Innhold

1 Innledning **7**

1.1 Hensikten med oppgaven 7

1.2 Hvorfor et kompleksitetsmål? 7

1.3 Hva bidrar til kompleksitet? 8

1.4 Krav til kompilatoren vi skal bruke som basis 8

1.5 Free Software Foundation og GNU prosjektet 9

1.6 Språket i denne oppgaven 9

2 Om kompleksitetsmål **11**

2.1 Kildekodens tekstlige størrelse 12

2.2 Halsteads Software Science 12

2.2.1 Kildekodens volum og nivå 13

2.2.2 Anstrengelse 13

2.3 Kontrollflyt 14

2.4 McCabes kompleksitetsmål 15

2.5 Modularisering 17

2.6 Psykologiske teorier 17

2.7 Programforståelse ved hypotesetesting 18

2.8 Sammendrag 19

3 Kompleksitetsmålet **20**

3.1 Kompleksitet fra hvilken synsvinkel 20

3.2 Hva gjør det vanskelig å forstå en tekst 21

3.3 Symboler 21

3.4 Et mål for informasjonsmengde 23

3.5 Informasjonsmengde og kompleksitet 25

3.6 Symbolkategorier 26

3.7 Kompleksiteten av et segment 27

3.8 Kompleksiteten av en programtekst 28

3.9 Entropi 28

3.10 Følsomhet for struktur 29

3.10.1 Tradisjonell programmering 29

3.10.2 Objektorientert programmering 29

3.11 Test av kompleksitetsmålet 30

3.12 Sammendrag 30

4 Litt kompilatorteknikk 31
4.1 Kompilatorer generelt 31
4.2 Kontekst-frie grammatikker 31
4.3 Et standard design for kompilatorer 34
4.4 Kompilatorens front-end 34
4.5 Bottom Up parsing 35
4.6 BISON — et kompilatorverktøy 36
4.7 Oversettelsesskjema 37

5 Kompleksitetsmålet i lys av C++ 39
5.1 Bruk av identifikatorer i C++ 39
5.2 Symbolkategorier for C++ 40
5.3 C++ programtekst 43
5.4 Seksjonsbegrepet 43
5.5 Språkets seksjonskonstruksjoner 44
5.6 Eksterne deklarasjoner 46
5.7 Eksterne definisjoner 47
5.8 Prosedyredefinisjoner 48
5.9 Sammensatte setninger 49
5.9.1 Nesting av sammensatte setninger 50
5.10 Klassedefinisjoner 51
5.10.1 Foroverreferanser i attributtdeklarasjoner 53
5.11 Andre eksterne definisjoner 54
5.11.1 Enumeratordefinisjoner 54
5.11.2 Typedef definisjoner 55
5.12 Aksess til objekters attributter 55
5.13 Sære detaljer ved C++ 56
5.14 Headerfiler 57
5.14.1 Konsekvenser for kompleksitetsmålet 57
5.15 Overloading 58
5.15.1 Riktig bruk av overloading 59
5.15.2 Overloading av operatorer 59
5.15.3 Overloading i lys av kompleksitetsmålet 60
5.15.4 Redefinisjon av kardinalitet 60
5.16 Standard prosedyreargumenter 61
5.17 Preprosessering 62
5.18 Sammendrag 63

6 Metode 64
6.1 Hovedidé 64
6.2 Relativ og absolutt kompleksitet 65
6.3 Kompositt kompleksitet 66

6.4 Assimilasjon . . . 68
6.5 Strategi . . . 68

7 Implementasjon 70
7.1 Generelt om G++ . . . 71
7.2 Implementasjon i BISON . . . 71
7.3 Utnyttelse av funksjonaliteten i BISON . . . 72
7.4 Endringer på BISON runtime-kjerne . . . 75
7.5 Modifikasjoner på leksikalsk analysator . . . 77
7.5.1 Registrering av operatorforekomster . . . 79
7.5.2 Symbolforekomster i headerfiler . . . 80
7.6 Modifikasjoner på parser . . . 80
7.6.1 Sammensatte setninger . . . 81
7.6.2 Prosedyredefinisjoner . . . 81
7.6.3 Klassedefinisjoner . . . 82
7.6.4 Prosedyrekropper i klassedefinisjoner . . . 82
7.6.5 Aksess til objekters attributter . . . 84
7.7 Beregning av alfabetets størrelse . . . 84
7.7.1 Betraktninger omkring alfabetets størrelse . . . 85
7.7.2 G++ kompilatorens interne tabellverk . . . 86
7.7.3 Opptelling av deklarasjonslageret . . . 89
7.7.4 Deklarasjoner fra headerfiler . . . 89
7.7.5 Kompilatorgenererte deklarasjoner . . . 90
7.7.6 Overloading . . . 91
7.7.7 Opptelling av attributter . . . 91
7.7.8 Literaler . . . 92
7.8 Sammendrag . . . 93

8 Eksperimenter med kompleksitetsmålet 94
8.1 Observasjonene . . . 95
8.2 Variasjoner i kompleksitet . . . 95
8.2.1 Analyse av resultatene for Meta . . . 96
8.2.2 Analyse av resultatene for Nora . . . 97
8.3 Variasjoner i entropi . . . 98
8.4 Modulorientert bruk av headerfiler . . . 99
8.5 Målet relatert til intuitiv kompleksitet . . . 99
8.6 Ustrukturert ekvivalent . . . 100
8.6.1 Optimal prosedyreinndeling for G++ . . . 102
8.7 Sammendrag . . . 102

9 Diskusjon 107
9.1 Målets struktursensitivitet . . . 107
9.1.1 Begrensning av alfabetet . . . 107
9.1.2 Prosedyrenavn som egen symbolkategori . . . 108

9.2 Informasjon og kompleksitet 109
9.2.1 Løkker 110
9.2.2 Et modifisert kompleksitetsmål 110
9.2.3 Implementasjon av det modifiserte målet 111
9.3 Støtte fra andre eksperimenter 112
9.4 Organisering av kildekode 113
9.5 Betraktninger om valget av C++ 114
9.5.1 Effekten av preprosessering 115
9.6 Videre arbeid 115
9.6.1 Målinger på flere systemer 115
9.6.2 Videre vurdering av målet 116
9.6.3 Eksperimenter med modifisert kompleksitetsmål 116
9.7 Konklusjon 116

A Notasjon **119**

B Operatorer og reserverte ord i C++ **120**

C Kompleksitetsfilens format **121**

D Kildekode for modifikasjonene på G++ **124**

Forord

Dette er en hovedfagsoppgave til Cand.Scient. graden i studieretning databehandling ved Institutt for Informatikk (IFI), Universitetet i Oslo. Arbeidet med oppgaven har pågått i perioden fra februar 1992 til mai 1993.

Bakgrunnen for oppgaven er et mål for kompleksitet av kildekode utviklet av min veileder, Arne Maus ved IFI. Dette målet trengte å bli testet i praksis på store mengder kildekode, så det var ønskelig å få laget et program som automatiserer beregningen av målet. Hovedvekten av oppgaven er lagt på å vise hvordan man kan implementere målet i en eksisterende kompilator slik at kompilatoren beregner kompleksitet for den kompilerte kildekoden.

Det finnes standard teknikker og verktøy for å konstruere kompilatorer, så mange kompilatorer ser ganske like ut innvendig. Vi presenterer en metode for implementasjon av målet i en slik "standard" kompilator. Metoden ble brukt til å modifisere en kompilator for programmeringsspråket C++. Mye av arbeidet på kompilatoren besto i å analysere virkemåten slik at vi kunne utnytte eksisterende datastrukturer og dermed minimalisere modifikasjonene. Det var også nødvendig å reflektere over hvordan den generelle definisjonen av kompleksitetsmålet skulle anvendes på C++ kildekode.

Den modifiserte C++ kompilatoren ble brukt til innledende målinger på to programsystemer. Dette ga oss et tallmateriale for en vurdering av kompleksitetsmålets sterke og svake sider. Resultatene av disse målingene ga ingen entydig støtte til målet, men ga innblikk i målets svakheter slik at vi kunne formulere noen forslag til forbedringer. Her følger en oversikt over oppgavens kapittelinndeling:

Kapittel 1 gir en innledning til problemstillingen.

Kapittel 2 gir en kort presentasjon av alternative kompleksitetsmål fra litteraturen.

Kapittel 3 presenterer Maus' kompleksitetsmål i generelle termer uten tilknytning til noe bestemt programmeringsspråk.

Kapittel 4 gir en oversikt over de teknikker og verktøy som ofte benyttes ved kompilatorkonstruksjon.

Kapittel 5 reflekterer over hvordan kompleksitetsmålet bør anvendes på C++ kildekode.

Kapittel 6 presenterer en generell metode for implementasjon av kompleksitetsmålet i kompilatorer som er utviklet med bestemte kompilatorverktøy.

Kapittel 7 viser hvordan vi brukte metoden til å implementere målet i C++ kompilatoren G++. G++ er utviklet med kompilatorverktøyet BISON, og egner seg godt som utgangspunkt for en implementasjon etter metoden.

Kapittel 8 presenterer resultater av målinger som ble utført på to programsystemer ved hjelp av den modifiserte kompilatoren.

Kapittel 9 presenterer mulige forbedringer av målet i lys av resultatene fra kapittel 8.

Til slutt vil jeg gjerne takke veileder for alle gode råd under arbeidet med en interessant og lærerik oppgave.

Blindern, mai 1993

Stein Jørgen Ryan

Kapittel 1

Innledning

1.1 Hensikten med oppgaven

Denne oppgaven har til hensikt å implementere et mål for kompleksitet av kildekode. Dette skal gjøres ved å modifisere en eksisterende kompilator slik at den i tillegg til å produsere objektkode også kan måle den kompilerte kildekodens kompleksitet som et tall. Kompleksitetsmålet som skal implementeres er utarbeidet av Arne Maus ved Institutt for Informatikk, Universitetet i Oslo og defineres i [1].

Med den modifiserte kompilatoren kan vi få beregnet kompleksiteten av eksisterende kildekode og undersøke hvor godt Maus' kompleksitetsmål stemmer overens med intuitiv kompleksitet. I denne forbindelse vil jeg også sammenligne med andre kompleksitetsmål som har vært foreslått i litteraturen.

1.2 Hvorfor et kompleksitetsmål?

Stadig større programsystemer med tiltagende kompleksitet gjør det vanskeligere å forstå og vedlikeholde programmer. Det blir mer å forholde seg til og vanskeligere å verifisere at programmet fungerer etter hensikten. Samtidig vet man at programmer med samme spesifikasjon kan oppvise forskjellig grad av kompleksitet. Mye arbeid har vært nedlagt i å finne programmeringsteknikker som bidrar til å senke programmers kompleksitet. Strukturert programmering og objektorientert programmering er to eksempler på dette. Brukt riktig kan disse teknikkene gi bedre programvare.

En kompilator som kan identifisere komplekse deler av et program gir oss en henvisning til hvilke deler av programmet vi bør konsentrere oss om og eventuelt søke å forenkle. Kompilatoren kan dermed bidra til at programmet blir mer strukturert. Dermed blir programmet også mer oversiktlig og lettere å vedlikeholde. Sannsynligheten for feil i kildekoden reduseres.

1.3 Hva bidrar til kompleksitet?

Når vi skal måle hvor kompleks en bit kildekode er, må vi først ta stilling til hvilke faktorer som bidrar til kompleksitet. Denne analysen er verdifull i seg selv, og mange av forskningsresultatene rundt kompleksitetsmål kan tjene som retningslinjer for nye, bedre programmeringsspråk. Poenget er ikke nødvendigvis å beregne et kompleksitetstall som ad magisk vei sier noe om hvor vanskelig det er å forholde seg til en gitt kildekode. De forskjellige kompleksitetsmål som har vært foreslått, skiller seg fra hverandre ved at de legger vekt på forskjellige faktorer:

1. Programmets flytskjema. Et slikt mål foreslås av McCabe i [2]. Målet gir en kompleksitet som vokser med antallet mulige veier gjennom et program, og gjør tester (`IF`, `CASE` etc) til en kompleksitetsmessig sett kostbar operasjon.

2. Antall variabler og antallet referanser til disse. Disse målene søker å fange opp det faktum at det blir vanskeligere å følge tråden i et program jo fler variabler og prosedyrer det inneholder. Hver prosedyre og variabel har knyttet til seg semantiske regler for bruk, og disse reglene må vi huske når vi skal forstå hvordan programmet fungerer. Jo flere variabler og prosedyrer, jo flere regler må man holde styr på.

3. Programmets tekstlige størrelse brukes ofte som et enkelt mål for kompleksitet. Jo mer tekst, jo mer må man lese og forstå.

Det kompleksitetsmålet vi skal implementere i denne oppgaven baserer seg på punkt 2. Det fokuserer på størrelsen av programmets forskjellige navnerom (prosedyrer og klasser gir opphav til lokale navnerom) og antall referanser til navnerommet. Da et strukturert program typisk vil søke å kapsle inn kompleksitet ved å innføre lokale navnerom, vil et slikt kompleksitetsmål være følsomt for programmets struktur, og ikke bare dets rent tekstlige størrelse. Dette er en viktig egenskap som vil kunne utforskes nærmere med den modifiserte kompilatoren som verktøy.

1.4 Krav til kompilatoren vi skal bruke som basis

Når vi skal modifisere en kompilator, trenger vi tilgang til kompilatorens kildekode. Dermed kan man utelukke å ta utgangspunkt i kommersielt tilgjengelige kompilatorer, da kildekoden for disse er konfidensiell. Samtidig bør man velge en kompilator for et språk som er i utstrakt praktisk bruk slik at det finnes en stor tilgjengelig mengde kildekode man kan måle kompleksiteten av og dermed få et statistisk grunnlag for å vurdere kompleksitetsmålet. Språket bør understøtte de vanligste programmeringsteknikkene:

- Programmering uten å dele opp i prosedyrer.
- Programmering med oppdeling i prosedyrer.
- Objekt-orientert programmering.

Vi kan da finne ut hvordan kompleksitetsmålet reflekterer strukturen i et program. Et mer strukturert program bør gi et lavere kompleksitetstall.

Kravene til utstrakt bruk og støtte for objekt-orientert programmering gjør at et språk som C++ utpeker seg som en interessant kandidat. C++ har i løpet av relativt kort tid oppnådd stor popularitet, noe som ikke minst skyldes at det er en videreføring av det noe mer tilårskomne programmeringsspråket C. C++ representerer en *evolusjonær* og ikke en *revolusjonær* utvikling. De som har brukt C finner seg fort til rette i C++. Opphavsmannen til C++, Bjarne Stroustrup, estimerer at antallet brukere av språket fordobler seg for hver 6.-8. måned.

Da C++ har fått stor aksept og er blitt omfattet av en meget stor interesse, har det også utviklet seg ikke-kommersielle implementasjoner av kompilatorer for språket.

1.5 Free Software Foundation og GNU prosjektet

Free Software Foundation er en heller løs sammenslutning ildsjeler med røtter i amerikansk universitetsmiljø som mener at programvare burde være gratis [3]. I GNU prosjektet har de lykkes med å lage implementasjoner av programsystemer som normalt koster en anseelig sum penger. En viktig del av prosjektet utgjøres av C kompilatoren GCC skrevet av Richard Stallman et al. Til denne er det videre blitt utviklet en C++ utvidelse kalt G++.

I tråd med prinsippet om gratis programvare, er full kildekode tilgjengelig for alle programmer utviklet i GNU prosjektet. Så også kildekoden for G++ og GCC. G++ utmerker seg dermed som et godt utgangspunkt for implementasjon av et kompleksitetsmål, og denne oppgaven vil bruke G++ som basis.

1.6 Språket i denne oppgaven

Hvis relevant norsk terminologi finnes, vil denne oppgaveteksten søke å bruke norske fremfor engelske faguttrykk. Dette under forutsetning av at de norske betegnelsene er godt innarbeidet. Forfatteren vil prioritere en klar fremstilling fremfor bruk av dårlig innarbeidet norsk terminologi som vanskeliggjør forståelsen. Engelske faguttrykk i norsk språkdrakt blir brukt der norsk terminologi ikke finnes eller er lite innarbeidet.

En del av oppgaven vil gå ut på å modifisere et eksisterende program (G++ kompilatoren). G++ kompilatoren er skrevet i C med engelske variabelnavn og kommentarer, og vi vil derfor fortsette å benytte engelske variabelnavn når vi modifiserer programmet slik at variabelnavnene forblir konsistente. Kommentarer i programteksten vil også være på engelsk.

Kapittel 2

Om kompleksitetsmål

For store programsystemer er det vanskelig å estimere kostnadene ved utvikling og vedlikehold, og overskridelser av budsjetter og tidsfrister er snarere regelen enn unntaket. Det synes spesielt vanskelig å estimere ressursbruken ved utvikling av nye systemer.

Ved vedlikehold av eksisterende systemer er utgangspunktet noe mindre diffust. Systemets kildekode er da gitt, og vanskelighetene med vedlikeholdet vil være sterkt relatert til hvor vanskelig det er å forstå kildekoden slik at man kan gjøre endringer i den. Erfaring tilsier at utgiftene til vedlikehold representerer omkring $\frac{2}{3}$ av kostnadene ved et programsystem [4]. Forskningen omkring kompleksitetsmål konsentrerer seg derfor i stor grad om å utvikle mål som kvantiserer den anstrengelse det representerer å sette seg inn i og forstå en gitt kildekode. I [5] tenker man seg slike mål brukt i kontraktørvirksomhet ved at kildekoden for et ferdig system må tilfredsstille krav til målt vedlikeholdbarhet før produktet godkjennes av oppdragsgiver.

Vi ønsker å estimere arbeidsmengden som nedlegges i forståelsesprosessen, men vi har liten kunnskap om hvordan forståelsesprosessen fungerer. Å måle denne arbeidsmengden er ikke mulig før vi vet mer om hva som skjer i hjernen når vi prøver å forstå noe. I [6] definerer Bill Curtis kompleksitet slik:

> Complexity is a characteristic of the software interface which influences the resources another system will expend while interacting with the software.

I denne sammenheng er "another system" den menneskelige hjerne, og vi trenger en modell av hjernens aktivitet for å kunne kvantisere ressursene hjernen må bruke for å oppnå forståelse. Å beregne kompleksitet blir etter denne definisjonen som å beregne arbeid i fysikkens forstand. En av retningene innen forskningen omkring programvare kompleksitet er da også sterkt inspirert av fysikken og naturlovene (Halsteads Software Science [7]).

Vår forståelse av prosessene i hjernen er imidlertid svært begrenset, og vil nok fortsatt være det i lang tid fremover. Derfor måler kompleksitetsmålene

aspekter ved kildekoden som bidrar til kompleksitet, og ikke *selve kompleksiteten*. Dette kan ikke understrekes sterkt nok.

Ved å måle flere aspekter ved kildekoden, kan vi på bakgrunn av tidligere erfaring danne oss et bilde av hvor krevende vedlikeholdet vil være. Vi bør imidlertid avstå fra å kombinere disse målene til ett mål for "den ene og sanne kompleksitet" så lenge vi ikke har noen modell av forståelsesprosessen. Et slikt mål vil ikke ha noen referanseramme, og mangle enhver verdi i fravær av empiriske valideringer.

Vi vil nå se nærmere på enkelte aspekter ved kildekode som kan tjene som indikatorer for kompleksitet, og kort omtale hvordan noen kompleksitetsmål beregner et måltall for kompleksitet basert på disse indikatorene.

2.1 Kildekodens tekstlige størrelse

Det er en innlysende sammenheng mellom kildekodens tekstlige størrelse og problemene forbundet med å forstå den. Å måle kildekodens størrelse i antall linjer er dessuten enkelt. Antall linjer kildekode har derfor lenge vært brukt som kompleksitetsmål, men er dessverre ikke særlig presist, da de fleste programmeringsspråk tillater at enhver programtekst kan skrives på én linje. Målet blir derfor helt avhengig av skrivestil hos programmereren. Spesielt kan mengden kommentarer og indentering gjøre store utslag. Hvis vi unnlater å telle kommentarer og blanke linjer, vil dog programmerere bruke omtrent like mange linjer på samme programtekst. Målet kan da tjene som en grov indikator for kompleksitet.

Definisjon 2.1 *$LOC(T)$ er antallet linjer i programteksten T som ikke er blanke eller består utelukkende av kommentarer.*

Dette betyr *ikke* at kommentarer er uinteressante for forståelsen. Et mekaniserbart mål kan imidlertid vanskelig vurdere kommentarenes verdi. I C++ fjernes dessuten kommentarene av en preprosessor før kompilatoren leser kildekoden.

2.2 Halsteads Software Science

Ved å telle antall *symboler* i kildekoden (altså reserverte ord, operatorer og deklarerte identifikatorer), får man en indikator som er ufølsom for oppdelingen av kildekoden i linjer. Halstead [7] utviklet en del mål som baserer seg utelukkende på følgende enkle data om kildekoden:

- Antall forekomster av operatorer, N_1.
- Antall forekomster av operander, N_2.

- Antall tilgjengelige operatorer, η_1.
- Antall tilgjengelige operander, η_2.

Kildekodens lengde kan nå beregnes som $N = N_1 + N_2$, og kildekodens vokabular som $\eta = \eta_1 + \eta_2$. Anta at en algoritme er programmert, slik at den er representert som kildekode i et programmeringsspråk. Vi kan da innhente N og η fra denne representasjonen av algoritmen. Inspirert av resultater innen fysikk, foreslo Halstead i [8] at alle målbare egenskaper ved en algoritme kan utledes fra to uavhengige egenskaper ved algoritmen. Han hevdet også at N og η var to slike uavhengige egenskaper. Den interessante konsekvensen er at alle målbare egenskaper ved en algoritme kan beregnes ut fra en enkel opptelling av operatorer og operander i en programtekst som implementerer algoritmen.

En algoritme kan imidlertid implementeres på et uendelig antall måter. For eksempel kan man ved gjentatt bruk av en idempotent operasjon (eksempelvis addisjon med null) øke N til en vilkårlig stor verdi slik at det ikke gir mening å snakke om N som en egenskap ved algoritmen. Halstead forutsetter at kildekoden er fri for denne typen "urenheter".

2.2.1 Kildekodens volum og nivå

Halstead definerer *volumet* V av en implementasjon som det minste antall bits man kan kode implementasjonen i uten tap av informasjon:

$$V = N \log_2 \eta \tag{2.1}$$

Volumet V vil være avhengig av programmeringsspråket: En implementasjon i et høynivå språk vil ha lavere volum enn en implementasjon i assembler. Laveste verdi for V får vi når vi bruker et språk som har en implementasjon av algoritmen innebygd i språket, for eksempel i form av en predefinert prosedyre p . Algoritmen kan da implementeres som et prosedyrekall $p(\cdots)$. Det minimale volumet benevnes V^*, og er en funksjon av antallet "konseptuelt unike parametre" til den tenkte prosedyren p. Implementasjonens nivå L kan nå uttrykkes som:

$$L = \frac{V^*}{V} \tag{2.2}$$

2.2.2 Anstrengelse

Basert på L og V utleder Halstead et uttrykk for anstrengelsen (effort) E forbundet med å skrive en implementasjon av en algoritme. Anta at denne

implementasjonen består av N symbolforekomster fra et vokabular av η symboler. Arbeidet med å skrive implementasjonen kan da betraktes som arbeidet forbundet med å instansiere N symbolforekomster med symboler fra vokabularet.

Å instansiere en symbolforekomst betrakter Halstead som et oppslag i en "mental database" over symbolene i vokabularet. Hvis oppslaget gjøres ved binærsøk, vil hvert oppslag i gjennomsnitt medføre $\log_2 \eta$ "mentale sammenligninger". For å instansiere alle N symbolforekomstene, kreves $N \log_2 \eta$ mentale sammenligninger. Vi kjenner igjen dette uttrykket som formelen for V i (2.1).

Anstrengelsen forbundet med hver mentale sammenligning er minst når vi bruker høynivå språk, og tiltar når språkets nivå avtar mot assembler. Hvis vi lar $\frac{1}{L}$ være et mål for anstrengelsen som ligger i en mental sammenligning, får vi:

$$E = \frac{1}{L}(N \log_2 \eta) = \frac{V}{L} \tag{2.3}$$

Enheten for E er "elementære mentale diskriminasjoner". Det er svært nærliggende å mistenke Halstead for å ha konstruert modellen om den "mentale databasen" etter at (2.3) ble postulert. Antagelsen om binærsøket på den mentale databasen synes grepet ut av luften, og han gjør ikke noe forsøk på å rettferdiggjøre denne forløsende antagelsen i [7].

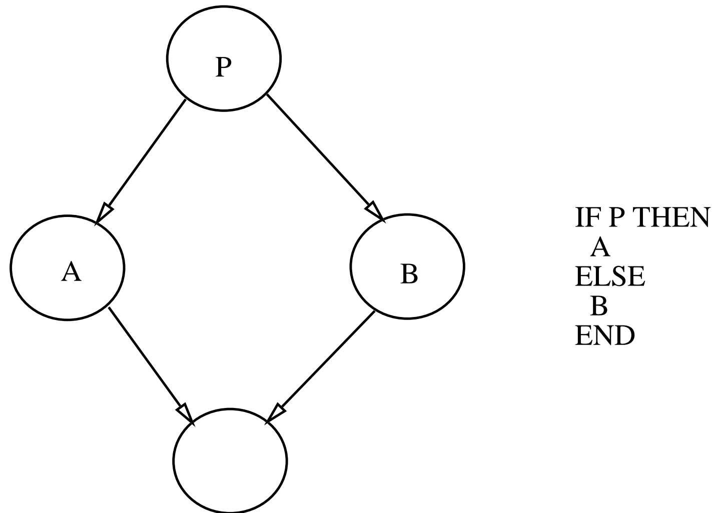
For å teste (2.3), antar Halstead at antallet "elementære mentale diskriminasjoner" pr sekund er konstant og tilsvarer Strouds konstant S. Strouds konstant stammer fra kognitiv psykologi, og uttrykker blant annet hjernens evne til å oppfatte enkelthendelser som separate hendelser. For eksempel må en film vises med mer enn $S = 18$ bilder i sekundet for at vi skal tolke filmen som sammenhengende bevegelse og ikke en serie enkeltbilder. Halstead tenker seg S som et mål for hjernens arbeidshastighet — omtrent som klokkefrekvensen i en datamaskin. Ved hjelp av S blir det mulig å teste (2.3) ved å måle tiden man bruker på å skrive en implementasjon I, og deretter sammenligne med tiden $\frac{E}{S}$ beregnet for I.

Anstrengelsen E forbundet med å *skrive* en implementasjon blir ofte brukt som et mål for hvor vanskelig det er å *forstå* den ved å lese programteksten [5].

2.3 Kontrollflyt

I [9] foreslår Dijkstra at vi fra naturens side er best utstyrt til å forstå statiske sammenhenger, og relativt dårlig utstyrt til å visualisere prosesser som utspenner seg i tid (som for eksempel utførelsen av et program).

Erfaring tilsier at sekvensering av setninger synes å gi en rimelig lettfattelig statisk beskrivelse av en dynamisk prosess. Enhver konstruksjon



Figur 2.1: Flytgraf for en programtekst

som gjør at den dynamiske utførelsen kan avvike fra den sekvensielle tekstlige representasjonen kompliserer arbeidet med å visualisere teksten som en prosess. Dijkstra bruker dette til å argumentere for at `goto` bør erstattes av andre kontrollstrukturer.

I vår sammenheng ønsker vi å tolke dette dithen at kontrollstrukturer indikerer kompleksitet: Å følge gangen i et program er enklest når programmet ikke inneholder hopp. Jo flere hopp kildekoden inneholder, jo vanskeligere blir det å danne seg et bilde av virkemåten.

2.4 McCabes kompleksitetsmål

Tanken om programmets kontrollflyt som en indikator for kompleksitet utgjør fundamentet i et stort antall kompleksitetsmål som bruker programmets flytgraf som utgangspunkt for grafteoretiske mål for kompleksitet. Det mest kjente av disse ble lansert i [2] av Thomas McCabe, og utløste en strøm av varianter over samme tema. Figur 2.1 gir et eksempel på en flytgraf.

En node i en flytgraf kan representere

- En maksimal sekvens av setninger som er slik at dersom *første* setning i sekvensen utføres, blir *alle* setningene i sekvensen utført sekvensielt.
- Et predikat viss verdi påvirker programflyten.

Antall mulige veier gjennom en flytgraf kan gjøre tjeneste som et kompleksitetsmål. Målet blir imidlertid meningsløst dersom grafen inneholder løkker

(antall mulige veier er da ikke begrenset av grafen). McCabe foreslår derfor at man bruker antallet *elementære* veier i grafen som mål. Enhver mulig vei gjennom grafen kan konstrueres som en kombinasjon av elementære veier.

Anta at vi har en programtekst T som består av et sett prosedyrer $P_1 \cdots P_p$. For hver P_i kan vi danne en flytgraf G_i med noder N_i og rettede kanter E_i slik at

- G_i har eksakt én startnode $s_i \epsilon N_i$ og én sluttnode $t_i \epsilon N_i$.
- Enhver $n \epsilon N_i$ kan nås fra startnoden s_i.
- Sluttnoden t_i kan nås fra enhver $n \epsilon N_i$.

Hvis vi tenker oss en kant fra t_i til s_i, vil det eksistere en vei mellom alle $a, b \epsilon N_i$. G_i er nå *sterkt koblet*. La G være grafen som består av de separate komponentene $G_1 \cdots G_p$. Antallet elementære veier i G er da gitt som:

$$V(G) = 2p + \sum_{i=1}^{p} \mid E_i \mid - \mid N_i \mid \tag{2.4}$$

$V(G)$ er McCabes mål for kompleksiteten av programteksten T.

Dersom P_i er *strukturert*, kan $V(G_i)$ beregnes direkte fra programteksten uten å konstruere G_i. P_i betegnes som strukturert når den er satt sammen av enkle setninger ved hjelp av Dijkstra-strukturer [10], det vil si sekvensering, `IF` og `WHILE`. Anta at π er antall predikater i en strukturert P_i. Da hevder McCabe [2] at vi har

$$V(G_i) = \pi + 1 \tag{2.5}$$

Dersom et predikat P er en konjunksjon av andre predikater, foreslår McCabe at hver konjunkt i P regnes som et eget predikat fordi en test på P kan omskrives til en nestet test på hver konjunkt.

De fleste programmeringsspråk inneholder spesielle kontrollstrukturer som ikke er Dijkstra-strukturer, og (2.5) kan derfor ikke benyttes generelt på kildekode skrevet i slike språk. Forekomster av spesielle kontrollstrukturer kan imidlertid alltid reduseres til Dijkstra-strukturer. Vi kan alltid danne en ekvivalent kildekode som bare inneholder Dijkstra-strukturer. Ved å benytte (2.5) på denne ekvivalente kildekoden, kan målet allikevel lett beregnes for kildekode som inneholder spesielle kontrollstrukturer: En `CASE` setning som velger mellom N tilfeller kan for eksempel reduseres til N `IF` setninger, og bidrar derfor med N predikater. En `CASE` setning med N alternativer gir imidlertid et bedre bilde av kildekodens struktur enn N `IF` setninger. Denne implisitte omskrivingen av `CASE` til `IF` har vært gjenstand for kritikk [11] fordi `CASE` setninger straffes urimelig hardt.

Forklaringen ligger i at McCabe tenkte seg målet brukt i forbindelse med testing av programvare. Slik testing bør omfatte alle elementære veier i

programmet, og McCabe foreslår derfor at man overholder en øvre grense $V(G_i) < 10$ for hver prosedyre P_i. Hver P_i kan da testes rimelig grundig hver for seg, slik at en grundig testing av programmet som helhet "faktoriseres" ned til testing av hver prosedyre.

2.5 Modularisering

Et programsystem vil i praksis bestå av mange moduler der hver modul har et grensesnitt som regulerer forholdet til andre moduler. Minimale grensesnitt mellom modulene vil sikre maksimal frihet ved endringer internt i modulene og forenkle vedlikeholdet [12]. Oppdelingen av systemet i moduler blir således svært viktig for utgiftene til vedlikehold og videreutvikling. Et godt kompleksitetsmål bør derfor være sensitivt for modulariseringen av systemet. Fordi slike mål prøver å danne et bilde av forhold utenfor hver enkelt modul, omtales de ofte som *makromål*. De målene vi har beskrevet til nå fokuserer i stor grad på programteksten internt i hver enkelt prosedyre, og kalles derfor *mikromål*. Litteraturen domineres av mikromål, og de få makromål som finnes er ofte mangelfulle.

Det synes vanskelig å finne enkle aspekter ved programteksten som indikerer god eller dårlig modularisering. Antallet avhengigheter mellom modulene har vært brukt som en grov indikator [13]. Hver modul har da to attributter av interesse for kompleksitetsmålet:

- Fan-in. Antallet moduler som refererer til denne modulen.
- Fan-out. Antallet moduler denne modulen refererer til.

En modul med høy fan-in og høy fan-out vil være vanskelig å endre på fordi endringer fort får store konsekvenser for omgivelsene. Et slikt mål kan brukes for å lokalisere moduler i systemet som bør splittes opp og reorganiseres.

2.6 Psykologiske teorier

Vi har nå sett hvordan tre forskjellige aspekter ved en gitt kildekode kan brukes som indikatorer for kompleksitet:

- Tekstlig størrelse
- Kontrollflyt
- Modularisering

Å kombinere disse indikatorene i ett måltall for kompleksitet er ikke ønskelig med mindre vi har en psykologisk velfundert teori å basere oss på. Uten en bakenforliggende teori vil måltallet mangle referanseramme. Noen kompleksitetsmål har gjort bruk av psykologiske teorier, men ofte trekkes svært vide konklusjoner på spinkelt grunnlag. For eksempel har Halsteads E vært gjenstand for sterk kritikk [6] [14] [15] i forbindelse med (mis)bruken av Strouds konstant S. S er relatert til hjernens evne til å oppfatte enkle sansestimuli, og ikke til de komplekse prosessene som inngår i kunsten å programmere [6].

2.7 Programforståelse ved hypotesetesting

Arbeidet med å forstå en gitt programtekst kan betraktes som arbeidet forbundet med å bygge en mental modell av programtekstens virkemåte. Ved å skumlese programteksten, oppfatter vi bruddstykker av informasjon som gjør at vi kan formulere hypoteser om virkemåten til isolerte deler av kildekoden [16] [17]. For eksempel kan en trenet programmerer merke seg små detaljer som kan tyde på at en bestemt løkke implementerer et binærsøk. Han fremsetter da en hypotese om at løkken implementerer et binærsøk, og studerer kildekoden nærmere i et forsøk på å verifisere hypotesen.

- Hvis hypotesen synes å holde, konkluderer han med at han sannsynligvis har forstått løkkens funksjon. Denne forståelsen er til hjelp når han senere forsøker å formulere hypoteser for tilstøtende deler av kildekoden.

- Hvis hypotesen synes å være gal, har arbeidet med falsifiseringen øket innblikket i kildekoden, slik at det blir lettere å formulere en bedre, alternativ hypotese.

- Dersom han ikke klarer å styrke eller svekke hypotesen, lar han den stå, og håper at hypotesen styrkes eller svekkes i lys av hypoteser om andre deler av kildekoden.

På denne måten bygger man opp et sett med hypoteser som understøtter hverandre. Når man føler at hypotesene er sydd sammen til et konsistent hele som i rimelig grad understøttes av programteksten, har man *forstått* programteksten og dannet seg en mental modell av den.

En god programmerer vil merke seg detaljer i kildekoden som gir grobunn for fruktbare hypoteser. Vi kaller disse detaljene *fyrtårn*. En dårligere programmerer har ikke et like stort repertoar av fyrtårn, og vil ha et dårligere utgangspunkt for å formulere nyttige hypoteser. Repertoaret av fyrtårn vil stort sett stamme fra tidligere programmeringserfaring og innsikt i problemet programmet befatter seg med. Dette forklarer hvorfor programmeringserfaring

gjør det lettere å forstå kildekode og hvorfor forskjellen i produktivitet mellom programmerere er så høy.

