

Kompilatorkonstruktion

Kjell Post
kpt@mdh.se

Innehåll

1. Kompilatorns beståndsdelar
2. Lexikalisk analys
3. Syntaktisk analys
4. Symboltabeller
5. Semantisk analys
6. Kodoptimering
7. Kodgenerering
8. Referenser

Kompilator \equiv översättare

Käll-
program

```
⋮  
sum = 0; /* init */  
⋮
```



Kompilator



Objekt-
program

```
⋮  
load r1,0  
⋮
```

- Exempel: Pascal, Ada, C \Rightarrow Assembler
- Kompilerad kod \sim 10 ggr snabbare än interpreterad.

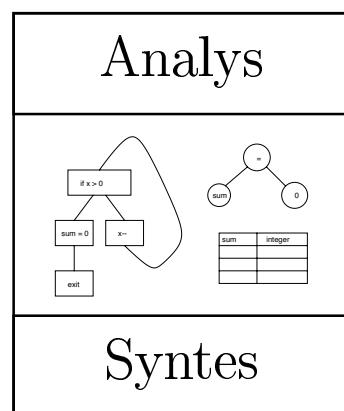
1 Kompilatorns beståndsdelar

Käll-
program

```
⋮  
sum = 0; /* init */  
⋮
```



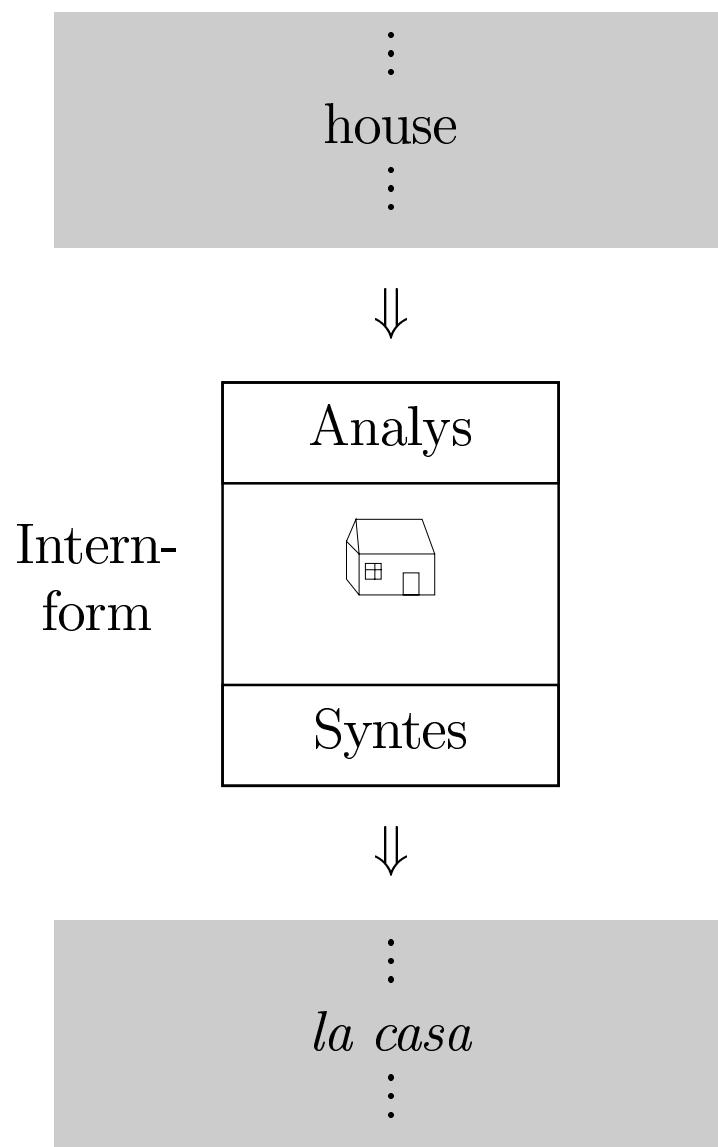
Intern-
form



Objekt-
program

```
⋮  
load r1,0  
⋮
```

Jämförelse med naturligt språk



Kompilatorns beståndsdelar (forts)

- **Analys** (eng. *front-end*)

Identifiera och gruppera symbolerna i källkoden.

Analysera om programmet är syntaktiskt korrekt.

Analysera om programmet är semantiskt korrekt.

Skapa en intern representation som återger programmets struktur och betydelse.

- **Syntes** (eng. *back-end*)

Analysera data- och programflöde.

Utföra optimeringar.

Generera objektkod.

Analysfasen

Tecken:

s u m = 0 ; / * i n i t * / ...
 ↓

Lexikalisk analys

Symboler:

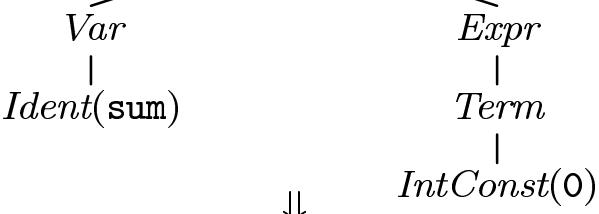
ident(sum) assign intconst(0) semicolon ...
 ↓

Syntaktisk analys

↓

Assignment

Härleddningsträd:



Symboltabell

sum	float
x	int
⋮	

Internform:

sum := 0

Lexikalisk analys

- Grupperar tecken till symboler (eng. *tokens*):
 - Reserverade ord: `for`, `if`, `break`, ...
 - Operatorer: `+`, `+=`, `!=`, ...
 - Separatorer: `;`, `(`, `)`, ...
 - Identifierare: `kalle`, `x9`, `X9`, ...
 - Konstanter: `123`, `-2.1e3`, `"hej"`, ...
 - Kommentarer och “white space”: plockas bort.
- Exempel: hur skrivs symbolen `≠` i olika språk?

FORTRAN: `.NE.`

Pascal: `<>`

PL/I: `/=`

C: `!=`

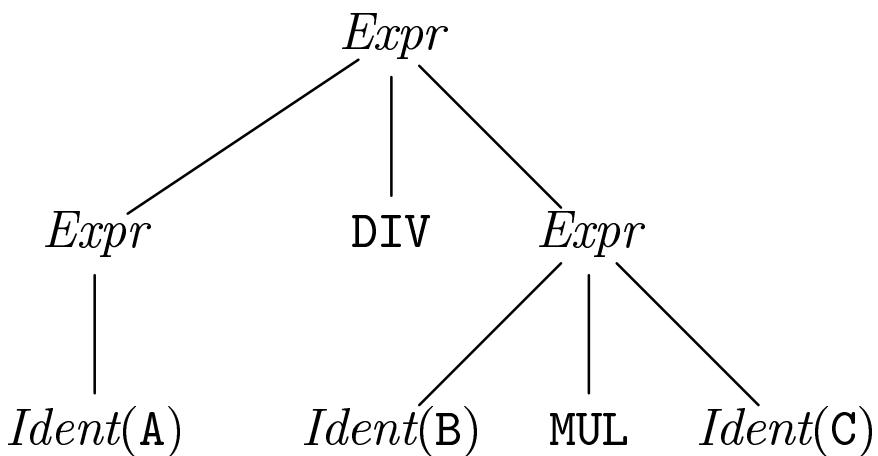
Modula-2: `#`

- Tokens beskrivs med *reguljära uttryck*.

Syntaktiskt analys

- Språket beskrivs med en grammatik.
- Grammatiken beskriver *hur* tokens får grupperas.
- En *parser* undersöker om en ström av tokens följer grammatiken.
- En parser bygger också ett träd som visar programmets struktur.

Exempel: A/B*C

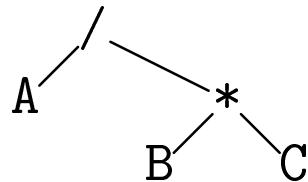


Semantisk analys

- Indata: parseträd och symboltabell.
- Semantisk analys kontrollerar sådant som inte kan beskrivas med en grammatik. Exempel:
 - Typcheckning: $i = 5 + 'x'$
 - Rätt antal och typ på argument till funktioner.
- Utdata: internform, t ex någon av följande:
 - *Stackkod:* A B C * /
 - *Treadresskod:*

T1	\leftarrow	B	*	C
T2	\leftarrow	A	/	T1

- *Abstrakt syntaxträd:*



Vad är syftet med internformen?

- Enklare än högnivåspråket
Färre och primitivare operationer.
- Inte inriktat mot någon speciell processor.
- Lämplig form för optimeringar.

Semantiska regler beskrivs oftast informellt och anropas från parsern när en viss konstruktion upptäckts.

Syntes

- Kodoptimering
- Kodgenerering

Kodoptimering

- Borde egentligen heta *kodförbättring*.
- Utförs på internformen: maskinoberoende.
- Exempel: eliminering av gemensamma deluttryck

Före	Efter
$A \leftarrow B + C * I$	$T \leftarrow C * I$
$D \leftarrow C * I + E$	$A \leftarrow B + T$ $D \leftarrow T + E$

Kodgenerering

- Instruktionsval för målmaskinen.
- Registerallokering och generering av assemblerkod.
- Minneshantering
- Maskinberoende optimeringar.

Exempel: *peephole*-optimering:

Före	Efter
load r0,M[a]	inc M[a]
add r0,1	
store r0,M[a]	

”Dyra” instruktioner ersätts med en billig.

- Schedulering av instruktioner.

Felhantering

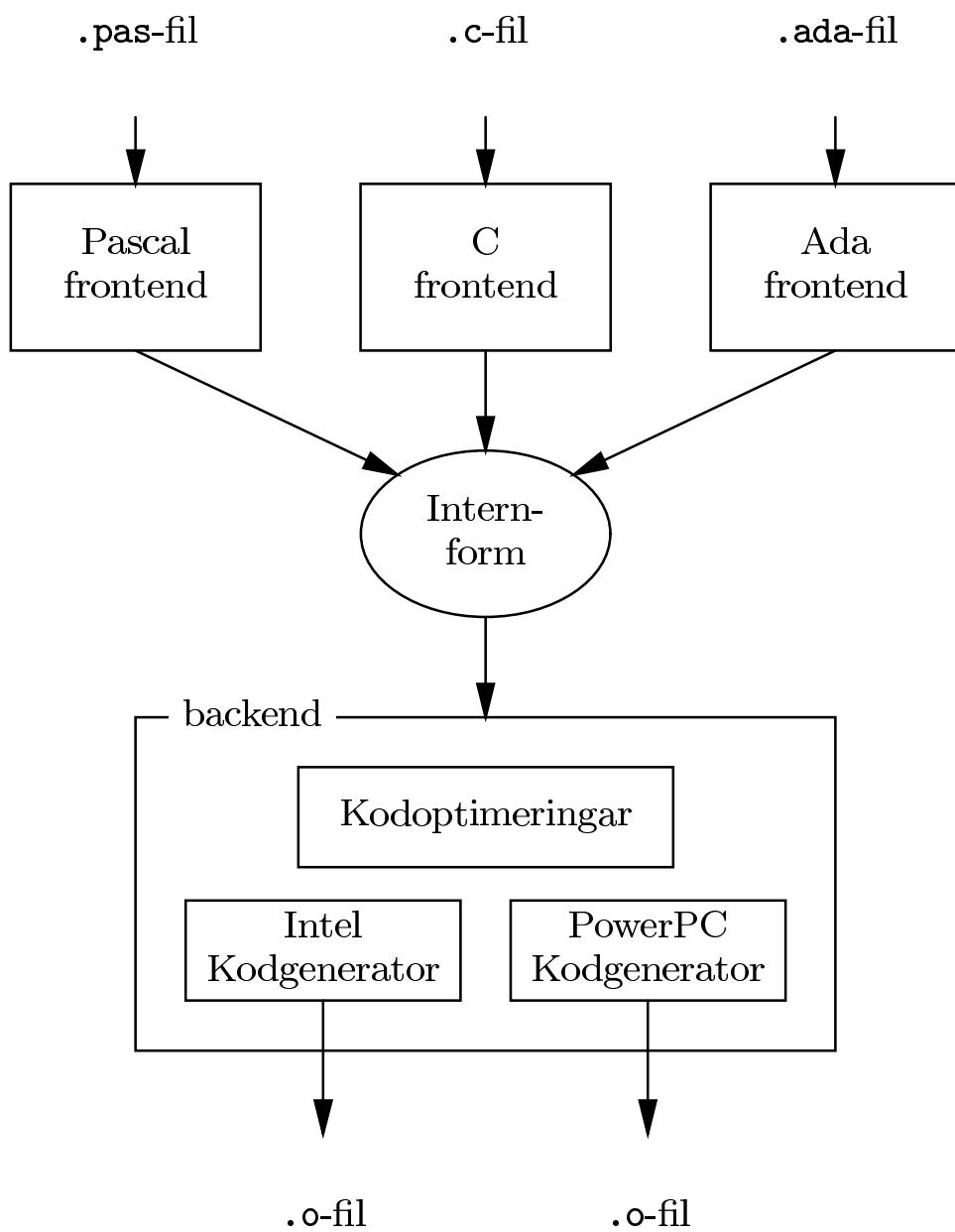
Exempel på fel som kan uppstå i olika faser:

- **Lexikalisk analys:** ö, 5EL, %K, 'hej
- **Syntaxanalys:** x = y + * z;
- **Semantisk analys:** Typkonflikt, t ex 3.14 + 'c'
- **Kodoptimering:** oinitierade variabler
- **Kodgenerering:** för många lokala variabler.
- **Symboltabellen:** dubbeldeklaration.

