Hjemmeeksamen 2 i INF3110/4110

Innleveringsfrist: onsdag 19. november kl. 1400

Innlevering

Besvarelsen av oppgave 2,3,4 og 5 skal leveres skriftlig på papir i IFI-ekspedisjonen. Merk denne med navn, kurskode, gruppe og brukernavn ved IFI.

Besvarelsen av oppgave 1 skal sendes per e-post. Hvis du er på gruppe *x*, så sender du e-post til `inf3110-`*x*`@ifi.uio.no`. Programkoden skal sendes som vedlegg og ha navnet *brukernavn*`.sml` hvor *brukernavn* er ditt brukernavn ved IFI.

Fristen er absolutt; det gis ingen utsettelse uten gyldig grunn. Det gis heller ingen mulighet for å ta denne hjemmeeksamen på nytt etter innlevering.

Samarbeid og egenerklæring

Denne oppgaven er ment som en individuell oppgave, siden resultatet av oppgaven inngår som en del av den samlede karakteren for kurset, men det er også tillatt å samarbeide to og to. I så tilfelle skal begge levere hver sin oppgave, men skrive klart og tydelig i både den skriftlige og den elektroniske besvarelsen hvem de har samarbeidet med. Begge vil da få samme uttelling for besvarelsen.

Vurdering

Eksamensform i dette kurset består av to hjemmeeksamener og en avsluttende 3-timers eksamen. Begge hjemmeeksamenene teller 20%; den avsluttende eksamen teller 60%. For å gjøre regningen noe enklere vil du kunne oppnå 20 *poeng* på hver hjemmeeksamen, samt 60 poeng på avsluttende eksamen. Til slutt vil din poengsum bli regnet om til en bokstavkarakter. Manglende innlevering av hjemmeeksamen gir 0 poeng.

Annen informasjon

Filen med programkode skal være *kjørbar* i SML/NJ, dvs. kunne hentes inn ved å bruke `use "`*brukernavn*`.sml";`.

Kommenter programkoden slik at den blir forståelig!

Hvis du ikke klarer alt i en oppgave, så forklar hvordan du har tenkt og hvor langt du har kommet.

Lykke til!

Analyse av programmeringsspråk

Her er en grammatikk for programeringsspråket[1] L2.

⟨*program*⟩ → ⟨*var decl*⟩$^?$ ⟨*proc decl list*⟩$^?$ **begin** ⟨*statement list*⟩ **end**
⟨*var decl*⟩ → **var** ⟨*name list*⟩ **;**
⟨*proc decl list*⟩ → ⟨*proc decl*⟩ ⟦**;** ⟨*proc decl*⟩⟧*
⟨*proc decl*⟩ → **procedure** ⟨*proc name*⟩ **:** ⟨*var decl*⟩$^?$ **begin** ⟨*statement list*⟩ **end**
⟨*name list*⟩ → ⟨*name*⟩ ⟦**,** ⟨*name*⟩⟧*
⟨*statement list*⟩ → ⟨*statement*⟩ ⟦**;** ⟨*statement*⟩⟧*
⟨*statement*⟩ → ⟨*proc call*⟩ | ⟨*assignment*⟩ | ⟨*input*⟩ | ⟨*output*⟩ | ⟨*if-statement*⟩
⟨*proc call*⟩ → **call** ⟨*proc name*⟩
⟨*proc name*⟩ → ⟨*name*⟩
⟨*assignment*⟩ → **assign** ⟨*expression*⟩ **>** ⟨*variable*⟩
⟨*variable*⟩ → ⟨*name*⟩
⟨*expression*⟩ → ⟨*term*⟩ ⟦⟨*ar-op*⟩ ⟨*term*⟩⟧*
⟨*ar-op*⟩ → **+** | **-** | ***** | **/**
⟨*term*⟩ → ⟨*number*⟩ | ⟨*variable*⟩
⟨*input*⟩ → **?** ⟨*variable*⟩
⟨*output*⟩ → **!** ⟨*variable*⟩
⟨*if-statement*⟩ → **if** ⟨*expression*⟩ **then** ⟨*statement list*⟩$^?$ **fi**