Bruken av hypoteser i forståelsesprosessen gjør at programteksten forstås som grupper av setninger, og ikke som en strøm av enkelttegn. Vi bruker betegnelsen *brokke* om et utsnitt av programteksten som forstås som en enhet (fra engelsk *chunk*). Hver brokke gir opphav til en hypotese som utgjør et fundament for den videre forståelsen av programteksten. En lang prosedyrekropp vil typisk bestå av mange brokker. Når vi har en rimelig forståelse av hver brokke, blir det mulig å forstå prosedyren som helhet.

2.8 Sammendrag

Dette kapitlet har gitt oss innblikk i de klassiske idéene som danner grunnlaget for en rekke kompleksitetsmål. Målet vi skal bruke i denne oppgaven er sterkt knyttet til Halsteads mål for volumet V av en programtekst. Den bærende idéen er at programmets kompleksitet er proporsjonal med den mengden informasjon vi må absorbere fra programteksten for å forstå programmet. Hvordan denne informasjonsmengden skal beregnes er tema for kapittel 3.

Kapittel 3

Kompleksitetsmålet

> If you can't measure it, you don't understand it.
>
> *Fritt etter Lord Kelvin*

Dette kapitlet presenterer kompleksitetsmålet vi skal implementere. Den opprinnelige definisjonen finnes i [1]. Presentasjonen av målet holdes i generelle termer, og er ikke knyttet til noe bestemt programmeringsspråk. Hvordan målet skal tilpasses bruk på C++ kildekode er tema for kapittel 5.

3.1 Kompleksitet fra hvilken synsvinkel

Vi kan grovt skille mellom to faser i et programs levetid:

Konstruksjonsfasen. Denne fasen konsentrerer seg om valg av algoritmer og oppdeling av problemet i delproblemer slik at programmet blir effektivt og enklest mulig å sette seg inn i. Konstruksjonsfasens største problem er ikke å forstå hvordan programmet fungerer, men å *fastsette* hvordan det skal fungere — bestemme programmets struktur.

Vedlikeholdsfasen. Denne fasen kan strekke seg over år og involvere mange personer som ikke var med på det opprinnelige utviklingsarbeidet. Vedlikeholdsarbeidet vil typisk bestå av feilretting og utvidelser, og de impliserte bør enklest mulig kunne sette seg inn i programmets virkemåte. Problemene forbundet med vedlikeholdet er direkte relatert til hvordan vi konstruerte programmet. Et godt strukturert program forenkler vedlikeholdet.

Å *konstruere* et program er en helt annen prosess enn å *sette seg inn i* et eksisterende program. Konstruksjon omfatter betraktninger som vi ikke behøver å befatte oss med når vi driver vedlikehold (med mindre vi skal foreta en total omskriving av programmet).

Mye taler for at vi bør skille mellom hvor komplekst det er å konstruere programmet og hvor komplekst det er å foreta endringer i det når det omsider foreligger.

Vårt kompleksitetsmål er en funksjon av kildekoden, og vil derfor gi et bilde av hvor vanskelig det er å forstå et *eksisterende* program. Det vil reflektere kompleksiteten sett fra vedlikeholdsprogrammererens synspunkt.

3.2 Hva gjør det vanskelig å forstå en tekst

En tekst forstås lettest når den kan leses sekvensielt uten stadig å måtte referere til andre deler av teksten. Samtidig er vår metode for konstruksjon av store systemer basert på "splitt og hersk". Vi deler opp problemet i mindre deler som løses hver for seg og sys sammen til et hele. Delene vil typisk være prosedyrer og klasser som benyttes andre steder i programmet.

Dette resulterer i at en programtekst til stadighet refererer til andre deler av teksten gjennom prosedyrekall og variabelreferanser. For å forstå meningen med et prosedyrekall må vi konsultere den kalte prosedyrens semantikk. Hvis programmet inneholder mange prosedyrer, blir det mye semantikk å holde rede på, og jo flere prosedyrekall som finnes, jo flere konsultasjoner mot semantikken blir det. Vanskeligheten med å forstå et program synes dermed å være en funksjon av to parametre:

- Antallet deklarerte entiteter (prosedyrer og variable). Dette gir et mål på hvor mye sematikk vi til en hver tid må ha klart for oss når vi leser programteksten.
- Antallet referanser til de deklarerte entitetene. Dette gir et mål på hvor mange ganger vi må konsultere semantikken for at teksten skal få mening.

Dette utgjør essensen i det kompleksitetsmålet vi skal implementere. Vi vil modifisere G++ kompilatoren slik at den bestemmer disse to parametrene for gitt kildekode, og beregner en verdi som sier noe om hvor vanskelig det er å sette seg inn i programmet ved å lese kildekoden.

3.3 Symboler

Naturlige språk som snakkes av mennesker har *ord* som minste meningsbærende enhet. Med programmeringsspråk er det mer passende å bruke betegnelsen *symbol* istedet for *ord*. Variabelnavn, operatorer og reserverte ord er eksempler på symboler. Et symbol kan forekomme flere ganger i programteksten, så vi skiller mellom symboler og *symbolforekomster*. En

programtekst kan da oppfattes som en sekvens av symbolforekomster. Symbolene som forekommer i programteksten er elementer i en symbolmengde kalt programtekstens *alfabet*.

Definisjon 3.1 *Programtekstens alfabet utgjør en endelig mengde symboler. Symbolene som forekommer i programteksten er en delmengde av programtekstens alfabet.*

Observér at alfabetet til en gitt programtekst altså kan inneholde symboler som tilfeldigvis ikke forekommer i den gitte programteksten. Vi kan ikke fastsette programtekstens alfabet ved å lese den. Et alfabet inneholder alltid predefinerte symboler som inngår i programmeringsspråket. Alle disse predefinerte symbolene forekommer ikke nødvendigvis i teksten, men regnes alltid med i alfabetet.

Legg merke til at ordet alfabet i tradisjonell forstand har en litt annen betydning, nemlig som den statiske mengde *bokstaver* som tjener som byggestener for symbolene. Det er imidlertid *symbolene* (ordene) som er meningsbærende, og ikke de enkelte bokstavene hver for seg. De enkelte bokstavene i symbolene er dermed uten interesse når vi skal måle hvor vanskelig det er å forstå (gi mening til) en gitt programtekst. Derfor fjerner vi bokstavene fra vår interesse-sfære ved å la *symbolene* være de minste enhetene i programteksten og definere alfabetet som en mengde *symboler* og ikke *bokstaver*.

I sin rolle som meningsbærer kan et symbol betraktes som en merkelapp for en mening (symbolets semantikk). De fleste programmeringsspråk tillater at man utvider alfabetet med nye symboler ved å *deklarere* deres mening. Deklarerte symboler vil typisk være navn på variabler, prosedyrer og typer. Det er også mulig å innføre nye symboler uten å deklarere dem først. Dette gjelder for eksempel tallkonstanter, som kan forekomme direkte i programteksten uten forutgående deklarasjon. Vår kjennskap til matematikk gjør at vi umiddelbart gjenkjenner et tall som et tall. Av og til er det ønskelig å la det gå klart frem av teksten hva tallet representerer, og mange programmeringsspråk tilbyr derfor muligheten til å deklarere tallkonstanter slik at programteksten kan snakke om "PI" og ikke "3.14", men dette er alltid en opsjon og ikke et krav. Vi kan foreløpig betrakte tallkonstanter etc som implisitt deklarerte symboler, og vil komme tilbake til dem senere.

For å muliggjøre deklarasjoner har programmeringsspråket en grunnmengde av forhåndsdeklarerte symboler som brukes for å angi meningen til nye symboler. Meningen til disse forhåndsdeklarerte symbolene inngår i definisjonen av programmeringsspråket. Alfabetet kan i tråd med dette deles inn i to disjunkte symbolmengder.

- En statisk del bestående av forhåndsdeklarerte reserverte symboler.
- En dynamisk del bestående av symboler innført ved deklarasjon.

Den dynamiske delen av alfabetet kan vokse og tilpasses oppgaven programmet skal løse slik at forskjellige programmer så godt som alltid har forskjellig alfabet. Større programmer vil være splittet opp i deler med forskjellige deloppgaver. De fleste programmeringsspråk tillater at hver slik del tilpasser alfabetet til sin deloppgave ved å innføre lokale navnerom som deklarerer lokale symboler (for eksempel lokale deklarasjoner i prosedyrer). Dette innebærer at ett og samme program kan ha forskjellig alfabet i forskjellige deler av programteksten. Fordi vårt kompleksitetsmål er en funksjon av antall deklarerte entiteter (og dermed alfabetets størrelse), er det behagelig å se på hver slik del isolert, og vi innfører begrepene *segment* og *seksjon*.

Definisjon 3.2 *Et segment defineres induktivt ved at*

- *En symbolforekomst er et segment.*
- *En sekvens av segmenter med samme alfabet er et segment.*

Intuitivt: Et segment er et utsnitt av programteksten som ikke omfatter mer enn ett navnerom.

Definisjon 3.3 *En seksjon er et segment som ikke kan utvides til et større utsnitt av teksten og fremdeles bare omfatte ett navnerom.*

Intuitivt: En seksjon er et maksimalt segment.

3.4 Et mål for informasjonsmengde

Idéene til det numeriske mål for kompleksitet som vi skal bruke i denne oppgaven har røtter i en klassisk artikkel av R.V.L. Hartley [18] publisert i 1928 som utarbeider et mål for informasjonsmengde. Forfatteren var opptatt av dette i forbindelse med beregning av behovet for båndbredde ved informasjonsoverføring, blant annet TV-overføring (i 1928!). Dette leder til en litt annen innfallsvinkel enn Maus bruker i [1].

Når vi leser et segment symbol for symbol, bidrar hvert leste symbol til at vår viten om segmentet øker. Det skal vise seg fruktbart å betrakte lesingen av en symbolforekomst som en eliminasjonsprosess som eliminerer alle andre symboler i alfabetet fra symbolforekomsten (sammenlign med avsnitt 2.2.2 som omhandler arbeidet forbundet med å *skrive* en implementasjon). Etterhvert som symboler leses, elimineres sekvenser av symbolforekomster. Når hele segmentet er lest, er alle mulige segmenter med samme alfabet og lengde eliminert. Vår viten om segmentet har da nådd sitt maksimum, og vi kan betrakte antallet eliminerte segmenter som et mål for hvor mye informasjon vi har tatt til oss.

Anta at vi har et segment som består av n symbolforekomster fra et alfabet med s symboler. Hvis alle symboler kan forekomme hvor som helst i

segmentet, kan symbolene settes sammen til s^n forskjellige segmenter. I tråd med det foregående kan vi la s^n representere segmentets informasjonsmengde fordi uttrykket sier oss hvor mange segmenter vi har eliminert som følge av gjennomlesningen ($s^n - 1$ segmenter er eliminert). Informasjonsmengden vil da øke eksponentielt med segmentets lengde n. Når n øker, vil bidraget til informasjonsmengden fra hvert tilføyd symbol øke dramatisk.

Eksempel Anta at alfabetet består av $s = 4$ symboler og vi utvider et segment fra én til to symbolforekomster. Da øker informasjonsmengden med 12 (fra $s^1 = 4$ til $s^2 = 16$). Hvis vi nå utvider til tre symbolforekomster, øker informasjonsmengden med 48 (fra $s^2 = 16$ til $s^3 = 64$).

Et slikt mål for informasjonsmengde er ikke i tråd med vår intuitive oppfatning av informasjonsmengde. Bidraget til informasjonsmengden fra én enkelt symbolforekomst må være uavhengig av forekomstens posisjon i segmentet slik at identiske utsnitt av et segment bidrar med samme informasjonsmengde. Vi ønsker et proporsjonalt mål for informasjonsmengde der informasjonsmengden til et hele er gitt som summen av informasjonsmengden til delene. La $\mathcal{C}(T)$ betegne informasjonsmengden for segmentet T. Vi vil at et segment T som består av segmentene $T_1 \cdots T_n$ skal ha

$$\mathcal{C}(T) = \sum_{i=1}^{n} \mathcal{C}(T_i) \tag{3.1}$$

Hvis vi lar $T_1 \cdots T_n$ være de enkelte symbolforekomstene i T (n er da lengden av T), og lar $\mathcal{C}(T_i)$ være bestemt av kun alfabetets størrelse, får vi

$$\mathcal{C}(T) = Kn \tag{3.2}$$

der K er en konstant som er avhengig av alfabetets størrelse. Vi skal nå utlede et slikt mål. La T_1 være et segment med lengde n_1, informasjonsmengde $K_1 n_1$ og et alfabet med s_1 symboler. La T_2 være et segment med lengde n_2, informasjonsmengde $K_2 n_2$ og et alfabet med s_2 symboler. Vi ønsker at K_1 og K_2 skal være slik at

$$K_1 n_1 = K_2 n_2 \tag{3.3}$$

når

$$s_1^{n_1} = s_2^{n_2} \tag{3.4}$$

Dvs to segmenter av forskjellig størrelse og med forskjellig alfabet skal representere samme mengde informasjon når segmentene har samme antall mulige kombinasjoner av symbolene i alfabetet. (3.4) kan omskrives til

$$n_1 \log s_1 = n_2 \log s_2 \tag{3.5}$$

Hvis vi dividerer (3.3) med (3.5) får vi

$$\frac{K_1}{\log s_1} = \frac{K_2}{\log s_2}$$

som holder for alle $s_1 > 0$ og $s_2 > 0$ dersom vi tar

$$K_1 = \log s_1 \tag{3.6}$$

og

$$K_2 = \log s_2 \tag{3.7}$$

Ved innsetting av $K = \log s$ i (3.2) får vi et uttrykk for informasjonsmengden av et segment som funksjon av segmentets og alfabetets størrelse:

Definisjon 3.4 *La T være et segment som består av n urestrikterte symbolforekomster fra et alfabet med s symboler. Da er informasjonsmengden av T gitt ved*

$$\mathcal{C}(T) = n \log s \tag{3.8}$$

Log-funksjonens grunntall bestemmer måleenheten for informasjonsmengde fordi

$$\log_a x = \frac{1}{\log_b a} \log_b x \tag{3.9}$$

Når vi bruker 10 som grunntall, kalles enheten en *Hartley* etter forfatteren av [18]. Når vi bruker 2 som grunntall, kalles enheten en *bit*. (3.8) angir da det minste antall bits vi kan bruke i en binær koding av T uten å miste informasjon slik at dekoding blir umulig. Legg merke til at dette tilsvarer Halstead's V i (2.1).

3.5 Informasjonsmengde og kompleksitet

Informasjonsmengden $\mathcal{C}(T)$ for en programtekst T gir oss et mål for hvor mye informasjon vi må forholde oss til når vi prøver å forstå T. Maus foreslår derfor i [1] at vi bruker $\mathcal{C}(T)$ som et kompleksitetsmål for T. I denne oppgaven velger vi å bruke grunntall 2 for logaritmefunksjonen i (3.8), og enheten for kompleksitet blir dermed *bits*. Vi kunne valgt et hvilket som helst annet grunntall, men vi er kun interessert i kompleksitetstallet som et relativt mål vi kan bruke til å sammenligne programtekster. Kompleksitetstallets numeriske verdi i seg selv sier oss lite. Vi velger et lite grunntall fordi dette gir størst kompleksitetstall og dermed finere oppløsning ved sammenligning av kompleksiteter.

Legg merke til at (3.8) angir kompleksiteten av et segment der symbolforekomstene er urestriktert, dvs ethvert symbol kan forekomme hvor som helst i segmentet. Programmeringsspråk har syntaktiske og semantiske regler for hvordan symbolene kan kombineres, og dette begrenser antallet lovlige

kombinasjoner av n symboler. s^n angir et maksimum for antallet kombinasjoner, så når vi bruker (3.8), blir den beregnede kompleksiteten for høy. Vi velger å se bort fra dette da vi bare er interessert i å *sammenligne* kompleksiteten av segmenter. Når vi måler kompleksiteten av to segmenter, vil avviket introduseres i begge målingene, så ved sammenligning av de to kompleksitetene er dette av liten betydning.

3.6 Symbolkategorier

Når vi leser en programtekst symbol for symbol kan vi tenke oss at vi for hver symbolforekomst foretar et oppslag i en mental database som inneholder symbolenes mening (sammenlign med avsnitt 2.2.2). Etterhvert som teksten leses, vil meningen til hvert symbol bidra til meningen til hele teksten. Vi vet at den menneskelige hjernes hukommelse trenes opp ved bruk slik at de oppslagene som forekommer hyppigst betjenes hurtigst og med minst mental anstrengelse. Symbolene i den statiske delen av alfabetet har samme mening i alle programtekster, og en trenet programmerer vil derfor ha trenet opp sin hukommelse slik at de reserverte ordene i språket umiddelbart gir mening for ham. Meningen til symbolene i den dynamiske delen av alfabetet vil derimot variere fra program til program, og for å gi mening til et deklarert symbol kan man ikke trekke på erfaring fra tidligere programmer på samme måte som for de reserverte symbolene. Å gi mening til et deklarert symbol representerer dermed en større vanskelighet enn å gi mening til et reservert symbol.

En bestemt delmengde av de reserverte symbolene i et programmeringsspråk vil være spesielt innarbeidet hos programmerere. Det er de matematiske symbolene som brukes i uttrykk, dvs operatorer som $+$ og $-$. De matematiske symbolene brukes i all naturvitenskap, og leses med stor letthet av alle som befatter seg med programmering.

Av denne diskusjonen ser vi at ved kompleksitetsbetraktninger kan det være hensiktsmessig å skille mellom forskjellige klasser av symboler. Noen symboler gir lettere mening enn andre når vi leser programteksten.

Definisjon 3.5 *Vi deler symbolene i programmets alfabet inn i symbolkategorier. En symbolkategori er en mengde av symboler som er slik at alle forekomster av symbolene i mengden representerer omtrent samme mentale anstrengelse når vi prøver å gi mening til programteksten.*

Her følger en mulig inndeling i symbolkategorier for alfabetet til programmer som er skrevet i moderne programmeringsspråk (PASCAL, C etc).

- Deklarerte identifikatorer. Dvs prosedyre- og variabel-navn etc.
- Reserverte ord i språket.
- Operatorer av typen $+$ og $-$.

- Tekstlige literaler.
- Numeriske literaler.

Med literaler menes her konstanter som ikke er deklarert og derfor ikke er tilknyttet noen identifikator. For å kunne snakke presist om symboler og symbolkategorier, gir vi følgende definisjoner:

Definisjon 3.6 *La S være et symbol eller en forekomst av et symbol. Da er $\mathcal{K}(S)$ symbolkategorien for S.*

Definisjon 3.7 *La T være et segment med alfabet A. La K være en symbolkategori for A. Da er $\mathcal{S}(K,T)$ det antall av symboler s i A som har $\mathcal{K}(s) = K$.*

$\mathcal{S}(K,T)$ kalles *kardinaliteten* til kategorien K over segmentet T, og er generelt en funksjon av T fordi T kan innføre nye symboler ved deklarasjon og dermed utvide K.

Definisjon 3.8 *La K være en symbolkategori og T et segment. Da er $\mathcal{R}(K,T)$ det antall symbolforekomster f i T som har $\mathcal{K}(f) = K$.*

3.7 Kompleksiteten av et segment

Vi ønsker at kompleksitetsmålet skal reflektere den varierende grad av anstrengelse symboler fra forskjellige symbolkategorier representerer ved gjennomlesingen av et segment. Kompleksiteten til et segment som består av kun reserverte symboler, bør f.eks. være uavhengig av antallet deklarerte symboler. Vi skal nå modifisere kompleksitetsmålet for et segment slik at det blir sensitivt for fordelingen av forekomstene på symbolkategoriene.

I (3.8) er kompleksitetsbidraget fra hver symbolforekomst f i segmentet konstant med $\mathcal{C}(f) = \log s$ der s er antall symboler i *hele* alfabetet. Dette gjør at kompleksiteten av reserverte symboler blir avhengig av antallet deklarerte symboler. Vi kan unngå dette ved å la s være antallet symboler i *symbolforekomstens symbolkategori* og ikke i *hele alfabetet*, dvs $s = \mathcal{S}(\mathcal{K}(f), f,)$. s og dermed $\mathcal{C}(f)$ for en forekomst f av et reservert symbol vil da være uavhengig av antallet deklarerte symboler.

Dette gjør at vi får innført et skille mellom programmeringsspråkets kompleksitet (som er en funksjon av antallet reserverte symboler) og applikasjonens kompleksitet (som er en funksjon av antallet deklarerte symboler). Symboler fra små symbolkategorier vil bidra mindre til kompleksiteten enn symboler fra store kategorier. Ved å gruppere symbolene i symbolkategorier oppnår vi å bringe kompleksitetsmålet nærmere vår intuitive følelse av at kompleksitetsbidraget fra en forekomst av et reservert symbol er konstant fra program til program.

Definisjon 3.9 *La T være et segment med et alfabet som har n symbolkategorier $K_1 \cdots K_n$. Vi definerer da kompleksiteten til T som*

$$\mathcal{C}(T) = \sum_{i=1}^{n} \mathcal{R}(K_i, T) \log \mathcal{S}(K_i, T) \qquad (3.10)$$

3.8 Kompleksiteten av en programtekst

Vi utvider nå kompleksitetsmålet fra å gjelde et segment (kun ett navnerom) til å gjelde en generell programtekst (med flere navnerom). En programtekst vil etter våre definisjoner alltid være en sekvens av seksjoner, så vi kan utvide kompleksitetsmålet til å omfatte generelle programtekster ved å tillate T i (3.1) å være en generell programtekst P (dvs $T_1 \cdots T_n$ i (3.1) er seksjoner).

Definisjon 3.10 *La P være en generell programtekst som består av seksjonene $T_1 \cdots T_n$. Kompleksiteten til programteksten P er da gitt ved*

$$\mathcal{C}(P) = \sum_{i=1}^{n} \mathcal{C}(T_i) \qquad (3.11)$$

Intuitivt: Kompleksiteten til en programtekst er gitt ved summen av kompleksiteten til hver seksjon i programteksten. Kompleksiteten til hver seksjon beregnes ved (3.10). Med enkle modifikasjoner blir en kompilator et ypperlig verktøy for innsamling av de statistika som behøves for beregning av kompleksiteten av en seksjon etter (3.10). Kompleksiteten av konstruksjoner i programmet som består av flere seksjoner (flere navnerom) beregnes så etter (3.11) som summen av seksjonenes kompleksitet. Disse konstruksjonene vil være *klasser*, *prosedyrer* og ikke minst *hele programmet*.

3.9 Entropi

Gjennomsnittlig informasjonsmengde pr symbolforekomst kaller vi *entropi*.

Definisjon 3.11 *La n være antallet symbolforekomster i programteksten T. Vi definerer da entropi $\mathcal{H}(T)$ for T som*

$$\mathcal{H}(T) = \frac{\mathcal{C}(T)}{n}$$

$\mathcal{H}(T)$ sier oss noe om antallet entiteter vi må ha klart for oss når vi leser programteksten T. Variasjoner i $\mathcal{H}(T)$ skyldes hovedsaklig variasjoner i $\mathcal{S}(K,T)$ for dynamiske symbolkategorier K, så $\mathcal{H}(T)$ gir oss en pekepinn om den gjennomsnittlige størrelsen på alfabetet for T. Jo høyere $\mathcal{H}(T)$, jo mer semantikk må vi holde rede på når vi prøver å forstå T. For å unngå misforståelser bør man merke seg at Maus i [1] bruker betegnelsen entropi om $\mathcal{C}(T)$. Vår definisjon 3.11 av entropi er mer i tråd med den tradisjonelle bruken av begrepet [19].

3.10 Følsomhet for struktur

Ved å bruke lokale deklarasjoner fremfor globale der dette er mulig, knyttes deklarasjonene til de deler av kildekoden som faktisk bruker dem, og kildekoden blir mer strukturert. Lokale deklarasjoner bidrar til $\mathcal{C}(T)$ kun dersom de inngår i alfabetet til T ifølge reglene for leksikalsk skop, så $\mathcal{C}(T)$ vil belønne slike strukturforbedringer med redusert kompleksitet.

3.10.1 Tradisjonell programmering

Anta at et program med programtekst T er ustrukturert og ikke gjør bruk av prosedyrer eller lokale deklarasjoner. Alle deklarasjoner $D_1 \cdots D_n$ i T er globale. Dersom vi deler opp T i prosedyrer $P_1 \cdots P_m$, vil mange D_i kunne gjemmes bort inne i forskjellige P_j som lokale deklarasjoner, slik at det globale navnerommet avtar. Hver P_j introduserer imidlertid sitt prosedyrenavn i det globale navnerommet, så det eksisterer en optimal m som minimaliserer det globale navnerommet og $\mathcal{C}(T)$. Ved å sammenligne modeller av et ustrukturert program T og dets strukturerte ekvivalent T_s, utleder Maus i [1] det optimale antallet prosedyrer k_{opt} i T_s når prosedyrene har utsyn til globale variable:

$$k_{opt} = \sqrt{2n} - 1 \tag{3.12}$$

Han kommer videre fram til at k_{opt} vil være relativt liten sett i forhold til størrelsen av T når T er et *stort* program. Hver P_j i en slik optimal prosedyreinndeling blir derfor ubehagelig stor.

Ved å dele opp T i et antall *klasser* slik at $P_1 \cdots P_m$ er *klasseprosedyrer* tilhørende forskjellige klasser, kan vi dele opp T i flere prosedyrer (øke m) uten at det globale navnerommet øker. Dette fordi klasseprosedyrer ikke inngår det globale navnerommet, men i et navnerom lokalt til klassen. Objektorientert programmering har derfor klare fordeler over tradisjonell programmering av store systemer når fordelen måles som redusert gjennomsnittlig $\mathcal{C}(P_j)$ over alle $j \leq m$.

3.10.2 Objektorientert programmering

Klasser skiller seg fra andre datatyper ved at de knytter prosedyrer til datatypen. Hvis vi ser bort fra problematikken forbundet med arv, kan alle klasser implementeres som en `record` av type T med et sett prosedyrer som arbeider på variable av type T. Det er da vanskelig å få frem at disse prosedyrene representerer de eneste tillatte operasjoner på T. Når vi representerer T ved en klasse, kan vi få frem dette ved å la klassens instansvariable være beskyttet slik at de kun kan manipuleres av klassens prosedyrer. Klasseprosedyrene til T representerer da alle mulige operasjoner på T, og programmets struktur gjenspeiles derfor bedre i kildekoden.

Kompleksitetsmålet vil belønne implementasjon av T som en klasse fordi prosedyrene assosiert med T da er klasseprosedyrer (som ikke bidrar til det globale navnerommet). Klasseprosedyrene er i et navnerom lokalt til klassen. Når man ikke bruker klasser, vil prosedyrene som opererer på T inngå i det globale navnerommet. *Riktig bruk av klasser reduserer det globale navnerommet.* Klasseprosedyrene bringes inn i synsfeltet kun i forbindelse med operasjoner på objekter av klassen, og er ellers "ute av syne". Såfremt prosedyrene fordeles som klasseprosedyrer på et antall klasser, kan vi innføre et stort antall prosedyrer (slik at hver prosedyre blir av behagelig tekstlig størrelse) uten å bli straffet av kompleksitetsmålet slik tilfellet er ved tradisjonell programmering.

3.11 Test av kompleksitetsmålet

Hvis kompleksitetsmålet er i tråd med intuitiv kompleksitet, vil vi i lys av avsnitt 3.10.1 og 3.10.2 forvente å finne at prosedyrene i optimalt strukturerte *objektorienterte* programmer har en relativt fast størrelse, mens prosedyrene i optimalt strukturerte *tradisjonelle* programmer vil vokse med programmets totale størrelse. I [1] estimerer Maus at gjennomsnittlig prosedyrelengde for et objektorientert program vil være 5 til 16 setninger uavhengig av programmets størrelse. Vi ønsker å bruke dette som en test av kompleksitetsmålet ved å foreta målinger på et antall velstrukturerte objektorienterte og tradisjonelle systemer av en viss størrelse. Hvis vi finner signifikante forskjeller i prosedyrelengde og gjennomsnittlig kompleksitet pr prosedyre i favør av objektorientert programmering, vil vår tillit til kompleksitetsmålet styrkes.

3.12 Sammendrag

I dette kapitlet har vi definert kompleksitetsmålet i generelle termer uten å fokusere på noe bestemt programmeringsspråk. Den generelle definisjonen bygger på en inndeling av

- Programteksten i seksjoner.
- Alfabetet i symbolkategorier.

Når vi skal tilpasse målet for bruk på et bestemt programmeringsspråk, må vi spesifisere *hvordan* denne inndelingen skal gjøres for kildekode skrevet i språket. Dette er tema for kapittel 5, der målet tilpasses for bruk på C++ kildekode.

Vi har også sett hvordan optimalt strukturert kildekode bør se ut dersom kompleksitetsmålet gir et godt bilde av intuitiv kompleksitet slik den oppfattes av programmerere. Dette gir oss et grunnlag for å teste målet i kapittel 8 ved å utføre målinger på et par programsystemer.

Kapittel 4

Litt kompilatorteknikk

For å finne en hensiktsmessig strategi for implementasjon av Maus' kompleksitetsmål i G++ kompilatoren, skal vi i dette kapitlet gi en kort oversikt over nyttige teknikker som brukes i kompilatorer. Vi bør spesielt merke oss at kompilatorkonstruktører ofte gjør bruk av spesielle verktøy som forenkler arbeidet med å skrive en kompilator. I tilfellet G++ har opphavsmennene valgt å bruke verktøyet BISON for å lage den syntaksanalyserende delen av kompilatoren. BISON vil spille en viktig rolle i implementasjonen av kompleksitetsmålet.

4.1 Kompilatorer generelt

En kompilator har en kompleks oppgave. Den skal transformere et program skrevet i et språk A til et program med eksakt samme mening skrevet i et annet språk B. Programmet skrevet i språket A kalles *kildekoden*. Oversettelsen som er skrevet i språket B kalles *objektkoden*. For å kunne utføre denne oppgaven, må det foreligge en formell spesifikasjon av språkene A og B slik at en meningsbevarende oversettelse kan finne sted. Språket B er som oftest et maskinspråk for en prosessor, og er spesifisert av prosessor-produsenten.

Å skrive en kompilator er et omfattende foretagende som krever et godt gjennomtenkt design og de rette verktøy. Mye forskning har vært gjort rundt dette, og man har etter hvert kommet frem til en rimelig standard løsning for hvordan en kompilator bør være bygget opp. Løsningen er uavhengig av språkene A og B innen visse restriksjoner.

4.2 Kontekst-frie grammatikker

Formell spesifikasjon av språkene A og B er en nødvendighet hvis vi skal kunne garantere en meningsbevarende oversettelse. En slik spesifikasjon har to sider: Syntaks og semantikk. Vi skal her se bort fra hvordan man kan

beskrive et programmeringsspråks semantikk da det ikke er relevant for oppgaven, og kun konsentrere oss om den syntaktiske spesifikasjonen.

De fleste programmeringsspråks syntaks kan spesifiseres fullstendig i form av en *kontekstfri grammatikk*.

Definisjon 4.1 *En kontekstfri grammatikk angir alle former syntaktisk korrekt kildekode kan ha, og består av følgende komponenter:*

1. *En mengde symboler.*

2. *En mengde produksjoner. Hver produksjon er på formen* $V \rightarrow H_1 \cdots H_n$ *der* V *er et symbol kalt produksjonens* venstreside*, og* $H_1 \cdots H_n$ *er en sekvens av symboler kalt produksjonens* høyreside*.*

   Grammatikkens symboler kan deles opp i to disjunkte mengder. De symboler som bare opptrer i høyresider kalles terminalsymboler*. De symboler som opptrer i en eller flere venstresider kalles* ikke-terminaler*.*

3. *En ikke-terminal er utpekt som* startsymbol*.*

All kildekode skrevet i språket A er en sekvens av terminalsymboler fra A's grammatikk. For å vise hvordan en grammatikk angir formen for all syntaktisk korrekt kildekode, definerer vi først hva som menes med *substitusjon* og *avledning*.

Definisjon 4.2 *En substitusjon er en transformasjon av en sekvens av grammatikksymboler* $S_1 \cdots V \cdots S_m$ *til en sekvens* $S_1 \cdots H_1 \cdots H_n \cdots S_m$ *i henhold til en produksjon* $V \rightarrow H_1 \cdots H_n$*.*

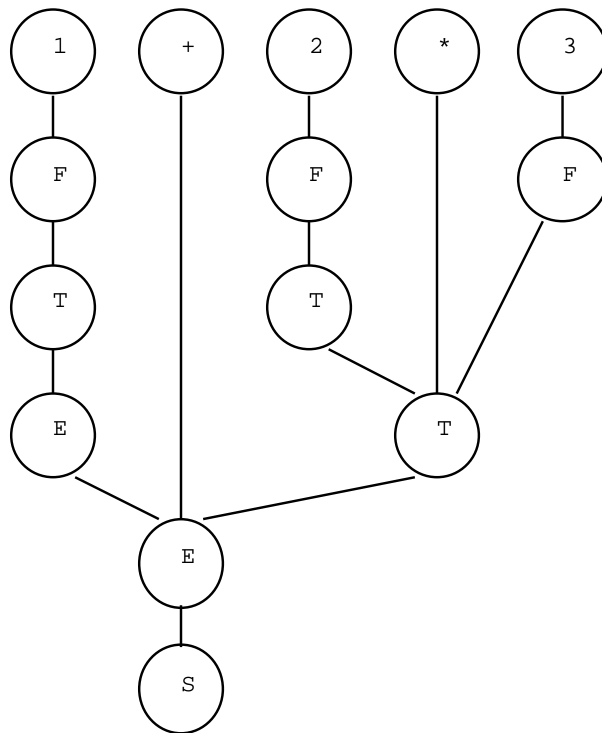
Definisjon 4.3 *En avledning er en serie substitusjoner utført på en sekvens av grammatikksymboler slik at sekvensen transformeres til en ny sekvens.*

Grammatikken definerer språket til å være de sekvenser av terminalsymboler som kan avledes fra startsymbolet, og utgjør en *induktiv* definisjon av språket. Pseudo-terminalsymbolet ϵ brukes for å betegne en tom tekst, og produksjoner $V \rightarrow \epsilon$ kalles *epsilon-produksjoner*. Epsilon-produksjoner kan betraktes som produksjoner med $n = 0$ grammatikksymboler i høyresiden, og brukes for å terminere rekursive definisjoner.

Figur 4.1 viser en grammatikk for enkle aritmetiske uttrykk der symbolet $\rightarrow$ er erstattet med :. Denne grammatikken gjør bruk av ikke-terminalene `S`,`E`,`T` og `F`, med `S` som startsymbol. `TALL` er et terminalsymbol som representerer et vilkårlig siffer. En avledning kan illusteres ved hjelp av et såkalt *parse-tre*. Figur 4.2 viser et parse-tre som illustrerer hvordan uttrykket `1+2*3` kan avledes fra startsymbolet `S` i grammatikken i figur 4.1. Noder for terminalsymbolet `TALL` inneholder det korresponderende sifferet.

```
S : E
E : E '+' T
E : T
T : T '*' F
T : F
F : '(' E ')'
F : TALL
```

Figur 4.1: En grammatikk med startsymbol S for enkle aritmetiske uttrykk



Figur 4.2: Parse-tre for et uttrykk

4.3 Et standard design for kompilatorer

I tråd med kompilatorens oppgave som translatør fra et språk til et annet, deles kompilatoren ofte inn i to deler der den ene delen fokuserer på kildekoden, mens den andre delen fokuserer på objektkoden.

Front-end leser kildekoden tegn for tegn, og avdekker hvordan kildekoden er avledet fra grammatikkens startsymbol. Denne prosessen kalles *parsing*, og resultatet av parsingen kan presenteres i form av et parse-tre som i figur 4.2.

Back-end genererer objektkode i språket B basert på parse-treet, og foretar som regel optimaliseringer underveis slik at objektkoden blir mest mulig effektiv. Back-end er sterkt knyttet til språket B, og er isolert fra språket A gjennom parse-treet. Denne delen av en kompilator er derfor av liten interesse for denne oppgaven, da vi er ute etter å måle kompleksitet av kildekode skrevet i språket A.

