svært kraftig, og gjør at språket er godt egnet til spesifikasjon av semantiske definisjoner og høynivå operasjoner.

AML systemet består av et spesifikasjonspråk for attributtgrammatikker, og en kompilator. Kompilatoren tar en spesifikasjon og genererer et system for konstruksjon og manipulasjon av abstrakt syntaks samt en evaluator for å beregne semantikk for uttrykk i språket som er beskrevet i spesifikasjonen. De genererte delene kan så kombineres med spesifikasjonsuavhengige deler til et system for ulike former for manipulasjon av uttrykk i det spesifiserte språket, et *attribueringsystem*. Systemets base-språk er SML; spesifikasjonspråket er en utvidelse av SML og attribueringsystemene implementeres i SML. Semantisk og syntaktisk informasjon for et språk, som et programmeringsverktøy skal utvikles for, spesifiseres i spesifikasjonspråket. Attribueringsystemet som AML systemets kompilator genererer på grunnlag av spesifikasjonen inngår så som en del av et språkbaserte verktøyet som skrives i SML.

Kapittel 1

Innledning

Programmerere møter mange praktiske problemer under utvikling og vedlikehold av store programsystemer. Det er derfor viktig å finne fram til metoder hvor datamaskinen kan anvendes for automatisk å støtte opp om programutvikling og vedlikehold av programvare. Utvikling av slike metoder har vært et aktivt forskningsfelt de siste ti-femten år. Feltet kan deles i tre hovedgrupper:

- Utvikling av bedre språk som gjør det enklere og raskere å utvikle og vedlikeholde programvare.
- Finne fram til bedre metoder og strategier for utvikling av programvare. Dette vil resultere i kortere utviklingstid for programvare og programvare som er enklere å forstå og vedlikeholde. Tidligere fokuserte en på strukturerte metoder for å systematisere utviklingsprosessen, nå skifter interessen mer mot objektorienterte metoder.
- Utvikling av bedre programmeringsverktøy. I tradisjonelle programutviklingsomgivelser brukes mye tid på trivielle, men nødvendige, oppgaver. Gode verktøy kan ta seg av mange av disse oppgavene og frigjøre tid til mer produktivt arbeide.

Utvikling av bedre programmeringsverktøy er uten tvil kjerneområdet innenfor dette feltet, men kan ikke sees isolert fra utviklingen innen språk og metode som stadig stiller høyere krav til programmeringsverktøyene.

Stor forskningsinnsats legges ned i utvikling av *språkbaserte omgivelser for programutvikling*. Dette er interaktive verktøy som søker å nyttigjøre informasjon om språket på en slik måte at programmereren blir, "on-line", gjordt oppmerksom på sine feil. Denne kunnskapen kan nyttigjøres på flere måter:

- Kjennskap til språkets syntaks kan brukes for å finne syntaktiske feil.
- Kjennskap til språkets semantikk kan benyttes til effektiv feilsøking eller generering og oppdatering av kode under editering.

Tabeller

2.1 *Vanlige SML typer og noen verdier* 16

3.1 *En enkel attributtgrammatikk.* 25

3.2 *Evalueringsplaner for den enkle attributtgrammatikken.* 28

6.1 *OAG planer partisjonert for inkrementell evaluering ved rekursive funksjoner.* 73

7.1 *Kompilatorstørrelse, batch evaluering* 79

7.2 *Kompilatorstørrelse, inkrementell endelig automat evaluering* 79

7.3 *Kompilatorstørrelse, inkrementell evaluering ved rekrusive funksjoner.* 79

7.4 *Antall beregnede hash-nøkler og hash-oppslag for inkrementell evaluering ved endelig automat og rekursive funksjoner.* 81

6.16 *Abstrakt klippbord.* 63
6.17 *Visit-sekvensbasert endringsforplantingsalgoritme* 66
6.18 *Intruksjoner for endelig automat for inkrementell evaluering av OAG.* 67
6.19 *Semantisk funksjon for beregning av* `value` *og* `error` *attributter i inkrementellt system.* 68
6.20 *Kontrollflyt ved attribuering av syntaks tre ved bruk av* `visit`*-funksjoner* 69
6.21 `visit-` *og* `suspend`*-funksjoner for OAG plan.* 71
6.22 *Kontrollflyt ved inkrementell evaluering ved rekurisve funksjoner.* 72
6.23 *Rekursive funksjoner.* 74
6.24 *Startfunksjon for produksjonen* `Sum`. 75
6.25 `suspend`*-funksjon for* `Sum` *som mottar kontroll fra barn 2.* 76

7.1 *Komposisjon av attribueringsystemer.* 78
7.2 *Intern/ekstern syklus.* 80

Figurer

3.1 *Attribuert tre for uttrykket* `"let a = 4 in 2 + a end".` 26

4.1 *Enkel attributtgrammatikk spesifisert i AML.* 31

5.1 *Representasjon av attribuert tre.* 36
5.2 *Deltre.* 37
5.3 *Oppbygning av AML-kompilator.* 38
5.4 *Avhengigheter mellom spesifikasjonsavhengige moduler.* 41
5.5 *Generering av SML modul.* 42
5.6 *Abstrakt representasjon av operasjoner for attributtevaluering.* 44
5.7 *Abstrakt representasjon av rekursive funksjoner.* 44
5.8 *Konstruksjon av oversettere og kode-generatorer.* 46