Hantering

1. Upptäcka felet.
2. Rapportera felet.
3. Korrigera felet (svårt) och fortsätt.

Översikt av en kommersiell kompilator



Ordlista

- **Retargeting:** ändra en kompilator för att generera kod åt en annan målmaskin.
- **Kompilatorgenerator:** en samling verktyg som tar beskrivningar av kompilatorns delar och genererar implementationer av dessa.
- **Pass:** det antal gånger som källprogrammet, i någon form, läses in. Varför flera pass?
 - Framåtreferenser.
 - Begränsat minnesutrymme.

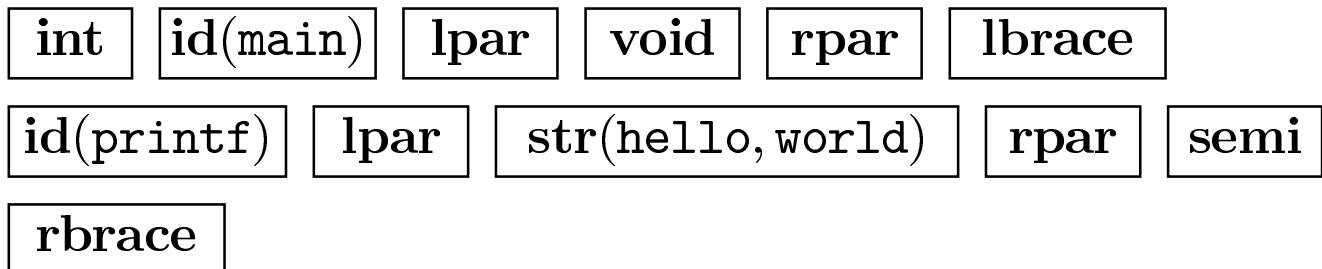
Kvalitetsmått

- Kompileringshastighet.
- Storlek och hastighet för genererad kod.
- Retargeting-egenskaper.

2 Lexikalisk analys

- Grupper tecken till så kallade *tokens*.

```
int main(void) { /* I'm a comment */
    printf("hello, world");
}
```



- Blanktecken och kommentarer kastas bort.
- Annat namn för en lexikalisk analysator: *scanner*.

Vad är en token?

- I ett naturligt språk:
Verb, substantiv, adjektiv, ...
- I ett programmeringsspråk:
Heltal: en icke-tom sekvens av siffror.
Reserverat ord: begin eller end eller ...
Identifierare: bokstav följt av bokstäver/siffror.
- Vissa tokens har också ett värde:
Heltal: dess numeriska värde
Identifierare: namnet
Relation: =, ≤, ≥, >, <, ...
- Tokens skickas vidare till parsern.
- Tokens beskrivs med *reguljära uttryck*.

Misstag från det förflutna

FORTRAN-regel: blanktecken saknar betydelse.

Exempel: VAR1 kan också skrivas VA R1.

Studera följande satser:

- DO 5 I = 1,25
- DO 5 I = 1.25

Den första satsen är DO 5 I = 1 , 25 (en loop).

Den andra satsen är D05I = 1.25 (en tilldelning).

Dvs, man kan inte avgöra om D05I är en variabel eller en DO-sats förrän man sett ”.” eller ”,”.

Detta kräver godtycklig lång *look-ahead*.

Mer misstag...

- C++: använder >> för inmatning.

```
cin >> x;
```

men < ... > används för templates:

```
List<int>
```

och dessa kan nästlas:

```
HashTable<List<int>>
```

- PL/I: saknar reserverade ord.

IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;

Väldigt besvärligt för parsern.

Reguljära uttryck

- Reguljära uttryck är
 - svagast av de formella språken, men ...
 - kompakta, lätt att skriva och förstå;
 - kan implementeras effektivt.

Definition: sträng

- En *sträng* är en ändlig sekvens av tecken.
Visas med detta typsnitt: `begin`, `-100`, ...
- Den *tomma strängen* betecknas ε .
Den tomma strängen innehåller noll tecken.
- För våra ändamål: en token är en sträng.

Definition: reguljärt uttryck

- Ett reguljärt uttryck beskriver en *mängd* strängar.
 1. ε är ett reguljärt uttryck.
 2. Ett godtyckligt tecken är ett reguljärt uttryck.
 3. Om A och B är reguljära uttryck så är följande uttryck också reguljära:

$A \mid B$	betecknar A eller B
$A \cdot B$	betecknar A följt av B
A^*	betecknar noll eller flera A
A^+	betecknar ett eller flera A

- Operatorerna är ordnade efter stigande prioritet.
- Oftast struntar man i ":" vid sammansättning.
- Paranteser kan användas för gruppering.

Några lagar för reguljära uttryck

$$\begin{aligned} A \mid B &= B \mid A \\ (A \ B) \ C &= A \ (B \ C) \\ A \ (B \mid C) &= A \ B \mid A \ C \\ A \cdot \varepsilon &= \varepsilon \cdot A = A \\ A^* &= (A \mid \varepsilon)^* \\ A^{**} &= A^* \\ A \cdot A^* &= A^+ \end{aligned}$$

Exempel

- Siffra: $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- Heltal: $(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+$
- Identifierare:
 $a \mid \dots \mid z \mid A \mid \dots \mid Z (a \mid \dots \mid z \mid A \mid \dots \mid Z \mid 0 \mid \dots \mid 9)^*$
- Relationsoperator:
 $< \mid <= \mid > \mid >= \mid != \mid ==$
- Reserverat ord:
`begin | end | if | then | ...`

Reguljära definitioner

Behändigt att kunna ge namn åt reguljära uttryck:

- Siffra:

$$\text{digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- Heltal:

$$\begin{aligned}\text{sign} &= + \mid - \mid \varepsilon \\ \text{integer} &= \text{sign} \text{ digit}^+\end{aligned}$$

- Identifierare:

$$\begin{aligned}\text{letter} &= a \mid \dots \mid z \mid A \mid \dots \mid Z \\ \text{ident} &= \text{letter} (\text{letter} \mid \text{digit})^*\end{aligned}$$

Övningsuppgift

Vi har beskrivit en identifierare som en bokstav följt av noll eller flera bokstäver eller siffror:

$$\text{ident} = \text{letter} (\text{letter} \mid \text{digit})^*$$

Kan man lika gärna skriva

$$\text{ident} = \text{letter} (\text{letter}^* \mid \text{digit}^*) \quad ?$$

Motivera ditt svar.

Mer exempel på reguljära uttryck

- Email-adresser: t ex kpt@idt.mdh.se

$$\begin{aligned} \text{name} &= \text{letter}^+ \\ \text{address} &= \text{name}'@'\text{name}'.'\text{name}'.'\text{name} \end{aligned}$$

(Riktiga adresser tillåter mer, men är reguljära.)

- Tal i Pascal/C, t ex 10, 3.14 och 1.6E10.

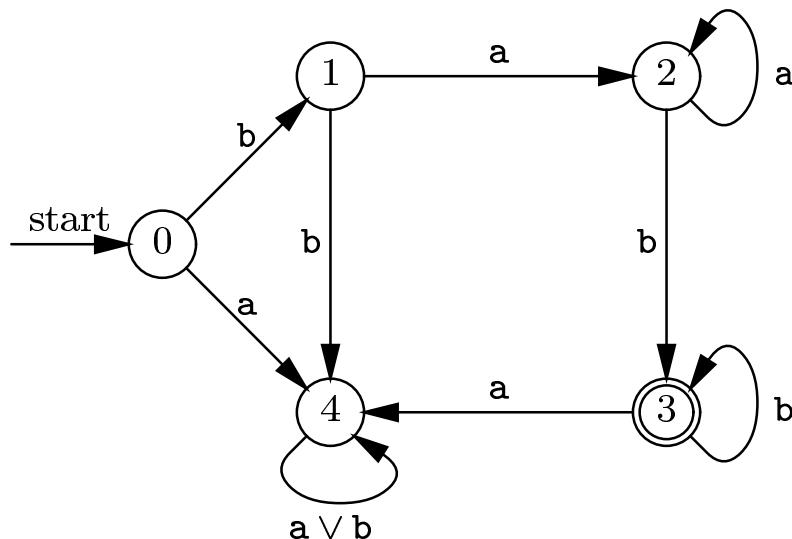
$$\begin{aligned} \text{digit} &= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{digits} &= \text{digit}^+ \\ \text{optFrac} &= '.' \text{ digits} \mid \varepsilon \\ \text{optExp} &= E (+ \mid - \mid \varepsilon) \text{ digits} \mid \varepsilon \\ \text{Num} &= \text{digits optFrac optExp} \end{aligned}$$

- Reguljära uttryck används flitigt i UNIX.

(Fast operatorerna har lite annan betydelse.)

Ändliga automater

- En *ändlig automat* är ett tillståndsdiagram.
 - Antalet tillstånd är ändligt.
 - Bågarna är märkta med tecken.
- Varje reguljärt uttryck har en ändlig automat.
- Automaten känner igen motsvarande r.u.
- Exempel: ändlig automat för ba^+b^+



- Tillstånd 3 är ett *sluttillstånd*.

Hur automaten används

1. Starta i tillstånd 0.
2. Så länge det finns fler tecken att läsa:
 - (a) Läs ett tecken.
 - (b) Följ en båge märkt med tecknet.
3. När är alla tecken är konsumerade:

Om vi är i ett sluttillstånd: *acceptera* strängen.

(Om ingen båge passar tecknet har ett fel uppstått.)

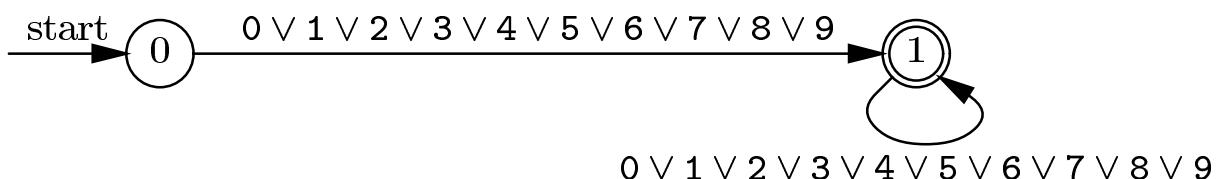
Exempel: Undersök om strängen baab matchar ba^+b^+

<i>Steg</i>	<i>Tillstånd</i>	<i>Tecken kvar</i>
1	0	baab
2	1	aab
3	2	ab
4	2	b
5	3	ε

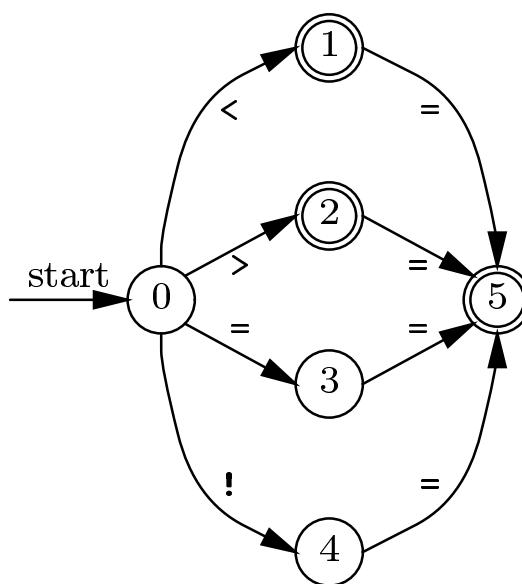
Inga tecken kvar och vi är i tillstånd 3 \Rightarrow acceptera.

Mer exempel på automater

- digits = $(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+$



- rellop = $< \mid <= \mid > \mid >= \mid != \mid ==$



Övningsuppgift

Rita en ändlig automat för ident.

$$\begin{aligned}\text{letter} &= a \mid \dots \mid z \mid A \mid \dots \mid Z \\ \text{ident} &= \text{letter} (\text{letter} \mid \text{digit})^*\end{aligned}$$

Beskriv vad som händer med följande strängar:

- foo123
- 9x
- y

Implementation av automat: övergångstabell

Tillstånd	a	b	accept?
0	4	1	nej
1	2	4	nej
2	2	3	nej
3	4	3	ja
4	4	4	nej

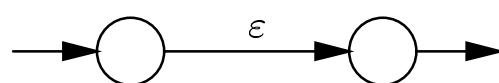
4 är ett "skräptillstånd" dit man går (och aldrig kommer från) om man får något som inte matchar ba^+b^+ .

SIMULATE-DFA

1. $s \leftarrow 0$
2. $\text{read}(c)$
3. **while** $c \neq \text{end-of-file}$
 4. $s \leftarrow \text{table}[s, c]$
 5. $\text{read}(c)$
6. **return** $\text{accept}[s]$

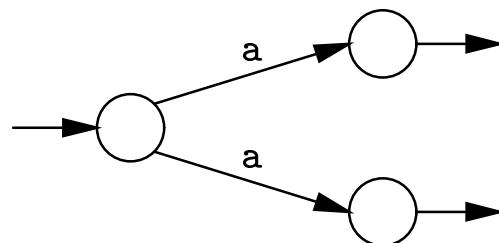
Icke-deterministiska automater

- De automater vi sett har varit *deterministiska*.
- En icke-deterministisk automat kan dessutom ha ε -bågar:



Dvs, automaten kan byta tillstånd utan att konsumera tecken!

- En icke-deterministisk automat kan också ha flera övergångar per tecken och tillstånd.

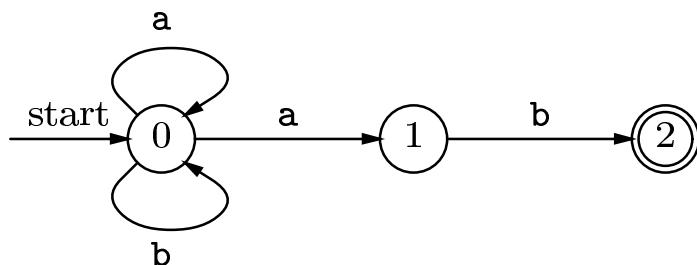


Dvs, automaten måste eventuellt hålla reda på flera tillstånd!