Ut fra denne beskrivelsen kan man få et inntrykk av at alle kompilatorer

- Først bygger et parse-tre.
- Deretter genererer objektkoden som en funksjon av parse-treet.

Det finnes imidlertid kompilatorer som ikke genererer parse-trær i det hele tatt, for eksempel kompilatorer skrevet etter recursive-descent prinsippet (se [20]). Parse-trær er et hjelpemiddel til å forstå hvordan en kompilator gir struktur til programteksten, og ikke alle kompilatorer lar denne strukturen nedfelle seg i en eksplisitt node-struktur som så tolkes av en back-end. Parse-treet kan altså være mer eller mindre implisitt.

4.4 Kompilatorens front-end

Kildekode foreligger i praksis som en sekvens av enkelt-tegn (som oftest en ASCII tekstfil). Alle grammatikker for programmeringsspråk kan derfor spesifiseres med ASCII alfabetets tegn i rollen som terminalsymboler. Dette er imidlertid lite hensiktsmessig all den tid de minste enhetene i syntaksen til et programmeringsspråk er reserverte ord og deklarerte identifikatorer. Da er det mer fruktbart å bruke de reserverte ordene og de deklarerte identifikatorene som terminalsymboler. Det er derfor vanlig at kompilatorens front-end splittes opp i to deler:

Leksikalsk analysator leser kildekoden tegn for tegn og genererer terminalsymbolene i språket A's grammatikk.

Parser leser terminalsymbolene fra den leksikalske analysatoren, og avdekker kildekodens struktur.

Som regel er man ikke ute etter en representasjon av kildekodens struktur i form av et parse-tre, men et resultat (objektkoden!) av en kildekode med en slik struktur. Man kan gradvis bygge opp det ønskede resultatet ved at parseren tilkjennegir kildekodens struktur ved å utføre en bestemt *aksjon* hver gang den avdekker at en produksjon er blitt brukt i avledningen av kildekoden fra startsymbolet.

Hvert grammatikksymbol har da et sett attributter som kalles grammatikksymbolets *semantiske verdi*. Hver node i parsetreet inneholder således den semantisk verdien til det utsnittet av kildekoden som noden representerer. Aksjonene er ansvarlige for å beregne semantisk verdi etterhvert som parse-treet tar form. Når parse-treet er ferdig konstruert, inneholder nodenes attributter informasjon som gjør det mulig å generere den endelige oversettelsen.

For å avdekke kildekodens struktur leser parseren symbol for symbol fra den leksikalske analysatoren. Under denne prosessen refererer man til det sist leste terminalsymbolet som *lookahead-symbolet*. Det finnes et stort antall algoritmer for parsing av kontekstfrie språk. Enkelte kontekstfrie språk kan parses med enkle og effektive algoritmer, mens andre krever mer avanserte og tungrodde algoritmer. Algoritmene kan grovinndeles i to kategorier etter hvilken del av parse-treet de konstruerer først.

Bottom Up parsing avdekker parse-treets struktur fra bladene og oppover mot roten.

Top Down parsing avdekker parse-treets struktur fra roten og nedover mot bladene.

Bottom Up parsingalgoritmer er kraftigere enn Top Down algoritmer, og brukes i en rekke verktøy (parsergeneratorer) som genererer parsere automatisk ut fra en gitt grammatikk med aksjoner tilknyttet produksjonene. G++ bruker en maskingenerert Bottom Up parser. Vi vil derfor forbigå Top Down parsere i stillhet, og kun fokusere på virkemåten for en Bottom Up parser.

4.5 Bottom Up parsing

Ved Bottom Up parsing konstrueres parse-treet fra bladene og oppover mot roten. Vi skisserer her virkemåten for såkalte LR parsere. En LR parser gjør bruk av en stack. Denne stacken inneholder grammatikksymboler, og er initielt tom. Et grammatikksymbol $S(N)$ på stacken er assosiert med en foreldreløs node N i det uferdige parse-treet. En LR parser bygger parse-treet ved hjelp av de to operasjonene *shift* og *reduksjon*.

- En shift-operasjon etablerer en node N med $S(N)$ lik look-ahead-symbolet. Deretter legges $S(N)$ øverst på stacken, og et nytt look-ahead-symbol leses inn fra den leksikalske analysatoren.

- En reduksjon fjerner først de øverste $S(N_1)\cdots S(N_i)$ symbolene fra stacken. Disse symbolene tilsvarer høyresiden i produksjonen $S(N) \rightarrow S(N_1)\cdots S(N_i)$. Deretter etableres noden N med de eksisterende nodene $N_1 \cdots N_i$ som avkom, og til slutt legges S(N) øverst på stacken.

Sterkt forenklet foregår parsingen ved at man shift'er inntil toppen av stacken gjenkjennes som høyresiden i en produksjon. Da foretas en reduksjon hvor symbolene i høyresiden tas av stacken, og erstattes med venstresiden. Denne prosessen fortsetter inntil hele kildekoden er lest og stacken er tom. Man har da avdekket parse-treet for kildekoden.

4.6 BISON — et kompilatorverktøy

Basert på den skisserte standard-oppdelingen av en kompilator, har det dukket opp et vell av verktøy som forenkler implementasjonen av en kompilator. Å lage en `LR` parser for et programmeringsspråk involverer implementasjon av en tilstandsmaskin med typisk flere hundre tilstander. Arbeidet med å lage transisjonstabellen for denne tilstandsmaskinen kan med fordel overlates til en datamaskin, og det har vært laget en rekke verktøy som genererer en LR parser basert på grammatikken med aksjoner. G++ bruker en LR parser generert av programmet BISON [21].

BISON konstruerer en LALR parser basert på en grammatikk med aksjoner tilknyttet produksjonene. En LALR parser er en LR parser med visse restriksjoner på grammatikken slik at implementasjonen av parseren blir mest mulig effektiv. Umiddelbart før BISON utfører en reduksjon i henhold til en produksjon $S(N) \rightarrow S(N_1)\cdots S(N_i)$, utføres den tilhørende aksjonen. Aksjonen har tilgang til attributtene for de tenkte nodene N og $N_1 \cdots N_n$. Fordi parsetreet bygges opp fra bunnen (Bottom Up), vil attributtene til $N_1 \cdots N_n$ være kjent på dette tidspunktet, og attributtene til N kan beregnes fra attributtene til $N_1 \cdots N_n$. På denne måten syntetiseres attributtene opp gjennom parse-treet.

En aksjon spesifiseres som en sammensatt setning i programmeringsspråket C. Aksjonen har tilgang til attributtene gjennom de predefinerte pseudovariablene `$$` (for N) og `$1` ...`$n` (for $N_1 \cdots N_n$). Dersom en produksjon ikke har angitt noen aksjon, utføres automatisk tilordningen `$$=$1`. Typen til pseudovariablene avhenger av grammatikksymbolet de er assosiert med — forskjellige grammatikksymboler kan ha behov for forskjellige attributter. I utgangspunktet har pseudovariablene typen `int` for alle grammatikksymbolene. En kalkulator basert på grammatikken i figur 4.1 kan da implementeres i BISON som vist i figur 4.3.

```
S : E           {printf ("%d\n",$1);};
E : E '+' T     {$$=$1+$3;};
E : T;
T : T '*' F     {$$=$1*$3;};
T : F;
F : '(' E ')'   {$$=$2;};
F : TALL;
```

Figur 4.3: En kalkulator for enkle aritmetiske uttrykk i BISON

Ut fra en slik spesifikasjon genererer BISON en transisjonstabell for en tilstandsmaskin. Transisjonstabellen brukes av prosedyren `YYPARSE`, som implementerer en LALR parser. `YYPARSE` kaller en eksternt definert funksjon `YYLEX` for å lese et terminalsymbol hver gang den foretar en shift operasjon. Hver gang `YYPARSE` foretar en reduksjon, utføres den reduserende produksjonens aksjon, og attributter beregnes.

For kalkulatoren i figur 4.3, vil hver node i parse-treet inneholde verdien av det deluttrykket noden representerer. Noder i bunnen av treet (som representerer terminalsymboler) får sine attributter satt av den leksikalske analysatoren `YYLEX`. Før `YYLEX` returnerer terminalsymbolet, legger den terminalsymbolets attributter i den globale variablen `yylval` som avleses av `YYPARSE` ved hver shift operasjon. I eksemplet i figur 4.3 må `YYLEX` sørge for å gi `yylval` riktig verdi før terminalsymbolet `TALL` returneres. "Riktig verdi" er verdien av tallkonstanten terminalsymbolet representerer.

BISON er Free Software Foundations implementasjon av YACC. Verktøyet YACC så dagens lys i begynnelsen av 1970-årene og brukes i dag i et stort antall kompilatorer. I tråd med filosofien til Free Software Foundation, er BISON laget med henblikk på at den skal være mest mulig "åpen" og lett å modifisere. En viktig forskjell mellom YACC og BISON er at kildekoden for BISONs `YYPARSE` funksjon er tilgjengelig og kan modifiseres, mens YACC har `YYPARSE` i kompilert form i et bibliotek.

Fordi kildekoden for `YYPARSE` er tilgjengelig, er det mulig å modifisere `YYPARSE` til å utføre visse standard aksjoner som utføres ved *hver eneste reduksjon* etter at aksjonen tilknyttet produksjonen er utført. Muligheten for slike standard aksjoner spiller en viktig rolle i vår implementasjon av kompleksitetsmålet.

4.7 Oversettelsesskjema

Spesifikasjonen av grammatikkens produksjoner og tilhørende aksjoner kalles et *oversettelsesskjema*. Figur 4.3 angir således et oversettelsesskjema for verdien av et konstantuttrykk. Et oversettelsesskjema definerer oversettelsen induktivt på grammatikken, og angir derfor hvordan *enhver* lovlig kildekode

kan oversettes. Den bærende idéen i denne oppgaven er å utarbeide et oversettelsesskjema for beregning av kompleksitet av C++ kildekode. Dette oversettelsesskjemaet må veves inn det oversettelsesskjemaet for objektkode som allerede eksisterer i G++, og blir tema for kommende kapitler.

Kapittel 5

Kompleksitetsmålet i lys av C++

Kompleksitetsmålet er ikke bundet til noe bestemt programmeringsspråk, og ble derfor beskrevet i generelle termer ved gjennomgåelsen i kapittel 3. Før vi kan måle kompleksitet av kildekode skrevet i et bestemt språk (i vårt tilfelle C++), må vi bringe klarhet i hvordan målet skal anvendes på slik kildekode. Vi må ta stilling til

- Hvordan vi skal dele opp alfabetet i *symbolkategorier*.
- Hvordan vi skal dele opp en vilkårlig programtekst i *seksjoner*.

Disse vurderingene vil tilpasse den generelle definisjonen av kompleksitetsmålet til C++, og er hovedtema for dette kapitlet. Vi ser først på hvordan alfabetet for en C++ programtekst kan deles inn i symbolkategorier, og vurderer deretter hvordan seksjonsbegrepet bør anvendes på C++.

Arbeidet vil i stor grad ta utgangspunkt i språkdefinisjonen for C++ [22]. G++ kompilatoren implementerer C++ versjon 2.0. Denne versjonen av C++ mangler noen av mekanismene som beskrives i [22], blant annet parametriserte typer (såkalte *templates*). Disse mekanismene har ingen interesse ved implementasjon av kompleksitetsmålet i G++, og vil ikke bli belyst.

C++ inneholder mange spesielle mekanismer som dekkes dårlig av den generelle definisjonen av kompleksitetsmålet. Vi tenker her spesielt på såkalt *overloading* av prosedyrer og operatorer. Den siste delen av kapitlet diskuterer hvordan kompleksitetsmålet bør håndtere slike spesialiteter.

5.1 Bruk av identifikatorer i C++

Motivasjonen for å innføre symbolkategorier var å gjøre kompleksiteten til en symbolforekomst ufølsom for antallet symboler i andre kategorier (se avsnitt

```
int compare (int a,int b)
{
  if (a > b)
    goto a;
  if (b > a)
    goto b;
  return (0);
  a: return (1);
  b: return (-1);
}
```

Figur 5.1: Samme identifikator som merkelapp og variabel

3.6). Dermed oppnår vi for eksempel at kompleksiteten til forekomster av velkjente matematiske operatorer som $+$ og $-$ vil holde seg konstant og uavhengig av for eksempel antall deklarerte variabler. Vår kategorisering av symbolene i alfabetet vil i stor grad basere seg på de kategoriene vi skisserte i avsnitt 3.6.

Når det gjelder den skisserte symbolkategorien for deklarerte identifikatorer, må vi imidlertid ta hensyn til at én og samme identifikator kan brukes til å omtale flere logiske entiteter i et C++ program. Hvilken entitet identifikatoren referer til vil da være avhengig av kontekst. Denne "gjenbruk" av identifikatorer kan skje ved

- Nesting av deklarasjoner i flere nivåer.
- Overloading av prosedyrer.
- Bruk av virtuelle prosedyrer.
- Bruk av samme navn på entiteter av forskjellig entitetstype. Samme identifikator kan for eksempel opptre i rollen som merkelapp (for `goto`), strukturert type og variabel i samme segment.

Slik gjenbruk av identifikatorer blir gjenstand for diskusjon fordi den passer dårlig inn i det bildet vi presenterte i kapittel 3. Der antok vi at antall symboler i alfabetet gir et godt bilde av mengden semantikk vi må holde rede på. Ved gjenbruk av identifikatorer vil antall symboler bare være en nedre grense for mengden semantikk.

Bruk av samme navn på entiteter av forskjellig entitetstype er spesielt interessant for inndelingen av alfabetet i symbolkategorier, og behandles i neste avsnitt. De andre tilfellene behandles etterhvert som vi omtaler språkkonstruksjonene de opptrer i (prosedyrer og klasser).

5.2 Symbolkategorier for C++

```
struct point
{
  int x,y;
} point;
```

Figur 5.2: Samme identifikator som strukturnavn og variabelnavn

Figur 5.1 viser en prosedyre der en identifikator opptrer som navn på mer enn én entitet. Prosedyren `compare` gjør bruk av identifikatorene `a` og `b` i to forskjellige roller:

- Som variablene `a` og `b`.
- Som merkelappene `a` og `b`

For å avgjøre hvilken entitet identifikatorene refererer til, må kompilatoren ta hensyn til hvilken kontekst identifikatorene forekommer i. Figur 5.2 viser en lignende situasjon der en variabel og en strukturert type bruker samme identifikator. Figuren inneholder en deklarasjon av entitetene:

- Variablen `point`.
- Strukturnavnet `point`.

Eksemplene viser at identifikatorer kan opptre som navn på mange forskjellige typer deklarerte entiteter. Nærmere bestemt kan de deklarerte identifikatorene i et C++ program fungere som navn på følgende typer entiteter:

- Variabler.
- Prosedyrer.
- Typenavn deklarert med `typedef`.
- Enumererte konstanter deklarert med `enum`.
- Merkelapper som markerer målet for hopp med `goto`.
- Strukturer deklarert med `struct`, `union`, `enum` og `class`.

Disse *entitetstypene* kan igjen deles inn i tre grupper. Innen hver slik gruppe har alle entiteter som er deklarert på samme nivå unike identifikatorer, mens entiteter fra forskjellige grupper kan ha identiske identifikatorer (til tross for at de er deklarert på samme nivå). I C++ har vi tre slike *entitetsgrupper*:

- Identifikatorer som betegner variabler, prosedyrer, typenavn og enumererte konstanter. Vi kaller slike identifikatorer *navn*.

- Identifikatorer som betegner målet for hopp ved `goto`. Vi kaller slike identifikatorer *merkelapper*.

- Identifikatorer som betegner strukturer deklarert med `struct`, `union`, `enum` og `class`. Vi kaller slike identifikatorer *strukturnavn*. På engelsk brukes betegnelsen *tags*.

Ved deklarasjon av en struktur ved hjelp av nøkkelordet `struct`, `union`, `enum` eller `class` gjør en C++ kompilator automatisk en `typedef` av et *typenavn* med samme identifikator som *strukturnavnet*. Deklarasjon av strukturer gir dermed opphav til både et *navn* og et *strukturnavn*. Dette gjøres av bekvemmelighetshensyn fordi et *typenavn* innført ved `typedef` kan brukes direkte i deklarasjoner. Til sammenligning må et *strukturnavn* alltid prefikses med `struct`, `union`, `enum` eller `class` ved deklarasjon av entiteter av typen. Vi skal betrakte en identifikator som et *strukturnavn* kun når den forekommer etter et av disse reserverte ordene. Alle andre forekomster av identifikatorer er enten *navn* eller *merkelapper*. Den eneste grunnen til at man har strukturnavn i C++ skyldes at man vil beholde bakoverkompatibilitet med programmeringsspråket C.

I kapittel 3 definerte vi en symbolkategori som en mengde symboler, og antok uten videre at hvert symbol i denne mengden representerte *én* entitet. Identifikatorene kan ikke utgjøre en symbolkategori etter denne definisjonen fordi samme symbol kan betegne *flere* forskjellige entiteter avhengig av kontekst. Vi må splitte opp identifikatorene i flere symbolkategorier. Dette understøttes av at vi lett ser av kontekst hvilken entitet en gitt identifikatorforekomst betegner:

- Merkelapper opptrer kun som argument til `goto` og i prosedyrekropper etter et kolon.

- Strukturnavn opptrer kun etter nøkkelordene `struct`, `union`, `enum` og `class`.

På samme måte som kompleksiteten av reserverte ord bør være uavhengig av antallet deklarerte identifikatorer, bør derfor kompleksiteten til identifikatorer i entitetsgruppe A kun være avhengig av antallet entiteter i A og uavhengig av antallet entiteter i andre entitetsgrupper.

Det synes nå naturlig å splitte symbolkategorien *deklarerte identifikatorer* fra avsnitt 3.6 opp i tre symbolkategorier — en symbolkategori for hver entitetsgruppe. Vi får da følgende symbolkategorier for C++:

1. Reserverte ord.

2. Operatorer.
3. Navn.
4. Merkelapper (labels).
5. Strukturnavn (tags).
6. Tekstlige literaler.
7. Numeriske literaler.

Dette konkluderer vurderingene omkring inndelingen av alfabetet i symbolkategorier. Det gjenstår nå å vurdere hvordan vi kan dele opp enhver C++ kildekode i seksjoner slik at kompleksiteten til hele programteksten kan beregnes etter definisjon 3.10. Et godt utgangspunkt for disse vurderingene får vi ved å se nærmere på det bildet en C++ kompilator har av kildekoden.

5.3 C++ programtekst

Et program skrevet i C++ kan bestå av mange kildefiler som kompileres separat slik at man får produsert en objektfil pr kildefil. Når alle kildefilene er kompilert, sammenføyes objektfilene til én utførbar programfil av en *linker*. Fordi kompilatoren bare får se innholdet av én kildefil hver gang den kjøres, kan den bare utføre kompleksitetsberegninger på hver kildefil. Den kan ikke regne ut kompleksiteten til hele programmet med mindre programmet består av kun én kildefil. Vi løser dette problemet ved at kompilatoren skriver informasjon om kildefilens kompleksitet til fil slik at denne informasjonen blir gjort permanent og ikke forsvinner når kompilatoren terminerer.

Hver kildefil vil nå i tillegg til en objektfil få generert en "kompleksitetsfil" som inneholder detaljer om kompleksiteten til kildefilen. Kompleksitetsfilen vil inneholde kompleksiteten til de prosedyrer og klasser som deklareres i den korresponderende kildefilen samt kompleksiteten til hele kildefilen. Når alle kildefilene som inngår i programmet er kompilert, kan vi danne oss et bilde av kompleksiteten til hele programmet ved å se på kompleksitetsfilene kompilatoren har etterlatt seg. Kompleksiteten til hele programmet kan beregnes som summen av kompleksiteten til hver kildefil (som kan leses inn fra den tilhørende kompleksitetsfilen). Kompleksitetsfilens format spesifiseres i vedlegg C.

5.4 Seksjonsbegrepet

Når vi skal dele opp en kildefil i seksjoner, er det nyttig å ha klart for seg at programteksten ikke forstås tegn for tegn, men i *brokker* (se avsnitt 2.7).

Vi vil bruke språkets regler for leksikalsk skop som en rettesnor for oppdelingen i seksjoner, men bare ned til et visst detaljnivå. Når et lite utsnitt T av programteksten (for eksempel en sekvens av enkle setninger) naturlig oppfattes som en brokke, vil vi ikke bryte opp T i mindre seksjoner selv om alfabetet for T ikke er konstant. Lokale deklarasjoner i T kan gjøre at alfabetet i T varierer litt fra sted til sted i T, men i lys av teorien om programforståelse ved hypotesetesting, blir det allikevel mest naturlig å betrakte T som en seksjon. Brokkebegrepet vil således fungere som moderator når vi klarlegger begrepet seksjon med utgangspunkt i programmeringsspråkets regler for leksikalsk skop. Spesielt i tilfeller med liten tekstlig avstand mellom deklarasjoner, faller det naturlig å legge mindre vekt på skopreglene, og mer vekt på at deklarasjonene og bruken av dem utgjør en naturlig brokke. For å forstå et lite utsnitt må vi ha klarhet i samtlige entiteter som deklareres der, og hvordan disse lokale deklarasjonene er sekvensielt plassert i teksten blir mindre interessant fordi teksten skumleses mange ganger.

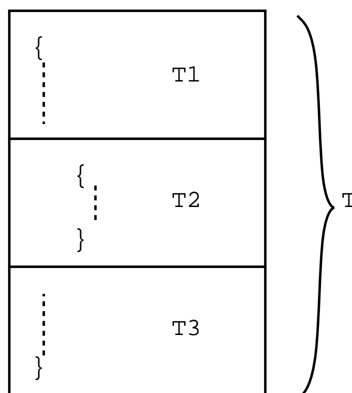
5.5 Språkets seksjonskonstruksjoner

Vi kan spesifisere hvordan en kildefil skal inndeles i seksjoner ved å angi en mengde L av ikke-terminaler fra programmeringsspråkets grammatikk. Et utsnitt T av en programtekst vil utgjøre en seksjon dersom T er avledet fra et av symbolene i L uten at noen av substitusjonene i avledningen introduserer symboler inneholdt i L. Dette svarer til at en seksjon ikke kan inneholde andre seksjoner. Hver ikke-terminal i L representerer en konstruksjon i språket som avledes til én eller flere seksjoner, og vi kaller en slik språklig konstruksjon for en *seksjonskonstruksjon*. Som vi senere skal se, kan en sammensatt setning gjøre tjeneste som en seksjonskonstruksjon i C++.

En instans T av en seksjonskonstruksjon inneholder andre seksjonskonstruksjoner dersom substitusjonene i avledningen av T introduserer symboler inneholdt i L. En slik T utgjør ikke *én* seksjon, men *flere* seksjoner. En sammensatt setning kan for eksempel inneholde en annen sammensatt setning, og består da av *flere* seksjoner. Figur 5.3 viser hvordan en sammensatt setning T som inneholder en sammensatt setning T_2 kan deles inn i seksjonene T_1, T_2 og T_3. T_1 og T_3 ville utgjort én seksjon i fravær av T_2, og vi skal derfor anta at T_1 og T_3 har identisk alfabet. Dette er en viktig observasjon som vi benytter når vi i avsnitt 6.5 skisserer hvordan målet kan implementeres som et oversettelsesskjema for *kompleksitet* basert på et eksisterende oversettelsesskjema for *objektkode*.

La oss kalle inndelingen av en programtekst T i seksjoner for $\mathcal{A}(T)$.

Definisjon 5.1 *$\mathcal{A}(T)$ for programtekst T er en sekvens av programtekster $T_1 \cdots T_n$ som er slik at $T = T_1 T_2 \cdots T_n$ og alle T_i er seksjoner.*



Figur 5.3: Inndeling av en sammensatt setning T i seksjoner T_1, T_2 og T_3 når T inneholder en sammensatt setning T_2.

$\mathcal{A}(T)$ avhenger av seksjonskonstruksjonene for programmeringsspråket, og vi forklarer nå hvordan vi kan finne $\mathcal{A}(T)$ for enhver syntaktisk korrekt T.

- Dersom T inneholder instanser av seksjonskonstruksjoner, men ikke selv utgjør en seksjon, kan T skrives på formen

$$T_1 K_1 T_2 K_2 \cdots T_n K_n T_{n+1} \tag{5.1}$$

  der K_i er instanser av seksjonskonstruksjoner (for eksempel en sammensatt setning), og T_i er utsnitt av programteksten som *ikke* inneholder seksjonskonstruksjoner (merk at et utsnitt T_i kan være et tomt utsnitt). Vi har da

$$\mathcal{A}(T) = T_1 \mathcal{A}(K_1) T_2 \mathcal{A}(K_2) \cdots T_n \mathcal{A}(K_n) T_{n+1} \tag{5.2}$$

- Ellers har vi

$$\mathcal{A}(T) = T \tag{5.3}$$

Ved å anvende $\mathcal{A}(T)$ på innholdet T av en kildefil, får vi delt opp kildefilen i seksjoner. Vår oppgave blir nå å identifisere *alle* seksjonskonstruksjonene for C++ slik at $\mathcal{A}(T)$ er definert for enhver C++ programtekst T. Vi har allerede foreslått at sammensatte setninger er seksjonskonstruksjoner. Vi ønsker å finne de resterende seksjonskonstruksjonene ved å gå systematisk gjennom grammatikken for C++, og begynner derfor med å se på strukturen til C++ kildefiler, som er avledninger fra C++ grammatikkens startsymbol.

5.6 Eksterne deklarasjoner

En C++ kildefil er en sekvens *eksterne deklarasjoner* av

- Variable.
- Prosedyrer.
- Datatyper.

"Eksterne" refererer til at deklarasjonene ikke gjøres inne i andre deklarasjoner, og derfor opptrer utenfor (eksternt) alle deklarasjoner. En ekstern deklarasjon kan variere kraftig i tekstlig størrelse, og omfatte alt fra en enkel variabeldeklarasjon til en kompleks prosedyredeklarasjon med stor prosedyrekropp. Omfangsrike deklarasjoner vil typisk inneholde mange refereranser til andre deklarasjoner. I forbindelse med beregning av kompleksitet er det viktig å merke seg at en ekstern deklarasjon D kun kan referere til entiteter som er deklarert tidligere i programteksten. Foroverreferanser til entiteter deklarert eksternt senere i filen tillates ikke i D. Slike entiteter er derfor ikke med i alfabetet til D. En ekstern deklarasjon D representerer en introduksjon av et nytt symbol, og gjør at D og etterfølgende deklarasjoner har et utvidet alfabet (deklarasjoner kan omtale seg selv i forbindelse med rekursivitet). Deklarasjoner som gjøres inne i eksterne deklarasjoner kaller vi *lokale* deklarasjoner. Lokale variable i prosedyrer utgjør en stor del av de lokale deklarasjonene.

La oss inntil videre anta at ingen ekstern deklarasjon D inneholder lokale deklarasjoner eller andre elementer som gjør at alfabetet i D varierer. D vil da utgjøre en *seksjon* (se definisjon 3.3) fordi

1. Alfabetet er det samme overalt i D. I henhold til definisjon 3.2 er D dermed et segment.
2. Segmentet som utgjøres av D kan ikke utvides til å omfatte neste eksterne deklarasjon E i kildefilen fordi E har et alfabet som er utvidet (med symbolet deklarert i E). Eksterne deklarasjoner utgjør derfor maksimale segmenter, som pr definisjon 3.3 er seksjoner.

I tråd med avsnitt 5.5 utgjør derfor eksterne deklarasjoner seksjonskonstruksjoner — en ekstern deklarasjon gir alltid opphav til *minst én seksjon.* Vi vil i det følgende analysere de forskjellige typer eksterne deklarasjoner en C++ kildefil kan inneholde. Gjennom denne analysen vil vi få avdekket eventuelle andre seksjonskonstruksjoner i C++. En C++ kildefil kan inneholde følgende typer eksterne deklarasjoner:

- Prosedyredeklarasjoner.

```
int max (int a, int b);

int max (int a, int b)
{
  if (a > b)
    return (a);
  else
    return (b);
}
```

Figur 5.4: Deklarasjon og definisjon av en prosedyre

- Klassedeklarasjoner.
- Enumeratordeklarasjoner.
- `typedef` deklarasjoner.

Utgangspunktet for analysen er å undersøke om de forskjellige typene eksterne deklarasjoner kan huse lokale deklarasjoner. I praksis inneholder de fleste eksterne deklarasjoner lokale deklarasjoner, og en slik deklarasjon utgjør derfor ikke en seksjon hvis vi legger leksikalsk skop til grunn for inndeling i seksjoner. Vi må vurdere hvordan eksterne deklarasjoner kan deles opp i seksjoner avhengig av deres lokale deklarasjoner. Av de nevnte deklarasjonene er spesielt prosedyre- og klasse-deklarasjoner interessante, da de utgjør mesteparten av programteksten og typisk inneholder mange lokale deklarasjoner.

5.7 Eksterne definisjoner

En ekstern deklarasjon kan ta form av en *definisjon*. Definisjoner er interessante fordi de typisk inneholder mange lokale deklarasjoner og passer dårlig med antagelsen om at eksterne deklarasjoner ikke inneholder lokale deklarasjoner.

En kildefil kan inneholde mange eksterne deklarasjoner av samme entitet (prosedyre, variabel etc), men kun én av disse deklarasjonene kan være en definisjon. Definisjoner inneholder en fullstendig beskrivelse av entiteten, mens deklarasjoner som ikke er definisjoner kun gir en delvis spesifikasjon av entiteten, og typisk brukes for å omgå restriksjonene på forover-referanser ved implementasjon av rekursive datatyper og prosedyrer.

Figur 5.4 viser to eksterne deklarasjoner av prosedyren `max`, der kun den siste deklarasjonen er en definisjon. Den første deklarasjonen av `max` inneholder all informasjon kompilatoren trenger for å verifisere at et kall på `max` er korrekt angitt, mens *definisjonen* i tillegg spesifiserer prosedyrekroppen og på denne måten angir hviken effekt prosedyren har når den kalles.

```
struct point;

struct point {int x,y;};
```

Figur 5.5: Deklarasjon og definisjon av en type

Typer kan også deklareres og defineres. Figur 5.5 viser to deklarasjoner av klassen `point`, der kun den siste deklarasjonen er en definisjon. Den første deklarasjonen av `point` angir bare eksistensen av klassen slik at man kan deklarere *pekere* til objekter av klassen. *Objekter* av klassen kan ikke deklareres før klassedefinisjonen er lest, for det er først *der* det spesifiseres hvordan objekter av klassen skal representeres i maskinhukommelsen.

For å spesifisere den deklarerte entiteten fullt ut, vil definisjoner som oftest inneholde lokale deklarasjoner av:

- Lokale variable for prosedyrer.
- Attributter for klasser.

Vi skal derfor i det følgende se nærmere på de forskjellige formene for eksterne definisjoner som forekommer i C++ og hvordan seksjonsbegrepet kan anvendes på dem. Målet er transformere kildefilen fra en sekvens eksterne deklarasjoner til en sekvens seksjoner ved å dele opp hver eksterne deklarasjon i seksjoner. Da kan kompleksiteten til en vilkårlig programtekst beregnes etter definisjon 3.10.

5.8 Prosedyredefinisjoner

Prosedyredefinisjoner er prosedyredeklarasjoner som skiller seg fra andre prosedyredeklarasjoner ved at de i tillegg til *prosedyrehodet* spesifiserer *prosedyrekroppen*.

Prosedyrehodet spesifiserer prosedyrens navn, returtype og parametre. Parameterdeklarasjonene er lokale til prosedyredefinisjonen, og i prosedyrekroppen brukes parametre som om de var lokale variable. I motsetning til de fleste andre språk tillater C++ *anonyme* parametre som ikke kan refereres av prosedyrekroppen (se [22] r.8.3). Dette er tenkt brukt ved vedlikeholdsarbeid når en parameter blir gjort overflødig. Vi kan markere at en parameter er tatt ut av bruk ved å utelate parameternavnet fra prosedyredefinisjonens prosedyrehode. En slik anonym parameterdeklarasjon sender et signal til kildekodens lesere om at prosedyrekroppen ikke bruker parameteren. Samtidig er parametrenes antall og type uendret slik at vi unngår å endre alle kallene på prosedyren.

Prosedyrekroppen har form av en sammensatt setning, og angir den setningssekvens som blir utført når prosedyren kalles. I C++ kan en setning være en lokal deklarasjon, og prosedyrekroppen kan derfor i egenskap av å være en setningssekvens inneholde lokale deklarasjoner. En slik lokal deklarasjon betraktes som en setning fordi den kan angi en initialverdi for den deklarerte entiteten, og deklarasjoner reduseres da til en spesiell form for tilordningssetninger.

I forbindelse med standard prosedyreargumenter (omtales i avsnitt 5.16) og brukerdefinerte datatyper, kan prosedyrehodet referere til andre deklarasjoner. Bortsett fra definisjoner av `inline` prosedyrer i klassedefinisjoner (omtales i avsnitt 5.10), er alle prosedyredefinisjoner i C++ eksterne. Prosedyrehodet kan derfor bare referere til *eksterne* deklarasjoner, så den dynamiske delen av alfabetet for et prosedyrehode består av

- Symbolene for prosedyrens parametre.
- Symboler innført ved ekstern deklarasjon foran prosedyrehodet.

På grunn av muligheten for anonyme parametre, viste det seg vanskelig å få G++ kompilatoren til å bestemme parameterdeklarasjonenes bidrag til prosedyrehodets alfabetet. Vi valgte derfor å utelukke parametrene fra alfabetet for prosedyrehodet, og brukte antallet globale deklarasjoner som en tilnærming. Dette gjelder også for prosedyredeklarasjoner som ikke spesifiserer prosedyrekroppen.

I prosedyredefinisjoner inkluderes parametrene i alfabetet for *prosedyrekroppen*, som utgjør den interessante delen av prosedyredefinisjoner. Vi skal nå se nærmere på prosedyrekropper og sammensatte setninger generelt.

5.9 Sammensatte setninger

Anta at vi har en sammensatt setning T som inneholder n lokale deklarasjoner. Dersom vi legger rent leksikalske betraktninger til grunn for oppdelingen av T i seksjoner, vil T ha minst n seksjoner — hver lokale deklarasjon utvider alfabetet og gir opphav til en seksjon. En slik betraktning ser på arbeidet med å forstå T som én sekvensiell gjennomlesing av T hvor forståelse oppnås tegn for tegn — lokale deklarasjoner som vi ikke har lest enda bidrar ikke til kompleksitet. I lys av teorien om programforståelse ved hypotesetesting (se avsnitt 2.7) synes dette urimelig. Ifølge denne teorien vil T bli skumlest mange ganger, og antallet entiteter man har i sin mentale database ved gjennomlesing (se avsnitt 3.2) vil omfatte alle lokalt deklarerte entiteter. Dersom T ikke inneholder andre instanser av seksjonskonstruksjoner, tilsvarer dette at alfabetet er det samme i hele T og at T utgjør en seksjon. Sammensatte setninger er derfor seksjonskonstruksjoner. Dersom en sammensatt

```
int a;
void f (void)
{
  int a;
  {
    int a;
    S;
  }
}
```

Figur 5.6: Nesting av lokale deklarasjoner som gir skygging

setning inneholder andre seksjonskonstruksjoner (for eksempel andre sammensatte setninger), vil setningen splittes opp i flere seksjoner etter prinsippet i avsnitt 5.5 som vist i figur 5.3.

5.9.1 Nesting av sammensatte setninger

Sammensatte setninger (og dermed lokale deklarasjoner) kan nestes til vilkårlig dybde. En setning har utsyn til alle entiteter som er deklarert foran den

1. i den sammensatte setningen som setningen inngår i.
2. i omsluttende sammensatte setninger.
3. eksternt.

Vi har inntil nå antatt at entiteter i samme entitetsgruppe har unike identifikatorer (se avsnitt 5.2). Nestede sammensatte setninger som inneholder lokale deklarasjoner kan deklarerere flere entiteter med samme identifikator i samme entitetsgruppe (på forskjellig nivå i nestingen), og bryter derfor med denne antagelsen. Vi skal nå se nærmere på hvordan kompleksitetsmålet bør håndtere dette.