6.1 `termtype` *og* `prodtype` *deklarasjoner* 47
6.2 `type` *og* `datatype` *representasjon* 48
6.3 *Syntaks tre* 48
6.4 *Signaturen til strukturen* `Tree`. 49
6.5 *Navigasjon i syntaks-trær med merkede noder.* 50
6.6 *Signatur for attributtlagerets underliggende datastruktur.* 52
6.7 *Disjunkt union for attributtverdier* 53
6.8 *Attributt funksjoner.* 54
6.9 *Kopi funksjon for attributter.* 54
6.10 *Algoritme for attributtevaluering av OAG.* 58
6.11 *Implementasjon av endelig automat for evaluering av OAG.* 59
6.12 *Instruksjoner for endelig automat* 60
6.13 *Utsnitt av* `plan-` *og* `mapV-tabeller` 60
6.14 *Semantisk funksjon for beregning av* `value` *og* `error` *attributter.* 61
6.15 *Visit-funksjon for produksjon* `Sum`. 62

7 Konklusjon **77**

7.1 Diskusjon 77

7.1.1 Gjenbruk av kompilatorkode 77

7.1.2 Kobling av attribueringsystemer mot språkbasert strukturediter 80

7.1.3 Sammenligning av inkrementelle evalueringsteknikker for klassen OAG . . 81

7.2 Konklusjoner 82

7.2.1 Forslag til videre arbeid 83

Litteratur **85**

4.3 Semantiske regler 33

5 Generering av attribueringsystemer for AML 35

5.1 Attribueringsystemer 35

5.1.1 Attribueringsystemer for AML 36

5.2 AML kompilatoren 38

5.2.1 Front-end 39

5.2.2 Grammatikk-analysator 40

5.2.3 Back-end 40

5.3 Back-end design 42

5.3.1 Generering av spesifikasjonsavhengige moduler 42

5.3.2 Generering av Evaluator modul 43

5.3.3 Konstruksjon av oversettere og kode-generatorer 45

6 Attribueringsystemer for AML 47

6.1 Brukerdefinisjoner og abstrakt syntaks 47

6.2 Manipulasjon av abstrakt syntaks 48

6.3 Lagring av attributtverdier 51

6.3.1 Underliggende datastruktur 51

6.3.2 Grensesnitt mot attributtlager 52

6.4 Manipulering av attribuerte trær 55

6.4.1 Konvertering mellom attribuerte og ikke-attribuerte trær 55

6.4.2 Navigasjon 55

6.4.3 Strukturmodifikasjon etter klippbord paradigmet 56

6.5 Evaluering av OAG ved fast-plan evaluatorer 56

6.5.1 Fast-plan evaluatorer for OAG 56

6.5.2 Implementasjon av endelig automat-basert OAG evaluator 59

6.5.3 Implementasjon av OAG evaluator basert på rekursive funksjoner 61

6.6 Abstrakt klippbord 63

6.6.1 Cut, Copy og Paste 63

6.6.2 Klippbordparadigmet og inkrementell evaluering 63

6.6.3 Inkrementell reevaluering ved endringsforplantning og faste planer 64

6.6.4 Endelig automat-basert inkrementell OAG-evaluator 65

6.6.5 Inkrementell evaluering av OAG med rekursive funksjoner 67

Innhold

1 Innledning **11**

1.1 Målsetting 13

1.2 Oppbyggning 13

1.3 AML Gruppen 14

2 Standard ML **15**

2.1 Kjernespråket 16

2.1.1 Grunnleggende typer 16

2.1.2 Bindinger og Omgivelser 16

2.1.3 Tupler og Poster 16

2.1.4 Funksjoner og Polymorfisme 17

2.1.5 Datatyper og mønstergjennkjenning 18

2.1.6 Referanser 19

2.1.7 Exceptions 20

2.2 Modularisering av SML programmer 20

2.2.1 Strukturer og Signaturer 21

2.2.2 Funktorer 22

3 Attributtgrammatikker **23**

3.1 Attributtevaluering 25

3.1.1 Evalueringsteknikker 27

3.2 OAG 27

3.3 Språkbasert editering og inkrementell evaluering 28

4 AML: Attribution in ML **30**

4.1 Abstrakt syntaks 32

4.2 Attributter 32

Forord

Når jeg nå endelig er ferdig med denne oppgaven vil jeg benytte anledningen til å takke veilederen min, førsteamanuensis Khalid A. Mughal. Han har alltid vært tilgjengelig, og har hatt mye å bidra med.

En stor takk går også til samboeren min, Hilde, som har holdt ut mye "spennende" snakk om attributtgrammatikker. Hun har alltid hatt noe oppmuntrende å si når arbeidet har stått i stampe.
Min far har kommet med mange gode råd i forbindelse med skrivingen.

Hovedfagstudentene ved instituttet har gjort studietiden til en uforglemmelig opplevelse.

Lars Søraas,
September 1994

Generering av attribueringsystemer for AML spesifikasjoner

Hovedfagsoppgave

Lars Søraas
Institutt for Informatikk
Universitetet i Bergen

26. september 1994