- DFA = Deterministic Finite Automaton.
- NFA = Non-deterministic Finite Automaton.

NFA vs DFA

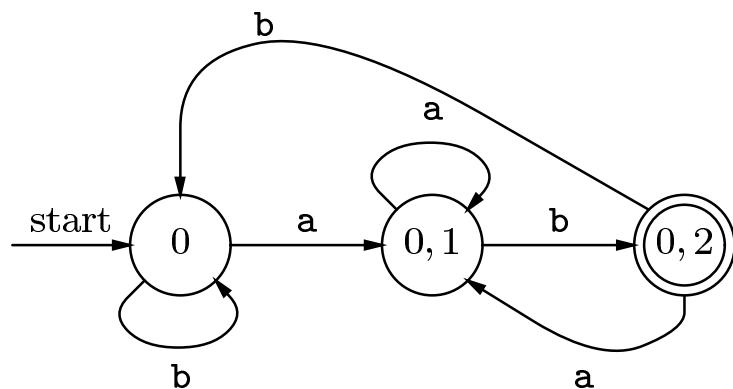
- Varför bry sig om NFA?
Reguljära uttryck visas enklast med NFA.
- En NFA kan alltid transformeras till en DFA.
- Exempel: NFA för $(b \mid a)^* a \ b$.



Exempel: Undersök om strängen baab accepteras.

Steg	Tillstånd	Tecken kvar
1	{0}	baab
2	{0}	aab
3	{0, 1}	ab
4	{0, 1}	b
5	{0, 2}	ε

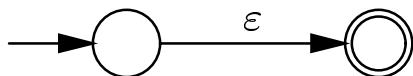
- Motsvarande DFA:



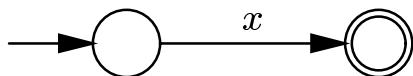
- En NFA kan alltså befina sig i flera tillstånd samtidigt.
- Hur många olika?
- NFA:n befinner sig i någon delmängd av alla tillstånd.
- Hur många delmängder finns det?
- Jo, 2^n , där n är antalet tillstånd.
- Vi återkommer till transformeringen NFA → DFA.

Reguljära uttryck \rightarrow NFA

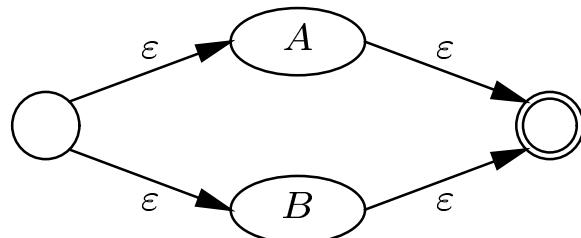
- För ε :



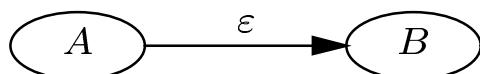
- För ett tecken x :



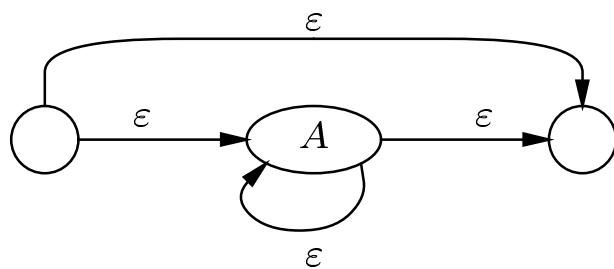
- För $A \mid B$:



- För $A \cdot B$:

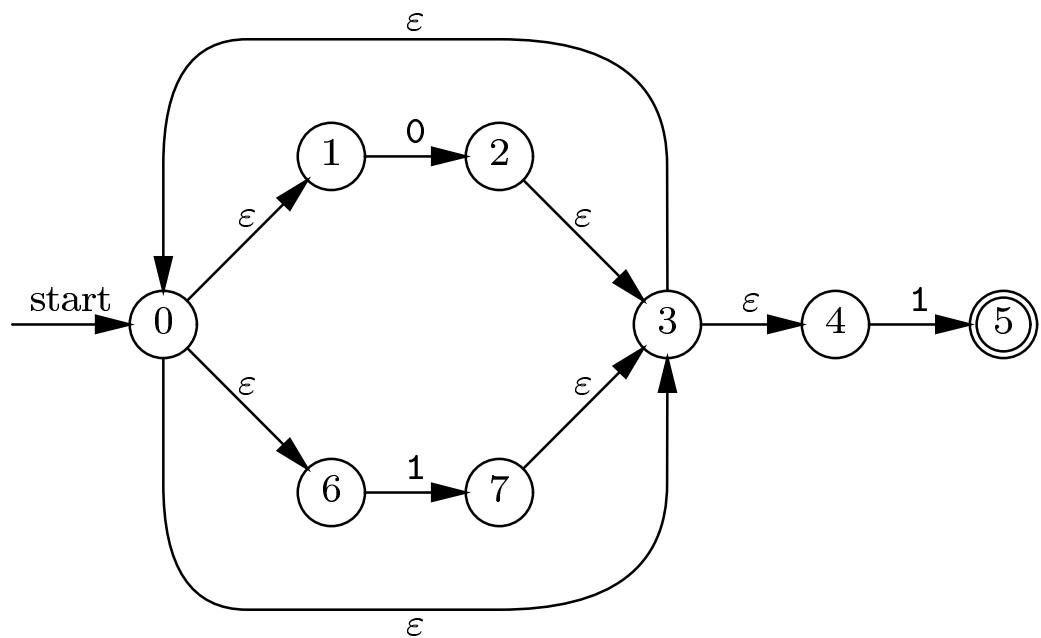


- För A^* :



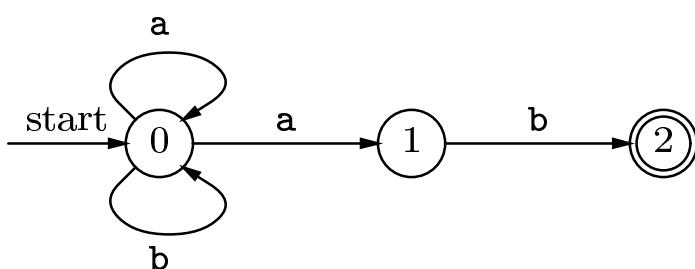
Exempel: $(0 \mid 1)^* 1$

- NFA:



NFA → DFA

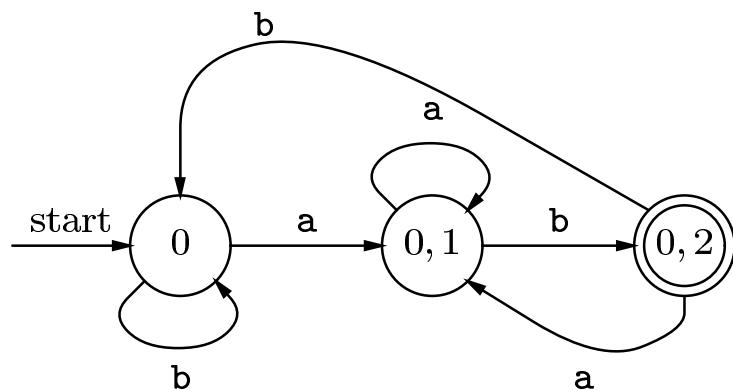
- *Insikt:* Varje tillstånd i DFA:n motsvarar en delmängd av tillstånden i NFA:n.
- Låt oss studera NFA:n för $(b \mid a)^* a \ b$.



- Från starttillståndet 0 kan man inte gå via ε till andra tillstånd, så DFA:n måste börja med delmängden $\{0\}$.
- Från $\{0\}$ kan man via a gå till 0 eller 1, så det måste finnas en båge från $\{0\}$ till $\{0, 1\}$ märkt a .
- Från $\{0, 1\}$ kan man via a bara komma tillbaks.
- Från $\{0\}$ kan man via b bara komma tillbaks.
- Från $\{0, 1\}$ kan man via b komma till $\{0, 2\}$.

- Från $\{0, 2\}$ kan man via a komma till $\{0, 1\}$.
- Från $\{0, 2\}$ kan man via b komma till $\{0\}$.

Detta leder till följande DFA:



I en generell NFA måste man komma ihåg att även utforska ε -bågar som ju kan följas "gratis".

Större exempel

Låt oss implementera en scanner för ett enkelt språk med identifierare, heltalet, relationsoperatorer och tre reserverade ord: `if`, `then` och `else`.

- Reguljära definitioner:

```
blank  =  ' ' | '\t' | '\n'  
letter =  a | ... | z | A | ... | Z  
digit  =  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
num    =  digit+  
ident  =  letter (letter | digit)*
```

- Specifikation av tokens:

- Om blanksteg hittas returneras ingen token, utan scannern returnerar nästa token.
- Vi antar vidare att `if`, `then` och `else` inte får användas som identifierare.

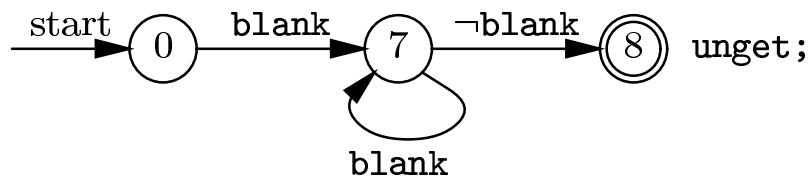
<i>Reguljärt uttryck</i>	<i>Token</i>	<i>Värde</i>
blank ⁺	—	—
if	if	—
then	then	—
else	else	—
ident	id	sträng
num	num	heltalsvärde
<	relop	LT
<=	relop	LE
==	relop	EQ
!=	relop	NE
>	relop	GT
>=	relop	GE

- Om scannern ser <=, ska den returnera LT och lämna kvar =, eller returnera LE?

Maximum-munch-regeln: konsumera längsta möjliga teckenföld och returnera LE.

Automater

- blank^+ :

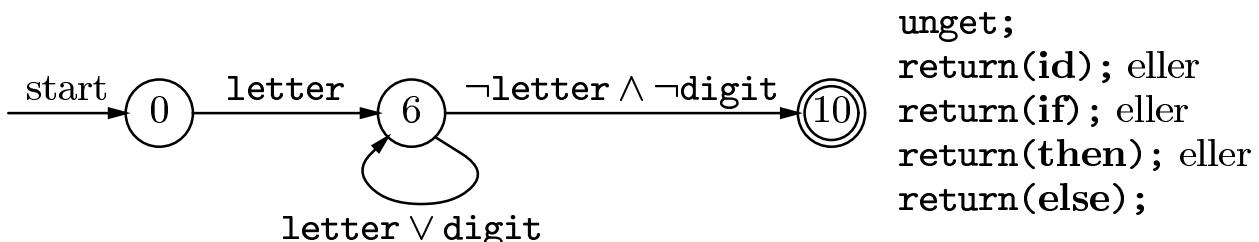


unget gör att senast lästa tecknet lämnas tillbaks.

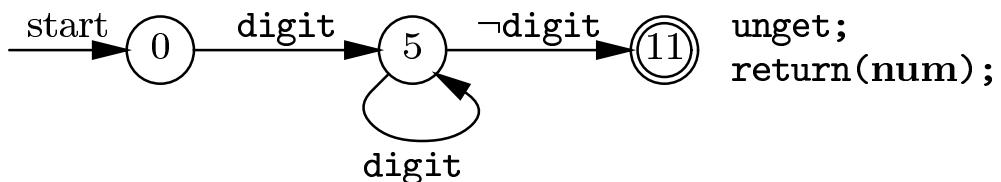
- if, then och else.

Trick: vi behandlar dessa som identifierare och undersöker om någon av dessa har hittats när vi accepterar ident.

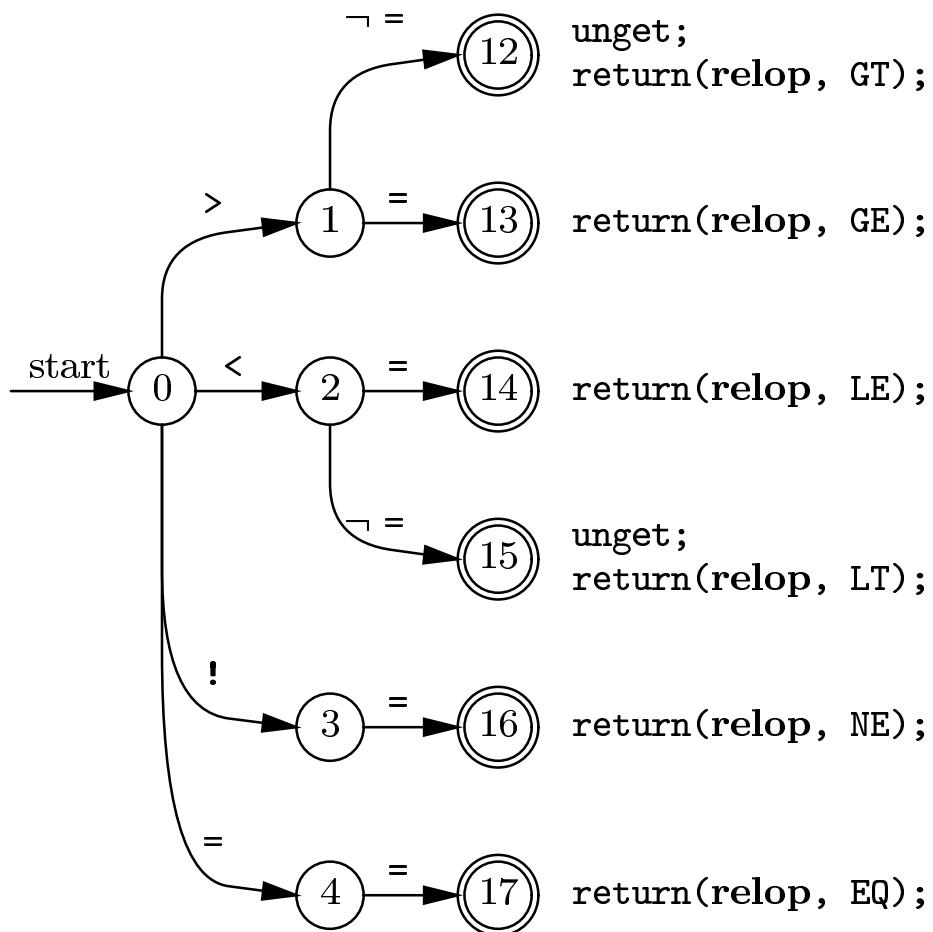
- ident:



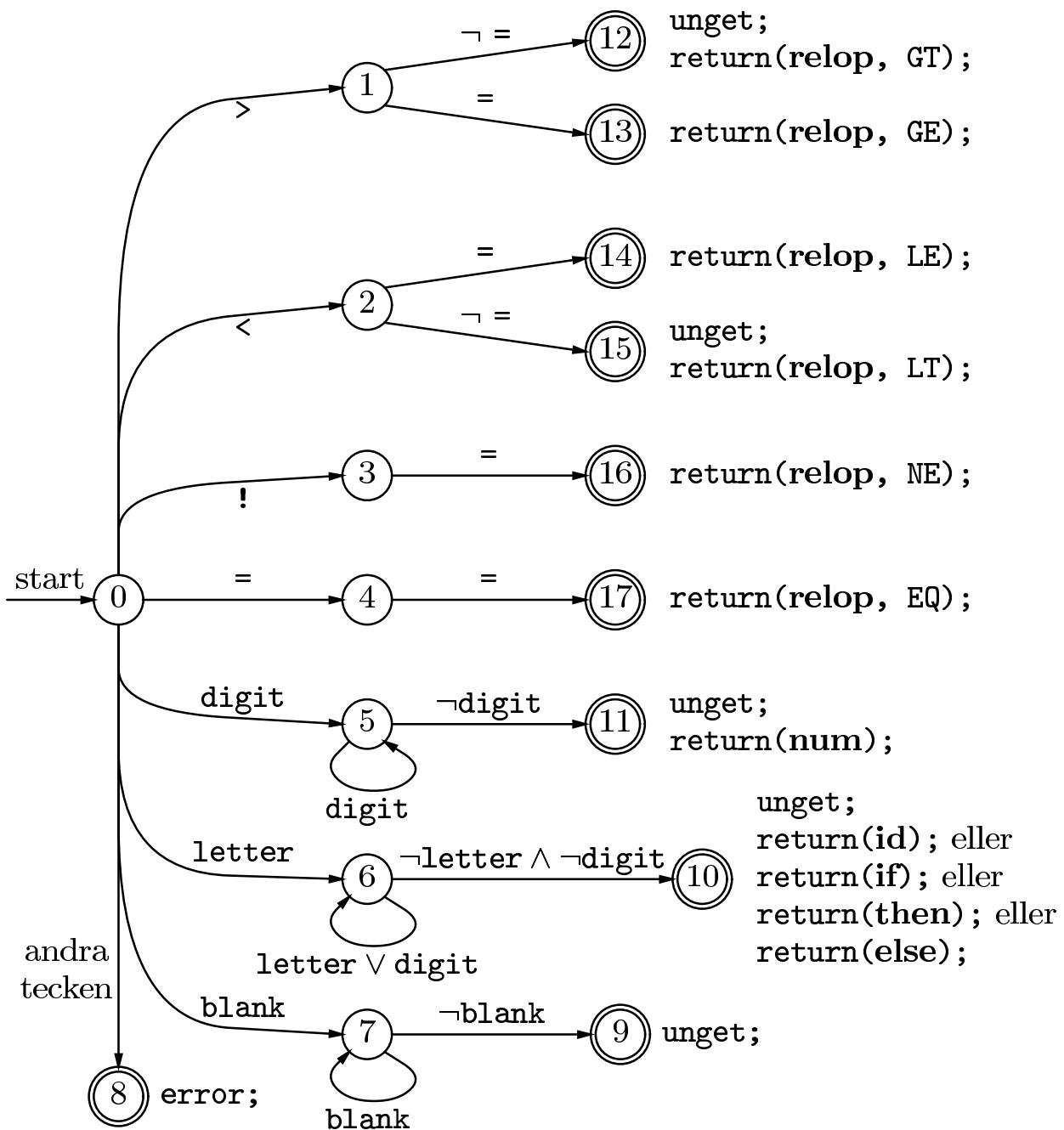
- num:



- Relationsoperatorer:



- Den fullständiga automaten (en DFA!)

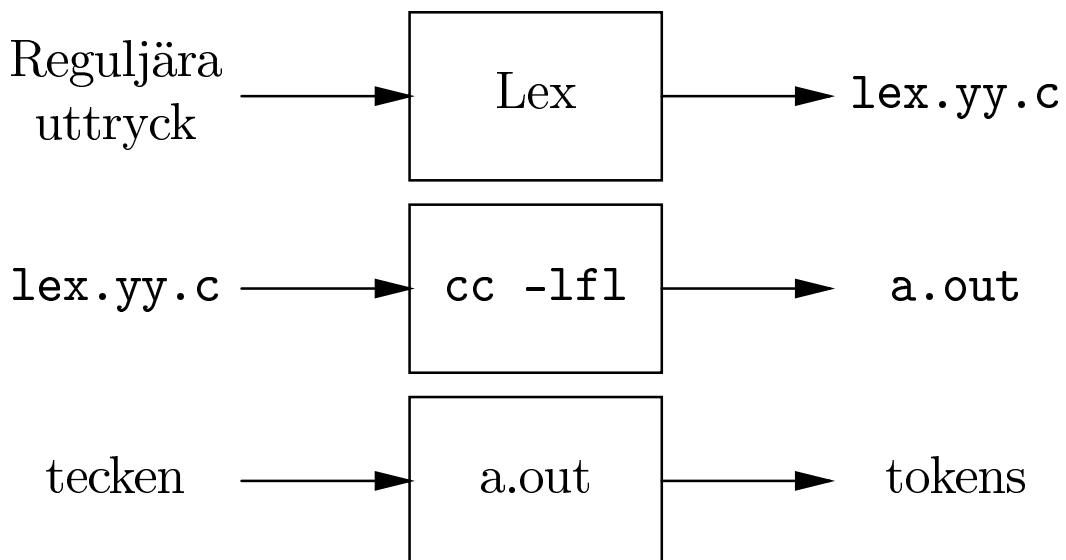


Automater kan också kodas med en switch-sats

```
Token scan() {
    int state = 0;
    for (;;) {
        switch (state) {
            case 0:
                GET(c);
                if (c == '>') state = 1;
                else if (c == '<') state = 2;
                else if (c == '!') state = 3;
                else if (c == '=') state = 4;
                else if (isdigit(c)) state = 5;
                else if (isalpha(c)) state = 6;
                else if (isspace(c)) state = 7;
                else state = 8;
                break;
            case 1:
                GET(c);
                if (c == '=') state = 13; else state = 12;
                break;
            case 2:
                GET(c);
                if (c == '=') state = 14; else state = 15;
                break;
        }
    }
}
```

(Se bilaga för fullständig listning och körexempel.)

Lexikalanalysgeneratorn Lex



Utökad notation för reguljära uttryck

<i>Uttryck</i>	<i>Betyder</i>
[abc]	a b c
[0-9]	0 ... 9
a?	a ε
.	godtyckligt tecken ≠ \n

Lex tillåter mycket mer, se manualsidan.

Indata till Lex

```
%{  
    C-deklarationer  
%}  
    Lex-definitioner  
%%  
    r1      { action1 }  
    r2      { action2 }  
        ...  
    rn      { actionn }  
%%  
    C-funktioner
```

- r_i är reguljära uttryck.
- $action_i$ är C-kod.
- Lex genererar `yylex()` som returnerar nästa token.

Vår scanner, skriven för Lex

```
%{  
    /* C-deklarationer utelämnade här */  
}  
  
blank          [ \t\n]  
letter         [a-zA-Z]  
digit          [0-9]  
num            {digit}+  
ident          {letter}({letter}|{digit})*  
  
%%  
  
{blank}+       { /* no action and no return */ }  
if             { return IF; }  
then           { return THEN; }  
else           { return ELSE; }  
{ident}        { yyval.name = malloc(strlen(yytext) + 1);  
                  strcpy(yyval.name, yytext); return ID; }  
{num}          { yyval.numval = atoi(yytext); return NUM; }  
"<"            { yyval.relop = LT; return RELOP; }  
"<="           { yyval.relop = LE; return RELOP; }  
"=="           { yyval.relop = EQ; return RELOP; }  
"!="           { yyval.relop = NE; return RELOP; }  
">"            { yyval.relop = GT; return RELOP; }  
">>="           { yyval.relop = GE; return RELOP; }  
.              { complain(yytext[0]); }  
  
%%
```

Reguljära uttryck kan inte räkna

Exempel

Försök beskriva strängar på följande form:

$$, (,^n ,) ,^n \quad (n > 0)$$

Dvs, matchande vänster- och högerparantespar.

()

(())

(((()))

OSV

- Går inte att beskriva med reguljära uttryck.
 $, (,^* ,) ,^*$ fungerar inte. Varför?
- Kan dock beskrivas med kontextfri grammatik.

Sammanfattning

- Implementera scannern för hand:
 1. Beskriv tokens med reguljära uttryck.
 2. Rita motsvarande NFA.
 3. Översätt till DFA.
 4. Koda DFA:n som en tabell eller med `switch`.
- Eller använd Lex och eliminera steg 2–4.

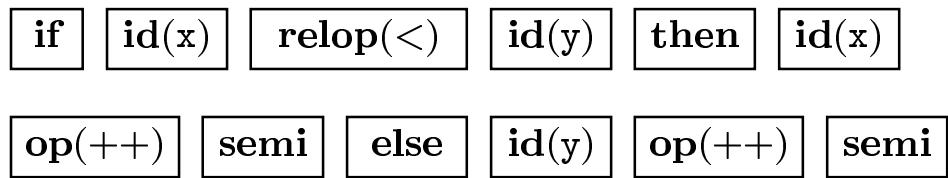
3 Syntaktisk analys

- Indata: en sekvens av tokens från scannern.
- Utdata: ett parse-träd för programmet.
- Exempel:

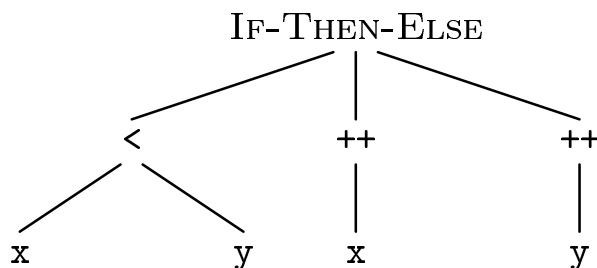
- Källprogram:

```
if x < y then x++; else y++;
```

- Indata till syntaktisk analys:



- Utdata från syntaktisk analys:



Parserns uppgift

- Inte alla symbolsekvenser bildar giltiga program:

```
if then else x ; y > ++ x
```

- Vi behöver
 1. En *notation* för att beskriva giltiga symbolsekvenser.
 - Reguljära uttryck är inte kraftfulla nog.
 - Kontextfri grammatik klarar mycket mer.
 2. En *metod* för att skilja på giltiga och ogiltiga symbolsekvenser.
 - En *parser* kan avgöra detta.

Notation	Igenkännare
Reguljärt uttryck	Ändlig automat
Kontextfri grammatik	Parser

Rekursiva definitioner

Konstruktioner i programmeringsspråk beskrivs ofta *rekursivt*.

- Om E_1 och E_2 är två *aritmetiska uttryck* så är
$$E_1 + E_2 \quad E_1 - E_2 \quad E_1 * E_2 \quad E_1 / E_2$$
också aritmetiska uttryck.

- Om S är en *sats* så är

`while` E do S

`repeat` S `until` E

också satser.

Kontextfri grammatik

Ett naturligt sätt att beskriva rekursiva strukturer.

$$\begin{aligned} E &\rightarrow \text{num} \\ E &\rightarrow \text{ident} \\ E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) \end{aligned}$$

Varje regel $X \rightarrow Y_1 Y_2 \dots Y_n$ kallas för en *produktion*.

Produktioner

En produktion $X \rightarrow Y_1 Y_2 \dots Y_n$ kan tolkas som

” X kan ersättas med $Y_1 Y_2 \dots Y_n$.”

En samling produktioner med samma vänsterled, t ex

$$\begin{aligned} E &\rightarrow \text{num} \\ E &\rightarrow \text{ident} \\ E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) \end{aligned}$$

kan skrivas mer kortfattat som

$$E \rightarrow \text{num} \mid \text{ident} \mid E + E \mid E - E \mid E * E \mid E / E \mid (E)$$

Icke-terminala symboler

Om X har en regel $X \rightarrow \dots$ kallas X *icke-terminal*.

I vårt exempel är endast E icke-terminal.

Varje grammatik har också en speciell *start-symbol*.

Terminala symboler

Om X saknar produktion $X \rightarrow \dots$ kallas X *terminal*.

En terminal är synonymt med token, t ex **num**, +, (.

Mer exempel på kontextfri grammatik

Sentence → *Subject Predicate*
Subject → *Adjective Noun*
Predicate → *Verb*
Adjective → **old | big | strong**
Noun → **harry | billy**
Verb → **jogs | snores | sleeps**

Beskriver meningar som t ex **old billy snores**.

Hur avgöra om en viss mening är syntaktiskt korrekt?

Kontrollera om det finns en *härlädning*.