At en setning har utsyn til en entitet A betyr ikke nødvendigvis at setningen faktisk kan omtale entiteten A ved dens identifikator. Hvis en deklarert entitet faller inn under punkt 2 eller 3 foran, kan den bli *skygget for* av entiteter med samme navn. Figur 5.6 viser et eksempel på slik skygging.

Referanser til deklarerte entiteter foregår utelukkende ved navn, og siden setningen `S` i følge regelen over har utsyn til *tre* entiteter med samme navn `a`, følger at kun én av disse vil være tilgjengelig som `a` i `S`. Dette vil være den `a` som er deklarert nærmest `S`.

Når en programmerer leser en programtekst der enkelte deklarasjoner skygger for andre deklarasjoner som i figur 5.6, må han assosiere flere entiteter med samme navn, og ved hjelp av språkets skopregler slutte seg til hvilken entitet som omtales på et gitt sted. Å identifisere hvilken entitet som

```
class circle:public shape
{
  private:
  struct point {int x,y;};
  point center;
  int radius;
  public:
  double area (void);
  circle (point _center,int _radius)
  {center = _center; radius = _radius;}
};

double circle::area (void) {return (PI*radius*radius);}
```

Figur 5.7: En klassedefinisjon

omtales blir nå mer komplisert: I tillegg til at entiteter fra forskjellige entitetsgrupper kan ha identiske identifikatorer, kan også entiteter innen samme entitetsgruppe ha identiske identifikatorer dersom de deklareres på forskjellig nivå. Selv om kun én entitet er tilgjengelig under navnet `a` i `S`, kan vi ikke under feilsøking ignorere de `a`-entitetene som skygges for. Det kan tenkes at en eventuell feil skyldes nettopp skygging: En lokal variabel er introdusert, og kom i skade for å skygge for en global variabel. Dermed kan det hende at kode som tidligere aksesserte den globale variablen nå aksesserer den lokale, og prosedyrens effekt er blitt endret drastisk av en tilsynelatende uskyldig introduksjon av en ny variabel. For å oppdage feil som skyldes skygging, må programmereren i tillegg til de entiteter som faktisk kan omtales direkte ved navn også ha oversikt over de entiteter som skygges for.

Kompleksitetsmålet er en funksjon av antallet navngitte entiteter vi må ha forståelse av for å kunne gi mening til kildekoden og finne eventuelle feil (se avsnitt 3.2). Ved beregning av kardinalitet $\mathcal{S}(K,T)$ for segmentet T må vi derfor telle med *alle* entiteter i kategorien K som segmentet T har utsyn til, også de entiteter som skygges for. Dette representerer et avvik fra definisjon 3.7 av kardinalitet, der vi antok at hvert symbol i K refererer til kun én entitet. C++ inneholder også en annen mekanisme som gjør at vår innledningsvise definisjon av kardinalitet bør modifiseres: Såkalt overloading av prosedyrer og operatorer gjør at prosedyrer og operatorer kan forekomme i mange varianter med samme navn, og vi venter derfor med å redefinere kardinalitet inntil overloading er blitt behandlet i avsnitt 5.15.

5.10 Klassedefinisjoner

En klassedefinisjon spesifiserer hvilke attributter klassens objekter skal ha, og hvilke deler av kildekoden som skal ha adgang til hvilke av disse attributtene. Figur 5.7 viser et eksempel på en klassedefinisjon.

Attributtene innføres ved lokale deklarasjoner i klassedefinisjonen samt eventuelt ved arv fra eksisterende klasser. Klassen `circle` i eksemplet arver attributter fra klassen `shape`, og introduserer dessuten nye attributter ved deklarasjoner lokalt i `circle`. Disse attributtdeklarasjonene innfører:

- Instansvariable. Alle instanser (objekter) av klassen inneholder et separat sett av disse variablene, som dermed bestemmer hvordan klassens objekter representeres i maskinens hukommelse. I eksemplet utgjør `center` og `radius` klassens instansvariable. Instansvariable som er deklarert `static` inngår ikke i hvert objekt, men deles av alle objektene (se [22] r.9.4). De er derfor egentlig ikke instansvariable, men spesielle globale variable som er tilknyttet en bestemt klasse.

- Klasseprosedyrer. Kroppen til disse prosedyrene har adgang til klassens attributter, og man innskrenker gjerne adgangen til klassens instansvariable slik at kun klassens prosedyrer kan manipulere dem. Dette markeres med nøkkelordene `private` og `protected`. Alle attributtdeklarasjoner som følger etter `private`, vil kun være synlige for klassens prosedyrer. Attributtdeklarasjoner som følger etter `protected` kan i tillegg manipuleres av prosedyrer i subklasser. Denne beskyttelsesmekanismen beskytter attributter mot utidig manipulasjon. I eksemplet er instansvariablene `center` og `radius` beskyttet på denne måten. Når alle instansvariable er beskyttet, kan omverdenens bruk av objektet kun skje gjennom kall på klasseprosedyrene. Figur 5.7 definerer klasseprosedyrene `area` og `circle` etter nøkkelordet `public`. Disse prosedyrene er da ubeskyttede og tilgjengelige for alle brukere av klassen. Klasseprosedyren `circle` utgjør klassens *konstruktor*, og initialiserer objekter av klassen.

- Typer. Lokale deklarasjoner av typer i en klasse brukes typisk av klassens instansvariable. Instansvariable er ofte ikke synlige utenfra. Da kan det hende at eventuelle definisjoner av instansvariablenes type er lite interessante utenfor klassen, og kan gjemmes bort inne i klassen. I eksemplet er `point` en lokal type, som imidlertid ikke er helt uinteressant utenfor klassen, da et punkt må angis som argument til konstruktoren når et `circle` objekt deklareres. Mange implementasjoner av C++ betrakter derfor slike lokale typedeklarasjoner som globale deklarasjoner (se [22] r.18.3.5).

Klassens prosedyrer blir som regel ikke *definert*, men bare *deklarert* inne i klassedefinisjonen. Den fulle prosedyredefinisjonen gis som en ekstern deklarasjon prefikset med klassens navn, og kan legges i en annen kildefil enn klassedefinisjonen hvis ønskelig. I figur 5.7 blir klasseprosedyren `area` definert under navnet `circle::area` utenfor klassedefinisjonen. I praksis legges ofte klassedefinisjoner i en fil h, mens de tilhørende prosedyredefinisjonene

legges i en separat kildefil for hver klasse. En kildefil kan da bruke klassene ved å inkludere *h* gjennom direktivet `#include<h>` fordi klassedefinisjonene i *h* forteller kompilatoren hvordan klassenes prosedyrer kalles og hvordan klassene skal representeres i hukommelsen. Vi kommer tilbake til praktisk bruk av multiple kildefiler i 5.14.

5.10.1 Foroverreferanser i attributtdeklarasjoner

Hvis en prosedyre *defineres* inne i en klassedefinisjon, er dette et hint til kompilatoren om at prosedyrens kode skal ekspanderes på hvert kallsted og ikke implementeres som kall på en subrutine. Prosedyren `circle` i figur 5.7 er et eksempel på en slik `inline` prosedyre. `inline` prosedyrer opptrer ofte i sammenheng med at man ønsker at all adgang til objekter skal skje gjennom kall på klassens prosedyrer. Hvis alle prosedyrekall ble implementert ved kall på subrutiner, kunne dette resultere i dårlig effektivitet for helt enkle observatorfunksjoner som kanskje ikke gjør annet enn å returnere en av objektets instansvariable. Med `inline` prosedyrer unngår man effektivitetstapet ved subrutinekall, og kan derfor akseptere at også helt enkle observatorfunksjoner angis som klasseprosedyrer. Fordi objektkoden vil inneholde en kopi av `inline` prosedyren for hvert kall på den, er det viktig at `inline` prosedyrer er enkle og korte slik at ikke programmets størrelse vokser alt for mye.

Det er et prinsipp at kroppen til en klasseprosedyre skal ha tilgang til de av klassens attributter som ikke er arvet fra andre klasser. Når kompilatoren leser kroppen til en `inline` prosedyre definert i klassedefinisjonen, er ikke hele klassedefinisjonen lest enda, og prosedyrekroppen kan derfor referere til attributter som kompilatoren enda ikke har sett. Vi må derfor vente med å beregne kompleksiteten av en slik prosedyrekropp inntil hele klassedefinisjonen er lest. *Først da* vet vi størrelsen på det alfabetet prosedyrekroppen har utsyn til.

La oss midlertidig se bort fra prosedyrekroppene som forekommer i eventuelle prosedyredefinisjoner inne i klassedefinisjoner. Slike prosedyrekropper må uansett behandles spesielt. Vi kommer tilbake til hvordan dette løses under gjennomgangen av de modifikasjoner som ble gjort på G++ kompilatoren. I mellomtiden minner vi om at foroverreferanser representerer et problem også ved kompilering, slik at en løsning allerede må foreligge i en C++ kompilator.

Når vi ser bort fra prosedyrekropper i attributtdeklarasjonene, kan en attributtdeklarasjon bare referere til eksterne deklarasjoner og andre attributter som er deklarert foran den. Reglene for referanser mellom attributtdeklarasjoner er de samme som for referanser mellom eksterne deklarasjoner — forover-referanser tillates ikke. I eksemplet er variablen `center` deklarert til å ha type `point`, der `point` er en datatype som er innført ved en *forutgående* attributtdeklarasjon i klassen. En attributtdeklarasjon muliggjør

```
enum color
{
  red=0xFF0000,green=0x00FF00,blue=0x0000FF,
  white=red|green|blue
};
```

Figur 5.8: En enumeratordefinisjon

omtale av attributtet i etterfølgende attributtdeklarasjoner, og representerer derfor en utvidelse av alfabetet. Alfabetet i klassedefinisjonen vil altså vokse etterhvert som nye attributter innføres ved deklarasjon. Ut fra betraktninger om leksikalsk skop kan derfor ikke en klassedefinisjon med flere attributter utgjøre en seksjon. Klassedefinisjonen er imidlertid å betrakte som en naturlig brokke av teksten. Vi velger derfor å legge mindre vekt på det leksikalske aspektet, og betrakter en klassedefinisjon som en seksjonskonstruksjon.

5.11 Andre eksterne definisjoner

Prosedyre- og klasse-definisjoner utgjør mesteparten av teksten i et C++ program. Det finnes også andre typer eksterne deklarasjoner, men disse er på langt nær så interessante, da de ikke kan inneholde setninger, og sjelden representerer noen tankemessig utfordring. Vi omtaler dem allikevel i korthet slik at vi får en komplett omtale av alle de forskjellige typene eksterne deklarasjoner.

5.11.1 Enumeratordefinisjoner

En enumeratordefinisjon innfører et sett navngitte konstanter og eventuelt et typenavn (typenavnet kan sløyfes). Typens verdimengde er mengden av konstantene. Figur 5.8 viser et eksempel på en enumeratordefinisjon.

Dersom `color` deklareres eksternt, representerer dette en introduksjon av fem nye navn i det globale navnerommet:

- Strukturnavnet `color`.
- Enumeratorkonstantene `red`, `green`, `blue` og `white`.

Deklarasjonene av enumeratorkonstantene blir derfor å betrakte som eksterne deklarasjoner selv om de forekommer inne i definisjonen av den enumererte typen. Som vi ser av eksemplet, kan verdien til en enumeratorkonstant angis som et konstant-uttrykk der tidligere deklarerte enumeratorkonstanter kan inngå. Hver deklarasjon av en enumeratorkonstant representerer derfor en utvidelse av alfabetet. Vi velger å se bort fra dette rent

```
typedef char *string;

string p;
```

Figur 5.9: En typedef definisjon

leksikalske aspektet og ønsker i stedet å betrakte enumeratordefinisjonen i sin helhet som en brokke (se avsnitt 2.7).

Enumeratorkonstantene er ikke lokale deklarasjoner, men "flyter" ut i det omkringliggende navnerommet, slik at en enumeratordefinisjon ikke gir opphav til noe lokalt navnerom. Vi velger derfor å *ikke* betrakte enumeratordefinisjoner som seksjonskonstruksjoner. En *lokal* enumeratordefinisjon utgjør *ikke* en seksjon, men en *ekstern* vil utgjøre en seksjon i egenskap av å være en ekstern deklarasjon.

5.11.2 Typedef definisjoner

I C++ er to typer kompatible dersom de har samme struktur, og variable og prosedyrer deklareres med en kryptisk notasjon som spesifiserer typen. I deklarasjonen `char *p`, angir `*` foran `p` at `p` er en peker til en `char`. Dersom variable av mer sammensatte typer skal deklareres, blir deklarasjonene fort uleselige. For å hjelpe på dette, har man gjort det mulig å definere typenavn som så kan brukes som en erstatning for den kryptiske deklarator-notasjonen.

I figur 5.9 deklareres typenavnet `string` som kan brukes til å deklarere variable av denne typen, og dette øker lesbarheten. En `typedef` definisjon kan ikke inneholde lokale deklarasjoner, og introduserer kun ett nytt navn. Vi betrakter derfor ikke `typedef` definisjoner som seksjonskonstruksjoner.

5.12 Aksess til objekters attributter

En klasses instansvariable blir instansiert for hvert objekt av klassen, og må derfor omtales i tilknytning til et bestemt objekt. Man bruker notasjonen `a.b` for å omtale attributtet `b` for objektet `a`. Dersom `a` er en peker til et objekt, benyttes formen `a->b`. Samme notasjon brukes når `b` er en klasseprosedyre.

Man kan deklarere attributter som ikke instansieres for hvert objekt, men deles av alle klassens objekter. Dette gjelder for eksempel instansvariable som er deklarert `static`, og datatyper lokale til klassen. Slike attributter omtales ved at man knytter dem til *klassen* og ikke et *objekt* av klassen. Man bruker da notasjonen `a::b` for å omtale attributtet `b` for klassen `a`.

Operatoren `::` brukes dessuten til å løse opp tvetydigheter som oppstår når attributter fra arvede klasser har samme navn. Når klassene `a` og `c` begge

deklarerer attributtet `b`, og klassen `d` arver klassene `a` og `c`, blir referanser til `b` i klasseprosedyrene for `d` tvetydige, og må endres til `a::b` og `c::b`.

C++ ser på hele programmet som en slags klasse der de eksterne deklarasjonene utgjør attributtene. I tråd med dette refererer uttrykket `::b` til den eksternt deklarerte entiteten `b`. Denne notasjonen er nyttig ved omtale av eksternt deklarerte entiteter i klasseprosedyrer. Man kan da fritt innføre nye attributter i klassen uten å tenke på om de nye attributtene skygger for eksterne deklarasjoner og dermed utilsiktet endrer klasseprosedyrenes effekt.

Operatorene i uttrykkene `a.b`, `a->b` og `a::b` representerer en innsnevring av alfabetet for symbolforekomsten `b`. `b` *må* nå være et attributt i en bestemt klasse. I uttrykket `::b` *må* `b` være deklarert eksternt. Vi ønsker at kompleksitetsmålet skal belønne objektorientert programmering ved å ta hensyn til denne innsnevringen av alfabetet. Dette får vi til ved å betrakte `b` som en seksjonskonstruksjon. $\mathcal{C}(b)$ blir da redusert fordi kardinaliteten til symbolkategorien for `b` er begrenset til antall *synlige attributter* i `a` (antall *eksterne deklarasjoner* i tilfellet `::b`). Legg merke til at dette vil belønne bruk av nøkkelordene `private` og `protected` for å begrense innsynet til attributtene. Vi føler at en slik belønning er riktig fordi begrensning av innsynet til attributtene forenkler vedlikeholdet av kildekoden ved at klassene blir mer uavhengige av hverandre.

5.13 Sære detaljer ved C++

Vi har nå sett hvilke seksjonskonstruksjoner det er hensiktsmessig å definere for C++, og vi har definert symbolkategoriene. Dette burde i utgangspunktet være tilstrekkelig for å tilpasse målet til bruk på et bestemt programmeringsspråk. C++ inneholder imidlertid en del sære detaljer som bør belyses.

- For at kompilatoren skal kunne foreta typesjekking av referanser til entiteter som defineres i andre filer, benytter man *headerfiler*. Headerfiler er et ganske primitivt konsept som C++ har arvet fra C. Vi skal se at den tradisjonelle bruken av headerfiler er uheldig for kompleksitetsmålet og krever spesialbehandling.

- Overloading av prosedyrer og operatorer leder til at samme symbol kan betegne mange entiteter. I kapittel 3 antok vi at antallet deklarerte symboler ga et godt bilde av antallet deklarerte entiteter. Overloading bryter ned denne antagelsen, og vi må vurdere hvordan målet skal håndtere overloading.

Vi fokuserer først på problemene forbundet med headerfiler, og gjør deretter noen betraktninger omkring overloading.

```
/* fil h */

extern void Pa (void);
extern void Pb (void);
```

```
/* fil a */

#include"h"
void Pa (void) {Pb();}
```

```
/* fil b */

#include"h"
void Pb(void) {Pb();}
```

Figur 5.10: Innholdet av filene a, b og h som illustrerer bruken av headerfiler

5.14 Headerfiler

En prosedyre `Pa` definert i filen `a` kan kalle en prosedyre `Pb` definert i filen `b` kun dersom fil `a` inneholder en deklarasjon av `Pb`. `Pb` må deklareres i `a` før den kan kalles fra `Pa`. Da kompilatoren ikke har tilgang til fil `b` når den kompilerer fil `a`, kan den ikke kontrollere at deklarasjonen av `Pb` i `a` stemmer overens med definisjonen av `Pb` i `b`. Hvis deklarasjonen av `Pb` i `a` avviker fra definisjonen av `Pb` i `b` (for eksempel forskjellig antall parametre), vil kompilatoren generere feil kode for kall på `Pb` fra `Pa`. Man garderer seg mot feil deklarasjon av `Pb` i `a` ved å legge deklarasjonen av `Pb` i en headerfil `h` som inkluderes i både `a` og `b` (se [22] avsnitt 4.3). Filene `a`, `b` og `h` blir som i figur 5.10.

Hvis deklarasjonen i `h` er feil, vil man få feil under kompilering av definisjonen av `Pb` i `b` fordi deklarasjonen av `Pb` i `h` ikke stemmer overens med definisjonen. Deklarasjonen av `Pb` i `h` som brukes under kompilering av fil `a` vil derfor alltid være korrekt, og kompilatoren vil sjekke at alle kall på `Pb` i `a` er syntaktisk korrekte.

En klassedefinisjon spesifiserer en datatype som kan benyttes i mange datafiler. En kildefil `f` kan benytte klassen dersom klassedefinisjonen legges i en headerfil som inkluderes i `f`. I lys av dette kan vi forvente at headerfiler hovedsaklig inneholder prosedyredeklarasjoner og klassedefinisjoner.

5.14.1 Konsekvenser for kompleksitetsmålet

I tråd med den omtalte teknikken for kontroll av prosedyrekall på kryss av kildefiler, består mange C++ programmer av kildefilene $f_1 \cdots f_n$ og en prosjekt-headerfil h som inneholder deklarasjoner av alle prosedyrer som defineres i $f_1 \cdots f_n$. Alle filene $f_1 \cdots f_n$ begynner med en inklusjon av h, og enhver f_i kan derfor inneholde kall på prosedyrer som er definert i en vilkårlig f_j.

Preprosessor-direktivet `#include"h"` i begynnelsen av hver kildefil f_i vil innføre en stor mengde prosedyrenavn i navnerommet for f_i. La oss betegne prosedyrene som deklareres i h som $P_1 \cdots P_m$. Hvis antallet kildefiler n er

høyt, er sannsynligheten liten for at en fil f_i omtaler alle P_j. Inklusjon av h i f_i vil typisk deklarere et stort antall prosedyrer P_j som aldri blir omtalt i f_i. Dette kan gjøre at det globale navnerommet i f_i blir meget stort sett i forhold til innholdet av f_i. f_i utgjør imidlertid bare en *del* av et program, og vi finner det rimelig at $\mathcal{C}(f_i)$ er sensitiv for programmets totale størrelse. Antallet deklarasjoner som innføres ved inklusjon av h er en funksjon av programmets totale størrelse, og vil derfor bevirke at $\mathcal{C}(f_i)$ varierer med programmets totale størrelse og ikke bare innholdet av f_i.

Dersom f_i gjør bruk av rutiner fra bibliotek, inkluderer f_i dessuten andre headerfiler (for eksempel `stdio.h` og `math.h`) som inneholder deklarasjoner av rutiner i biblioteket. Slike headerfiler skal vi kalle system-headerfiler. Vi kan skille prosjekt- og system-headerfiler fra hverandre ved at prosjekt-headerfilene typisk inkluderes med direktivet `#include "h"`, mens system-headerfilene inkluderes med direktivet `#include <h>`. Sistnevnte syntaks angir at filen h befinner seg i en av flere mulige systemspesifiserte filkataloger. Forskjellen i syntaks representerer en mulighet for å differensiere mellom headerfiler når vi tester kompleksitetsmålet i praksis. System-headerfiler inneholder deklarasjoner av generelle, mest mulig uavhengige rutiner som ikke er knyttet til noe bestemt prosjekt. Det synes derfor urimelig å la dem virke inn på $\mathcal{C}(f_i)$ på samme måte som prosjekt-headerfiler. Fordi deklarasjonene i system-headerfiler typisk innfører uavhengige entiteter, synes det naturlig å la en slik deklarasjon medvirke til kardinalitet bare dersom den faktisk benyttes i f_i.

I lys av diskusjonen over, må alle deklarasjoner under kompilering av f_i få assosiert et flagg som indikerer om de har blitt omtalt i kildefilen f_i. Deklarasjoner som gjøres i f_i eller i en prosjekt-headerfil får dette flagget satt umiddelbart, mens deklarasjoner som foretas i en system-headerfil først får flagget satt når kompilatoren ser at deklarasjonen benyttes av f_i. Kun deklarasjoner som har dette flagget satt telles med i størrelsen av navnerommet ved beregningen av $\mathcal{C}(f_i)$.

5.15 Overloading

C++ tillater at man definerer flere prosedyrer med samme navn sålenge prosedyrenes parameterlister er forskjellige nok til at man alltid kan avgjøre hvilken prosedyredefinisjon et bestemt prosedyrekall refererer til. Dette kalles *overloading*, og brukt riktig kan det bidra til å øke kildekodens lesbarhet.

Figur 5.11 definerer *to* prosedyrer med navnet `print` — én for utskrift av tekst, og én for utskrift av heltall. Når kompilatoren kommer over et kall på `print`, undersøker den typen til den aktuelle parameteren. Hvis det er en peker til en tekst, konkluderer kompilatoren at vi ønsker å kalle den første prosedyren. Er parameteren et heltall, genereres et kall på den siste prosedyren.

```
void print (char *s)
{
  printf ("%s",s);
}

void print (int n)
{
  printf ("%d",n);
}
```

Figur 5.11: Overloading: To prosedyrer med samme navn

Overloading er kjent fra matematikken, der de samme operatorene brukes om flere forskjellige operasjoner som har sterke felles trekk. For eksempel brukes addisjonsoperatoren `+` til addisjon av både reelle og komplekse tall. Argumentenes type gjør det klart hvilken operasjon man sikter til. Alternativet er å innføre en operator for hver operasjon. Ved å bruke kun én operator drar man nytte av at operasjonene representerer samme abstraksjon.

5.15.1 Riktig bruk av overloading

I likhet med matematiske operatorer kan to prosedyrer representere en operasjon som må *implementeres* litt forskjellig avhengig av argumentenes type. Den *abstrakte* effekten skal være den samme. Effekten av prosedyren `print` i eksemplet er i begge tilfeller å skrive ut argumentets verdi. Hvordan dette skal gjøres rent praktisk avhenger imidlertid av argumentets type, slik at operasjonen må spesifiseres som to separate prosedyrer. Det overlates så til kompilatoren å velge hvilken prosedyre som skal benyttes i hvert kall.

Vi sa tidligere at overloading kan bidra til økt lesbarhet hvis det brukes riktig. I lys av diskusjonen over kan vi si mer om hva vi mener med *riktig* i denne sammenheng. En prosedyre skal representere en abstraksjon og ha en bestemt abstrakt effekt. Derfor er det viktig at to prosedyrer med samme navn har den samme effekten, og at overloading bare brukes når dette er tilfelle.

5.15.2 Overloading av operatorer

C++ tilbyr dessuten mulighet for overloading av *operatorer*, slik at man kan definere de velkjente matematiske operatorene for egendefinerte datatyper, og på denne måten dra nytte av operatorenes innarbeidede mening. Hvis man for eksempel vil definere + operatoren for en egendefinert datatype `complex` for komplekse tall, deklarerer man en prosedyre som i figur 5.12.

Man kan deretter beregne summen av to komplekse tall `a` og `b` ved uttrykket `a+b`. Man bør utvise meget stor forsiktighet ved overloading av op-

```
complex operator+ (complex a,complex b)
{
  return (complex (a.real + b.real,a.imag + b.imag));
}
```

Figur 5.12: Operator overloading

eratorer. Hvis man for eksempel skulle komme i skade for å introdusere en side-effekt i operatoren, vil man fort få programfeil som er meget vanskelige å finne. Ens egen oppfatning av operatoren som rent applikativ er så innarbeidet at den blir til "den blinde flekken" når man leter etter feilen.

5.15.3 Overloading i lys av kompleksitetsmålet

Anta nå at vi har en kildekode der overloading er brukt "riktig". Vi kunne da i utgangspunktet tenke oss å "belønne" overloading av et prosedyrenavn `a` med nedsatt kompleksitet for kildekoden som helhet. Dette kunne for eksempel gjøres ved å telle alle prosedyredefinisjonene for `a` som én entitet og ikke separate entiteter.

Dette forutsetter imidlertid at alle prosedyredefinisjonene for `a` faktisk implementerer samme abstraksjon. I en situasjon hvor vi prøver å finne en feil i et program, kan vi ikke uten videre anta dette. Feilen kan skyldes at en av prosedyrene for `a` faktisk har en litt annen effekt enn vi forventer, og vi må derfor se på hver prosedyredefinisjon separat. Vi har ikke oppnådd noen forenkling av dette arbeidet ved å bruke overloading — situasjonen er som om vi hadde definert alle prosedyrene med forskjellige navn. Hvis vi i tillegg har definert operatorer for egendefinerte datatyper, er ikke en gang forekomster av enkle operatorer som $+$ og $-$ nødvendigvis korrekte.

5.15.4 Redefinisjon av kardinalitet

Vi vil la kompleksiteten forbundet ved overloading av operatorer og prosedyrer komme til uttrykk i kompleksitetsmålet ved at

- Symbolkategorien for operatorer utvides med én for hver overloading av en operator. Hver forekomst av en operator vil da representere en større kompleksitet jo flere varianter av operatorene vi har deklarert.

- Symbolkategorien for prosedyrer (dvs symbolkategorien *navn*) utvides med én for hver overloading av en prosedyre. Hver referanse til en prosedyre vil da representere en større kompleksitet jo flere ganger vi gjør bruk av overloading.

```
void print (int n = 0)
{
  printf ("%d",n);
}
```

Figur 5.13: Standard argument til en prosedyre

```
void print (int n)
{
  printf ("%d",n);
}

void print (void);
{
  printf ("%d",0);
}
```

Figur 5.14: Standard argument implementert ved overloading

Konklusjonen blir altså at hver operatordefinisjon og prosedyredefinisjon bør bidra til kardinalitet i kompleksitetsbetraktningene selv om den representerer overloading og altså ikke introduserer noe nytt symbol i den tilhørende symbolkategorien.*Dette utgjør en redefinisjon av begrepet kardinalitet slik det opprinnelig ble innført i definisjon 3.7.*

Definisjon 5.2 *La K være en symbolkategori og T et segment. Vi definerer da $\mathcal{S}(K,T)$ som antallet entiteter som kan refereres til av de symbolene s i alfabetet for T som har $\mathcal{K}(s) = K$. Dette antallet skal inkludere entiteter som blir skygget for av entiteter med samme identifikator.*

Definisjon 5.2 erstatter heretter definisjon 3.7.

5.16 Standard prosedyreargumenter

C++ tillater at man unnlater å angi enkelte aktuelle parametre i et prosedyrekall. Verdien til de uspesifiserte argumentene må da være angitt i prosedyredeklarasjonen som *standard argumenter* som vist i figur 5.13.

Kallene `print()` og `print(0)` er nå ekvivalente. Vi observerer at *bruk av standard argumenter innebærer en implisitt overloading av prosedyren*: Man oppnår samme effekt ved å deklarere *to* funksjoner som i figur 5.14

Ved bruk av standard argumenter, må man være ekstra påpasselig med parametrene i prosedyrekall. Det er nå blitt "farlig enkelt" å gjøre et kall med gal parameterverdi ved simpelthen å glemme en parameter i prosedyrekallet.

Dette gjør at man har mer å forholde seg til, og vi kan la denne økte kompleksiteten komme til uttrykk ved at vi tar hensyn til den implisitte overloading som finner sted: For hvert standard argument i prosedyrehodet tenker vi oss en implisitt deklarert prosedyre som telles med i antallet entiteter. Definisjonen av `print` med ett standard argument i figur 5.13 vil telle som definisjon av *to* entiteter som i figur 5.14.

5.17 Preprosessering

En kompilator leser normalt kildekoden direkte fra kildefilen A. Kompilatorer for programmeringsspråkene C og C++ fraviker dette ved at kildefilen først må prosesseres av et annet program, den såkalte *C preprosessoren* (CPP). CPP leser kildefilen A, og lager en temporær fil B som deretter leses av kompilatoren (i noen systemer opprettes istedet en såkalt *pipe* mellom CPP og kompilatoren). Innholdet i B tilsvarer innholdet i A med visse endringer i henhold til *direktiver* til CPP i kildefilen A. Ved hjelp av direktiver i A kan man dermed "massere" den kildekoden kompilatoren leser. Dette er praktisk i mange situasjoner:

- Ved *betinget kompilering* (conditional compilation), vil angitte deler av filen A ikke bringes videre til filen B hvis et predikat P ikke er oppfylt. Variablene i P kan styres fra kommandolinjen, og dermed kan man manipulere kildekoden (og dermed det ferdige programmet) direkte fra kommandolinjen når man starter kompilatoren. Dette utnyttes ofte til å lage forskjellige versjoner av et program basert på samme kildekode.

- CPP lar deg definere *makroer*. En makro er en funksjon fra en tekst T_A til en annen tekst T_B. Ved å definere en makro ber man CPP om å oversette enhver forekomst av teksten T_A i filen A til teksten T_B i filen B. Makroer kan ha parametre, og brukes da ofte som en substitutt for `inline` prosedyrer.

- Innholdet av en angitt fil H kan injiseres i kildekoden på et angitt sted. Når CPP påtreffer et slikt direktiv, skriver den det prosesserte innholdet av H til B før den fortsetter å overføre innholdet av A til B i henhold til direktivene i A.

Preprosessering gjør at kompilatoren potensielt leser en annen kildekode enn en menneskelig leser. Spesielt er dette et problem ved inklusjon av headerfiler, der kompilatoren får servert mye mer tekst enn en menneskelig leser. Hvis kompilatoren kan detektere at en del av B stammer fra en inkludert fil, kan den kompensere for dette. Vi skal se at CPP legger igjen informasjon i fil B som gjør det mulig å detektere denne situasjonen og korrigere i kompilatoren.

5.18 Sammendrag

I dette kapitlet har vi vurdert hvordan kompleksitetsmålet kan anvendes på C++ kildekode. De to viktigste punktene i denne vurderingen er inndelingen av alfabetet i symbolkategorier og oppdelingen av kildefiler i seksjoner basert på seksjonskonstruksjonene for C++. Vi kom fram til følgende seksjonskonstruksjoner for C++:

- Ekstern deklarasjon.
- Klassedefinisjon.
- Sammensatt setning.
- Bruk av operatorene `.`, `->` og `::`.

Videre så vi hvordan headerfiler og overloading bør håndteres. Den viktigste beslutningen som ble tatt i dette kapitlet var nok at symboler som innføres ved deklarasjoner i system-headerfiler bare skal telles med i navnerommet dersom de faktisk brukes i den inkluderende kildefilen.

I lys av teknikkene for kompilatorkonstruksjon som ble introdusert i kapittel 4, kan vi nå skifte fokus over mot implementasjon av målet i en eksisterende kompilator.

Kapittel 6

Metode

Dette kapitlet har til hensikt å skissere en metode for implementasjon av Maus' kompleksitetsmål ved modifikasjon av en eksisterende kompilator. Metoden kan brukes på enhver kompilator som er basert på en tabelldrevet LR parser, f.eks. parsere laget med BISON.

6.1 Hovedidé

For å kunne regne ut kompleksiteten av kildekode som er skrevet i et bestemt programmeringsspråk, må vi opplagt ha en definisjon av kompleksitet av slik kildekode som vi kan basere utregningene på. Denne definisjonen av kompleksitet må være tuftet på definisjonen av syntaksen for språket slik at kompleksitet er definert for all syntaktisk korrekt kildekode. I kapittel 5 viste vi hvordan vi kan definere $\mathcal{C}(T)$ for C++ kildekode T ved å utpeke et sett språkkonstruksjoner som gir opphav til seksjoner i kildekoden. Hver slik *seksjonskonstruksjon* er tilknyttet en bestemt ikke-terminal i C++ grammatikken.

Grammatikksymbolene i språkets grammatikk representerer utsnitt av en tenkt kildekode. Hvert slikt utsnitt har en kompleksitet. Videre angir hver produksjon hvordan utsnitt (symbolene i høyresiden) kan kombineres til større utsnitt (symbolet i venstresiden). Vårt kompleksitetsmål er definert slik at kompleksiteten av et hele er gitt som summen av kompleksiteten av delene (3.1). Vi kaller denne viktige egenskapen for kompleksitetsmålets *additive egenskap* — kompleksiteten til venstresiden i en grammatikkproduksjon er lik summen av kompleksiteten til hvert grammatikksymbol i høyresiden.

La N og $N_1 \cdots N_n$ være noder i parse-treet og $S(N)$ grammatikksymbolet som noden N representerer. Når parseren bygger opp parse-treet, slår den sammen en sekvens av grammatikksymboler $S(N_1) \cdots S(N_n)$ til ett enkelt symbol $S(N)$ hver gang den foretar en reduksjon i henhold til en produksjon $S(N) \rightarrow S(N_1) \cdots S(N_n)$. Ved hver reduksjon opprettes en ny node N i parse-treet som representerer produksjonens venstreside. Vår idé er at

noden N som opprettes ved reduksjonen får beregnet en kompleksitet lik summen av kompleksiteten til nodens avkom i treet $N_1 \cdots N_n$ (som representerer grammatikksymbolene i høyresiden). Nodene i bunnen av parse-treet (terminalsymbolene) får sin kompleksitet satt av den leksikalske analysatoren (som er opphavet til alle terminalsymbolene). Hver node i parse-treet vil nå inneholde et mål for kompleksiteten til det utsnitt av kildekoden som noden representerer. Den leksikalske analysatoren forsyner oss med kompleksiteten til terminalsymbolene, som så syntetiseres opp gjennom parse-treet ved reduksjoner slik at de forskjellige konstruksjonene programmet består av får regnet ut sin kompleksitet. Når parsingen er avsluttet, vil parse-treets rotnode inneholde kompleksiteten til hele kildekoden.

Vår oppgave blir nå å angi et *oversettelsesskjema* (se avsnitt 4.7) som for hver produksjon i C++ grammatikken viser hvordan kompleksiteten til symbolet i venstresiden kan beregnes fra kompleksiteten til symbolene i høyresiden. Det kan her være nyttig å se tilbake på figur 4.3 som viser et oversettelsesskjema for aritmetiske uttrykk. Oversettelsesskjemaet viser hvordan verdien til et uttrykk i venstresiden skal beregnes fra verdien til deluttrykkene i høyresiden.