Noen kommentarer:

- Dette er et språk som likner på bokens C3; vi har mulighet for rekursjon, men ikke blokkstruktur. I hver prosedyre kan kun lokale variable deklareres; lokale prosedyrer er ikke tillatt. Vi antar at den eneste typen er heltall.
- ⟨*name*⟩ og ⟨*number*⟩ er leksikalske kategorier og kan betraktes som grunnsymboler. Store og små bokstaver regnes som forskjellige.
- L2 kan inneholde kommentarer. De starter med et %-tegn og varer ut linjen.
- Prosedyrer har ikke parametre. Et hovedprogram har dermed deklarasjoner av globale variable, deklarasjoner av prosedyrer og en kropp med setninger (⟨*statement*⟩'s).
- Semantikken til if-setningen er rett og slett at setningene bak then utføres dersom uttrykket etter **if** er ikke-negativt.

[1] Dette er en enklere variant av Logla-0 som har vært benyttet tidligere semestre.

Oppgave 1

I filen `compiler.sml` finner du en ferdig skrevet scanner og parser for dette språket, samt en SIMPLESEM-implementasjon. Det er også skrevet ferdig en kompilator for et *begrenset* fragment av språket. Denne tar en liste av tokens, akkurat som parseren, og genererer SIMPLESEM-instruksjoner som legges inn i et kodelager. Koden er skrevet slik at du skal slippe å tenke på *alle* detaljene underveis. F.eks. er det laget prosedyrer for å legge inn og ta ut uttrykk fra kode- og datalageret i SIMPLESEM.

Begrensingen i kompilatoren er følgende: Den forutsetter at programmene *ikke har* prosedyredeklarasjoner eller prosedyrekall. **Oppgaven går ut å utvide kompilatoren slik at denne begrensningen opphører.** Kompilatoren skal altså utvides slik at den også genererer kode for prosedyredeklarasjoner og prosedyrekall.

Begrensningen i språket svarer til følgende varianter av ⟨*program*⟩ og ⟨*statement*⟩:

⟨*program*⟩ → ⟨*var decl*⟩$^?$ **begin** ⟨*statement list*⟩ **end**
⟨*statement*⟩ → ⟨*assignment*⟩ | ⟨*input*⟩ | ⟨*output*⟩ | ⟨*if-statement*⟩

Utvidelsen kan du gjøre i flere steg:

a) Implementer først prosedyrer *uten lokale variable*. Dette svarer til følgende variant av ⟨*proc decl*⟩:

⟨*proc decl*⟩ → **procedure** ⟨*proc name*⟩ **: begin** ⟨*statement list*⟩ **end**

(Hint 1: `PROGRAM`-funksjonen må endres. Forslag: la `C[2]` inneholde et `JUMP`-uttrykk som gjør at instruksjonspekeren flyttes til begynnelsen på koden som inneholder ⟨*statement list*⟩.)

(Hint 2: `PROCDECLLIST-`, `PROCDECL-` og `PROCCALL`-funksjoner må legges til. Tenk over hva som bør skje *ved kall* på en prosedyre og *ved retur* fra en prosedyre. Sjekk hva som står i boken. . .)

b) Deretter, ta høyde for at prosedyrer *kan ha lokale variable*. Dette svarer til følgende variant av ⟨*proc decl*⟩ (samme som den opprinnelige grammatikken):

⟨*proc decl*⟩ → **procedure** ⟨*proc name*⟩ **:** ⟨*var decl*⟩$^?$ **begin** ⟨*statement list*⟩ **end**

(Hint: Her må prosedyrer for å parse lokale variable legges til. Hvordan bør datalageret organiseres for lokale variable?)

Merk godt hvor i koden du har gjort endringer slik at den blir lett å lese.