Härledningar

1. Börja med startsymbolen.
2. Ersätt en icke-terminal m h a en produktion.
3. Upprepa tills bara terminala symboler finns kvar.

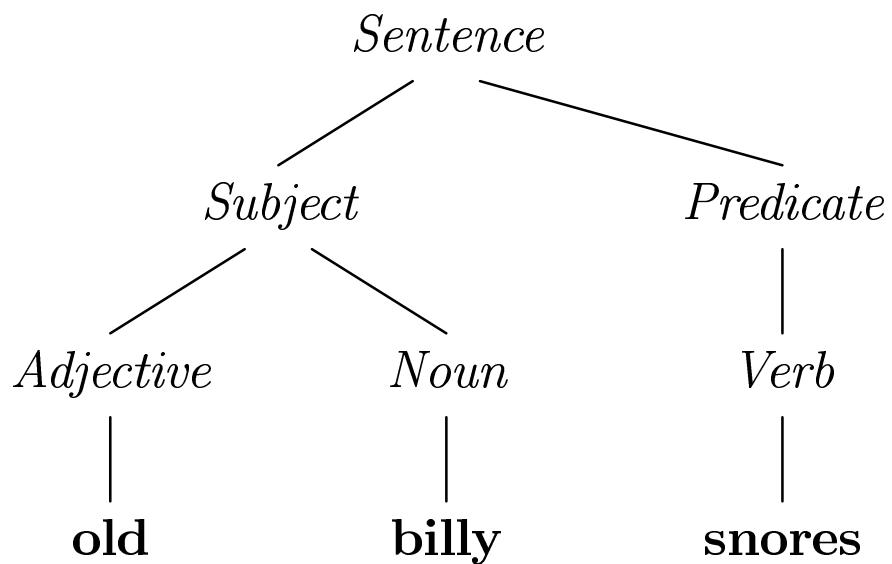
Exempel på härledning

Sentence \Rightarrow *Subject Predicate*
 \Rightarrow *Adjective Noun Predicate*
 \Rightarrow **old** *Noun Predicate*
 \Rightarrow **old billy** *Predicate*
 \Rightarrow **old billy** *Verb*
 \Rightarrow **old billy snores**

Formellt: $\alpha A \beta \Rightarrow \alpha \gamma \beta$ om $A \rightarrow \gamma$ är en produktion.

Parseträd

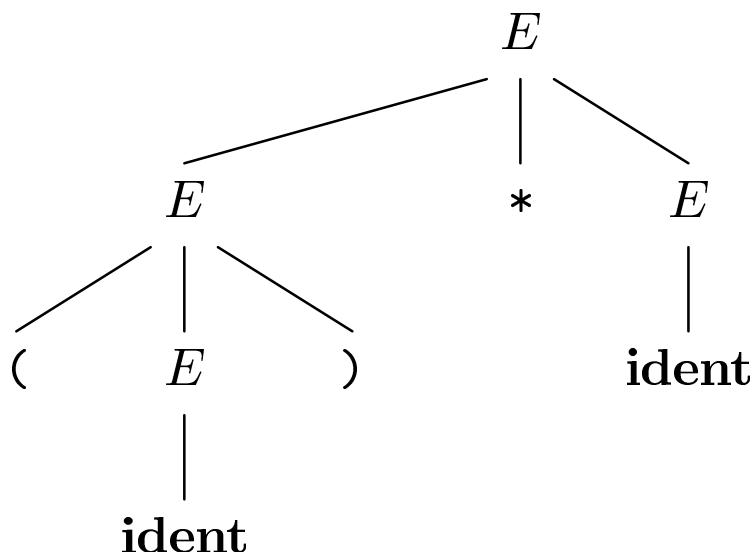
En härledning kan också åskådliggöras med ett *parseträd*.



Mer härledningar

$$E \rightarrow \text{ident} \mid E + E \mid E * E \mid (E)$$

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow (E) * E \\ &\Rightarrow (\text{ident}) * E \\ &\Rightarrow (\text{ident}) * \text{ident} \end{aligned}$$

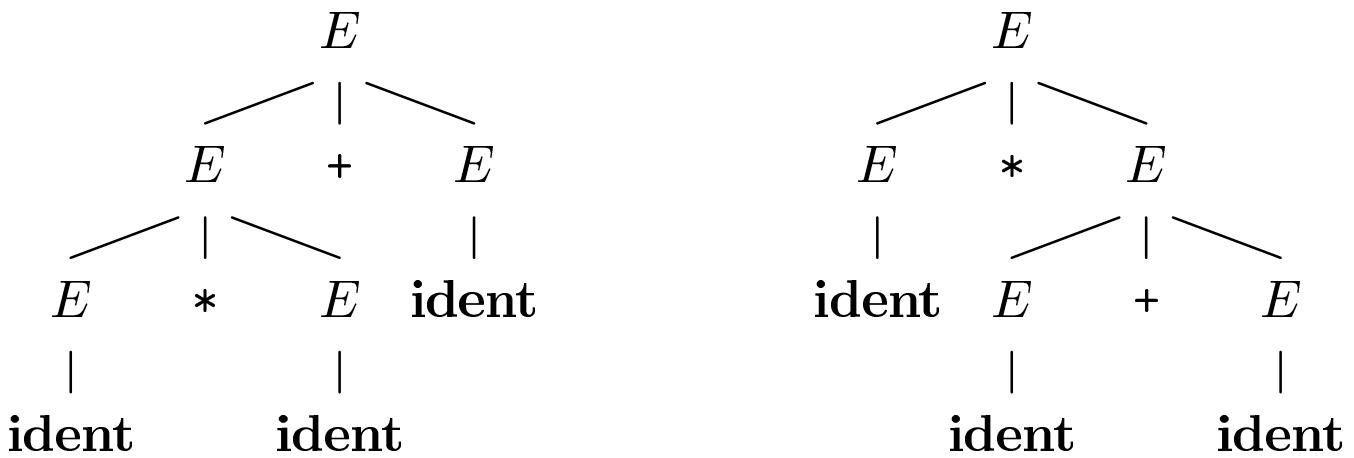


Tvetydig grammatik

En grammatik är *tvetydig* om en mening har flera parseträd. Följande grammatik är tvetydig:

$$E \rightarrow \text{ident} \mid E + E \mid E * E \mid (E)$$

Studera t ex **ident * ident + ident**.



- Tvetydighet \Rightarrow program kan tolkas på > 1 sätt.
- Grammatiken måste skrivas om.

Omskrivning av tvetydig grammatik

- Det finns inga givna regler för att skriva om tvetydigheter.
- I det här fallet vill vi att ***** ska ha prioritet över **+**.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & \text{ident} \mid (E) \end{array}$$

Övningsuppgift

Visa ett parseträd för **ident * ident + ident** med hjälp av den nya grammatiken.

Kan du konstruera mer än ett parseträd?

Varför heter det kontextfri grammatik?

När vi använder en produktion $X \rightarrow \dots$
byter vi ut X utan att bry oss om dess kontext.

Kontextfri grammatik > reguljära uttryck

Följande grammatik beskriver balanserade parantespar:

$$S \rightarrow (S) \mid \varepsilon$$

(ε är fortfarande tomma strängen.)

De meningar som kan härledas från S är

$\varepsilon, \quad (), \quad (((), \quad (((()), \quad \text{etc}$

Parsern

- En grammatik G beskriver en mängd meningar.
- En mening α består av terminala symboler

$$\alpha = t_1 \ t_2 \dots t_n$$

Exempel: $\alpha = \mathbf{ident} + \mathbf{ident} * \mathbf{ident}$

- Parserns uppgift:
 - Givet en grammatik G med startsymbol S ...
 - ... samt en mening α :
 - Kan en härledning för α konstrueras från S ?

$$S \Rightarrow \dots \Rightarrow \alpha$$

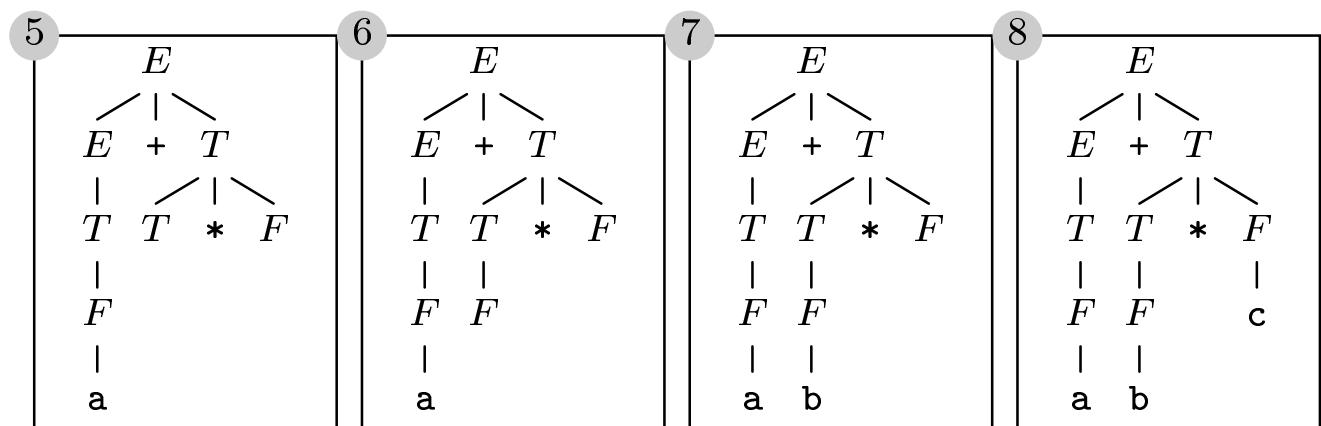
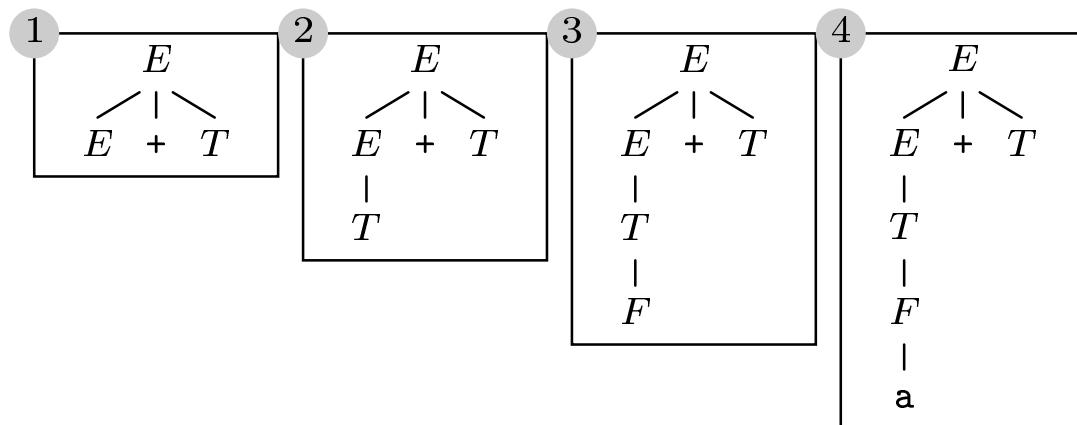
- Vi vill helst också veta härledningens parseträd.

Top-down eller bottom-up?

- Härledning kan utföras *top-down* eller *bottom-up*.
- Top-down är enklare att förstå.
- Bottom-up användas mest i praktiken (t ex Yacc).
- Båda ställer vissa krav på grammatiken.

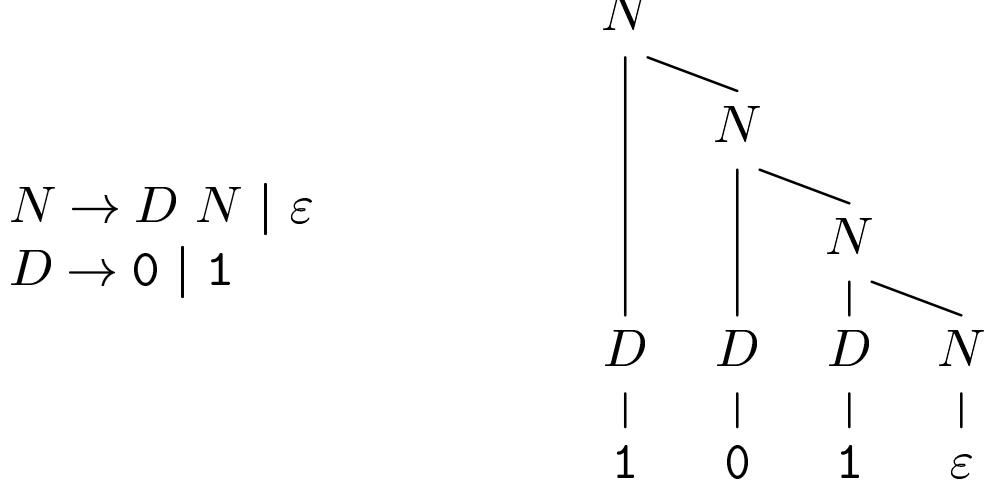
Top-down parsing av a + b * c

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow T * F \mid F \\
 F &\rightarrow \text{ident} \mid (E)
 \end{aligned}$$



Top-down parsing

Följande grammatik beskriver binära tal, t ex 101.



- Parseträdet är initialt N , aktuell token är 1.
- Parseträdet kan sedan expanderas mha någon av

$$\begin{array}{l} N \rightarrow D \ N \\ N \rightarrow \varepsilon \end{array}$$

- Vilken ska väljas? Beror på *nästa* token.