6.2 Relativ og absolutt kompleksitet

G++ er i likhet med mange andre kompilatorer en én-pass kompilator. Det vil si at den leser gjennom kildekoden kun én gang. Samtidig ser vi av (3.10) at kompleksiteten av en seksjon er en funksjon av størrelsen på seksjonens alfabet. Fordi seksjonen kan utvide alfabetet med deklarasjoner av nye symboler, vet vi ikke alfabetets størrelse før kompilatoren har prosessert alle deklarasjoner. Hvis vi antar at deklarasjoner kan forekomme hvor som helst i en seksjon, betyr dette at alfabetets størrelse først er kjent når kompilatoren har lest hele seksjonen og avdekket seksjonens parse-tre. Nodene i dette parse-treet ble introdusert før størrelsen av seksjonens alfabet var kjent. Når en ny node introduseres i parse-treet, skal noden få beregnet en kompleksitet, men dette kan nødvendigvis ikke gjøres ved (3.10), da alfabetets størrelse er ukjent når noden introduseres. Løsningen på dette problemet er å gi nodene en kompleksitet som er *relativ* i forhold til alfabetets størrelse.

Det er viktig at den relative kompleksiteten har den additive egenskapen slik at vi ved hver reduksjon kan beregne relativ kompleksitet for den noden reduksjonen introduserer. Når en hel seksjon er lest, vil rotnoden til seksjonens parse-tre inneholde den relative kompleksiteten for hele seksjonen. På dette tidspunktet er vi istand til å bestemme størrelsen på seksjonens alfabet, og trenger ikke lenger å regne kompleksiteten relativt i forhold til et alfabet av ukjent størrelse. Alfabetets størrelse er kjent, og vi kan regne ut det absolutte kompleksitets-tallet for seksjonen etter (3.10).

Vi ser av (3.10) at absolutt kompleksitet av en seksjon T er en funksjon

av størrelsen $\mathcal{S}(K,T)$ av symbolkategoriene og *antallet symbolforekomster* $\mathcal{R}(K,T)$ for hver symbolkategori K. På bakgrunn av den forutgående diskusjonen synes dermed antallet symbolforekomster $\mathcal{R}(K,T)$ fra hver symbolkategori K å kunne tjene som mål for relativ kompleksitet for en seksjon. *Å beregne relativ kompleksitet for en seksjon innebærer da å telle opp for hver symbolkategori K de symbolforekomster F i seksjonen som har $\mathcal{K}(F) = K$.*

Definisjon 6.1 *Anta at alfabetet til segmentet T er delt inn i j symbolkategorier. Relativ kompleksitet $\mathcal{C}_r(T)$ er da gitt som en vektor*

$$\mathcal{C}_r(T) = \begin{pmatrix} \mathcal{R}(K_1,T) \\ \vdots \\ \mathcal{R}(K_j,T) \end{pmatrix}$$

Hvis vi lar segmentet T bestå av segmentene $T_1 \cdots T_n$ har vi

$$\mathcal{C}_r(T) = \sum_{i=1}^{n} \mathcal{C}_r(T_i) = \begin{pmatrix} \sum_{i=1}^{n} \mathcal{R}(K_1,T_i) \\ \vdots \\ \sum_{i=1}^{n} \mathcal{R}(K_j,T_i) \end{pmatrix} \tag{6.1}$$

som viser at dette målet for relativ kompleksitet har den ønskede additive egenskapen. Ved en reduksjon i henhold til produksjonen $T \rightarrow T_1 \cdots T_n$, beregnes $\mathcal{C}_r(T)$ ved (6.1), og relativ kompleksitet kan dermed syntetiseres opp gjennom parse-treet ved reduksjoner og munne ut i en relativ kompleksitet for en hel seksjon når seksjonen er lest. Da kan vi regne ut seksjonens absolutte kompleksitet.

6.3 Kompositt kompleksitet

Kompleksiteten assosiert med en node i parse-treet kan altså være gitt som relativ eller absolutt kompleksitet.

- Dersom noden representerer en seksjon, vil den ha absolutt kompleksitet fordi nodens alfabet er kjent når noden introduseres.
- Dersom noden representerer et segment (men ikke en seksjon), vil den ha relativ kompleksitet fordi segmentet kanskje kan utvides til å omfatte mer tekst som kan huse nye deklarasjoner som utvider alfabetet.

I forbindelse med nesting av seksjonskonstruksjoner kan en node ha både absolutt og relativ kompleksitet. De nestede sammensatte setningene i figur 5.3 kan genereres av grammatikken i figur 6.1. En node N som representerer symbolet `stmt` kan ha enten relativ eller absolutt kompleksitet.

```
stmt     : compstmt
stmt     : goto label ';'
stmt     : ....
stmts    : stmts stmt
stmts    : stmt
compstmt: '{' stmts '}'
compstmt: '{' '}'
```

Figur 6.1: Grammatikk for sammensatte setninger (fra G++)

- Dersom N er innført ved reduksjon med produksjonen `stmt:compstmt`, er N bærer av den absolutte kompleksiteten for seksjonskonstruksjonen `compstmt` (en sammensatt setning).
- Dersom N er innført ved reduksjon av en enkel setning (for eksempel `goto label`), er N bærer av den relative kompleksiteten for den enkle setningen. Setningen utgjør et segment som kanskje kan utvides med etterfølgende enkle setninger (som kan inneholde deklarasjoner), så setningens alfabet er ukjent.

En node N for symbolet `stmts` representerer en sekvens av setninger (både enkle og sammensatte). Fordi enkle setninger har relativ kompleksitet og sammensatte setninger har absolutt kompleksitet, kan en slik setningssekvens ha både relativ og absolutt kompleksitet. Generelt må vi representere kompleksiteten til en node i form av en relativ og en absolutt komponent.

Definisjon 6.2 *La T være et utsnitt av programteksten. Vi definerer $\mathcal{C}_r(T)$ som den relative kompleksitetskomponenten, og $\mathcal{C}_a(T)$ som den absolutte kompleksitetskomponenten til T. $\mathcal{C}(T)$ kan nå angis i to komponenter som tuppelet $(\mathcal{C}_a(T), \mathcal{C}_r(T))$. Vi kaller denne tokomponents representasjonen av kompleksitet for kompositt kompleksitet.*

Hvis vi lar alle grammatikksymboler i enhver produksjon $V \rightarrow H_1 \cdots H_n$ ha en assosiert kompositt kompleksitet, kan vi regne ut den kompositte kompleksiteten til venstresiden V fra høyresiden ved

$$(\mathcal{C}_a(V), \mathcal{C}_r(V)) = (\sum_{i=1}^{n} \mathcal{C}_a(H_i), \sum_{i=1}^{n} \mathcal{C}_r(H_i)) \tag{6.2}$$

I forbindelse med utregningen av $\mathcal{C}_r(V)$ bringes det imidlertid inn et krav som må holde dersom kompositt kompleksitet skal kunne representere kompleksiteten til ethvert symbol ved parsing av kildekode.

Relativ kompleksitet regnes i forhold til et alfabet, og to relative kompleksitetskomponenter må referere til *samme* alfabet for at man skal kunne addere dem etter (6.1). Hvis det i en produksjon eksisterer to eller flere H_i

med $\mathcal{C}_r(H_i) \neq 0$ som refererer til forskjellige alfabet, klarer vi ikke å syntetisere $\mathcal{C}_r(V)$. I neste avsnitt skal vi se hvordan vi kan løse dette problemet ved å regne om relativ kompleksitet til absolutt kompleksitet på riktig tidspunkt.

6.4 Assimilasjon

Anta at $V \rightarrow H_1 \cdots H_n$ er en produksjon, og la H_l og H_r, $l < r \leq n$ være grammatikksymboler i produksjonens høyreside. Anta videre at $\mathcal{C}_r(H_l) \neq 0$ og $\mathcal{C}_r(H_r) \neq 0$. For å syntetisere kompositt kompleksitet opp gjennom parse-treet, må vi beregne $\mathcal{C}_r(V)$ ved (6.1), så $\mathcal{C}_r(H_l)$ og $\mathcal{C}_r(H_r)$ må referere til samme alfabet. Dersom de *ikke* refererer til samme alfabet, stammer $\mathcal{C}_r(H_l)$ og $\mathcal{C}_r(H_r)$ fra *forskjellige seksjoner* med *forskjellig alfabet*. Seksjoner dannes av språkets seksjonskonstruksjoner, så da er H_l eller H_r seksjonskonstruksjoner. *Men da er alltid alfabetet for H_l eller H_r kjent slik at vi kan beregne absolutt kompleksitet for $\mathcal{C}_r(H_l)$ eller $\mathcal{C}_r(H_r)$.* Vi kan *assimilere* den relative kompleksitetskomponenten i den absolutte kompleksitetskomponenten:

Definisjon 6.3 *La T være et utsnitt av en programtekst med kompositt kompleksitet $(\mathcal{C}_a(T), \mathcal{C}_r(T))$. Å assimilere $\mathcal{C}_r(T)$ vil si å beregne absolutt kompleksitet for $\mathcal{C}_r(T)$ ved (3.10), addere resultatet til $\mathcal{C}_a(T)$, og så nullstille $\mathcal{C}_r(T)$. Den relative kompleksitetskomponenten er da blitt assimilert i den absolutte komponenten.*

Etter assimilasjon har vi $\mathcal{C}_r(H_l) = 0$ eller $\mathcal{C}_r(H_r) = 0$, og nullstilte relative komponenter kan alltid adderes.

6.5 Strategi

La grammatikkens produksjoner være på formen $V \rightarrow H_1 \cdots H_n$. Dersom produksjonenes aksjoner modifiseres til å assimilere $\mathcal{C}_r(H_i)$ for ikketerminaler H_i som representerer seksjonskonstruksjoner, kan kompositt kompleksitet syntetiseres opp gjennom parse-treet ved (6.2). Legg merke til at $\mathcal{C}_r(H_i)$ generelt kan stamme fra flere forskjellige seksjoner *med identisk alfabet* når utsnittet representert ved H_i omslutter andre seksjonskonstruksjoner i programteksten (se avsnitt 5.5).

Når hele parse-treet er konstruert, har hver node i treet som representerer en seksjonskonstruksjon fått beregnet sin kompleksitet. Av disse nodene er vi primært interessert i prosedyredefinisjonene, som huser alle programmets setninger. Implementasjon av denne strategien på det eksisterende oversettelsesskjemaet i G++ vil kreve at

- Aksjonene må ha tilgang til den kompositte kompleksiteten for utsnittet som assosieres med hvert symbol V og $H_1 \cdots H_n$ i den tilhørende

produksjonen. Alle grammatikksymboler må ha kompositt kompleksitet som et *attributt*. I avsnitt 4.6 så vi at kompilatorverktøyet BISON lar oss knytte attributter til grammatikksymbolene V og $H_1 \cdots H_n$ via pseudovariablene `$$` og `$1` ... `$n`. Figur 4.3 viser hvordan pseudovariablene kan benyttes for å beregne *verdien* av enkle aritmetiske uttrykk. Da representerer `$i` *verdien* av deluttrykket H_i. Hvis vi ønsker å beregne *kompleksiteten* av uttrykk istedet for deres *verdi*, lar vi `$i` være den *kompositte kompleksiteten* og ikke *verdien* av deluttrykket. Assimilasjon gjøres i aksjonene ved beregninger på `$i`.

- Etter eventuelle assimilasjoner på `$i`, må enhver aksjon regne ut kompositt kompleksitet for V fra `$1` ... `$n` ved (6.2). Resultatet legges i `$$`.

Det er to haker ved denne strategien:

- Hver eneste aksjon må modifiseres til å beregne `$$` fra `$1` ... `$n`. Oversettelsesskjemaet for G++ består av ca. 550 produksjoner, så dette er ingen liten modifikasjon.

  Beregning av `$$` bør istedet skje *automatisk* etter utførelsen av hver aksjon. En aksjon blir da kun ansvarlig for å utføre nødvendige assimilasjoner slik at (6.2) alltid er gyldig *etter* at aksjonen er utført. Assimilasjon er nødvendig i kun 19 av produksjonene i G++, så når `$$` beregnes automatisk, holder det å modifisere 19 av de 550 produksjonene.

- Når vi skal modifisere et eksisterende oversettelsesskjema, vil pseudovariablene `$$` og `$1` ... `$n` allerede være i bruk til andre ting, og vi kan ikke uten videre kapre dem til vårt bruk.

I neste kapittel skal vi se at BISON inneholder et par hyggelige overraskelser som bidrar til enkle løsninger på disse problemene.

Kapittel 7

Implementasjon

I dette kapitlet viser vi hvordan vi kan modifisere G++ kompilatoren etter metoden i forrige kapittel slik at den regner ut Maus' kompleksitetsmål for den kompilerte kildekoden. Arbeidet med å implementere målet kan deles inn i fire deler:

1. Modifisere den leksikalske analysatoren til å beregne kompositt kompleksitet for hvert terminalsymbol den gir til parseren. Dette innebærer å kategorisere terminalsymbolet i riktig symbolkategori.

2. Modifisere aksjonene for produksjoner $V \rightarrow H_1 \cdots H_n$ der en eller flere H_i er seksjonskonstruksjoner. Disse aksjonene må sørge for å assimilere $\mathcal{C}_r(H_i)$ i $\mathcal{C}_a(H_i)$ for de H_i som er seksjonskonstruksjoner. Det blir da mulig å beregne $\mathcal{C}_r(V)$ ved addisjon av $\mathcal{C}_r(H_1) \cdots \mathcal{C}_r(H_n)$ (se avsnitt 6.3).

3. Modifisere parseren til å beregne kompositt kompleksitet for venstresiden ved (6.2) *etter* at den reduserende produksjonens aksjon er utført. Punkt 2 sikrer at beregningen alltid har mening. Vi skal se at BISON inneholder en mekanisme som vi kan bruke i dette arbeidet.

4. Beregne kardinalitet for alle symbolkategorier i forbindelse med assimilasjon av relativ kompleksitet. Vi skal se at kompilatorens deklarasjonstabeller inneholder all informasjon vi trenger for å bestemme antallet deklarerte symboler.

Disse modifikasjonene gir oss et oversettelsesskjema for *kompleksitet* basert på et eksisterende oversettelsesskjema for *objektkode*. Kompositt kompleksitet syntetiseres opp gjennom parse-treet, og vi får beregnet *absolutt* kompleksitet for hver instans av en seksjonskonstruksjon. En ekstern deklarasjon er definert som en seksjonskonstruksjon, så vi får beregnet absolutt kompleksitet for hver eksterne deklarasjon. Av disse er prosedyredefinisjoner spesielt interessante, da de inneholder programmets setninger.

7.1 Generelt om G++

G++ implementerer versjon 2.0 av C++, og er en utvidelse av C-kompilatoren GCC. Fordi GCC følger standard-oppdelingen av en kompilator i front-end og back-end, er dette en forholdsvis enkel utvidelse. GCCs front-end er blitt modifisert til å håndtere C++ konstruksjoner som klasser, overloading av prosedyrer og operatorer m.m. Back-end er beholdt så å si uendret. Fordi C++ i grove trekk er et supersett av C, får G++ mye gratis fra GCC.

G++ front-end er delt inn i leksikalsk analysator og parser etter tradisjonelle prinsipper som beskrevet i kapittel 4. G++ kompilatoren er skrevet i C, og bruker en LALR parser som er implementert ved hjelp av BISON. I utgangspunktet er det ikke mulig å lage en LALR parser for C++ eller C. Dette skyldes i grove trekk at betydningen av identifikatorer i deklarasjoner er kontekstavhengig [21]. Ved hjelp av noen mindre elegante triks er det allikevel mulig å arbeide seg rundt dette slik at man kan implementere parseren med en LALR parser generator som BISON. Ved å bruke BISON får man en strukturert og oversiktlig implementasjon av parseren, som dermed blir enkel å modifisere. Dette er av stor verdi når vi skal utvide kompilatoren til å beregne kompleksitet av kildekode etter strategien vi skisserte i avsnitt 6.5.

7.2 Implementasjon i BISON

I avsnitt 6.5 så vi at relativ kompleksitet $\mathcal{C}_r(T)$ for en instans T av en seksjonskonstruksjon (for eksempel en sammensatt setning) må regnes om til absolutt kompleksitet så fort hele T er lest av parseren. Slik *assimilasjon* av $\mathcal{C}_r(T)$ (se definisjon 6.3) kan foretas fra aksjonene dersom kompositt kompleksitet for utsnittene representert ved grammatikksymbolene $H_1 \cdots H_n$ i høyresiden kan manipuleres fra aksjonene. BISON gir oss mulighet til å assosiere verdier med gammatikksymbolene i produksjoner gjennom pseudovariablene `$$` og `$i`, der $1 \leq i \leq n$. I avsnitt 4.6 så vi hvordan `$` variablene kan brukes til å beregne verdien av enkle aritmetiske uttrykk. I avsnitt 6.5 foreslo vi `$` variablene brukt som bærere av den kompositte kompleksiteten til nodene i parse-treet. Datatypen til `$` variablene heter `YYSTYPE`, og det er nærliggende å utvide G++ sin definisjon av `YYSTYPE` med en representasjon av kompositt kompleksitet. Aksjonene har da tilgang til kompositt kompleksitet for $H_1 \cdots H_n$ via `$1` ... `$n`, og kan assimilere $\mathcal{C}_r(H_i)$ for alle seksjonskonstruksjoner H_i ved beregninger på `$i`. Når så alle seksjonskonstruksjoner H_i har $\mathcal{C}_r(H_i) = 0$, kan aksjonen syntetisere kompositt kompleksitet for venstresiden etter (6.2), og svaret legges i `$$`. Det er to problemer med denne idéen.

- Aksjonene i G++ sin parser bruker allerede `$` variablene som informasjonsbærere for annen informasjon som brukes til kompilatorens opprinnelige oppgave — å generere objektkode. `YYSTYPE` er definert slik at den passer til denne oppgaven. Typen til en `$` variabel

varierer med det assosierte grammatikksymbolet. `$` variable tilknyttet forskjellige grammatikksymboler kan ha forskjellig type. BISON forutsetter derfor at `YYSTYPE` er definert som en union av alle de forskjellige datatypene `$` variable kan anta. Vi ønsker at alle grammatikksymboler skal ha en kompleksitetsrepresentasjon, og må i så fall modifisere alle de alternative typene i `YYSTYPE` til å inneholde en slik kompleksitetsrepresentasjon. Dette er høyst ustrukturert.

- Vi ønsker å unngå å modifisere hver eneste aksjon i G++ grammatikken til å regne ut kompositt kompleksitet i `$$` fra kompositt kompleksitet i `$1` ... `$n` etter (6.2). Denne beregningen bør utføres automatisk etter at hver aksjon er utført.

Hvis vi implementerer en operasjon som etter utførelsen av hver aksjon automatisk adderer de kompositte kompleksitetene for $H_1 \cdots H_n$, kan de fleste av parserens aksjoner forbli urørt. Vi trenger da kun å modifisere aksjonene til de produksjonene hvor en eller flere H_i symboliserer seksjonskonstruksjoner. Disse aksjonene må besørge assimilasjon av $\mathcal{C}_r(H_i)$ i $\mathcal{C}_a(H_i)$. Etter at en aksjon er utført, vil alle `$1` ... `$n` ha relative kompleksitetskomponenter som kan adderes, så kompositt kompleksitet `$$` for venstresiden kan alltid beregnes etter (6.2). Fordi denne beregningen alltid er gyldig, kan den automatiseres slik at aksjonene ikke behøver å befatte seg med den.

Det viser seg at BISON knytter et ekstra sett attributter til grammatikksymbolene (i tillegg til attributtene i `$` variablene). Disse spesielle attributtene blir beregnet automatisk for venstresiden ved hver reduksjon, og egner seg derfor utmerket som bærere av kompositt kompleksitet. Vi viser nå hvordan denne mekanismen kan utnyttes for implementasjon av et oversettelsesskjema for *kompleksitet* som kan leve i symbiose med det eksisterende oversettelsesskjemaet for *objektkode*.

7.3 Utnyttelse av funksjonaliteten i BISON

I tillegg til å assosiere en verdi `$1` ... `$n` med hvert grammatikksymbol $H_1 \cdots H_n$ i høyresiden av en produksjon, tilbyr BISON også pseudovariablene `@1` ... `@n`. `@i` inneholder normalt informasjon om hvilke linjenummer konstruksjonen representert ved grammatikksymbolet H_i strekker seg over, og er tenkt brukt til å angi stedsanvisninger i kildekoden i forbindelse med feilmeldinger under kompilering. Ved hver reduksjon beregner BISON automatisk denne informasjonen for venstresiden, så det finnes ingen pseudovariabel `@@` analog til `$$` tilgjengelig fra aksjonene. Informasjonen har form av en record av type `YYLTYPE` som vises i figur 7.1.

Aksjoner kan referere til linjenummeret for starten av konstruksjonen H_i gjennom uttrykket `@i.first_line`. Den leksikalske analysatoren er ansvarlig for å sette opp denne informasjonen i en global variabel `yylloc` av type

```
typedef struct
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

Figur 7.1: Den original definisjonen av YYLTYPE

```
@@.first_line = @1.first_line;
@@.first_column = @1.first_column;
@@.last_line = @n.last_line;
@@.last_column = @n.last_column;
```

Figur 7.2: Beregning av linjeinformasjon for reduserende symbol

`YYLTYPE` hver gang den returnerer et terminalsymbol til parseren. Med denne informasjonen som basis, beregner BISON den tekstlige utstrekningen og posisjonen til alle noder i parse-treet. Hver gang parseren foretar en reduksjon i henhold til en produksjon $V \rightarrow H_1 \cdots H_n$, beregner BISON `YYLTYPE` informasjonen for V som funksjon av `@1` og `@n` som vist i figur 7.2. For illustrasjonens skyld gjør figuren bruk av `@@` for å betegne linjeinformasjonen til venstresiden. Som tidligere bemerket er ikke `@@` tilgjengelig som en pseudovariabel i aksjonene.

Vi kan løse de to problemene vi tidligere skisserte ved å

1. Utvide `YYLTYPE` til å inneholde symbolets kompositte kompleksitet. Strukturen til `YYLTYPE` er definert som en record, og det er en smal sak å utvide `YYLTYPE` med attributter som representerer symbolets kompositte kompleksitet. Utvidelse av `YYSTYPE` var til sammenligning vanskelig fordi `YYSTYPE` er en *union* av mange datatyper som er tilpasset kompilatorens primære oppgave. Strukturen til `YYLTYPE` har færre restriksjoner på seg fordi den brukes mindre, og er derfor lettere å modifisere.

2. Modifisere den delen av runtime-kjernen til BISON som automatisk kalkulerer linjeinformasjonen i `@@` for det reduserende symbolet V ved hver reduksjon. Modifikasjonen består i at vi beregner den kompositte kompleksiteten til V fra de kompositte kompleksitetene i `@1` ... `@n` ved (6.2).

Det viser seg at aksjonene i G++ kompilatorens parser ikke benytter seg av `@` variablene. For å spare plass i kompilatorens hukommelse kan vi derfor

```
typedef struct
{
  double abs [YYNLEXCAT];
  int    rel [YYNLEXCAT];
  int    ant [YYNLEXCAT};
  int    first_line;
  int    last_line;
} YYLTYPE;
```

Figur 7.3: Ny definisjon av YYLTYPE for kompleksitetsberegninger. Konstanten YYNLEXCAT betegner antallet symbolkategorier.

omdefinere `YYLTYPE` (og dermed `@n`) til å passe vårt formål. Figur 7.3 viser vår nye definisjon av `YYLTYPE`.

Attributtene i den nye `YYLTYPE` strukturen representerer aspekter ved kompleksiteten til en node i et (tenkt) parse-tre. En slik node representerer et utsnitt T av programteksten, og nodens attributter er bærere av kompositt kompleksitet for T. Attributtene i nodens `YYLTYPE` struktur inneholder følgende informasjon:

abs er absolutt kompleksitetskomponent $\mathcal{C}_a(T)$. Hvert element i `abs` angir bidraget til $\mathcal{C}_a(T)$ fra symbolforekomster i T som stammer fra en bestemt symbolkategori.

rel er relativ kompleksitetskomponent $\mathcal{C}_r(T)$. Hvert element i `rel` inneholder antallet symbolforekomster pr symbolkategori i segmenter av T med identisk men ukjent alfabet. Når programteksten T i figur 5.3 skal reduseres i henhold til produksjonen *stmt* $\rightarrow$ *compstmt* inneholder `rel` antallet *reserverte ord*, *navn* etc i T_1 og T_3. `abs` inneholder total kompleksitet for T_2 ($\mathcal{C}_r(T_2)$ er blitt assimilert). Til sammen angir `abs` og `rel` kompositt kompleksitet for T.

ant er antallet symbolforekomster i T fra hver symbolkategori. Denne informasjonen brukes sammen med den absolutte kompleksiteten for T for å beregne entropi $\mathcal{H}(T)$ (se avsnitt 3.9).

first_line er antallet linjer foran begynnelsen av utsnittet T som ikke er blanke eller inneholder kun kommentarer.

last_line er antallet linjer foran slutten av utsnittet T som ikke er blanke eller inneholder kun kommentarer. `first_line` og `last_line` brukes for å beregne antall linjer kildekode $LOC(T)$. $LOC(T)$ er det kompleksitetsmålet som er mest brukt i praksis. Ved å sammenholde $LOC(T)$ med $\mathcal{C}(T)$ for målinger på større mengder kildekode, kan vi danne oss et bilde av hvorvidt $\mathcal{C}(T)$ fanger opp aspekter ved kildekode som ikke kommer til uttrykk i $LOC(T)$.

```
const double log_2 = 0.69314718;

void make_complexity_absolute (YYLTYPE *etp,namespace *p)
{
  int i;
  for (i = 0; i < YYNLEXCAT; i++)
    {
      if (p->size[i] > 0)
        etp->abs[i] += ((double) (etp->rel[i])) *
                       log ((double)(p->size [i])) /
                       log_2;
      etp->rel[i] = 0;
    }
}
```

Figur 7.4: Assimilasjon av relativ kompleksitet

Med definisjonen av `YYLTYPE` i figur 7.3 kan aksjonene referere til absolutt komponent $\mathcal{C}_a(H_i)$ og relativ komponent $\mathcal{C}_r(H_i)$ for utsnittet representert av grammatikksymbolet H_i i høyresiden som henholdsvis:

```
@i.abs /* absolutt komponent */
```

og

```
@i.rel /* relativ komponent */
```

For hver ikke-terminal H_i som symboliserer en seksjonskonstruksjon i en produksjons høyreside, må den tilhørende aksjonen utføre kallet

```
make_complexity_absolute(&@i,p);
```

Prosedyren `make_complexity_absolute` regner om den relative kompleksitetskomponenten `@i.rel` til absolutt kompleksitet etter (3.10) som adderes til `@i.abs`, og nullstiller så `@i.rel`. Figur 7.4 viser hvordan dette gjøres for kompositt kompleksitet `etp`. For å kunne assimilere relativ kompleksitet, må prosedyren vite størrelsen på alfabetet. Denne informasjonen oversendes som en peker `p` til en vektor `size` der hvert element angir kardinaliteten for en symbolkategori. I avsnitt 7.7 kommer vi tilbake til hvordan denne informasjonen om alfabetet skaffes til veie.

7.4 Endringer på BISON runtime-kjerne

Vi viser nå hvordan vi modifiserte BISON til å beregne kompositt kompleksitet for venstresiden automatisk ved hver reduksjon. Uten denne modifikasjonen ville det vært nødvendig å modifisere *alle* aksjonene i oversettelsesskjemaet for G++ til å beregne kompositt kompleksitet for den tilhørende

produksjonens venstreside. Modifikasjonen sto derfor helt sentralt i arbeidet med å minimalisere endringene på kompilatoren.

Når man bruker BISON til å generere en LALR parser for en gitt grammatikk med aksjoner, blir sluttproduktet en fil som inneholder kildekode i C for en prosedyre `YYPARSE`. `YYPARSE` er en LALR parser som leser kildekoden via en leksikalsk analysator `YYLEX` og utfører en aksjon hver gang den foretar en reduksjon. Parseren som BISON genererer for bruk i G++ kompilatoren blir skrevet til filen `cplus-tab.c`. Denne filen blir senere kompilert som en del av kildekoden til G++ kompilatoren. Prosedyren `YYPARSE` kalles fra kompilatorens oppstartrutine. `YYPARSE` leser et terminalsymbol fra den leksikalske analysatoren ved hver shift-operasjon, og utfører aksjoner etterhvert som kildekodens parse-tre avdekkes. BISON genererer `cplus-tab.c` basert på to filer:

- `cplus-parse.y` inneholder alle grammatikkens produksjoner og de tilhørende aksjonene, og utgjør en spesifikasjon av et oversettelsesskjema (se avsnitt 4.7) fra kildekode til objektkode.
- `bison.simple` inneholder den delen av kildekoden til `YYPARSE` som er lik for alle grammatikker, dvs en generell parser-maskin.

Vi beskriver nå hvordan BISON genererer `cplus-tab.c` fra `cplus-parse.y` og `bison.simple`. BISON nummererer først hver produksjon, og lager

1. En stor `switch` setning som inneholder ett alternativ for hver produksjon. `switch` setningen er skrevet slik at den som funksjon av nummeret til en produksjon utfører produksjonens aksjon.
2. En samling tabeller som regulerer hvordan en automat skal foreta shift-operasjoner og reduksjoner for å parse den gitte grammatikken.

Den generelle parser-maskinen i `bison.simple` er tabell-drevet, og inneholder et merke der aksjonene skal utføres. Ved å kopiere `switch` setningen inn i `bison.simple` på markert sted og likeså grammatikkens tabellverk, forandres `bison.simple` fra en generell parser-maskin til en parser for en spesifikk grammatikk. Resultatet finner vi i vårt tilfelle på filen `cplus-tab.c`. Det står nå klart at vi må lete i `bison.simple` etter det stedet hvor BISON beregner `YYLTYPE` informasjonen for venstresiden etter en reduksjon, og modifisere denne koden til å regne ut venstresidens kompositte kompleksitet. Denne koden må utføres *etter* at `switch` setningen med aksjonene er utført. Aksjoner har da fått sjansen til å assimilere $\mathcal{C}_r(H_i)$ for seksjonskonstruksjoner H_i i høyresiden slik at addisjon av relative kompleksitetskomponenter alltid har mening. Beregningen av venstresidens `YYLTYPE` informasjon gjøres originalt av programutsnittet i figur 7.5 hentet fra filen `bison.simple`. Dette utsnittet fjernes og erstattes av utsnittet i figur 7.6.

```
/* These lines are executed by the BISON parser engine on
every reduction. (yylsp) is an array containing a YYLTYPE
element for each of the (yylen) grammar symbols in the
right side of the production. The textual span of the left
side of the production is computed from the span of the
first and last symbols of the right side and placed in
yylsp[0] overwriting the information for the leftmost
grammar symbol in the right side. (yylloc) is the span of
the lookahead symbol. */

if (yylen == 0) /* Epsilon production */
  {
    yylsp->first_line = yylloc.first_line;
    yylsp->first_column = yylloc.first_column;
    yylsp->last_line = (yylsp-1)->last_line;
    yylsp->last_column = (yylsp-1)->last_column;
  }
else            /* Normal production */
  {
    yylsp->last_line = (yylsp+yylen-1)->last_line;
    yylsp->last_column = (yylsp+yylen-1)->last_column;
  }
```

Figur 7.5: Beregning av tekstlig utstrekning for venstresiden som utføres automatisk ved hver reduksjon. Utsnitt av den originale generelle parsermaskinen i BISON.

7.5 Modifikasjoner på leksikalsk analysator

Den leksikalske analysatoren får nå en ny oppgave i tillegg til sin vanlige. I tillegg til å returnere det neste terminalsymbolet, må den også sette opp den globale variablen `yylloc` av type `YYLTYPE` til å inneholde den komposítte kompleksiteten til det returnerte terminalsymbolet. Parseren avleser `yylloc` ved hver shift-operasjon, og de avleste verdiene danner grunnlaget for alle kompleksitetsberegningene i parse-treet. Fordi et terminalsymbol alltid utgjør et segment av en seksjon, kan den komposítte kompleksiteten til ethvert terminalsymbol uttrykkes som en ren relativ kompleksitet (dvs $\mathcal{C}_a(T) = 0$ for alle terminalsymboler T). Alle elementer i `yylloc.abs` er da alltid lik null, og element k i vektoren `yylloc.rel` vil representere det antallet symboler fra kategori nummer k som det returnerte terminalsymbolet består av. Fordi et terminalsymbol pr definisjon består av ett eneste symbol, vil alle elementene i `yylloc.rel` være 0 unntatt ett element som vil være 1. *Å sette opp* `yylloc.rel` *innebærer dermed å kategorisere det returnerte terminalsymbolet i riktig symbolkategori.*

Hvis den leksikalske analysatoren skal kunne foreta denne kategoriseringen, må symbolkategorien til en symbolforekomst være entydig bestemt ut fra symbolets tekstlige representasjon alene. Vi har allerede sett at i C++ kan de deklarerte identifikatorene deles inn i tre symbolkategorier, og at deklar-

```
/* These lines are executed by the modified BISON parser engine
after the action associated with the reducing production has
been carried out. The composite complexity of the left side of
the production is computed from the composite complexity of the
symbols in the right side of the production. (yylsp) is an
array containing a modified YYLTYPE element for each of the
(yylen) grammar symbols in the right side. Each such element
carries the composite complexity of the text represented by the
grammar symbol. When this code has been run, yylsp[0] contains
the composite complexity of the left side of the reducing
production.*/

if (yylen == 0) /* Epsilon production - zero complexity */
  {
    int i;
    yylsp->first_line = yylloc.first_line;
    yylsp->last_line = (yylsp - 1)->last_line;
    for (i = 0; i < YYNLEXCAT; i++)
      {
        yylsp->abs [i] = 0.0;
        yylsp->rel [i] = 0;
        yylsp->ant [i] = 0;
      }
  }
else if (yylen > 1)
  {
    int i,j;
    yylsp->last_line = (yylsp + yylen - 1)->last_line;
    for (i = 0; i < YYNLEXCAT; i++)
      for (j = 1; j < yylen; j++)
        {
          yylsp->abs [i] += yylsp [j].abs [i];
          yylsp->rel [i] += yylsp [j].rel [i];
          yylsp->ant [i] += yylsp [j].ant [i];
        }
  }
```

Figur 7.6: Beregning av kompositt kompleksitet for venstresiden som utføres automatisk ved hver reduksjon. Utsnitt av den modifiserte generelle parsermaskinen i BISON.

erte identifikatorer *ikke* kan kategoriseres ut fra sin tekstlige representasjon alene: Kategorisering av deklarerte identifikatorer forutsetter at identifikatorens kontekst er kjent. Informasjon om kontekst har ikke den leksikalske analysatoren tilgang til. Den arbeider på et lavere abstraksjonsnivå i kompilatoren der terminalsymbolene enda ikke er relatert til hverandre. Den leksikalske analysatoren kan derfor ikke med sikkerhet fastsette symbolkategorien for en deklarert identifikator. Vi løser dette problemet ved at den leksikalske analysatoren konsekvent kategoriserer enhver deklarert identifikator som et navn (kategori nummer k) og ikke et strukturnavn eller en merkelapp (kategori nummer l). Hvis det på et senere punkt i parsingen blir klart at terminalsymbolet er et strukturnavn (eller en merkelapp), kan aksjonen til den grammatikkproduksjonen der dette avdekkes (terminalsymbolet utgjør H_i i produksjonens høyreside) omkategorisere terminalsymbolet fra kategori k til kategori l ved å utføre