Oppgave 2

Er grammatikken LL(1)? Gi en presis begrunnelse for svaret ditt. (Hint: bruk definisjonene; skriv evt. om til klassisk BNF.)

Oppgave 3

a) Hva trenger man av (1) dynamiske lenker og (2) statiske lenker for å implementere (i SIMPLESEM) fragmentet fra **(1a)**, hvor vi har prosedyrer, men hvor lokale variable ikke er tillatt? Forklar hvorvidt disse er *nødvendige* eller ikke for å organisere minnet. Gi en begrunnelse for svaret ditt. (Maks. 200 ord.)

b) Hva trenger man av (1) dynamiske lenker og (2) statiske lenker for å implementere (i SIMPLESEM) hele L2? Forklar hvorvidt disse er *nødvendige* for å organisere minnet. Gi en begrunnelse for svaret ditt. (Maks. 200 ord.)

Oppgave 4

Se på følgende grammatikker. For hver grammatikk skal du avgjøre om den er LL(1) eller ikke. Gi presise begrunnelser for hvorfor eller hvorfor ikke. For de grammatikkene som ikke er LL(1) skal du konstruere LL(1)-grammatikker for det samme språket, samt vise at LL(1)-kravet nå er oppfylt.

a)

$\langle A \rangle \rightarrow \mathbf{a}\ \langle A \rangle\ \langle F \rangle \mid \mathbf{a}\ \langle B \rangle\ \langle F \rangle \mid \mathbf{a}$
$\langle B \rangle \rightarrow \langle B \rangle\ \mathbf{b} \mid \mathbf{c}$
$\langle F \rangle \rightarrow \mathbf{f}$

b)

$\langle S \rangle \rightarrow \langle A \rangle\ \mathbf{a} \mid \mathbf{b}$
$\langle A \rangle \rightarrow \langle S \rangle\ \langle B \rangle$
$\langle B \rangle \rightarrow \mathbf{ab}$

c)

$\langle T \rangle \rightarrow \langle T \rangle \vee \langle F \rangle \mid \langle F \rangle$
$\langle F \rangle \rightarrow \langle F \rangle \wedge \langle S \rangle \mid \langle S \rangle$
$\langle S \rangle \rightarrow \neg\ \langle P \rangle \mid \langle P \rangle$
$\langle P \rangle \rightarrow \mathbf{(}\langle T \rangle\mathbf{)} \mid \mathbf{true} \mid \mathbf{false}$

Oppgave 5

a) Kan LL(1)-grammatikker være flertydige? Gi en forklaring på hvorfor/hvorfor ikke.
b) Forklar hvorfor venstrerekursive grammatikker ikke er LL(1).

Eksempel på bruk av kompilatoren i `compiler.sml`:

For å vise hvordan kompilatoren kan brukes, viser vi kjøringen av et enkelt program. Følgende kode ligger i filen `maks2.l2`;

```
% Reads two numbers and returns the greatest of them.

var a,b;
begin
?a;
?b;
if a - b then !a fi;
if b - a then !b fi
end
```

For å kompilere filen kan vi gjøre følgende:

```
- compileFile "maks2.l2";
Compiling succeeded!
val it = [] : token list
```

For å kjøre den genererte koden kan vi bruke `run()`:

```
- run();
Give a number: 7
Give a number: 3
The result is: 7
Execution done!
val it = () : unit
```

For å se på den genererte koden kan vi bruke `printCode()`:

```
- printCode();
0: Set 0, 2
1: Set 1, 4
2: Set 2, READ
3: Set 3, READ
4: Jumpt 6, D[2] - D[3] < 0
5: Set WRITE, D[2]
6: Jumpt 8, D[3] - D[2] < 0
7: Set WRITE, D[3]
8: Halt
9: Halt
⋮
```

For å se på datalageret kan vi bruke `printData()`:

```
- printData();
D[0]=2  D[1]=4  D[2]=7  D[3]=3
val it = () : unit
```

SLUTT