Prediktiv top-down parsing

- Antag att det finns flera produktioner att välja på

$$A \rightarrow \alpha_1 \mid \dots \mid \alpha_n \mid \varepsilon$$

- Antag att nästa token är X .
- Om någon mening som α_i kan härleda *börjar med* X , använd $A \rightarrow \alpha_i$, men ...
... *bara om* ingen annan α_j ($j \neq i$) kan härleda fram en mening som börjar med X .
- Om det finns flera val, ge upp eller backtracka.
- Använd $A \rightarrow \varepsilon$ bara om ingen α_i kan härleda en mening som börjar med X .
- I vårt exempel:

Använd $N \rightarrow D N$ om nästa token är 0 eller 1.
Använd $N \rightarrow \varepsilon$ annars.

Prediktiv top-down parsing

Parsern kan kodas enligt följande principer.

- Skriv en procedur för varje icke-terminal.
- Anropa `scan()` efter att en token konsumerats.
- Anropa startsymbolen från `main()`.
- Expandera sedan nästa symbol i parseträdet:
 - Om det är en terminal symbol:
Jämför med aktuell token, anropa `scan()`.
 - Om det är en icke-terminal symbol:
Anropa proceduren för den symbolen.
- Anropa en felrutin om nästa token inte passar.

```
void D() {
    if (token == '0')          /* D -> 0 */
        scan();
    else if (token == '1')     /* D -> 1 */
        scan();
    else
        error();
}

void N() {
    if (token == '0' || token == '1') { /* N -> D N */
        D();
        N();
    } else ;                      /* N -> */
}

int main(void) {
    scan();                      /* get the next token */
    N();                         /* call start symbol */
    if (token != EOF)
        error();
}
```

Vänsterrekursion

- Produktioner av typen

$$E \rightarrow E + T$$

leder till en procedur som anropar sig själv *utan att konsumera några tokens.*

```
void E() {  
    E();  
    if (token == '+')  
        scan();  
    else  
        error();  
    T();  
}
```

- Parsern hamnar i en oändlig loop.
- Grammatiken måste skrivas om.

Eliminering av vänsterrekursion

Vänsterrekursiva produktioner på formen

$$A \rightarrow A \alpha \mid \beta$$

genererar strängar på formen $\beta\alpha\dots\alpha$ men kan skrivas

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

Exempel

Skriv om $E \rightarrow E + T \mid T$.

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T \mid \varepsilon \end{aligned}$$

Exempel: igenkänning av aritmetiska uttryck

- Ursprunglig grammatik (tvetydig)

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{num}$$

- Omskrivning 1: eliminera tvetydigheter.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \text{num} \mid (E) \end{aligned}$$

- Omskrivning 2: eliminera vänsterrekursion:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T \mid \varepsilon \\ T &\rightarrow F T' \\ T' &\rightarrow * F \mid \varepsilon \\ F &\rightarrow \text{num} \mid (E) \end{aligned}$$

- Implementation med prediktiv top-down parsing.

```
#include <stdio.h>
#include <ctype.h>

int token, tokens[] = { '(', '1', '+', '2', ')', '*', '3', EOF };

void scan() { /* instead of a real scanner */
    static int *tokenp = &tokens[0];
    token = *tokenp++;
}

void error(char *s) {
    fprintf(stderr, "syntax error: %s\n", s);
    exit(1);
}

E() { /* E -> T E' */
    T();
    E1();
}

E1() {
    if (token == '+') { /* E' -> + T */
        scan();
        T();
    } else ; /* E' -> */
}

T() { /* T -> F T' */
    F();
    T1();
}
```

```
T1() {
    if (token == '*') {                  /* T' -> * F */
        scan();
        F();
    } else ;                            /* T' -> */
}

F() {
    if (isdigit(token))             /* F -> Num */
        scan();
    else if (token == '(') {         /* F -> ( E ) */
        scan();
        E();
        if (token == ')')
            scan();
        else
            error("expected )");
    } else
        error("expected number or (");
}

int main(void) {
    scan();                          /* get the next token */
    E();                            /* call the start symbol */
    if (token != EOF)
        error("expected EOF");
    exit(0);
}
```

Vänsterfaktorisering

Betrakta följande beskrivning av if-then-else-satser S .

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S \\ S &\rightarrow \text{if } E \text{ then } S \text{ else } S \end{aligned}$$

Efter att ha sett **if** kan inte parsern välja!

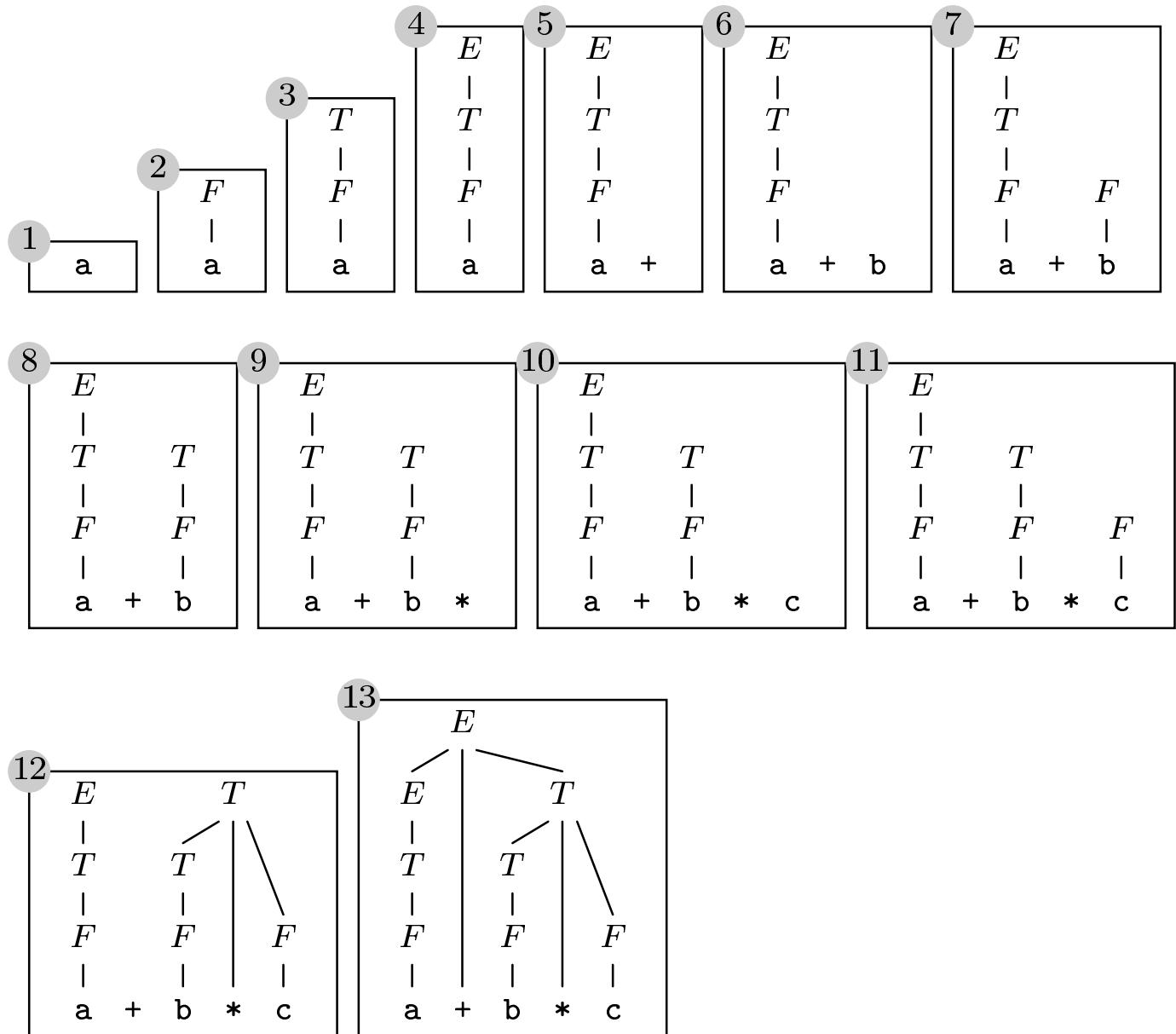
Lösning: skriv om grammatiken

$$\begin{aligned} S &\rightarrow \text{if } E \text{ then } S' \\ S' &\rightarrow \text{else } S \mid \varepsilon \end{aligned}$$

Slutsatser om top-down parsing

- Kan implementeras för hand, men ...
- Ställer vissa krav på grammatiken, t ex ingen vänsterrekursion.
- Kan alltid skrivas om, men brukar ge andra problem.
Associativiteten på t ex ”-” blir fel.
- Bottom-up parsing är kraftfullare, fast teorin är svårare att förstå, men ...
- Yacc gör det enkelt att använda!

Bottom-up parsing av $a + b * c$



Bottom-up parsing

- Nackdelen med top-down parsing:

Måste avgöra vilken produktion som ska väljas efter att ha sett endast en token i högerledet.

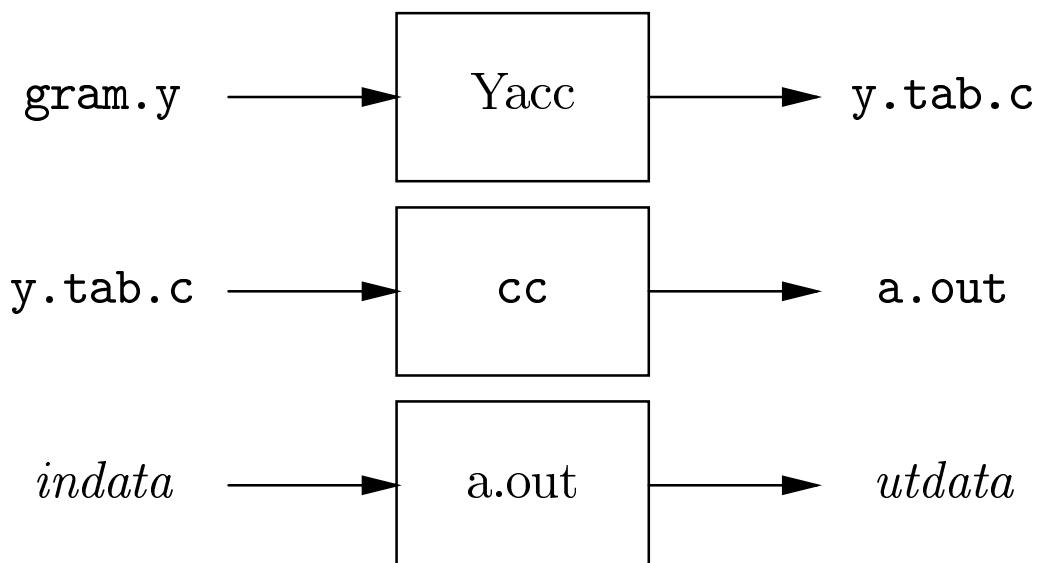
- Fördelen med bottom-up parsing:

Kan vänta med beslutet tills alla symboler i ett av högerleden har setts.

- En bottom-up parser *skiftar* in nya tokens tills den har ett högerled α och kan *reducera* α till A (för någon produktion $A \rightarrow \alpha$) osv.

shift a
reduce $F \rightarrow \mathbf{ident}$
reduce $T \rightarrow F$
reduce $E \rightarrow T$
shift +
⋮

- Hur vet parsern när den ska skifta/reducera?
- Sådan information lagras i en tabell som skapas utifrån grammatiken.
Exakt hur detta går till är överkurs.
- Yacc skapar en parsetabell från en grammatik.



- *gram.y* innehåller en kontextfri grammatik.
- *y.tab.c* anropar `yylex()` för att få nästa token.
- Vanligt att använda Lex och Yacc tillsammans.

Yacc exempel

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \text{digit} \mid (E) \end{aligned}$$

```
%{

#include <ctype.h>
#include <stdio.h>
%}

%token DIGIT

%%

line      : expr '\n' ;
expr      : expr '+' term | term ;
term      : term '*' factor | factor ;
factor    : DIGIT | '(' expr ')' ;
```

```
%%
yylex() {          /* hand coded, could have used Lex */
    int c = getchar();
    return (isdigit(c) ? DIGIT : c);
}

main()           /* yyparse() starts the parser */
{
    (void) yyparse();
}

yyerror(char *s) /* called whenever parsing fails */
{
    fprintf(stderr, "%s\n", s);
}
```

```
$ a.out
1+2*3
$ a.out
1+*3
syntax error
```

Yacc och Lex tillsammans

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow \text{num} \mid (E) \end{aligned}$$

- Filen `lex.l`:

```
%{
#include <stdio.h>
#include "y.tab.h" /* #define NUM 257 */
%}

blank      [ \t]
digit      [0-9]
num        {digit}+

%%

{blank}+    { /* skip blanks */ }
{num}       { return NUM; }
\n|.        { return yytext[0]; }
```

- Filen `gram.y`:

```
%{  
#include <ctype.h>  
#include <stdio.h>  
%}  
  
%token NUM  
  
%%  
line      : expr '\n' ;  
expr      : expr '+' term | term ;  
term      : term '*' factor | factor ;  
factor    : NUM | '(' expr ')' ;  
  
%%  
main()  
{  
    (void) yyparse();  
}  
  
yyerror(char *s)  
{  
    fprintf(stderr, "%s\n", s);  
}
```

- Kompilering (UNIX):

```
yacc -d gram.y      # -d producerar filen y.tab.h  
lex lex.l  
cc y.tab.c lex.yy.c -lfl
```

Sammanfattning

- Top-down parsing är lätt att förstå/implementera.
- Top-down parsing är inte lika kraftfull som bottom-up.

En bottom-up parser har t ex inga problem med vänsterrekursiva produktioner.

- I praktiken används parsergenerator för bottom-up parsers, t ex Yacc.
- Men vi vill ju inte bara känna igen uttryck.
Vi vill också kunna t ex beräkna ett uttrycks värde!
- Nästa del handlar om *syntaxstyrda översättningar*.