```
@i.rel[k]--;
@i.rel[l]++;
```

Dette håndterer de tilfeller der symbolkategorien er kontekst-avhengig. Vi vil i slike tilfeller velge k og l slik at vi får færrest mulig aksjoner å modifisere. I vårt tilfelle impliserer dette at den leksikalske analysatoren kategoriserer enhver deklarert identifikator som et navn, og et lite antall aksjoner modifiseres til å foreta omkategorisering til strukturnavn og merkelapp som skissert over.

7.5.1 Registrering av operatorforekomster

Den leksikalske analysatoren betrakter programteksten som en strøm av symbolforekomster, og skal kategorisere hvert av disse symbolene i korrekt symbolkategori etterhvert som symbolene leses. Enkelte operatorer passer dårlig inn i dette mønsteret, da de består av flere tegn som omslutter operatorens argument. En forekomst av disse operatorene er spredd utover en større del av teksten, med andre terminalsymboler mellom de tegnene som utgjør operatorforekomsten. Dette gjelder

- Indeksering med `[]`.
- Parentetisering med `()`.
- Sammenstilling av setninger i sammensatte setninger med `{}`.

Det er ikke uten videre gitt hvordan vi skal forholde oss til opptelling av forekomster av disse operatorene. Skal vi telle hvert tegn som en selvstendig operator (fordi tegnene kan stå langt fra hverandre), eller skal vi telle dem samlet som én operator?

C++ tillater overloading av slike operatorer, og kardinaliteten vil øke med én for hver overloading. Det vil derfor være naturlig å telle tegnene

samlet som én operator. Dette er i tråd med praksis i [8] og [11]. Fordi tegnene ikke brukes i andre sammenhenger, kan vi løse dette ved å la den leksikalske analysatoren telle det første tegnet (dvs `(`, `[` og `{`) som en operator og ignorere det andre tegnet operatoren består av.

7.5.2 Symbolforekomster i headerfiler

Som nevnt i avsnitt 5.17 leser ikke en C++ kompilator samme kildefil som en menneskelig leser. Kildefilen A prosesseres først av en preprosessor som produserer en ny fil B i henhold til direktiver i A. Kompilatoren leser så fil B, som er en mer eller mindre modifisert utgave av A.

Hvis A refererer til prosedyrer definert i bibliotek eller i andre kildefiler, vil B typisk være utvidet med et sett deklarasjoner inkludert fra en headerfil H (se avsnitt 5.14). Anta at H inneholder symbolforekomstene $h_1 \cdots h_n$. Vi ønsker ikke at $\mathcal{C}(h_i)$ skal inngå i $\mathcal{C}(A)$ når forekomsten h_i ikke stammer fra A. Vi løser dette ved at den leksikalske analysatoren nullstiller *alle* elementene i `yylloc.rel` når den returnerer terminalsymbolet for h_i. Dette innebærer $\mathcal{C}(h_i) = 0$, og $\mathcal{C}(h_i)$ vil derfor ikke bidra til $\mathcal{C}(A)$.

Denne løsningen på problemet med inkluderte filer krever imidlertid at den leksikalske analysatoren vet når den leser fra en headerfil. Hovedfilen og de inkluderte filene er blitt smurt sammen til én fil B, så dette er i utgangspunktet et problem. I forbindelse med stedsangivelse ved feilmeldinger har imidlertid kompilatoren behov for å vite hvilken fil teksten i B for øyeblikket stammer fra, og preprosessoren legger derfor igjen små "bokmerker" i B som forteller hvilken fil den etterfølgende teksten er hentet fra. Ved å følge med på disse "bokmerkene" vet den leksikalske analysatoren om et symbol stammer fra den inkluderende filen A eller ikke, og kan sørge for at bare symbolforekomster fra A bidrar til $\mathcal{C}(A)$.

Hvis man ønsker å måle informasjonsmengden i en headerfil, kan man kompilere headerfilen separat som om den var en selvstendig kildefil.

7.6 Modifikasjoner på parser

I avsnitt 6.5 så vi hvordan vi må modifisere aksjoner i parseren når den tilhørende produksjonens høyreside $H_1 \cdots H_n$ inneholder et grammatikksymbol H_i som er definert som en seksjonskonstruksjon. Vi definerte seksjonskonstruksjonene for C++ i kapittel 5. Når H_i symboliserer en seksjonskonstruksjon, må aksjonen besørge at $\mathcal{C}_r(H_i)$ assimileres i $\mathcal{C}_a(H_i)$ slik at $\mathcal{C}_r(H_i) = 0$ etter at aksjonen er utført. Beregningen av kompositt kompleksitet for venstresiden ved (6.2) er da alltid gyldig, og gjøres automatisk av BISONs runtime kjerne umiddelbart etter utførelsen av aksjonen. Vi skal nå se på noen utvalgte detaljer ved implementasjonen av denne strategien

på G++ parserens aksjoner. Vedlegg D inneholder kildekoden for modifikasjonene på aksjonene.

7.6.1 Sammensatte setninger

I G++ betraktes en lokal variabeldeklarasjon som en setning. En sammensatt setning T (en sekvens av setninger) kan derfor inneholde variabeldeklarasjoner blant setningene. Skopet til en variabel strekker seg over etterfølgende setninger og stopper ved den sammensatte setningens slutt. Dette innebærer at alfabetet rent leksikalsk betraktet kan variere på forskjellige steder i T. Som vi så i avsnitt 5.8, bør vi til tross for dette betrakte T som en seksjonskonstruksjon. En T viss setninger ikke inneholder seksjonskonstruksjoner (ingen nestede sammensatte setninger etc) vil utgjøre en seksjon.

Når kompilatoren leser T, er vår strategi å representere den hittil leste delen av T som relativ kompleksitet inntil hele T er lest. Alle deklarasjonene i T er da effektuert, og vi kjenner alfabetet til T fordi C++ krever at alle deklarerte entiteter *med unntak av merkelapper* skal være deklarert foran der de refereres. Vi vil derfor alltid kunne beregne absolutt kompleksitet for T så fort T er lest dersom T ikke inneholder referanser til merkelapper i omsluttende sammensatte setninger.

En T som inneholder merkelapper vil beholde en relativ komponent i form av antallet forekomster av merkelapper i T. Alle de andre elementene i $\mathcal{C}_r(T)$ (som har form av en vektor med ett element for hver symbolkategori) er blitt assimilert i $\mathcal{C}_a(T)$ og nullstilt. Vi gjør nå den sentrale observasjon at *prosedyrer har kun ett navnerom for merkelapper* uavhengig av nestingen av setninger i prosedyrekroppen. Det er ikke mulig å introdusere lokale navnerom for merkelapper. Derfor vil alltid relativ kompleksitet som stammer fra merkelapper kunne adderes — forekomstene av merkelapper vil alltid referere til samme merkelapp-alfabet (det finnes pr definisjon bare ett pr funksjon). Vi trenger derfor ikke å assimilere *hele* $\mathcal{C}_r(T)$. Den delen av $\mathcal{C}_r(T)$ som stammer fra forekomster av merkelapper kan få stå urørt inntil vi har lest hele prosedyrekroppen T er en del av.

7.6.2 Prosedyredefinisjoner

Når kompilatoren har lest hele den sammensatte setningen T som utgjør prosedyrens kropp, vil $\mathcal{C}_a(T)$ gi oss absolutt kompleksitet for prosedyrekroppen med unntak av eventuelle forekomster av merkelapper. Kompleksiteten til forekomstene av merkelapper vil ligge avleiret i $\mathcal{C}_r(T)$, som dessuten inkluderer relativ kompleksitet for prosedyrehodet. Som vi så i avsnitt 5.8, kan vi bruke antallet eksterne deklarasjoner så langt i filen som en tilnærming til størrelsen på alfabetet for prosedyrehodet. Det viste seg

vanskelig å bestemme bidraget til alfabetet fra parameterdeklarasjonene. I lys av målingene som presenteres i kapittel 8 synes tilnærmingen rimelig.

Når hele prosedyrekroppen er lest, har vi sett alle deklarasjoner av merkelapper. Vi kjenner derfor antallet merkelapper som deklareres i T, så $\mathcal{C}_r(T)$ kan assimileres i $\mathcal{C}_a(T)$ slik at vi får regnet ut den endelige absolutte kompleksitet for prosedyredefinisjonen. Resultatet skrives til kompleksitetsfilen.

7.6.3 Klassedefinisjoner

I avsnitt 5.10 definerte vi en klassedefinisjon som en seksjonskonstruksjon, og valgte å se midlertidig bort fra de tilfeller der klassedefinisjoner inneholder prosedyredefinisjoner for `inline` prosedyrer. Slike prosedyredefinisjoner er problematiske fordi de kan inneholde foroverreferanser til attributter av klassen. Anta først at listen T av attributtdeklarasjoner i en klassedefinisjon ikke inneholder prosedyredefinisjoner. Med unntak av attributter som er deklarert `static` eller er lokale typer, kan et attributt `b` kun omtales i tilknytning til et objekt av klassen (for eksempel gjennom uttrykket `a.b`). En klassedefinisjon knytter ikke noe objekt til attributtene, så attributtdeklarasjonene i T kan ikke referere til andre attributter i T. Vi ignorerer da muligheten for referanser til lokale typer og `static` attributter i T. Alfabetet som kan brukes av attributtdeklarasjonene i T er dermed tilnærmet lik det globale navnerommet. Under gjennomlesing av T kan kompilatoren representere kompleksiteten av den hittil leste delen av T som relativ kompleksitet. Når hele T er lest, beregnes absolutt kompleksitet for T ved hjelp av størrelsen på det globale navnerommet.

Dersom T inneholder prosedyredefinisjoner, vil kompilatoren foreta en omskriving av kildekoden som er hjemlet i språkdefinisjonen for C++. Denne omskrivingen flytter prosedyredefinisjonene ut av klassedefinisjonen slik at situasjonen blir som over. Prosedyrekroppene til eventuelle prosedyredefinisjoner inne i en klassedefinsjon stues unna inntil hele klassedefinisjonen er lest. *Først da* kjenner vi attributtene prosedyrekroppene kan omtale slik at de kan analyseres som beskrevet i avsnitt 7.6.1 og 7.6.2.

7.6.4 Prosedyrekropper i klassedefinisjoner

Muligheten for foroverreferanser til attributter i prosedyrekropper er ubehagelig for utviklere av C++ kompilatorer fordi dette er det eneste stedet foroverreferanser tillates hvis man ser bort fra hopp forover til en merkelapp. Foroverreferansene vanskeliggjør en enkel implementasjon av kompilatorens frontend i ett pass over kildekoden. Foroverreferanser til merkelapper løses lett ved såkalt backpatching (se [20]), mens prosedyrekropper er mer kompliserte strukturer som krever mer kompliserte løsninger. Til hjelp i denne forbindelse spesifiserer C++ standarden (se [22] avsnitt r.9.3.2) at en prosedyredefinisjon inne i en klassedefinisjon er ekvivalent med at

prosedyredefinisjonen merkes med nøkkelordet `inline` og flyttes ut av klassedefinisjonen. Prosedyren deklareres så på vanlig måte inne i klassedefinisjonen ved en prosedyredeklarasjon uten prosedyrekropp. Basert på denne spesifikasjonen kan en kompilator skrive om litt på kildekoden ved å "klippe ut" prosedyrekroppens tekst og "lime den inn" rett etter klassedefinisjonen som i eksemplet under.

```
class circle
{
private:
  double radius;
public:
  double get_radius (void) {return (radius);}
};
```

blir etter litt "klipping og liming" internt i kompilatoren til det ekvivalente

```
class circle
{
private:
  double radius;
public:
  double get_radius (void);
};
inline double circle::get_radius {return (radius);}
```

Programteksten passer nå for analyse i ett pass. Når G++ påtreffer en prosedyrekropp T_i inne i en klassedefinisjon D, "klipper" den ut prosedyrekroppen T_i og venter med å analysere T_i til etter at hele klassedefinisjonen D er lest. Når kompilatoren når slutten av D har den altså potensielt bare analysert deler av D. La oss omtale den analyserte teksten i D som A og de "utklipte" prosedyrekroppene som $T_1 \cdots T_n$. Når hele D er lest, kan en absolutt $\mathcal{C}(A)$ beregnes, men ikke $\mathcal{C}(T_i)$ (T_i er ikke parset enda), og vi kjenner derfor ikke $\mathcal{C}(D)$. Kompilatoren instruerer nå sin leksikalske analysator om å lese fra $T_1 \cdots T_n$ slik at disse omsider parses, nå som alle klassens attributter er definert. Så fort alle T_i er analysert og alle $\mathcal{C}(T_i)$ beregnet, finner vi

$$\mathcal{C}(D) = \mathcal{C}(A) + \sum_{i=1}^{n} \mathcal{C}(T_i)$$

Beregningen av $\mathcal{C}(D)$ representerer en ubehagelig løsning på et ubehagelig problem som skyldes at C++ her avviker fra den vanlige restriktive holdningen til foroverreferanser. Mesteparten av problemet er imidlertid løst av kompilatoren i utgangspunktet, fordi foroverreferanser også representerer et problem ved kompilering.

7.6.5 Aksess til objekters attributter

I avsnitt 5.12 så vi at i uttrykk av typen `a.b` må `b` betraktes som en seksjonskonstruksjon. Vi modifiserer grammatikkens produksjoner til å regne ut en absolutt $\mathcal{C}(b)$ umiddelbart etter at `b` er lest av kompilatoren. Dette er mulig all den tid C++ krever at klassen k for `a` må defineres før denne typen uttrykk tillates. Da er antallet synlige attributter n i klassen (og dermed alfabetet for `b`) kjent slik at $\mathcal{C}(b)$ kan beregnes ved (3.10). n avhenger av klassen k og hvilken prosedyre P uttrykket `a.b` forekommer i. Noen prosedyrer har innsyn til færre attributter enn andre prosedyrer:

- Dersom P er en klasseprosedyre for klassen k eller en klasse som arver k, vil n reguleres av nøkkelordene `private`, `protected` og `public` i klassedefinisjonene.
- Dersom P er deklarert som `friend` av klassen k, blir n som om P skulle vært en klasseprosedyre av k. Nøkkelordet `friend` gjør det mulig å gi bestemte prosedyrer utvidet tilgang til attributter, og omtales i kapittel r.11.4 i [22].
- Ellers er n gitt som antallet `public` attributter i klassehierarkiet for k.

Fordi størrelsen på alfabetet for `b` avhenger av aksessdeklarasjonene i klassehierarkiet for k, vil kompleksitetsmålet belønne strengest mulig innkapsling av klassens attributter. Slik innkapsling gjør klassene mer uavhengige av hverandre, og kildekoden blir enklere å vedlikeholde. Vi finner derfor en slik belønning naturlig.

7.7 Beregning av alfabetets størrelse

Vi har sett at aksjonene for produksjoner $V \rightarrow H_1 \cdots H_n$, der en eller flere H_i er seksjonskonstruksjoner, må modifiseres til å utføre prosedyrekallet

```
make_complexity_absolute(&@i,p);
```

slik at $\mathcal{C}_r(H_i)$ assimileres i $\mathcal{C}_a(H_i)$. Når dette kallet utføres, må kardinaliteten (se definisjon 3.7) for hver symbolkategori være kjent og ligge i parameteren `p`, slik at vi kan benytte (3.10) til å regne om relativ kompleksitet for H_i til absolutt kompleksitet. Vi skal nå se hvordan vi kan skaffe til veie informasjonen i `p`.

For noen symbolkategorier er kardinaliteten konstant for enhver instans av H_i og derfor alltid kjent. Dette gjelder i vårt tilfelle symbolkategoriene for

- Reserverte ord.

- Operatorer (dersom vi ser bort fra operator overloading).

For de andre symbolkategoriene må kardinaliteten beregnes for hver instans av H_i. For de ulike formene for deklarerte identifikatorer, dvs

- Variabelnavn, prosedyrenavn, typenavn og enumererte konstanter.
- Merkelapper.
- Strukturnavn.

vedlikeholder kompilatoren tabeller. Kardinalitet for disse symbolkategoriene kan lett beregnes basert på innholdet av disse tabellene. Kompilatoren samler allerede inn all den informasjon vi trenger.

Denne strategien forutsetter at deklarasjonstabellene gir et riktig bilde av alfabetet for H_i når parseren reduserer med produksjonen $V \to H_1 \cdots H_n$. Dette er ikke nødvendigvis tilfelle, da aksjonene som er blitt utført ved parsing av $H_j \cdots H_n$, $j > i$ kan ha modifisert deklarasjonstabellene. Når parseren reduserer med $V \to H_1 \cdots H_n$, vil deklarasjonstabellene typisk inneholde de deklarasjonene som er effektive på punktet umiddelbart etter V. For noen produksjoner vil dette stemme overens med de deklarasjonene som er effektive på slutten av den aktuelle H_i. For andre produksjoner må vi skrive om litt på grammatikken slik at produksjonen

$$V \to H_1 \cdots H_n$$

splittes opp i produksjonene

$$V \to H_1 \cdots H_{new} \cdots H_n$$

$$H_{new} \to H_i$$

Når parseren reduserer med $H_{new} \to H_i$, vil deklarasjonstabellene gi et riktig bilde av antallet deklarasjoner som er synlige ved slutten av H_i, og vi kan flytte assimilasjonen av $\mathcal{C}_r(H_i)$ til aksjonen for denne produksjonen. Slike enkle endringer i grammatikken skaper ingen konflikter med det eksisterende oversettelsesskjemaet i G++.

7.7.1 Betraktninger omkring alfabetets størrelse

Vi forutsetter at alfabetet for seksjonene i en instans T av en seksjonskonstruksjon er kjent når T er lest av parseren. C++ er definert slik at kompilering kan gjøres i ett pass over kildekoden, og deklarasjonene som er plassert *etter* T er ikke effektuert når $\mathcal{C}(T)$ beregnes ved assimilasjon av $\mathcal{C}_r(T)$ i $\mathcal{C}_a(T)$. Dette medfører at antallet deklarasjoner som innføres *etter* T ikke har noen innvirkning på $\mathcal{C}(T)$. Som vi så i avsnitt 2.7 leser ikke en menneskelig leser kildekoden tegn for tegn slik en kompilator gjør. Et menneske

leser kildekoden mange ganger på jakt etter "fyrtårn" som gir grobunn for hypoteser om kildekodens virkemåte. Når vi leser T, kan deklarasjonene som er plassert etter T være i vår "mentale database" selv om T ikke kan referere til disse deklarasjonene ifølge språkets skopregler.

Vi har definert eksterne deklarasjoner som seksjonskonstruksjoner. En C++ kildefil er en sekvens av eksterne deklarasjoner. Av disse deklarasjonene er det prosedyredefinisjonene som er av størst interesse fordi de typisk utgjør det meste av filen. Av diskusjonen over ser vi at de første eksterne definisjonene i en fil vil ha et mye mindre alfabet enn de siste. Dette synes galt fordi et menneske vil lese filen f *mange* ganger og ikke bare *én* gang slik en kompilator gjør. En typisk kildefil f begynner imidlertid med å inkludere en prosjekt-headerfil h som inneholder deklarasjoner av eksterne definisjoner i f. Dette garanterer at deklarasjonene i h stemmer overens med definisjonene i f. Andre kildefiler enn f kan da kalle prosedyrer definert i f ved å inkludere h (se avsnitt 5.14). Når f inkluderer h, vil definisjonene i f bli en del av det globale navnerommet før de analyseres av parseren. *Alle definisjonene i f har da utsyn til hverandre uavhengig av rekkefølgen de defineres i.*

Bruken av headerfiler gjør at alfabetet ved beregning av $\mathcal{C}(T)$ blir mer i tråd med innholdet av den mentale databasen til en menneskelig leser. Den valgte implementasjonsstrategien burde gi rimelig gode beregninger av kompleksitet for kildekode som bruker headerfiler slik vi har beskrevet her.

7.7.2 G++ kompilatorens interne tabellverk

Den leksikalske analysatoren fungerer som kompilatorens eneste kilde til informasjon om kildekoden. Alle tabeller som vedlikeholdes av kompilatoren bygges opp basert på informasjon fra den leksikalske analysatoren. Kompilatoren har hele tiden behov for å slå opp informasjon om navngitte entiteter (variabler, merkelapper etc). Et slikt oppslag gjøres typisk hver gang entiteten refereres til av kildekoden. Det er derfor viktig at slike oppslag er implementert så effektivt som mulig. Vi skal her se nærmere på hvordan G++ kompilatoren løser dette, og får samtidig innsyn i hvordan vi best kan hente ut nyttig informasjon om alfabetet fra kompilatorens tabeller.

Identifikatornoder

Når den leksikalske analysatoren konkluderer at det neste terminalsymbolet er en deklarert identifikator, returnerer den terminalsymbolet `IDENTIFIER`, og lar symbolets semantiske verdi være gitt som en peker til en datastruktur. Denne datastrukturen kalles en *identifikatornode*. Den leksikalske analysatoren returnerer samme identifikatornode for alle forekomster av samme identifikator uavhengig av forekomstens kontekst.

Identifikatornodene inneholder pekere inn til deklarasjonsnoder i kompilatorens deklarasjonslager. Identifikatornoden for en identifikator er tilgjen-

gelig fra parserens aksjoner som den semantiske verdien til terminalsymbolet `IDENTIFIER`, og aksjonene har dermed via pekere fra identifikatornoden adgang til de deklarasjoner som er tilknyttet identifikatoren på dette punktet i kildekoden.

Deklarasjonsnoder

Hver deklarerte entitet representeres ved en *deklarasjonsnode* som inneholder all informasjon om entiteten. Forskjellige typer entiteter har forskjellige typer deklarasjonsnoder med plass til forskjellig informasjon:

- En deklarasjonsnode for en variabel inneholder en peker til variablens type.
- En deklarasjonsnode for en merkelapp angir en posisjon i objektkoden som utgjør målet for hopp med `goto`.

En identifikator kan tjene som navn på flere entiteter samtidig (for eksempel en merkelapp og en variabel, se avsnitt 5.2), så en identifikatornode må kunne peke på flere deklarasjonsnoder samtidig. Hvis vi ser bort fra overloading av funksjoner (som behandles spesielt), kan ikke entiteter fra samme entitetsgruppe deklarert på samme nivå ha identisk identifikator, så det påkrevde antallet pekere fra en identifikatornode til deklarasjonsnoder er begrenset oppad av antallet entitetsgrupper. I G++ kompilatoren inneholder en identifikatornode `id` følgende pekere til deklarasjonsnoder:

- `IDENTIFIER_GLOBAL_VALUE (id)` peker til en ekstern deklarasjon av en global variabel, type eller prosedyre.
- `IDENTIFIER_CLASS_VALUE (id)` peker til en deklarasjon av et attributt i en klasse.
- `IDENTIFIER_LOCAL_VALUE (id)` peker til en deklarasjon av en lokal variabel i en funksjon.
- `IDENTIFIER_LABEL_VALUE (id)` peker til en deklarasjon av en merkelapp (label) i en prosedyrekropp.
- `TREE_TYPE (id)` peker til en deklarasjon av et strukturnavn.

Disse pekerne oppdateres etterhvert som identifikatorenes rolle i programteksten endrer seg i takt med nye deklarasjoner. Pekere som for øyeblikket ikke peker på noen deklarasjonsnode settes til NIL (dvs ingen entitet i denne entitetsgruppen eksisterer for denne identifikatoren på dette punktet i kildekoden).

Hvilken entitet en identifikatorforekomst faktisk representerer, bestemmes endelig av den kontekst identifikatoren opptrer i. Når en aksjon utføres, er

identifikatorens kontekst gitt av den tilhørende produksjonens høyreside. I BISON-produksjonen

```
stmt: goto identifier ';'
```

kan aksjonen referere direkte til merkelappens deklarasjonsnode via pekeren `IDENTIFIER_LABEL_VALUE($2)`. Dette fordi vi ved reduksjonen har avdekket nok om kontekst til å vite at identifikatoren benyttes som merkelapp.

Oppdatering av pekerne i identifikatornodene

Ved nesting av sammensatte setninger (se avsnitt 5.9.1) må vi ta hensyn til at lokale deklarasjoner kan skygge for andre deklarasjoner. Deklarasjonsnoder innført ved deklarasjoner i sammensatte setninger er tilgjengelige fra sin identifikatornode `id` som `IDENTIFIER_LOCAL_VALUE(id)`. Dersom flere nestede lokale deklarasjoner bruker samme identifikator `id` (skygging), vil `IDENTIFIER_LOCAL_VALUE(id)` peke ut den deklarasjonen D som er assosiert med navnet `id` i den dypest nestede sammensatte setningen. Når kompilatoren forlater denne sammensatte setningen, må pekeren som returneres av `IDENTIFIER_LOCAL_VALUE(id)` oppdateres til å peke på den lokale deklarasjonen som assosieres med `id` i den omsluttende sammensatte setningen. Hvis en slik lokal D ikke eksisterer, settes `IDENTIFIER_LOCAL_VALUE(id)` til `NIL`.

For å kunne utføre oppdateringen av pekerne i identifikatornodene ved utgangen fra en sammensatt setning, vedlikeholder kompilatoren to lister av deklarasjonsnoder for en sammensatt setning som den er i ferd med å kompilere:

- En liste A inneholder alle deklarasjonsnoder innført i et omsluttende navnerom som er blitt frakoblet sine identifikatornoder fordi de skygges for av en lokal deklarasjon i den sammensatte setningen.
- En liste B inneholder alle deklarasjonsnoder for lokale deklarasjoner i den sammensatte setningen.

Når en sammensatt setning forlates, traverseres liste B, og deklarasjonsnodene i listen frakobles sin identifikatornode ved at pekeren fra identifikatornoden til deklarasjonsnoden settes lik NIL. Deretter traverseres liste A, og disse deklarasjonsnodenes identifikatornoder oppdateres til å peke på deklarasjonsnodene i listen.

Med dette er alle entiteter som ble innført ved lokale deklarasjoner i den avsluttede sammensatte setningen blitt frakoblet identifikatornodene, og entiteter som ble frakoblet fordi de ble skygget for, er blitt tilkoblet igjen. Fordi sammensatte setninger kan nestes til vilkårlig dybde og følger et strengt parentetisk regime, lagres listene A og B på en stack. Elementet i bunnen av stacken representerer det globale navnerommet, og inneholder en liste over alle eksterne deklarasjoner. Stacken gir oss dermed oversikt over alle entiteter innført

- ved ekstern deklarasjon.
- ved lokal deklarasjon i prosedyrekropper.
- som parametre til prosedyrer.

Prosedyreparametre behandles som om de var lokale variable deklarert foran den første setningen i prosedyrekroppen. Kompilatoren legger listen av parametre i et nytt element på deklarasjonsstacken så fort prosedyrehodet er lest.

7.7.3 Opptelling av deklarasjonslageret

Vi har nå sett hvordan G++ kompilatoren organiserer sitt deklarasjonslager, og er derfor i stand til å utnytte kompilatorens eksisterende struktur til vårt formål. Ved å traversere listene over deklarasjonsnoder på deklarasjonsstacken, kan vi telle opp antallet deklarerte entiteter i hver dynamiske symbolkategori K, og på denne måten bestemme kardinalitet $\mathcal{S}(K,T)$ over segmentet T.

Vi må bestemme $\mathcal{S}(K,T)$ fra innholdet av deklarasjonsstacken hver gang vi ønsker å regne om relativ kompleksitet til absolutt kompleksitet. Dette skjer hver gang kompilatoren har lest en seksjonskonstruksjon, så beregning av kardinalitet som en funksjon av deklarasjonsstacken vil skje relativt ofte.

Å traversere listene på deklarasjonsstacken er i denne sammenheng en lite effektiv beregningsmåte. Det er mer effektivt å vedlikeholde en teller for hver av deklarasjonslistene. Telleren for en deklarasjonsliste inneholder antallet deklarasjoner i listen. En slik teller angir antallet entiteter innført i en symbolkategori som et resultat av deklarasjoner i en sammensatt setning eller eksternt. Vi modifiserer kompilatoren til å oppdatere en teller for den inneværende sammensatte setningen hver gang en ny lokal deklarasjon i setningen effektueres. Å beregne antallet deklarerte entiteter for en symbolkategori reduserer seg da til å summere tellerne i alle deklarasjonslistene på deklarasjonsstacken. Antallet elementer på stacken tilsvarer nivået av nesting av sammensatte setninger, som typisk er ganske lite (< 10), så dette vil være en effektiv beregningsmåte.

Ved utgang fra en sammensatt setning, vil de entiteter som utgår fra skopet bli automatisk trukket fra antallet aktive lokale entiteter ved at det øverste elementet på stacken fjernes. Tellerne i dette elementet vil da ikke lenger inngå i oppsummeringen av antallet deklarerte entiteter.

7.7.4 Deklarasjoner fra headerfiler

Å inkludere en deklarasjon i deklarasjonstellerne så fort deklarasjonen er effektuert gir problemer fordi ikke *alle* deklarasjonsnoder nødvendigvis skal telles med fra det øyeblikk de entrer en liste på deklarasjonsstacken. Vi har

tidligere sett (avsnitt 5.14.1) at bruk av system-headerfiler tilsier at enkelte deklarasjoner kun skal bidra til kardinalitet dersom de faktisk omtales i hovedfilen.

Generelt skal en ny deklarasjonsnode bidra til økt kardinalitet kun dersom visse betingelser er oppfylt. Disse betingelsene vil typisk være et predikat på flagg i deklarasjonsnoden (for eksempel et flagg som viser om deklarasjonen har blitt benyttet i den inkluderende kildefilen). Når telling av deklarasjonsnoder er betinget, blir de deler av kompilatoren som modifiserer flagg i deklarasjonsnodene ansvarlig for at deklarasjoner telles når modifikasjoner på flaggene gjør at noden bringes til å tilfredsstille betingelsen. Dessverre foregår denne oppdateringen av flagg i deklarasjonsnodene spredd utover store deler av kompilatorens kildekode og er tildels svært uoversiktlig. Dette vanskeliggjør en korrekt implementasjon av deklarasjonstellere med betingelser. Vi har problemer med å skaffe oss oversikt over de modifikasjoner dette vil kreve.

Det viser seg imidlertid at alle forekomster av identifikatorer i *uttrykk* innføres av en bestemt produksjon i grammatikken. I C++ betraktes et uttrykk som en setning, og ethvert prosedyrekall er definert som et uttrykk. Denne produksjonen tjener som en sentral innfallsport for all omtale av entiteter, og aksjonen for denne produksjonen blir utført hver gang en identifikator opptrer i et uttrykk. All overvåking av omtale av entiteter kan derfor sentraliseres til denne aksjonen.

Vi modifiserer aksjonen slik at alle entiteter som deklareres i system-headerfiler adderes til deklarasjonstellerne først når de faktisk refereres i et uttrykk i hovedfilen. Vi kan detektere om en deklarasjon gjøres i en system-headerfil ved å følge med på "bokmerkene" som preprosessoren har etterlatt (se avsnitt 7.5.2). Bokmerkene for filer inkludert med `#include<fil>` skiller seg fra bokmerkene for filer inkludert med `#include"fil"`, og vi kan bruke dette til å skille mellom system-headerfiler og prosjekt-headerfiler (se avsnitt 5.14.1). Deklarasjoner i prosjekt-headerfiler bidrar til kardinalitet så fort deklarasjonen er sett, mens deklarasjoner i system-headerfiler "holdes på is" inntil de refereres.

Dette forutsetter at alle deklarasjonsnoder utvides med et merke som indikerer om noden er talt med i deklarasjonstellerne på deklarasjonsstacken. Vi bruker dette merket for å sikre at hver deklarasjon bare telles én gang selv om den omtales mange ganger.

7.7.5 Kompilatorgenererte deklarasjoner

Variabler opprettes ikke bare som følge av deklarasjoner i kildekoden. Kompilatoren selv kan også deklarere variabler for bruk til lagring av temporære resultater under evaluering av uttrykk etc. Det er viktig at disse temporære variablene ikke telles med i navnerommet under kompleksitetsberegningene. Kompleksiteten til en programtekst skal kun avhenge av programteksten og må være uavhengig av hvordan kompilatoren tilfeldigvis bruker temporære

variable for å implementere objektkoden.

Til stor glede for oss vedlikeholder G++ diverse flagg i deklarasjonsnodene som indikerer om deklarasjonen ble gjort av kompilatoren eller stammer fra kildekoden. Disse flaggene brukes blant annet ved generering av informasjon for "debuggere". Det er derfor en smal sak å utelate kompilatorgenererte deklarasjoner fra navnerommet ved å teste på disse flaggene.

7.7.6 Overloading

Internt i kompilatoren har alle prosedyredeklarasjoner to navn:

- Det deklarerte navnet slik det spesifiseres i prosedyredeklarasjonen. Ved overloading har flere prosedyrer samme deklarerte navn.
- Et internt navn som består av det originale navnet med et prefiks avledet fra parametrenes datatyper.

Ved overloading må parametrenes datatyper (prosedyrens *signatur*) være forskjellige nok til at man alltid kan fastslå statisk hvilken prosedyre et gitt prosedyrekall refererer til. Siden prefikset i det interne navnet er avledet fra signaturen, vil alle prosedyrer med samme deklarerte navn ha forskjellig prefiks. Det interne navnet er derfor unikt selv om flere prosedyrer har samme deklarerte navn.

Når kompilatoren slår opp i deklarasjonslageret, bruker den det interne navnet som nøkkel, og hver signatur har derfor sin egen deklarasjonsnode i deklarasjonslageret. Bruk av standard argumenter medfører flere mulige signaturer (se avsnitt 5.16), og kompilatoren oppretter da automatisk en deklarasjonsnode for hver signatur. Siden hver signatur representeres av en deklarasjonsnode, vil vår opptelling av deklarasjonslageret gi en kardinalitet i samsvar med definisjon 5.2 på side 61. Kardinalitet gjenspeiler antall *deklarasjoner*, og ikke antall deklarerte navn.

Å generere unike navn ut fra prosedyresignaturen kalles på engelsk "name mangling", og er nødvendig fordi andre verktøy forutsetter at alle eksterne definisjoner har unike navn. En linker bruker for eksempel navnene til å koble sammen alle objektfilene til en eksekverbar fil, og antar at det interne navnet kan fungere som primærnøkkel for de eksterne definisjonene i programmet.

7.7.7 Opptelling av attributter

Hvis vi ser bort fra klasseprosedyrer som er deklarert `static`, har alle klasseprosedyrene direkte tilgang til klassens attributter som om disse var deklarert eksternt (eventuelt begrenset av nøkkelordet `private` for attributter innført ved arv). Disse klasseprosedyrene tar en peker til et objekt av klassen som en implisitt parameter. I prosedyrekroppen kan denne implisitte parameteren

omtales som `this`, og klasseprosedyrens direkte referanser til attributter henspeiler på attributter i objektet som utpekes av `this`. Dette gjør at direkte referanser til attributter i `this` ser ut som referanser til eksternt deklarerte (globale) entiteter, og bidrar til kardinalitet i prosedyrekroppen. Når vi beregner kardinalitet for seksjoner av en slik prosedyre, må vi passe på å inkludere alle synlige attributter.

Deklarasjonsnodene for klasseprosedyrer inneholder en peker til klassens deklarasjonsnode. Vi modifiserer deklarasjonsnodene for klasser til å inneholde antallet attributter i hver beskyttelseskategori (`public`, `protected` og `private`) som deklareres av klassen. Disse tellerne initialiseres umiddelbart etter at klassedefinisjonen er lest, og inkluderer kun de attributter som eksplisitt deklareres i klassedefinisjonen — arvede attributter er *ikke* inkludert.

Hver klassenode inneholder en liste over klassene den arver. Basert på denne informasjonen, kan man lett beregne det totale antallet synlige attributter i prosedyrekroppen til en klasseprosedyre `A::P`. Dette antallet inkluderer

1. Attributter i klassen A som ble *deklarert* i A og ikke *arvet*.

2. Alle arvede attributter som er `public` eller `protected`. Deklarasjonsnoden for A inneholder pekere til deklarasjonsnodene for klassene den arver, og fordi klassenes deklarasjonsnoder inneholder separate tellere for hver beskyttelses-kategori, kan vi utelate fra opptellingen arvede attributter som er deklarert `private`.

7.7.8 Literaler

Vi har sett at vi ved beregning av antallet deklarerte entiteter (dvs variabler, prosedyrer, strukturer etc) har stor glede av kompilatorens deklarasjonstabeller. Dessverre vedlikeholder ikke G++ kompilatoren liknende tabeller for tekstlige og numeriske literaler. Å bestemme kardinalitet for tekstlige og numeriske literaler i kildekoden krever derfor at vi lager våre egne tabeller over kildekodens literaler.

Literaler er spesielle i den forstand at man kan referere til dem direkte uten forutgående deklarasjon. Vi skal derfor tenke oss at alle literaler er forhåndsdeklarert, og at kun de literaler som faktisk benyttes i kildekoden bidrar til kardinalitet under beregning av kompleksitet for literalforekomster. Dette blir analogt til behandlingen av prosedyredeklarasjoner i system-headerfiler, der kun de deklarasjoner som faktisk refereres, telles med.

To identiske literaler kan brukes i forskjellige sammenhenger som gjør at vi tenker på dem som forskjellige størrelser til tross for at de er identiske tekstlig sett:

```
printf('pi=%d\n',3.14);
```

```
printf('Gjennomsnittsdybde for hull i Oslos gater=%dm\n',3.14);
```

Ovenstående to setninger vil skrive ut verdien av to forskjellige konstanter som tilfeldigvis har samme verdi og derfor kan representeres ved samme literal. En menneskelig leser vil oppfatte at de to literalforekomstene i eksemplet representerer to forskjellige konstanter til tross for at de har samme verdi. En kompilator vil derimot ikke kunne registrere at de to literalene betegner to logisk forskjellige størrelser, og kan vanskelig beregne en kardinalitet som reflekterer programmererens vanskeligheter med å gi mening til literalene.

Hvis man deklarerer konstanter (ved bruk av det reserverte ordet `const`) og deretter refererer til dem ved navn, vil kompilatorens kardinalitetsberegninger være mer i samsvar med antallet entiteter programmereren må ha en forståelse av.

Bruk av literaler direkte i programteksten anses som lite strukturert, da man får et stort oppdateringsproblem hvis man senere får behov for å endre på verdien til konstanter. Vi antar at kildekoden som skal analyseres av kompilatoren hovedsaklig benytter seg av navngitte konstanter deklarert med `const` og ikke sprer "magiske tall" rundt omkring. Disse deklarerte konstantene betraktes i C++ som skrivebeskyttede variable, og vil således komme i samme leksikalske kategori som vanlige variable.

7.8 Sammendrag

Vi har nå sett hvordan G++ ble modifisert til å beregne Maus' kompleksitetsmål for C++ kildekode. Ved å dra maksimalt nytte av maskineriet i BISON klarte vi å minimalisere modifikasjonene på kompilatoren. Vi fant at kompilatoren allerede vedlikeholder tabeller for mye av den informasjonen vi ønsker å samle inn om kildekoden. Mye av arbeidet besto derfor i å sette seg inn i kompilatorens virkemåte slik at vi kunne trekke ut de nødvendige data fra de eksisterende datastrukturer. Det viste seg spesielt fruktbart og strukturerende å betrakte arbeidet med modifikasjonene som en implementasjon av et oversettelsesskjema for *kompleksitet* basert på et eksisterende oversettelsesskjema for *objektkode*. Oversettelsesskjemaet for objektkode ga oss et godt utgangspunkt for å implementere målet med et minimum av endringer på kompilatoren.

Kapittel 8

Eksperimenter med kompleksitetsmålet

Med den modifiserte kompilatoren som verktøy kan vi foreta målinger på kildekode og skaffe til veie et statistisk grunnlag for en vurdering av kompleksitetsmålet. I lys av avsnitt 3.11 er det spesielt interessant å sammenligne målinger for objektorientert kildekode og mer tradisjonelt strukturert kildekode. Vi kan da avdekke i hvilken grad kompleksitetsmålet er relatert til intuitiv kompleksitet. I dette kapitlet presenterer vi resultater av målinger på to programsystemer.

Nora er et typisk objektorientert system, der klassemekanismen utnyttes for å strukturere kildekoden. Systemet er laget for å håndtere pasientjournaler i sykehusmiljø, og tilbyr et grafisk brukergrensesnitt mot en avansert SQL database. Av kommersielle grunner fikk vi bare tilgang til deler av kildekoden. Total størrelse for Nora estimeres til ca. 25.000 linjer kildekode (kommentarer og blanke linjer medregnet). Av dette hadde vi tilgang til ca. 6.000 linjer som kunne kompileres separat. Denne delen av kildekoden består av 10 kildefiler og 38 headerfiler. Det store antallet headerfiler skyldes at man har valgt å legge hver klassedefinisjon i en egen headerfil. Dette ble gjort for å unngå unødig rekompilering av moduler ved endringer på klassedefinisjoner. De fleste globale deklarasjoner innføres ved inklusjon av et utvalg headerfiler. Avhengig av hvilke headerfiler som inkluderes i en kildefil, kan antallet synlige globale deklarasjoner variere en del mellom kildefilene i Nora.

Meta er tradisjonelt strukturert, med inndeling av kildekoden i prosedyrer, og gjør ikke bruk av klasser. Systemet er et enkelt dokumentproduksjonsystem med et trivielt kommandolinjeorientert brukergrensesnitt. Hele kildekoden var tilgjengelig. Total størrelse er ca. 5.000 linjer kildekode (kommentarer og blanke linjer medregnet). Meta består av 10 kildefiler og en prosjekt-headerfil `meta.h` som inneholder deklarasjoner av alle prosedyrer som kalles på tvers av kildefiler (se avsnitt 5.14 og

5.14.1). Alle kildefilene som er av interesse ved kompleksitetsmålinger inkluderer `meta.h`. De fleste globale deklarasjoner innføres fra `meta.h`, og de interessante kildefilene har derfor utsyn til grovt sett det samme antallet globale deklarasjoner.

Ideelt sett ønsker vi å utføre målinger på to implementasjoner av samme programsystem — en objektorientert og en "tradisjonell" implementasjon. Slike målinger er direkte sammenlignbare og vil gi oss et godt bilde av hvordan kompleksitetsmålet belønner objektorientert programmering. I mangel av slik luksus må vi klare oss med Nora og Meta, og kan ikke trekke like kraftige konklusjoner av målingene. Når vi sammenligner målingene for Nora og Meta må vi for eksempel huske på at Nora totalt sett er et mye større system enn Meta.

8.1 Observasjonene

Når man introduserer et nytt kompleksitetsmål, er det å håpe at målet er mer sensitivt for interessante aspekter ved kildekoden, og dermed "bedre" enn eksisterende mål. I praksis brukes som regel antall linjer kildekode $LOC(T)$ som et enkelt kompleksitetsmål for kildekoden T. Vi ønsker derfor å sammenligne $\mathcal{C}(T)$ med $LOC(T)$ for et større antall T slik at vi kan vurdere hvor følsom $\mathcal{C}(T)$ er for strukturen til T. Figur 8.1 og 8.2 viser sammenhengen mellom $\mathcal{C}(T)$ og $LOC(T)$ for hver prosedyredefinisjon T i henholdsvis Nora og Meta.

I forbindelse med analyse av sammenhengen mellom antallet synlige deklarasjoner og entropi $\mathcal{H}(T)$, er det ønskelig med målinger der vi ignorerer alle symbolforekomster som ikke er *navn*. Slike målinger er ufølsomme for variasjoner i andre symbolkategorier, og bedre egnet når vi ønsker å danne oss et bilde av hvordan målet påvirkes av antallet synlige deklarasjoner.

Definisjon 8.1 *$\mathcal{C}_{navn}(T)$ er kompleksitet for de n forekomstene av navn i utsnittet T av kildekoden. Vi har*

$$\mathcal{H}_{navn}(T) = \frac{\mathcal{C}_{navn}(T)}{n}$$

Vi skal se at målingene på kildefilene for Nora og Meta hovedsaklig varierer i takt med antallet globale deklarasjoner som er synlige fra kildefilene. Vi skal bruke g som betegnelse på antallet globale deklarasjoner som er synlige fra en kildefil.

8.2 Variasjoner i kompleksitet

Av figur 8.1 og 8.2 aner vi en klar lineær sammenheng mellom $\mathcal{C}(T)$ og $LOC(T)$ slik vi rimeligvis kan forvente. På denne underliggende lineære

sammenhengen er det overleiret en viss spredning som kan tolkes dithen at $\mathcal{C}(T)$ fanger opp aspekter ved T som ikke registreres av $LOC(T)$. Spredningen synes å øke med $LOC(T)$. Dette kan forklares med at det er mer rom for struktur i en lang prosedyre enn i en kort, og at variasjonene i struktur fanges opp av $\mathcal{C}(T)$.

$LOC(T)$ er i motsetning til $\mathcal{C}(T)$ avhengig av den rent tekstlige organiseringen av T. $\mathcal{C}(T)$ betrakter T som en strøm av symboler, og er ikke sensitiv for inndelingen av T i linjer. Vi kan få et bedre bilde av struktursensitiviteten til $\mathcal{C}(T)$ dersom vi holder $\mathcal{C}(T)$ opp mot *antall symboler* i T. Figur 8.3 og 8.4 viser resultatet for henholdsvis Nora og Meta med inntegnet regresjonslinje. Spredningen er nå mye mindre, så mye av spredningen i figur 8.1 og 8.2 kan tilsynelatende tilskrives det faktum at $LOC(T)$ er et unøyaktig mål for størrelsen av T. Ut fra figurene kan vi like gjerne bruke antall symboler som kompleksitetsmål.

8.2.1 Analyse av resultatene for Meta

For Meta kan vi forklare den sterke lineære sammenhengen med at det globale navnerommet er svært stort i forhold til lokale navnerom. La l være antallet lokale deklarasjoner i en vilkårlig prosedyredefinisjon P i en kildefil F, og la g være antallet globale deklarasjoner som er synlige i filen F. En forekomst f av et navn i en P vil da typisk ha kompleksitet

$$\mathcal{C}(f) = \log(g + l) \tag{8.1}$$

Vi ignorerer da forekomster av typen `a.f`, der $\mathcal{C}(f)$ bestemmes utelukkende av antall tilgjengelige attributter i `a` — effekten av slike forekomster ser ut til å være neglisjerbar.

Når g er svært stor i forhold til typiske verdier for l, vil $\mathcal{C}(f)$ være lite sensitiv for variasjoner i l. Tabell 8.1 og 8.2 viser hvilke verdier g antar i kildefilene for henholdsvis Meta og Nora. Vi vil typisk ha $l < 10$, så med en g hentet fra tabell 8.1 vil $\mathcal{C}(f)$ være tilnærmet konstant og uavhengig av hvilken prosedyre P forekomsten f hentes fra. Målinger viser at gjennomsnittlig en tredjedel av symbolforekomstene i en prosedyredefinisjon er navn. Resten er hovedsaklig symboler fra statiske symbolkategorier, og vi skal regne med at et symbol fra en statisk symbolkategori gjennomsnittlig bidrar med en konstant kompleksitet s uavhengig av hvor den opptrer. Når l er liten i forhold til g, blir $\mathcal{C}(P)$ tilnærmet proporsjonal med antall symboler n i P:

$$\mathcal{C}(P) \approx \frac{2}{3}ns + \frac{1}{3}n\mathcal{C}(f) \approx \frac{2s+k}{3}n, k = \log g \approx \mathcal{C}(f)$$

Fordi de fleste globale deklarasjoner i Meta innføres ved inklusjon av prosjekt-headerfilen `meta.h`, vil g være av samme størrelsesorden i nesten alle Metas kildefiler (filene `prolog.c` og `unixUiO.c` inkluderer *ikke* `meta.h`).

Kildefil F	$LOC(F)$	$\mathcal{C}(F)$	g	$\mathcal{H}(F)$	$\mathcal{H}_{navn}(F)$
`aloc.c`	44	1323	69	5.40	6.15
`eval.c`	179	6350	82	5.47	6.05
`flsh.c`	482	17141	101	5.61	6.24
`hash.c`	60	1961	66	5.15	5.21
`meta.c`	407	14504	109	5.66	6.52
`mtrc.c`	192	31874	64	6.03	6.06
`prolog.c`	336	4459	8	5.94	2.97
`prop.c`	314	14166	73	5.68	6.18
`read.c`	135	5093	79	5.68	6.30
`unixUiO.c`	18	489	3	4.29	2.37

Tabell 8.1: Resultater av målingene på kildefilene for Meta

Antall symboler n i prosedyren P er da et like godt mål for kompleksitet som $\mathcal{C}(P)$ når vi ønsker å identifisere den mest komplekse prosedyren i Meta.

Når man plasserer alle prosedyredeklarasjoner i én headerfil (`meta.h`) vil det globale navnerommet i kildefilene bli kunstig stort. Ved å bruke flere headerfiler der hver headerfil deklarerer en gruppe av prosedyrer som *hører naturlig sammen*, kan man inkludere kun de prosedyrer som har interesse for den inkluderende kildefilen F. Antallet globale deklarasjoner som er synlige i F reflekterer da bedre det antallet entiteter vi må være kjent med for å forstå en prosedyre P i F, og $\mathcal{C}(P)$ vil være et bedre mål enn antall symboler i P når vi sammenligner prosedyrer som defineres i *forskjellige* kildefiler.

8.2.2 Analyse av resultatene for Nora

Fordi Nora gjør bruk av *klasser*, må vi modifisere (8.1) til å inkludere antallet attributter a i den klassen hvor prosedyren P er en klasseprosedyre. En forekomst f av et navn i P har da kompleksitet

$$\mathcal{C}(f) = \log(g + a + l) \tag{8.2}$$

Som for Meta vil l være liten i forhold til resten av navnerommet, slik at

$$\mathcal{C}(P) \approx \frac{2s + k}{3} n, k = \log(g + a) \approx \mathcal{C}(f)$$

To klasseprosedyrer for *samme* klasse vil typisk ha identisk $g + a$ dersom de defineres i samme kildefil. Antall symboler n i P blir da et like godt mål som $\mathcal{C}(P)$ når vi vurderer hvilken av de to som er mest kompleks. To klasseprosedyrer for *forskjellige* klasser kan generelt ha sterkt avvikende verdier for a, så det er overraskende at den lineære sammenhengen er såpass sterk for Nora. Noe av forklaringen kan ligge i at mange av klassene i Nora arver

Kildefil F	$LOC(F)$	$\mathcal{C}(F)$	g	$\mathcal{H}(F)$	$\mathcal{H}_{navn}(F)$
`accdb.cxx`	165	5895	123	5.91	7.16
`access.cxx`	152	4129	121	5.87	6.95
`acfilt.cxx`	35	666	104	5.37	6.44
`acfsm.cxx`	365	10632	168	6.11	7.48
`achook.cxx`	153	3021	93	5.44	6.25
`acproc.cxx`	65	1510	84	5.57	6.01
`windisp.cxx`	573	21046	172	6.19	7.58
`winmenu.cxx`	156	5416	201	6.15	7.47

Tabell 8.2: Resultater av målingene på kildefilene for Nora

et såkalt "application framework" for et grafisk brukergrensesnitt. Klassene arver da et stort antall attributter, så a er i utgangspunktet stor. Mange av klassene i Nora utviser derfor liten *prosentvis* forskjell i a, og antall symboler n i P blir et like godt mål som $\mathcal{C}(P)$ *til tross for* at prosedyrene tilhører forskjellige klasser med litt forskjellig a.

8.3 Variasjoner i entropi

Vi forventer at entropi $\mathcal{H}(F)$ (informasjonsmengde pr symbol) for en fil F øker med antall globale deklarasjoner g som er synlige i F. I tabell 8.3 ser vi at gjennomsnittlig entropi for prosedyrer i Nora er signifikant høyere enn i Meta. Dette stemmer bra med at Nora totalt sett er et mye større system enn Meta, og har mange flere globale deklarasjoner. Målet fanger her opp et aspekt ved kildekoden som ikke gjenspeiles i antall linjer kode eller antall symboler. Vi finner igjen den gjennomsnittlige entropien som stigningstallet for regresjonslinjene i figur 8.3 og 8.4.

Av tabell 8.2 ser vi at for filene i Nora vokser $\mathcal{H}(F)$ med g som forventet. De avvikene vi finner skyldes typisk at filene inneholder klasseprosedyrer for forskjellige klasser med litt varierende antall attributter a. Tabell 8.1 gir samme data for Meta. Vi merker oss at filene `mtrc.c` og `prolog.c` har merkelig høy $\mathcal{H}(F)$ i lys av g for F. Meta gjør ikke bruk av klasser, så dette kan ikke skyldes klasser med et stort antall attributter. De to filene viser seg å inneholde store tabeller initialisert med et stort antall literaler. Mesteparten av $\mathcal{C}(F)$ stammer derfor fra forekomster av *literaler*, og ikke fra forekomster av *navn*. Dette forklarer hvorfor $\mathcal{H}(F)$ er uventet høy i lys av den lave verdien for g.

Gjennomsnittlig informasjonsmengde pr forekomst av et *navn* $\mathcal{H}_{navn}(F)$ er en funksjon av antallet synlige deklarasjoner og er ikke sensitivt for bruken av literaler. Som forventet ser vi at $\mathcal{H}_{navn}(F)$ utviser en bedre sammenheng

med g enn $\mathcal{H}(F)$ for `mtrc.c` og `prolog.c`.

Observasjonene for `mtrc.c` og `prolog.c` reiser spørsmål ved om vi behandler literaler på en formålstjenlig måte. Dersom vi ønsker å bruke målet til å lokalisere de deler av kildekoden som bør restruktureres, gir målet i sin nåværende form sterke *falske* signaler om at `mtrc.c` bør analyseres nærmere. `mtrc.c` inneholder bare en meget enkel prosedyre som gjør et oppslag i en stor initialisert tabell, og representerer ingen stor mental utfordring gitt at tabellen er initialisert korrekt.

Figur 8.5 og 8.6 viser $\mathcal{H}(P)$ for hver prosedyre P i henholdsvis Nora og Meta. Spredningen i $\mathcal{H}(P)$ synes å avta når P øker i størrelse. Forskjellige symbolkategorier K kan ha svært forskjellig kardinalitet i P, så et utvalg korte prosedyrer kan ha store variasjoner i $\mathcal{H}(P)$ avhengig av hvilke symbolkategorier de få symbolene stammer fra. I store prosedyrer vil dette jevne seg ut, og spredningen i $\mathcal{H}(P)$ avtar. Denne betraktningen må imidlertid modereres av at vi har mange flere små prosedyrer enn store. Den grafiske fremstillingen kan da lett gi oss et galt bilde av spredningen.

8.4 Modulorientert bruk av headerfiler

Nora gjør bruk av *mange* prosjekt-headerfiler der hver headerfil inneholder prosedyrer og klasser som hører naturlig sammen. Nora bruker headerfiler for å beskrive moduler, og ikke bare for konsistenssjekk av prosedyrekall slik vi beskrev i avsnitt 5.14. I Meta brukes headerfilen `meta.h` kun til konsistenssjekk, og beskriver ikke en modul. Kildefilene i Nora kan (i motsetning til kildefilene i Meta) være selektive ved inklusjon av headerfiler slik at det globale navnerommet ikke blir kunstig stort. De fleste klasseprosedyrene i Nora tilhører imidlertid klasser som arver et stort arsenal av klasseprosedyrer fra "application framework". Dette gjør at navnerommet i en *klasse*prosedyre blir kunstig stort selv om det globale navnerommet begrenses ved selektiv inklusjon av headerfiler. Modulorientert bruk av headerfiler løser ikke problemet med kunstig store navnerom for klasseprosedyrer i store klassehierarkier.

8.5 Målet relatert til intuitiv kompleksitet

Nora og Meta er kommersielle systemer som er skrevet av dyktige programmerere. Mye arbeid har vært lagt ned for å sikre at programmene har god struktur. Vi kan derfor anta at kildekoden for Nora og Meta er delt opp i prosedyrer og klasser på en måte som letter vedlikehold og forståelse maksimalt.

Som vi så i avsnitt 3.11, vil et optimalt strukturert *objektorientert* program ha prosedyrer med gjennomsnittlig 5 til 16 setninger uavhengig av pro-

Parameter	*Nora*	*Meta*
Gjennomsnitt kompleksitet for prosedyrer i bits	503	943
Gjennomsnitt linjer kildekode for prosedyrer	16	25
Gjennomsnitt terminalsymboler i prosedyrer	83	168
Gjennomsnitt entropi for prosedyrer bits/symbol	6.05	5.51
Deklarasjoner i ustrukturert ekvivalent	—	230
Optimalt antall prosedyrer	—	21
Faktisk antall prosedyrer	101	62

Tabell 8.3: Nøkkeltall for Nora og Meta

grammets totale størrelse [1]. Gjennomsnittlig prosedyrelengde for optimalt strukturerte *tradisjonelle* programmer vil derimot øke med programstørrelsen. Under forutsetning av at:

1. Nora og Meta faktisk *er* optimalt strukturerte programmer.

2. Programmene er av en viss størrelse, slik at optimal prosedyrestørrelse for Meta blir større enn den universelle prosedyrestørrelsen for objektorientert kildekode.

3. $\mathcal{C}(T)$ oppviser en god sammenheng med intuitiv kompleksitet for en programtekst T.

Da kan vi ut fra betraktningene i 3.11 forvente at prosedyrene i Meta gjennomsnittlig er større og mer komplekse enn prosedyrene i Nora. Resultatene i tabell 8.3 bekrefter at så er tilfelle. Teorien om optimal inndeling i klasser og prosedyrer undergraves ikke av målingene. Vi føler oss derfor tryggere på at $\mathcal{C}(T)$ oppviser en viss sammenheng med den intuitive kompleksitet programmererne la til grunn da de programmerte Nora og Meta.

8.6 Ustrukturert ekvivalent

Av tabell 8.3 ser vi at gjennomsnittlig prosedyrelengde for Nora (16 *linjer*) stemmer godt overens med den universelle prosedyrelengden for optimalt strukturert *objektorientert* kildekode (mellom 5 og 16 *setninger*) som estimeres i [1]. For optimalt strukturert *prosedyreorientert* kildekode forventer vi at gjennomsnittlig prosedyrelengde *ikke* er konstant, men *øker* med programmets lengde. Det synes derfor interessant å undersøke om det faktiske antallet prosedyrer i Meta stemmer overens med formel (3.12) for det optimale antallet prosedyrer k_{opt} i et tradisjonelt strukturert program. Da vil argumentasjonen i 8.5 om at målet er godt relatert til intuitiv kompleksitet

styrkes ytterligere. For å beregne k_{opt} tenker vi oss et ustrukturert program uten prosedyrer som er ekvivalent med Meta. Antallet deklarasjoner n i denne ustrukturerte ekvivalenten (UE) bestemmer k_{opt} ved (3.12):

$$k_{opt} = \sqrt{2n} - 1$$

To mengder av deklarasjoner i Meta er interessante for beregning av n for en tenkt UE:

- Alle lokale deklarasjoner i prosedyrer med unntak av parametre. Denne mengden lokale deklarasjoner kaller vi L.
- Alle globale deklarasjoner som ikke er prosedyredeklarasjoner. Denne mengden globale deklarasjoner kaller vi G.

Vi skal estimere n som

$$n = \mid L \mid + \mid G \mid \tag{8.3}$$

Kompilatoren kan beregne $\mid L \mid$ ved hjelp av deklarasjonsstacken som ble omtalt i avsnitt 7.7.2. Vi kan lett holde prosedyreparametre utenfor $\mid L \mid$ fordi parametre ligger i egne elementer på deklarasjonsstacken. Beregning av $\mid G \mid$ bør gjøres manuelt. En og samme globale entitet kan deklareres i mange kildefiler, så maskinell opptelling kan komme i skade for å telle globale entiteter flere ganger.

Resultatet av målingene finner vi i tabell 8.3. Vi ser at det optimale antallet prosedyrer k_{opt} er mye mindre enn det faktiske antallet prosedyrer. I [1] estimerer Maus at et 50% intervall omkring k_{opt} vil gi en nær optimal struktur. På tross av dette slingringsmonnet finner vi det vanskelig å ta resultatet til inntekt for (3.12) under antagelsen om at Meta faktisk *har* en nær optimal struktur. Avviket mellom beregnet verdi og faktisk verdi er hele 300%. Det er mer nærliggende å anta ett av følgende:

- Meta er *ikke* optimalt strukturert.
- Beregnet k_{opt} er riktig dersom vi måler kompleksitet med kompleksitetsmålet, men målet korellerer dårlig med den følelse av intuitiv kompleksitet som programmererne la til grunn da de programmerte Meta.
- Meta er "mer objektorientert" enn tradisjonelle prosedyreorienterte programmer, og passer dårlig inn med de antagelser Maus gjorde i utledningen av (3.12). Mange av prosedyrene i Meta arbeider på samme datastruktur, og kan betraktes som klasseprosedyrer for denne datastrukturen slik vi skisserte i avsnitt 3.10.2. Klasseprosedyrer vil typisk være korte, så antallet prosedyrer vil øke når vi simulerer klasser på denne måten.

Vi håpet å finne en god sammenheng mellom k_{opt} og det faktiske antallet prosedyrer i Meta slik at sammenhengen mellom kompleksitetsmålet og intuitiv kompleksitet skulle bli sterkere underbygget. Resultatene for k_{opt} gir oss imidlertid ingen ytterligere støtte for at resultatene i 8.5 kan tas til inntekt for kompleksitetsmålets sammenheng med intuitiv kompleksitet.

8.6.1 Optimal prosedyreinndeling for G++

Det er interessant å merke seg at G++ kompilatoren (som er skrevet i C uten bruk av klasser) inneholder en del *meget* store prosedyrer. G++ kompilatoren består av ca. 150.000 linjer kildekode, og inneholder flere prosedyrer som er mellom 1.000 og 2.000 (!) linjer lange. Vi antar at opphavsmennene til G++ har strukturert programmet slik det best lar seg gjøre uten bruk av klasser. Når prosedyrene til tross for dette blir ekstremt store og uhåndterlige, understøtter det Maus' påstand om at optimal inndeling i prosedyrer vil gi ubehagelig store prosedyrer for store programmer. De enorme prosedyrene kan imidlertid like godt skyldes dårlig programmering. For å bringe klarhet i dette, bør vi analysere et større antall store programsystemer skrevet av *forskjellige* programmerere. Dessverre klarer ikke G++ å kompilere seg selv, så vi får ikke utført målinger på G++.

8.7 Sammendrag

Hvis vi ønsker å bruke målet til å identifisere den mest komplekse prosedyren i et program der hver kildefil har utsyn til et stort antall (flere hundre) globale deklarasjoner, kan vi like gjerne bruke antall symboler pr prosedyre som kompleksitetsmål. Å bruke én prosjekt-headerfil som inkluderes i alle kildefilene er spesielt uheldig for målet. Målets struktursensitivitet har en tendens til å "drukne" i et stort antall globale deklarasjoner.

Dersom vi ønsker å sammenligne kildekode hentet fra *forskjellige* programmer, vil målet i en viss grad være sensitivt for programmets totale størrelse. Antallet globale deklarasjoner vil øke med programstørrelsen og øke entropien, som utgjør proporsjonalitetskonstanten i den tilsynelatende lineære sammenhengen mellom antall symboler og kompleksitetsmålet.

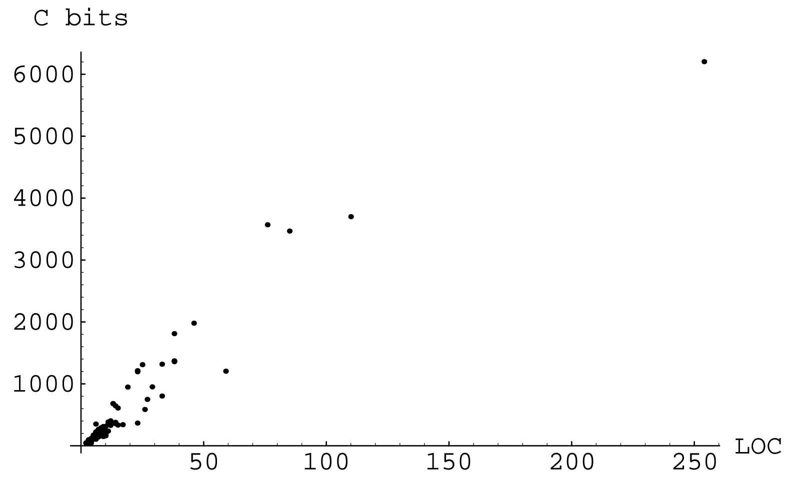
Hvilket mål vi velger for kompleksitet har mye å si for hvordan programmer blir seende ut dersom programmererne lar seg styre av kompleksitetsmålet. Derfor er det ikke likegyldig om vi bruker Maus' mål eller antall symboler som kompleksitetsmål:

- Vi føler at teorien bak Maus' mål vil lede programmererne til å skrive mer strukturert, med en fornuftig bruk av klasser og prosedyrer.

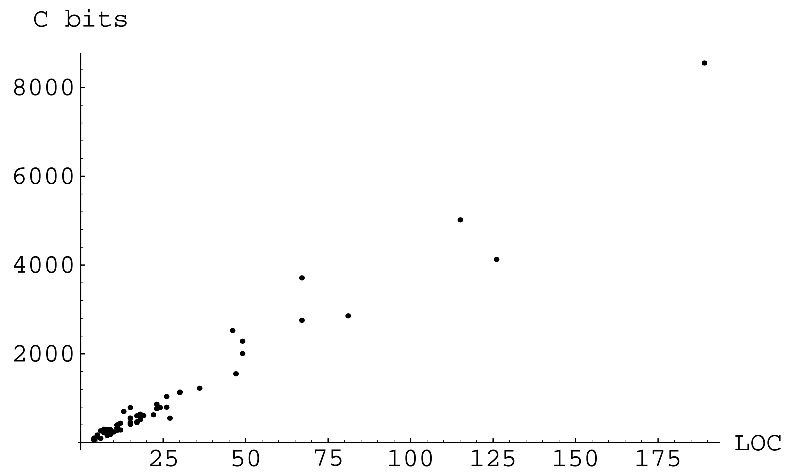
- Antall symboler brukt som kompleksitetsmål belønner kortest mulige programmer med minimal bruk av klasser og prosedyrer. Dersom pro-

grammerernes arbeid vurderes i lys av et slikt mål, vil de kvie seg for å innføre prosedyrer fordi prosedyredefinisjoner bare er plasskrevende innpakningspapir for prosedyrekroppene.

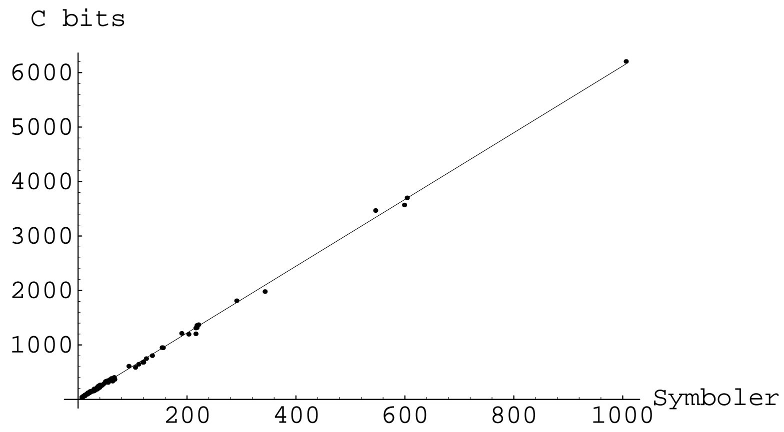
Vi tror derfor det vil være sunnest å bruke Maus' mål ved vedlikeholds-programmering. Filosofien bak dette målet er konstruktiv, og til hjelp når vi restrukturerer kildekode for å lette vedlikehold.



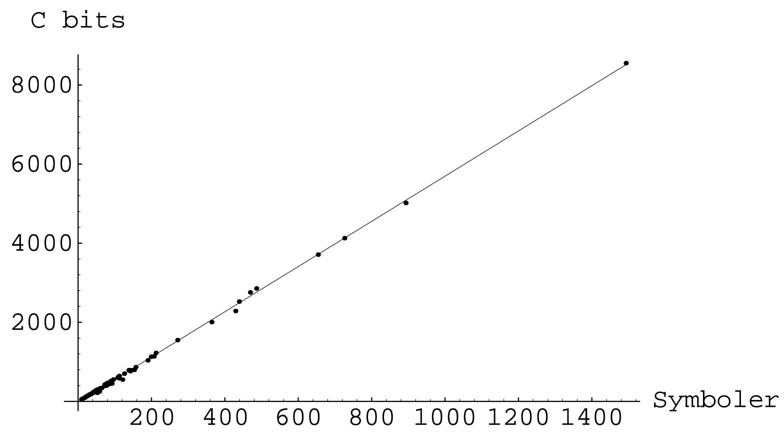
Figur 8.1: Kompleksitet og linjer kode for prosedyrer i Nora



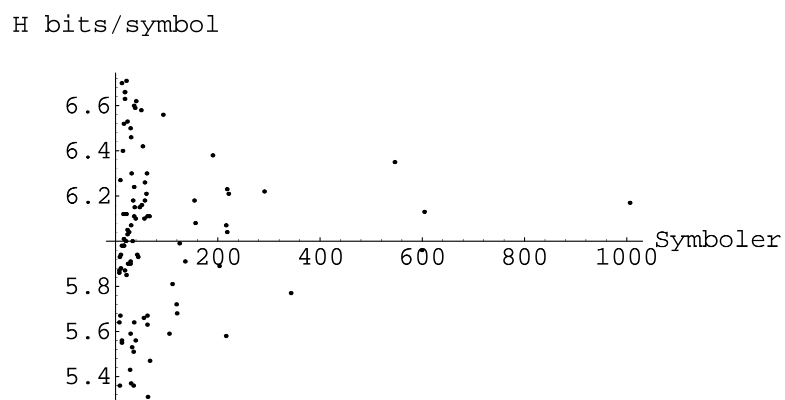
Figur 8.2: Kompleksitet og linjer kode for prosedyrer i Meta



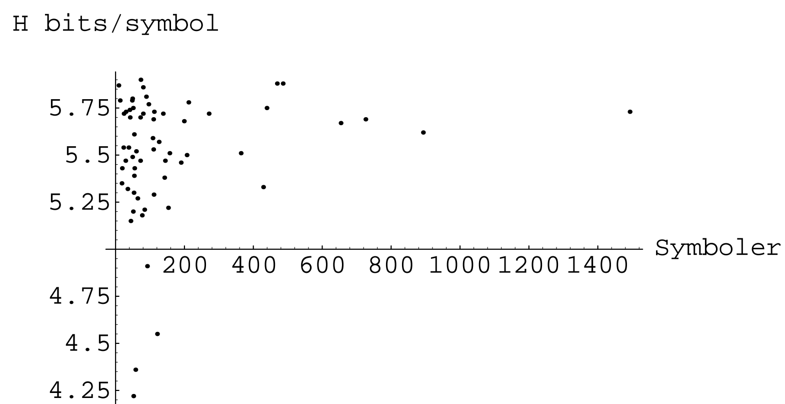
Figur 8.3: Kompleksitet og antall symboler for prosedyrer i Nora med regresjonslinje



Figur 8.4: Kompleksitet og antall symboler for prosedyrer i Meta med regresjonslinje



Figur 8.5: Entropi og antall symboler for prosedyrer i Nora



Figur 8.6: Entropi og antall symboler for prosedyrer i Meta

Kapittel 9

Diskusjon

I kapittel 8 fikk vi erfare målets svakheter. Basert på disse erfaringene ønsker vi i dette kapitlet å presentere noen forslag til forbedringer av målet. Da forsøkene med målet var ganske begrensede, presenterer vi et forsøk beskrevet i litteraturen som synes å støtte kompleksitetsmålet.

I lys av erfaringene med arbeidet på G++ synes det klart at den tradisjonelle måten å organisere kildekode på er langt fra optimal sett fra et vedlikeholdssynspunkt. Vi belyser derfor noen konsepter som kan gjøre kildekode bedre egnet for vedlikehold og modifikasjoner.

9.1 Målets struktursensitivitet

I avsnitt 8.2 så vi at målets struktursensitivitet vil avta med økende antall globale deklarasjoner g. En forekomst f av et navn i en prosedyrekropp med l lokale deklarasjoner har kompleksitet:

$$\mathcal{C}(f) = \log(g + l)$$

Det er ønskelig å modifisere målet slik at $\mathcal{C}(f)$ ikke domineres så sterkt av g (som typisk er mye større enn l). Vi skal nå presentere to alternative modifikasjoner som kan gjøre g mindre dominerende.

9.1.1 Begrensning av alfabetet

Når en programmerer leser et uttrykk i en prosedyrekropp og må tolke forekomstene av deklarerte identifikatorer i uttrykket, vil han ofte søke etter deklarasjonene for disse identifikatorene. Han vil starte søket blant de lokale deklarasjonene i prosedyren fordi de er sterkest bundet til den delen av kildekoden han prøver å forstå. Kun dersom dette søket ikke fører frem, vil søket bli utvidet til å omfatte deklarasjoner utenfor prosedyren. Generelt vil søket etter deklarasjoner foregå i flere iterasjoner. Søkeområdet utvides

til omsluttende navnerom ved hver iterasjon inntil den søkte deklarasjonen påtreffes.

Basert på denne modellen, kan vi forvente at forekomster av lokalt deklarerte entiteter er mindre anstrengende enn forekomster av globalt deklarerte entiteter. Lokale deklarasjoner er både tekstlig og logisk sterkt knyttet til det utsnittet av programteksten vi fokuserer på. De fleste globale deklarasjoner er ikke relevante for forståelsen av et lite utsnitt av programmet. Lokale deklarasjoner er alltid relevante, og fungerer bedre som "fyrtårn" under dannelsen av hypoteser om utsnittet (se avsnitt 2.7).

Anta at f er en forekomst av et navn i en prosedyrekropp. $\mathcal{C}(f)$ er en funksjon av antallet unike navn s som er synlige i den seksjonen der f forekommer. Vi har

$$\mathcal{C}(f) = \log s$$

Alle f i samme seksjon har således identisk $\mathcal{C}(f)$ uavhengig av om f er deklarert lokalt eller globalt. Hvis vi lar s variere med hvor f ble deklarert, kan vi få $\mathcal{C}(f)$ mer i takt med betraktningene over: *Jo mer lokal deklarasjonen av f er i forhold til forekomsten f, jo mindre $\mathcal{C}(f)$*. Anta at symbolforekomsten f er omgitt av n nivåer med deklarasjoner. Vi lar d_i representere antallet deklarasjoner som innføres på nivå i.

- d_1 er antallet deklarasjoner på nivået nærmest f. d_1 vil typisk være antallet lokale deklarasjoner i prosedyren.
- d_n er antallet globale deklarasjoner.

Dersom f deklareres på nivå k, kan vi sette opp følgende alternative kompleksitetsmål:

$$\mathcal{C}(f) = logs, s = \sum_{i=1}^{k} d_i \tag{9.1}$$

Kompleksitetsbidraget fra forekomster av lokalt deklarerte navn er nå uavhengig av antallet globale navn. Vi forventer derfor at denne modifikasjonen hindrer målet i å "drukne" i globale deklarasjoner.

9.1.2 Prosedyrenavn som egen symbolkategori

Følgende observasjoner synes fruktbare:

- De fleste globale (eksterne) deklarasjoner er prosedyredeklarasjoner.
- Prosedyrekall kan lett identifiseres ved at prosedyrenavnet alltid må etterfølges av parenteser. Vi kan derfor la prosedyrenavn være en egen symbolkategori.

Med *prosedyrenavn* som egen symbolkategori, vil de fleste symboler som tidligere ble kategorisert som *navn* bli kategorisert som *prosedyrenavn*. Kardinalitet for symbolkategorien *navn* vil avta dramatisk. Dersom f er en forekomst av et variabelnavn i en prosedyre, og vi lar g_v og l betegne henholdsvis antall globale og lokale *variable*, får vi nå

$$\mathcal{C}(f) = \log (g_v + l)$$

Fordi g_v typisk er en brøkdel av det *totale* antall globale deklarasjoner g (de fleste globale deklarasjoner er prosedyrer), vil ikke $\mathcal{C}(f)$ domineres like sterkt av g, men variere mer i takt med l. Studier av kildekoden for Meta indikerer dessuten at antallet variabelreferanser er mye større enn antall prosedyrekall. Vi kan derfor vente at denne modifikasjonen gir signifikante utslag på målingene.

Den eksisterende implementasjonen av målet kan modifiseres til å kategorisere en identifikator som *prosedyrenavn* så fort parseren avdekker at identifikatoren opptrer i rollen som prosedyrenavn (se avsnitt 7.5). Deklarasjonsnodene i kompilatorens deklarasjonslager (se avsnitt 7.7.2) inneholder nok informasjon til at vi kan skille mellom deklarasjoner av prosedyrer og variable, så vi kan lett beregne kardinalitet for de to symbolkategoriene *navn* og *prosedyrenavn*.

Fordi C++ ikke tillater lokale prosedyredefinisjoner, kan vi forvente at prosedyrenavn som egen symbolkategori vil gi omtrent samme resultater som modifikasjonen i avsnitt 9.1.1.

9.2 Informasjon og kompleksitet

Det implementerte kompleksitetsmålet er simpelthen et mål for informasjonsmengden av programteksten. Prinsippet er at *jo mer informasjon vi må fordøye under forståelsesprosessen, jo vanskeligere er det å etablere en forståelse for hvordan et system fungerer.* Denne tilnærmingsmåten har både positive og negative sider:

- Fordi informasjonsmengde er eksakt definert som det minste antallet bits som kreves for å kode informasjon slik at dekoding er mulig, er kompleksitetsmålet ikke et resultat av subjektive vurderinger av hvilke språkkonstruksjoner som bidrar mest til kompleksitet og i hvilken grad.

- Samme informasjonsmengde kan imidlertid representere forskjellig grad av anstrengelse i forståelsesprosessen — dvs forskjellig grad av kompleksitet. Dette er særlig tydelig i forbindelse med løkker og hopp. Vårt mål fanger for eksempel ikke opp den induktive anstrengelse som er involvert i forståelsen av en løkke.

Det kunne være ønskelig å gjøre $\mathcal{C}(T)$ mer følsom for kontrollstrukturen til T. I sin nåværende form utviser $\mathcal{C}(T)$ en viss følsomhet for struktur ved at inndelingen av T i seksjoner er en funksjon av inndelingen av T i navnerom, men $\mathcal{C}(T)$ indikerer ikke hvor vanskelig det er å "følge tråden" i T slik McCabes mål gjør (se avsnitt 2.4).

9.2.1 Løkker

De største utfordringene under arbeidet med å forstå et program møter man som regel når man prøver å forstå *løkker*. For å forstå ei løkke, må man lese løkkekroppen T gjentatte ganger på jakt etter et mønster som vedlikeholdes av hver iterasjon gjennom løkka. Dette mønsteret kalles løkkas *invariant*. Vår forståelse av T øker med hver gjennomlesing. Etterhvert som forståelsen øker, representerer T mindre og mindre ny informasjon til bruk i forståelses-prosessen. *Informasjonsmengden som absorberes i hver gjennomlesing er avtagende.*

Definisjon 9.1 *$\mathcal{C}_i(T)$ er mengden informasjon som absorberes ved $i+1$'te gjennomlesing av løkkekroppen T. $\mathcal{C}_0(T)$ utgjør således informasjonsmengden ved første gjennomlesing.*

Definisjon 9.2 *$\mathcal{L}(T)$ er den totale informasjonsmengden vi må absorbere ved gjentatte gjennomlesinger av løkkekroppen T for å forstå løkka.*

Dersom vi leser T n ganger før vi forstår løkka, har vi

$$\mathcal{L}(T) = \sum_{i=0}^{n-1} \mathcal{C}_i(T)$$

Vi kan ta $\mathcal{L}(T)$ som et grovt mål for den induktive anstrengelsen som er forbundet med å forstå løkka. Vi kan imidlertid ikke si noe om hvor mange ganger vi må lese T, eller hvordan $\mathcal{C}_i(T)$ avtar med økende i, så vi kan vanskelig si noe om hvordan $\mathcal{L}(T)$ skal beregnes.

9.2.2 Et modifisert kompleksitetsmål

I [23] estimeres at hver gjentatte lesing av en programtekst tar $\frac{2}{3}$ så lang tid som ved forrige gjennomlesing. Vi skal ta dette til inntekt for

$$\mathcal{C}_i(T) = k^i \mathcal{C}_0(T) = k^i \mathcal{C}(T), k = \frac{2}{3} \tag{9.2}$$

Basert på (9.2) vil $\mathcal{L}(T)$ være gitt som summen av en geometrisk rekke:

$$\mathcal{L}(T) = (1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + \cdots)\mathcal{C}(T)$$

Denne summen går mot en grenseverdi når antallet gjennomlesinger vokser:

$$\mathcal{L}(T) = \lim_{n\to\infty} \sum_{i=0}^{n} (\frac{2}{3})^i \mathcal{C}(T) = 3\mathcal{C}(T) \tag{9.3}$$

Dersom (9.2) holder, vil den totale informasjonsmengden som absorberes under arbeidet med å forstå ei løkke som er fri for nestede løkker ikke overstige $3\mathcal{C}(T)$. Vi kan derfor bruke $3\mathcal{C}(T)$ som et konservativt mål for informasjonsmengden vi må absorbere for å forstå ei slik løkke.

Hvor vellykket denne modifikasjonen vil være, avhenger sterkt av hvor stor andel løkkekroppene utgjør av kildekoden. Dersom det meste av kildekoden er løkkekropper, vil det modifiserte målet være lineært avhengig av det originale målet, og lide av de samme svakhetene.

9.2.3 Implementasjon av det modifiserte målet

Vi kan lett modifisere den eksisterende implementasjonen av kompleksitetsmålet til å straffe løkker som beskrevet i 9.2.2 dersom vi antar at programmer ikke bruker `goto` for å lage løkker, men istedet bruker språkets løkkekonstruksjoner.

- `do`
- `while`
- `for`

En løkkekropp fremtrer da alltid som en ikke-terminal L i G++ grammatikkens produksjoner for løkkekonstruksjonene. Aksjonene tilknyttet disse produksjonene modifiseres til å multiplisere kompositt kompleksitet for L med tre (både relativ og absolutt komponent multipliseres).

Merk at dette vil straffe nestede løkker med en faktor som vokser eksponentielt med dybden av nesting l. Løkkekroppen T i den innerste løkka inngår i l løkkekropper, og vil derfor bidra med kompleksitet

$$3^l \mathcal{C}(T)$$

Hvis man erstatter løkkekroppen T med et kall på en prosedyre med prosedyrekropp T, vil kompleksiteten for løkkene avta dramatisk. Dette synes vi er i samsvar med vår intuitive oppfatning av kompleksitet.

9.3 Støtte fra andre eksperimenter

Forskningen omkring kompleksitetsmål er forbløffende fattig på eksperimenter der målene forsøkes verifisert i praksis. Dette har mange årsaker, blant annet:

- Programkoden som benyttes i forsøkene må være av en viss størrelse. Evaluering av feilrate etc hos programmerere sammenlignet med estimert kompleksitet vil derfor være ressurskrevende.
- Enkelte kompleksitetsmål er svært løst definert og derfor vanskelige å falsifisere.
- Få kompleksitetsmål kan beregnes maskinelt — mange mål inneholder elementer av skjønn. Et eksempel er Halsteads V^* for det minimale volumet av en algoritme (se avsnitt 2.2.1). For mål som er basert på skjønn, er det svært ressurskrevende å fremskaffe et stort nok tallmateriale til at man kan gjøre statistisk signifikante observasjoner.
- Variasjonen i effektivitet mellom programmerere er svært stor, og gjør det vanskelig å trekke konklusjoner av eksperimentene.

Noen interessante eksperimenter har imidlertid vært gjort, og jeg vil i det følgende referere til et forsøk som synes å gi en viss empirisk støtte til det implementerte kompleksitetsmålet.

I [24] argumenterer forfatterne for at utstrakt bruk av globale variable gjør en programtekst vanskelig å forstå. De foreslår at globale variabler normalt ikke bør være tilgjengelige inne fra en prosedyre, og begrunner dette med at slik bruk svekker prosedyrens rolle som abstraksjon. Dersom man allikevel velger å programmere slik at man har behov for adgang til variable i omsluttende skop, må dette behovet deklareres eksplisitt — man må "importere" disse variablene.

I [25] beskrives et omfattende forsøk der man ønsket å måle effekten av ymse enkle modifikasjoner på et programmeringsspråk, deriblant begrensning av utsynet til variable i omsluttende navnerom. Dersom konklusjonene i [24] er riktige, burde denne endringen gi gevinst i form av færre programfeil og enklere vedlikehold.

- Språket TOPPS følger den tradisjonelle linjen hvor deklarasjoner i et omsluttende navnerom arves inn i lokale navnerom slik at utsynet til dem bevares. Dermed har prosedyrer skrevet i TOPPS automatisk utsyn til globale variable.
- I språket TOPPSII arves *ikke* deklarasjoner fra omsluttende navnerom automatisk. Hvis man ønsker utsyn til en variabel `a` som er deklarert i et omsluttende navnerom, må dette angis eksplisitt ved importdeklarasjonen `knows a`.

Resultatene av disse forsøkene kan gi oss et bedre bilde av hvor god den bærende idéen i kompleksitetsmålet er: Kompleksiteten øker jo flere variable vi har utsyn til.

TOPPSII setter en effektiv stopper for en sær type programfeil som kan oppstå ved vedlikehold av programmer skrevet i språk der deklarasjoner i omsluttende navnerom arves. Når navn kan arves, tillater man som regel at lokale navn kan skygge for arvede navn, og det er derfor alltid en fare forbundet med å innføre et nytt lokalt navn. Et slikt navn kan komme til å "skygge for" et arvet navn. Dermed kan variabelreferanser som tidligere refererte til en global variabel nå referere til den nye lokale variablen (se avsnitt 5.9.1).

Når vi innfører den regel at navn fra omsluttende navnerom må importeres eksplisitt ved en importdeklarasjon, vil deklarasjonene i blokken omfatte alle variabelnavn som benyttes i blokken. Å finne et unikt navn til en ny lokal variabel blir da enkelt, og kompilatoren kan kontrollere at samme navn ikke deklareres to ganger.

Importdeklarasjoner vil forenkle vedlikeholdet av et program. En vedlikeholdsprogrammerer har ofte behov for å introdusere nye variable til bruk i modifisert kode. Den tradisjonelle arven av variabelnavn vil da stå som en åpen felle som klapper igjen i det øyeblikk vedlikeholdsprogrammereren velger et uheldig navn på sin nye lokale variabel. Han kan uten å være klar over det endre effekten av eksisterende programkode (som før refererte til en bestemt global variabel, men nå refererer til hans nye lokale variabel). Denne typen feil vil være meget vanskelig å lokalisere, da programmereren under feilsøkingen typisk vil arbeide etter hypotesen: Når et program konsekvent endrer oppførsel, må dette skyldes endringer i kildekodens imperativer. I vårt tilfelle holder imidlertid ikke denne hypotesen, da kildekodens imperativer godt kan være uendret. Det er selve deklarasjonen av en ny lokal variabel som gjør at kompilatorens oversettelse av imperativene blir en helt annen.

I [25] konkluderer man med at utførte forsøk gir statistisk signifikante indikasjoner på at bruk av importdeklarasjoner over tid gjør det lettere å finne feil i programmer. Vi tar derfor forsøkene til inntekt for at kompleksitetsmålets hovedidé er riktig.

9.4 Organisering av kildekode

I avsnitt 9.3 så vi at forbedringer av programmeringsspråk kan bidra til at kildekode blir lettere å forstå og vedlikeholde, og det dukker til stadighet opp nye programmeringsspråk som prøver å rette opp svake sider ved eksisterende språk. Basert på egne erfaringer i forbindelse med modifikasjonene på G++ kompilatoren, tror jeg imidlertid man kan lette vedlikeholdsarbeidet betraktelig uten å gjøre endringer på selve programmeringsspråket. Dette synes interessant, da endringer i programmeringsspråk fort kan vise seg dyrkjøpte

i nærvær av store mengder eksisterende kildekode.

Når man prøver å forstå en programtekst, har man hele tiden behov for å lokalisere prosedyredefinisjoner og andre deklarasjoner for å se hva referanser til dem egentlig innebærer. Den tradisjonelle løsningen med å legge deklarasjonene i tekstfiler er i denne sammenheng langt fra optimal. Kildefilene er da "flate filer" som egner se dårlig for søking, og under arbeidet med G++ kompilatoren ble søkeprogrammet `grep` et uvurderlig verktøy. Dette ble en opplevelse til ettertanke: Hvorfor ikke la datamaskinen tre støttende til med et mer spesialisert verktøy enn `grep`? Effektivitetsgevinsten ville vært stor.

Vi har sett at et C++ program er en samling eksterne deklarasjoner der programmets effekt er definert som kall på konstruktorer for globale objekter og prosedyren `main`. Hvordan de eksterne deklarasjonene organiseres rent tekstlig har ingen innvirkning på programmets effekt. Det er derfor ikke nødvendig å plassere deklarasjonene i kildefiler på noen bestemt måte, og vi kan like gjerne lagre hver deklarasjon som et objekt i en database. Å kutte opp kildekoden slik innebærer ikke noe tap av semantikk. Hvis denne databasen indekseres på en passende måte, eliminerer vi hele søkeprosessen, og kan slå opp enhver deklarasjon direkte. Når dette integreres med en syntaks-sjekkende editor som editerer hver enkelt definisjon isolert fra de andre definisjonene, blir det vesentlig enklere å følge tråder gjennom programkoden: Ved å peke på en identifikatorforekomst får man innsyn i definisjonen til den entitet identifikatoren refererer til. Slik inspeksjon kan nestes, og programvaren holder rede på hvor man er. Da unngår man å "miste tråden", noe som lett skjer når man prøver å følge programkode ved å simulere datamaskin. Verktøy av denne typen finnes i dag i form av såkalte "browsere".

En browser kan presentere kildekoden i mange forskjellige perspektiver avhengig av hva man ønsker klarhet i [26]. Et perspektiv kan for eksempel være kalltreet til en bestemt prosedyre, oversikt over et klassehierarki etc. Browseren introduserer også "knagger" som man kan henge intern dokumentasjon og kommentarer på. Slik dokumentasjon er da ikke lenger innskrenket til ren tekst, men kan også inneholde mer visuell dokumentasjon i form av figurer.

Datastrukturene i browserens database tillater at kildekoden kan struktureres utover det som er mulig ved hjelp av programmeringsspråkets konstruksjoner. Klasser som hører sammen kan for eksempel samles i en gruppe av klasser.

9.5 Betraktninger om valget av C++

I kapittel 8 så vi at kompleksitetsmålet er sensitivt for hvordan kildekoden bruker headerfiler. Ved uheldig bruk av headerfiler blir innholdet av kompilatorens deklarasjonstabeller en dårlig tilnærming til innholdet av program-

mererens "mentale database". Da kan vi like gjerne bruke antall symboler i programteksten som et kompleksitetsmål.

Det kan tenkes at målet passer bedre for programmeringsspråk som har et kraftigere modulbegrep enn C++. Kildefilene er da typisk mer selektive i sin import av deklarasjoner fra andre moduler slik at deklarasjonstabellene i kompilatoren bedre reflekterer det antallet deklarasjoner vi må ha klart for oss for å forstå prosedyrene i den importerende kildefilen.

9.5.1 Effekten av preprosessering

Preprosessoren (se avsnitt 5.17) gjør at kompilatoren og programmereren leser forskjellig kildekode. Vi har forsøkt å kompensere for dette (se avsnitt 7.5.2), men spesielt i forbindelse med bruk av makroer vil målingene lett gi et feilaktig bilde av den informasjonsmengden vedlikeholdsprogrammerere må absorbere. Nora og Meta bruker ikke makroer i særlig grad. G++ kompilatorens kildekode gjør derimot utstrakt bruk av makroer for å manipulere viktige datastrukturer. Målinger på G++ ville gitt kunstig høye verdier fordi makroene ekspanderes før kildekoden måles. Hvis vi hadde valgt et språk som ikke gjør bruk av en integrert preprosessor, hadde programmerer og kompilator lest samme kildekode, og målingene hadde vært bedre i tråd med den faktiske belastningen på programmereren.

9.6 Videre arbeid

Denne oppgaven har fokusert på hvordan målet kan *implementeres*, og eksperimentene med målet i kapittel 8 var ikke særlig omfattende. Den modifiserte kompilatoren burde danne et godt utgangspunkt for målinger på mange programsystemer slik at vi kan danne oss et mer nyansert bilde av kompleksitetsmålet.

9.6.1 Målinger på flere systemer

Kildekoden vi testet målet på holdt høy standard, og er kanskje ikke representativ for "typisk" kildekode. For eksempel så vi at Meta viste tegn på tilnærming mot et *objektorientert* design, og kanskje ikke egnet seg til testing av formel (3.12) for det optimale antallet prosedyrer i et *prosedyreorientert* program. Det vil være interessant å foreta målinger på et større antall prosedyreorienterte programmer slik at vi kan få et bedre grunnlag for en vurdering av (3.12).

Vi bør også foreta målinger på en del dårlig strukturert kildekode slik at vi kan vurdere hvor godt målet straffer dårlig struktur. Kildekode skrevet av ferske informatikkstudenter kunne tjene som kildemateriale.

9.6.2 Videre vurdering av målet

Å verifisere om et kompleksitetsmål gir et godt bilde av kildekodens vedlikeholdbarhet er svært ressurskrevende. Maus utledet i [1] hvordan optimalt strukturert kildekode bør se ut dersom målet stemmer godt overens med intuitiv kompleksitet. Vi har sammenlignet disse spådommene med kildekode vi *tror* er tilnærmet optimalt strukturert. Det er interessant å gjøre målinger som kan vise oss sammenhengen med intuitiv kompleksitet mer direkte: Som testmateriale kan vi bruke kildekode som har vært vedlikeholdt over lengre tid under kontroll av et versjonskontrollsystem. Ved hjelp av versjonskontrollsystemet kan vi få oversikt over hvilke prosedyrer som har vært gjenstand for flest modifikasjoner. Hvis vi antar at disse prosedyrene er de mest komplekse, kan vi kontrollere om *de originale versjonene* av disse prosedyrene pekes ut som spesielt komplekse av målet. Versjonskontrollsystemet lar oss rekonstruere tidlige versjoner av prosedyrene slik at de kan måles med kompleksitetsmålet.

9.6.3 Eksperimenter med modifisert kompleksitetsmål

Alle de foreslåtte modifikasjonene på kompleksitetsmålet kan realiseres på samme implementasjon. Vi kan modifisere implementasjonen i G++ kompilatoren, og undersøke om vi kan observere noen positiv effekt på målingene. Det synes spesielt interessant å undersøke om vekting av løkkekropper tjener noen hensikt for "typisk" kildekode.

9.7 Konklusjon

Vi har sett at målet lar seg implementere relativt enkelt ved å modifisere en kompilator. Målingene på Nora og Meta i kapittel 8 viste oss at Maus' kompleksitetsmål $\mathcal{C}(T)$ for et utsnitt T av et program er en tilnærmet lineær funksjon av størrelsen til T regnet i antall symboler. Proporsjonalitetskonstanten k bestemmes i stor grad av antall globale deklarasjoner, og k vil derfor øke med den totale programstørrelsen.

Den observerte proporsjonaliteten er trolig knyttet til bruken av headerfiler, og kan falle mer eller mindre bort ved målinger på andre programsystemer der headerfiler brukes annerledes. Målinger på flere programsystemer vil gi oss et bedre grunnlag for en vurdering av målet.

Referanser

[1] Arne Maus. *Entropy as a Complexity Measure, and the Optimal Module Size of Object Oriented Programs.* Proc. IFIP 12th World Computer Congress - Madrid 1992, Vol I, Ed: Jan van Leeuwen. North Holland 1992.

[2] Thomas J. McCabe. *A Complexity Measure.* IEEE Transactions on Software Engineering, Vol. SE-2, No 4, December 1976, pp.308 - 320

[3] Richard M. Stallman. *The GNU Manifesto.* Hentet via InterNet, datert 1985

[4] V. Cote, P. Bourque, S. Oligny, N. Rivard. *Software Metrics: An Overview of Recent Results.* The Journal of Systems and Software 1988, pp.121-131

[5] W. Harrison, K. Magel, R. Kluczny, A. DeKock. *Applying Software Complexity Metrics to Program Maintenance.* Computer, September 1982, pp.65-79

[6] B. Curtis. *In Search of Software Complexity.* Workshop on Quantitative Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art. IEEE, New York 1979, pp.95-106

[7] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York 1977

[8] M.H. Halstead. *Natural Laws Controlling Algorithm Structure?* SIGPLAN Notices, Februar 1972, pp.19-26

[9] E.W. Dijkstra. *Go To Statement Considered Harmful.* Communications of the ACM, Mars 1968, pp.147-148

[10] H.F. Ledgard, M. Marcotty. *A Genealogy of Control Structures.* Communications of the ACM, November 1975, pp.629-639

[11] W.J. Hansen. *Measurement of program Complexity by the Pair (Cyclomatic Number, operator Count).* SIGPLAN Notices, Mars 1978, pp.29-33

[12] D.L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the ACM, December 1972, pp.1053-1058

[13] D. Kafura, S. Henry. *Software Quality Metrics Based on Interconnectivity.* The Journal of Systems and Software 1981, pp.121-131

[14] P.G. Hamer, G.D. Frewin. *M.H. Halstead's Software Science - a Critical Examination.* Proc. 6th International Conference on Software Engineering 1982, pp.197-206

[15] A.M. Lister. *Software Science - The Emperor's New Clothes?* The Australian Computer Journal, Mai 1982, pp.66-70

[16] R. Brooks. *Towards a theory of the comprehension of computer programs.* International Journal of Man-Machine Studies 1983, pp.543-554

[17] S. Letovsky. *Cognitive Processes in Program Comprehension.* The Journal of Systems and Software 1987, pp.325-339

[18] R.V.L. Hartley. *Transmission of Information.* Bell System Technical Journal, Vol. 7, 1928, pp.535-63

[19] Norman Abramson. *Information Theory and Coding.* McGraw-Hill 1963

[20] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers. Priciples, Techniques and Tools.* Addison-Wesley 1986.

[21] Charles Donnelly, Richard Stallman. *BISON - the YACC Compatible Parser Generator V1.12.* Free Software Foundation, December 1990

[22] Bjarne Stroustrup. *The C++ Programming Language, Second Edition.* Addison-Wesley 1991.

[23] J.S. Davis, R.J. LeBlanc. *A Study of the Applicability of Complexity Measures.* IEEE Transactions on Software Engineering, September 1988, pp.1366-1372

[24] W. Wulf, M. Shaw. *Global Variable Considered Harmful.* SIGPLAN Notices, Feb. 1973, pp.28-34

[25] J.D. Gannon, J.J. Horning. *Language Design for programming Reliability.* IEEE Transactions on Software Engineering Vol. SE-1, No. 2, June 1975, pp.179-191

[26] M. Adar, R. Field, W. Kubalski, T. Chou. *The Objectworks Browsing Model - Understanding Complex Software.* ParcPlace Systems Inc. Sunnyvale, California 1993.

Vedlegg A

Notasjon

Vi gir her en kort oversikt over notasjonene som blir brukt i denne oppgaven og hvor hver enkelt notasjon defineres.

Notasjon	Betydning	Def.
$LOC(T)$	Antall linjer i kildekode T	2.1
$\mathcal{K}(T)$	Symbolkategori for symbolet T	3.6
$\mathcal{S}(K,T)$	Kardinalitet for symbolkategori K over segmentet T	5.2
$\mathcal{R}(K,T)$	Forekomster fra symbolkategori K i segmentet T	3.8
$\mathcal{C}(T)$	Maus' kompleksitetsmål for kildekode T	3.10
$\mathcal{H}(T)$	Entropi for kildekode T	3.11
$\mathcal{C}_a(T)$	Absolutt komponent av $\mathcal{C}(T)$	6.2
$\mathcal{C}_r(T)$	Relativ komponent av $\mathcal{C}(T)$	6.2
$\mathcal{C}_{navn}(T)$	Kompleksitet for alle navn i kildekoden T	8.1
$\mathcal{H}_{navn}(T)$	Entropi for navn i kildekoden T	8.1

Vedlegg B

Operatorer og reserverte ord i C++

Figurene B.1 og B.2 gir en oversikt over symbolene i symbolkategoriene for reserverte ord og operatorer som brukes i kompilatorens kompleksitetsberegninger. Disse to symbolkategoriene er bestemt av språkets definisjon i [22].

else	long	while	sizeof	extern	case
new	break	goto	this	class	delete
typeof	typedef	for	double	switch	auto
do	friend	volatile	continue	float	const
static	virtual	short	signed	public	struct
if	asm	union	private	operator	default
overload	int	char	return	void	protected
enum	inline	register	unsigned		

Figur B.1: Reserverte ord i C++

<table>
<tr><td>()</td><td>{}</td><td>~</td><td>-</td><td>+</td><td>!</td><td>:</td></tr>
<tr><td>*</td><td>&</td><td>=</td><td>[]</td><td>/</td><td>%</td><td>...</td></tr>
<tr><td>|</td><td>^</td><td>?</td><td>.</td><td>>=</td><td>!=</td><td>,</td></tr>
<tr><td>==</td><td><=</td><td>></td><td><</td><td><<</td><td>>></td><td>;</td></tr>
<tr><td>++</td><td>--</td><td>&&</td><td>||</td><td>-></td><td>::</td><td>+=</td></tr>
<tr><td>-=</td><td>&=</td><td>|=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td></tr>
<tr><td><<=</td><td>>>=</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Figur B.2: Operatorer i C++

Vedlegg C

Kompleksitetsfilens format

Når den modifiserte kompilatoren kompilerer en kildefil `f`, genereres en tekstfil `f.cplx` som inneholder resultatene av kompleksitetsberegningene på `f`. Filen `f.cplx` vil inneholde et antall linjer med data om forskjellige utsnitt av kildefilen `f`.

- En linje for hver prosedyredefinisjon i `f`.
- En linje for hver klassedefinisjon i `f`.
- En linje med data om hele filen `f`.

Hver linje inneholder et antall dataelementer separert med ett mellomrom. Tabellene C.1, C.2 og C.3 viser formatet for en linje for henholdsvis prosedyredefinisjon, klassedefinisjon og hele filen. Formatet for hvert element på linjen angis som i `fprintf()` setningen som skriver linjen til fil. Figur C.1 viser innholdet av kompleksitetsfilen `windisp.cxx.cplx` for kildefilen `windisp.cxx` i programmet Nora.

Legg merke til at vi ikke skriver ut prosedyrenes navn til kompleksitetsfilen. Ved analyse av kompleksitetsfilene for Nora og Meta var ikke prosedyrenavnene av interesse. Ved å utelate prosedyrenavn fra kompleksitetsfilene ble det dessuten enklere å importere data i andre programmer for analyse av målingene. Hver linje inneholder kompleksitet og entropi beregnet for

- Alle symboler.
- Bare for symboler i symbolkategorien *navn*.

Beregningene for symbolkategorien *navn* er nyttige når vi ønsker å se hvordan målet varierer med antallet globale deklarasjoner. Vi ønsker da å holde andre symbolkategorier utenfor (se avsnitt 8.3).

Element nr.	*Parameter*	*Format*
1	$\mathcal{C}(P)$	`%.1f`
2	$\mathcal{H}(P)$	`%.2f`
3	$\mathcal{C}_{navn}(P)$	`%.1f`
4	$\mathcal{H}_{navn}(P)$	`%.2f`
5	$LOC(P)$	`%d`
6	Antall symboler i P	`%d`
7	Antall navn i P	`%d`

Tabell C.1: Format for en linje med data om en prosedyredefinisjon P

Element nr.	*Parameter*	*Format*
1	Klassens navn	`[%s]`
2	$\mathcal{C}(K)$	`%.1f`
3	$\mathcal{H}(K)$	`%.2f`
4	$\mathcal{C}_{navn}(K)$	`%.1f`
5	$\mathcal{H}_{navn}(K)$	`%.2f`
6	$LOC(K)$	`%d`
7	Antall symboler i K	`%d`
8	Antall navn i K	`%d`

Tabell C.2: Format for en linje med data om en klassedefinisjon K

Element nr.	*Parameter*	*Format*
1	Filens navn	`<%s>`
2	$\mathcal{C}(F)$	`%.1f`
3	$\mathcal{H}(F)$	`%.2f`
4	$\mathcal{C}_{navn}(F)$	`%.1f`
5	$\mathcal{H}_{navn}(F)$	`%.2f`
6	$LOC(F)$	`%d`
7	Antall prosedyredefinisjoner i F	`%d`
8	Antall lokale deklarasjoner i F	`%d`
9	Antall synlige globale deklarasjoner i F	`%d`

Tabell C.3: Format for linjen med data om kildefilen F