Syntaxstyrda översättningar

- Översättningen styrs av den kontextfria grammatiken.
- Vi studerar teorin för bottom-up parsers.
- Symboler har *attribut*:
 - Ett attribut är ett värde (tal, sträng, adress).
 - Terminaler tilldelas attribut av scannern.
 - Icke-terminaler tilldelas attribut av så kallade *semantiska aktioner*.
- En semantisk aktion är kod som tillhör en produktion $X \rightarrow X_1 \ X_2 \ \dots \ X_n$.
Den beräknar (oftast) $f(X) := f(X_1, \dots, X_n)$.
- Aktionen körs när produktionen reduceras.

Beräkning av attribut i scannern

Ny lex.l: sätter yyval när ett heltal påträffas.

```
%{  
#include <stdio.h>  
#include "gram.tab.h"  
%}  
  
blank      [ \t]  
digit      [0-9]  
num        {digit}+  
  
%%  
  
{blank}+ { /* skip blanks */ }  
{num}     { sscanf(yytext, "%d", &yyval); return NUM; }  
\n.       { return yytext[0]; }
```

Beräkning av attribut med semantiska aktioner

Ny gram.y (endast produktionerna visas):

```
%%
line    : expr '\n' { printf("%d\n", $1); }
;

expr   : expr '+' term { $$ = $1 + $3; }
| term { $$ = $1; }
;

term   : term '*' factor { $$ = $1 * $3; }
| factor { $$ = $1; }
;

factor : NUM { $$ = $1; }
| '(' expr ')' { $$ = $2; }
;
```

Koden { ... } körs när motsv. produktion reduceras.

Beräkning av attribut

I en produktion $X \rightarrow X_1 \ X_2 \ \dots \ X_n$.

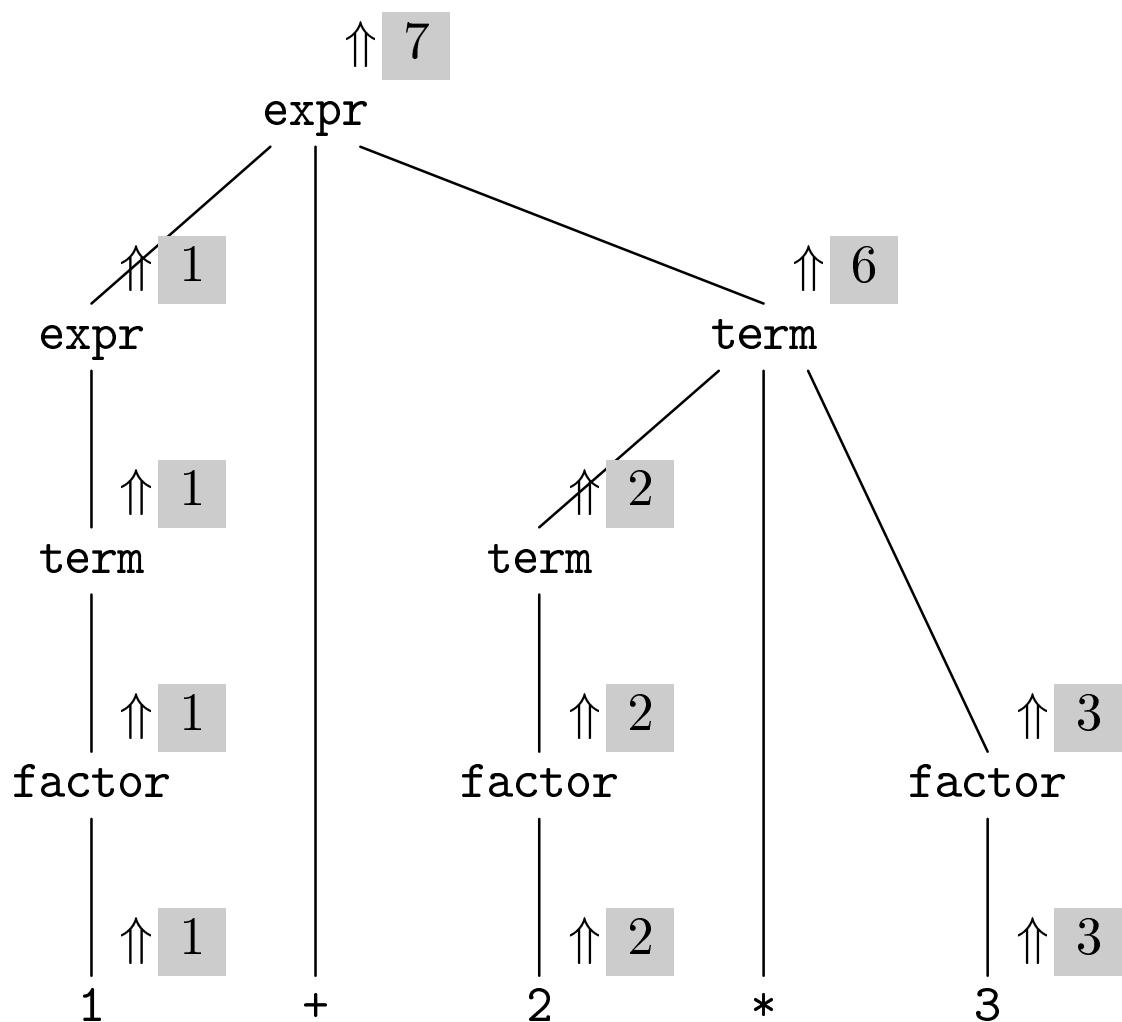
- $\$\$$ är attributet för X .
- $\$1$ är attributet för X_1 .
- $\$2$ är attributet för X_2 , osv

En semantisk aktion $\{ \ \$\$ = \$1; \ }$ behöver inte skrivas.

Körexempel

```
$ a.out  
1+2*3  
7
```

Beräkning av attribut, visat med ett parseträd



Mer om attribut

$$X \rightarrow X_1 \ X_2 \ \dots \ X_n$$

- Attribut som skickas uppåt i parseträdet kallas *syntetiserade*.

$$f(X) = f(X_1, \dots, X_n)$$

- Attribut som skickas nedåt kallas *ärvda*.

$$f(X_i) = f(X)$$

- Alla dessa beroenden mellan attributen leder till en viss evalueringsordning.
- Bara syntetiserade attribut \Rightarrow trivial evaluering.
- I praktiken kan också vissa ärvda attribut hanteras.

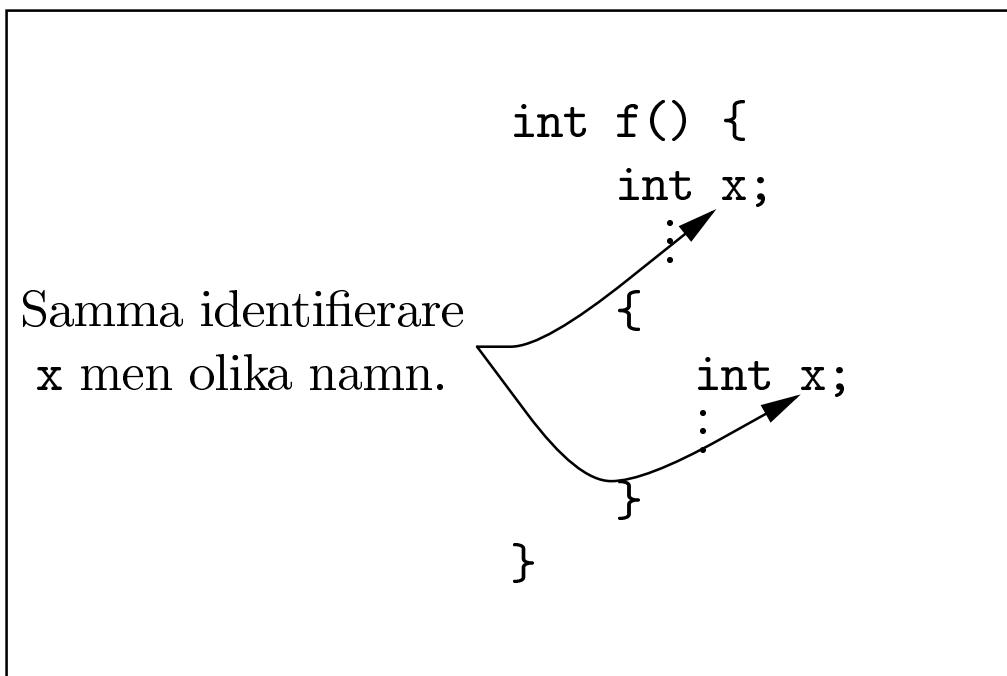
4 Symboltabeller

- Lagringsplatsen för alla *namn* i programmet.
- Ett namn kan vara en variabel, funktion, typ, etc.
- Håller reda på storlekar, adresser, typer, etc.
- Byggs upp under analysfasen.
- Används under syntesfasen.
 - Semantisk analys: typkonflikt?
 - Kodgenerering: hur mycket utrymme behövs?
 - Felhantering: information om namn.

Identifierare vs namn

- En *identifierare* är en sträng, t ex `main`.
- Ett *namn* betecknar ett minnesadress och har vissa attribut, t ex dess typ.

Exempel



Aliasing

Två namn betecknar
samma adress (C++)

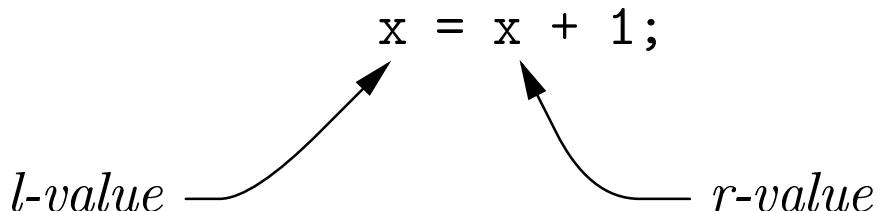
int f(int& x) {

:

f(y)

l-value vs r-value

Ett namn betyder olika saker på vänster och höger sida om en tilldelning.



- Ett *l-value* betecknar en minnesadress.
- Ett *r-value* betecknar ett värde.

Somliga uttryck saknar *l-value*, t ex `x+1` och `3`.

Men uttryck som `a[i]` har ett *l-value*.

Block-strukturer

- Pascal och Ada är språk som använder block.
- Block kan nästlas men inte överlappa.
- För dessa språk gäller *statiska* scope-regler:
 - Ett namn är synligt i det block det deklareras.
 - Om block B_2 är inuti B_1 så kommer namnen i B_1 också att synas i B_2 , såvida inte namnet har omdefinierats i B_2 .

```
B1: int x, y;  
{  
    B2:  
        int x;  
        :  
    }  
}
```

- Ett namns *scope* är de block där namnet syns.

Vilka operationer måste en symboltabell stödja?

- `begin_scope()`:

Markerar början på ett nytt block.

- `find_name(x)`:

Letar efter `x` i aktuellt block och utåt.

Returnera `NULL` om `x` inte finns.

- `add_name(x)`:

Lägger till `x` i symboltabellen.

- `end_scope()`:

Vid utgång ur ett block.

Alla namn som deklarerats i blocket tas bort.

En enkel och naturligt datastruktur för detta är en stack.

Symboltabellens utseende inuti p2

```
program prog;
var a, b, c : integer;

procedure p1;
var b, c : real;

procedure p2;
var c : real;
begin
    c := b + a;
end;
```

```
begin
    c := b + a;
end;
```

```
begin
```

```
...
```

```
end
```

c: real
c: real
b: real
c: int
b: int
a: int

Symboltabell

En stackbaserad implementation

- `begin_scope()`:

Pusha ett speciellt märke på stacken.

- `find_name(x)`:

Börja leta efter `x` överst, fortsätt nedåt tills `x` påträffas eller botten är nådd.

Returnera `NULL` om `x` inte finns.

- `add_name(x)`:

Pusha `x` på stacken.

- `end_scope()`:

Poppa alla element, till och med nästa märke.

I praktiken används hashtabeller (mycket effektivare).

5 Semantisk analys och internform

- Kontrollera statisk semantik och generera internform.
- Vad är statisk semantik?
 - Är alla variabler deklarerade?
 - Typkonflikter, stämmer operanderna med operatorerna, etc?
 - Stämmer typen och antal argument till funktioner?

Det finns formalismar för statisk semantik, p s s som kontextfri grammatik, men de har inte fått samma spridning.

- Varför generera internform?

Bra kod kan inte genereras i ett pass.

Optimeringar utförs enklast på en intern representation.

Var är vi?

Tecken:

s	u	m	=	0	;	/	*	i	n	i	t	*	/	...
---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----



Lexikalisk analys



Symboler:

ident(sum)	assign	intconst(0)	semicolon	...
------------	--------	-------------	-----------	-----



Syntaktisk analys

*Assignment*

Var

Ident(sum)

Expr

Term

IntConst(0)



Semantisk analys



:=

sum

0

Härläningsträd:

Symboltabell	
sum	float
x	int
:	

Internform:

Statisk analys

Följande fragment visar hur statisk semantik kan kontrolleras med syntaxstyrda översättningar.

- Vid en deklaration installeras namnet i symboltabellen:

```
var_decl      : IDENT : type_id
    { add_name($1, $3); }
```

- När en variabel används konsulteras symboltabellen.

```
factor        : IDENT
    { Name *p = find_name($1);
      if (p == NULL)
          error("%s not declared", $1);
      $$ = type(p);
    }
```

Generering av internform

- Bara en annan representation av källprogrammet.
- Internformen har dock följande fördelar:
 - Maskinberoende.
 - Inte avsett för något speciellt språk.
 - Lämpligt format för optimeringar.
 - Kan också interpereras.

Olika internformer

- Postfix notation (RPN).
- Abstrakt syntaxträd.
- Treadresskod.