```
119.3 6.63 69.1 7.68 5 18 9
1195.7 5.89 540.5 6.51 23 203 83
105.6 5.87 47.8 7.97 6 18 6
171.1 5.90 70.7 6.43 5 29 11
1358.4 6.23 619.5 7.46 38 218 83
171.5 5.91 71.1 6.47 5 29 11
1371.6 6.21 621.5 7.40 38 221 84
119.8 6.66 69.6 7.74 5 18 9
3569.6 5.96 1539.5 7.30 76 599 211
119.8 6.66 69.6 7.74 5 18 9
140.9 6.71 79.6 7.96 6 21 10
140.9 6.71 79.6 7.96 6 21 10
140.9 6.71 79.6 7.96 6 21 10
140.9 6.71 79.6 7.96 6 21 10
237.6 6.60 120.5 8.04 11 36 15
188.4 6.50 102.0 7.84 6 29 13
3700.6 6.13 1524.9 7.86 110 604 194
681.5 5.68 324.2 5.79 13 120 56
1212.9 6.38 587.7 7.73 23 190 76
80.5 6.70 47.0 7.83 4 12 6
609.8 6.56 293.7 8.39 15 93 35
328.8 6.58 166.0 8.30 12 50 20
356.8 6.26 139.9 8.23 14 57 17
104.4 6.52 55.7 7.95 5 16 7
264.7 6.62 132.6 8.29 7 40 16
372.4 6.21 161.6 7.70 12 60 21
195.4 6.30 84.2 8.42 10 31 10
250.5 6.59 123.9 8.26 10 38 15
3468.0 6.35 1472.7 8.27 85 546 178
<windisp.cxx> 21046.3 6.19 9382.4 7.58 573 29 70 172
```

Figur C.1: Innholdet av kompleksitetsfilen windisp.cxx.cplx for kildefilen windisp.cxx i Nora. Den siste linjen inneholder målinger på hele filen. Alle de andre linjene i filen inneholder målinger på prosedyrer.

Vedlegg D

Kildekode for modifikasjonene på G++

Modifikasjonene på G++ ble hovedsaklig gjort på

- Leksikalsk analysator `YYLEX()` i filen `cplus-lex.c`.
- Parserens aksjoner i filen `cplus-parse.y`.
- Deklarasjonslageret i filen `cplus-decl.c`.

Alle modifikasjonene er tydelig merket ved at de er omgitt av konstruksjonen

```
#ifdef COMPLEXITY
....
#endif
```

Modifikasjonene effektueres kun dersom makroen `COMPLEXITY` er definert i `Makefile`. Mange av modifikasjonene kan vanskelig løsrives fra sammenhengen, og det vil føre for langt å liste all relevant kildekode her. Vi viser kun de modifiserte delene av parserens oversettelsesskjema. Et oversettelsesskjema for produksjonene

$$V \rightarrow HS1$$

$$V \rightarrow HS2$$

blir angitt på formen

```
V : HS1
  { AKSJON }
  | HS2
  { AKSJON }
  ;
```

På denne måten blir alle produksjoner med samme venstreside listet sammen som en gruppe. Vi viser bare de relevante produksjonene i hver gruppe. Aksjonene i oversettelsesskjemaet ble hovedsaklig modifisert til å assimilere relativ kompleksitet (se avsnitt 7.6) og rekategorisere identifikatorer (se avsnitt 7.5).

Modifikasjonene på G++ ble forsøkt minimalisert ved at alle nye prosedyrer ble lagt i en egen fil `entropy.c` hvis mulig. Filen `complexity.h` inneholder deklarasjoner av prosedyrene i `entropy.c`. De modifiserte delene av G++ gjør bruk av prosedyrene i `entropy.c` ved å inkludere `complexity.h`.

Resultatet av kompleksitetsberegningene på en kildefil `f` skrives til filen `f.cplx`. For å gjøre det enklest mulig å tilpasse denne filens format til eventuelle fremtidige behov, ble alle prosedyrene for utskrift til filen lagt i en egen fil `cplxdump.c`. Dersom formatet i vedlegg C passer dårlig for import i et analyseprogram, kan formatet lett tilpasses ved enkle endringer i `cplxdump.c`.