Postfix notation

Infix	Postfix
a + b	a b +
a + b * c	a b c * +
(a + b) * c	a b + c *

- Operatorerna följer operanderna.
- Inga paranteser behövs.
- Stackmaskin, jfr HP-räknedosa.
- Användbart för uttryck utan villkor och hopp.

Evaluering av postfix-uttryck

Kräver en stack för att lagra mellanresultat.

Evaluera från vänster till höger:

- Numeriskt värde: pusha på stacken.
- Identiferare: pusha dess (r-)värde på stacken.
- Binär operator: poppa två element, applicera operatorn och pusha resultatet.

Evaluering av $(a + b) * c$ där $a=1$, $b=2$ och $c=4$.

<i>Stack</i>	<i>Indata</i>
	a b + c *
1	b + c *
1 2	+ c *
3	c *
3 4	*
12	

Utökning av postfix-notation

- **Tilldelning**

- Binär operator := med längsta prioritet.
- Använder l-value på sin första operand.

$x := 10 + k * 30 \Rightarrow x\ 10\ k\ 30\ *\ +\ :=$

- **Villkorliga satser**

Introducera ovillkorligt hopp

$<label>\ JUMP$

samt villkorligt hopp, om $\neq 0$

$<value>\ <label>\ JEQZ$

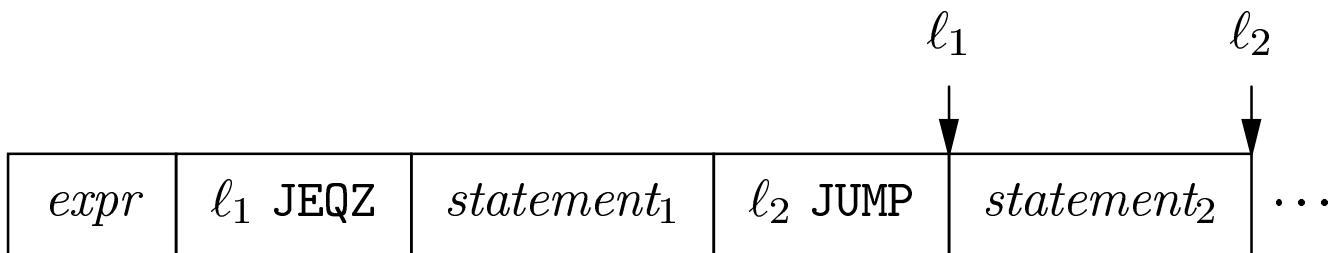
Om uttrycket lagras i en array så är label \equiv index.
Identifierare är egentligen index till symboltabellen.

Exempel

Översättning av

if expr then statement₁ else statement₂

I postfix form:



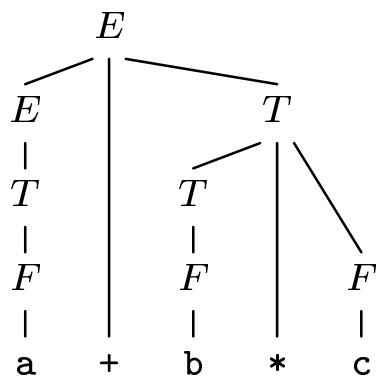
Övningsuppgift

Visa hur följande struktur kan översättas till postfixform.

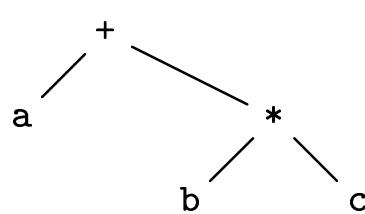
while expr do statement

Abstrakta syntaxträd

- Parsern producerar egentligen aldrig parseträdet.
Parseträdet har emellertid begränsat värde.
- Ett *abstrakt syntaxträd* påminner om ett parseträd.



Parseträd



Abstrakt syntaxträd

- Kan plattas ut och sparas på fil:

1	id	b	
2	id	c	
3	*	1	2
4	id	a	
5	+	4	3

Abstrakta syntaxträd

- Enkelt att bygga abstrakta syntaxträd mha semantiska aktioner.

```
expr      : expr '+' term          { $$ = mk_node('+', $1, $3); }
           | term                  { $$ = $1; }

;

term     : term '*' factor        { $$ = mk_node('*', $1, $3); }
           | factor                { $$ = $1; }

;

factor   : '(' expr ')',         { $$ = $2; }
           | NUM                  { $$ = mk_leaf(NUM, $1); }
           | IDENT                { $$ = mk_leaf(IDENT, $1); }

;
```

- Fördelar med abstrakta syntaxträd:
 - Enkelt att traversera och interperetera.
 - Pretty-printing möjligt genom inorder traversering.
 - Postorder traversering \Rightarrow postfix notation!
- Men det är en bit kvar till assembler...

Treadresskod

- En sekvens av instruktioner på något av följande format:

res \leftarrow *arg*₁ *op* *arg*₂

param *arg*

call *fcn* *n*

if *arg*₁ *relop* *arg*₂ goto *label*

goto *label*

- Exempel: tilldelningsats *a* := *b* * *c* + *d*

t1 \leftarrow *b* * *c*

a \leftarrow *t1* + *d*

- *t1* är en temporärvariabel, introduceras av kompilatorn.
- Argumenten och resultatet är egentligen index till symboltabellen.

Kontrollstrukturer med tressadresskod

```
if a = b then x := x + 1 else y := 20;
```

```
if a = b goto L1
x ← x + 1
goto L2
L1: y ← 20
L2:
```

Hoppadresserna (L1) och (L2) måste fyllas i efteråt, eftersom vi inte vet destinationen när hoppet skrivs.

Detta kan antingen göras i två pass, eller med s k *backpatching*.

Proceduranrop med tressadresskod

$f(a_1, a_2, \dots a_n);$

param	a_1
param	a_2
...	...
param	a_n
call	f, n

Array-referenser

- När referensen är ett l-value:

$a[i] \leftarrow b$

- När referensen är ett r-value

$b \leftarrow a[i]$

Exempel

Några exempel på översättning till internform med syntaxstyrda översättningar.

- Översättning till postfix notation.

```
line    : expr '\n'  
        ;  
expr    : expr '+' term { printf("+ "); }  
        | term  
        ;  
term    : term '*' factor { printf("* "); }  
        | factor  
        ;  
factor  : NUM           { printf("%d ", $1); }  
        | '(' expr ')',  
        ;
```

- Körexempel:

```
$ a.out  
1+2*3  
1 2 3 * +  
$ a.out  
(1+2)*3  
1 2 + 3 *
```

- Översättning av tilldelningssatser till treadresskod.

```
assign  : var '=' expr ';'      { gen($1, $3, '=', -1); }

var     : IDENT                  { $$ = lookup($1); }

expr    : expr '+' term        { $$ = gentemp();
                                gen($$, $1, '+', $3); }
          | term                 { $$ = $1; }

term   : term '*' factor       { $$ = gentemp();
                                 gen($$, $1, '*', $3); }
          | factor               { $$ = $1; }

factor : IDENT                  { $$ = lookup($1); }
          | '(' expr ')'
          ;
```

- `gen(res, a1, op, a2)` genererar en instruktion.
- `gentemp()` genererar en ny temporärvariabel och returnerar dess index.
- `lookup(s)` returnerar symboltabellsindex för `s`.

Representation av treadresskod

- Treadresskod kan lagras i en array. Exempel:

```
if a = b goto L1
x ← x + 1
goto L2
L1: y ← 20
L2:
```

- Representation:

	<i>op</i>	<i>arg</i> ₁	<i>arg</i> ₂	<i>res</i>
1	=	a	b	t1
2	JEQZ	t1		(5)
3	+	x	1	x
4	JUMP			(6)
5	:=	20		y
6				

6 Kodoptimering

- Hur vet vi om vi genererat optimal kod?
- Omöjligt att veta!
- Vi får nöja oss med *kodförbättringar*.
- Mål: snabbare och/eller mindre kod.
- Typer av optimeringar:
 1. Maskinberoende:
Utförs på internformen.
 2. Maskinberoende:
Utförs på genererad assemblerkod.

Konsekvenser av optimering

- Försvårar debugging.

Kod flyttas runt, variabler elimineras, etc.

- Kompileringen tar längre tid.

Många optimeringar kräver noggrann analys.

- Kan introducera numeriska problem.

$x*y - x*z \Rightarrow x*(y - z)$ är kanske OK.

$a + (b - c) \Rightarrow (a + b) - c$ kan ge *overflow*.

Typer av optimeringar

1. Algoritmiska förbättringar.

- Byt ut en långsam algoritm mot en snabbare.
- Val av algoritmer största faktorn i hastighet.
- Dock mycket svårt att automatisera.

2. Optimeringar på internformen.

- Lokala optimeringar, inuti så kallade *basic blocks*.
- Loop-optimeringar.
- Adressberäkningar för vektorer och poster.
- Globala optimeringar (på en hel funktion).
- Interprocedurella optimeringar (ovanligt).

3. Peephole-optimeringar.

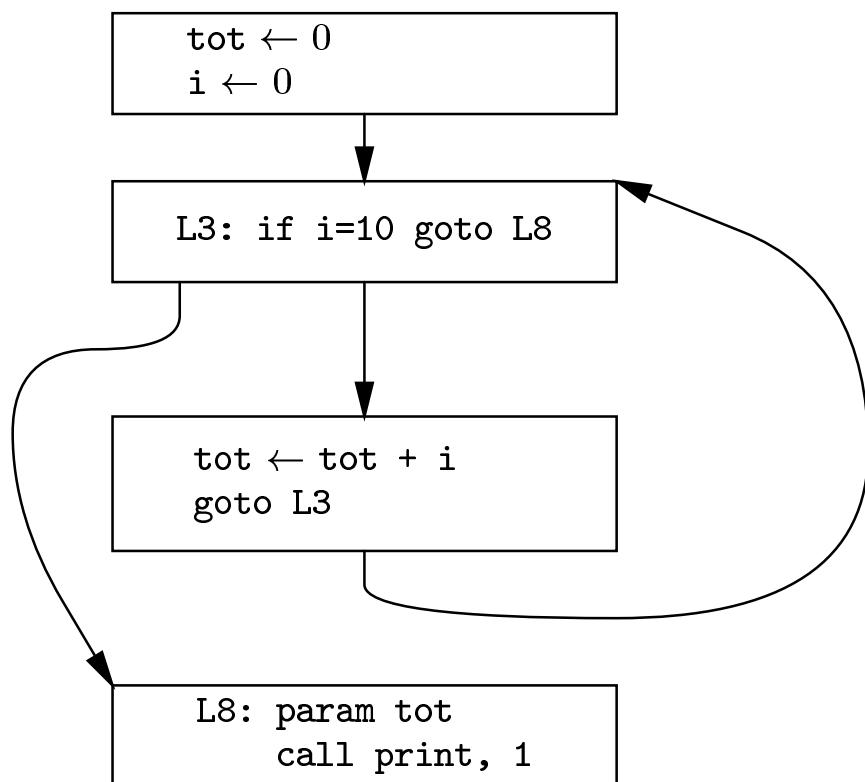
- Utförs på assembler-nivå (maskinberoende).

Basic block

- Ett *basic block* är en rak följd av instruktioner.
- Bara en ingång och en utgång.
- Inga hopp (förutom eventuellt sista instruktionen).

Exempel på basic block

```
tot = 0;  
for (i = 0; i < 10; i++)  
    tot = tot + i;  
print(tot);
```



Lokala optimeringar

- Optimeringar som utförs inom ett basic block utan information från andra block.
- Exempel: *constant folding*.

Konstanta uttryck beräknas under kompileringen.

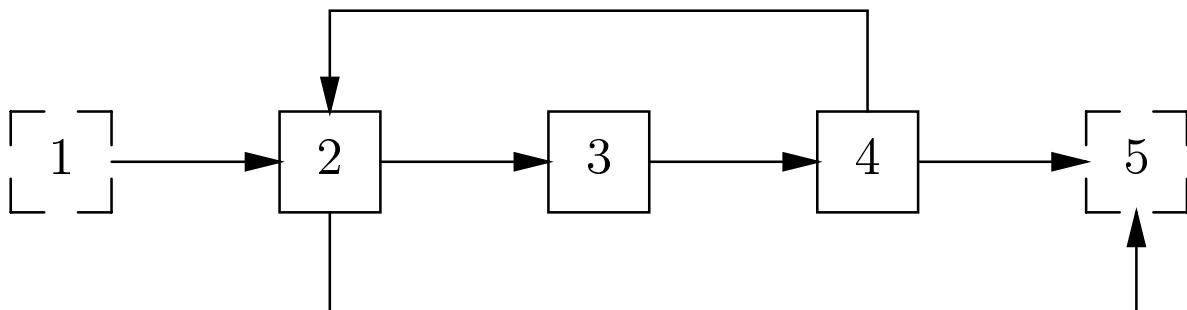
Före	Efter
const int N=10;	const int N=10;
...	...
i = N-1;	i = 9;
j = i*5;	j = 45;

- Exempel: *eliminering av gemensamma deluttryck*.

Före	Efter
a[i+1] = b[i+1];	t1 = i+1; a[t1] = b[t1];

Loop-optimeringar

- Målet är att minimera tiden i loopar.
- Försök reducera antalet operationer inne i loopen.
- En *kontrollflödesgraf* är en graf där
 - basic blocks är noder.
 - hopp är bågar.



- En *loop* är en mängd noder med en unik ingång och där alla kan nå varandra.
- I figuren är $\{2, 3, 4\}$ en loop.

Exempel på loop-optimeringar

- Exempel: *flytta loop-invarianter.*

<i>Före</i>	<i>Efter</i>
<pre>for (i=0; i<10; i++) { z = i + b/c ... }</pre>	<pre>t = b/c; for (i=0; i<10; i++) { z = i + t; ... }</pre>

- Exempel: *loop unrolling.*

Duplicera loop-kroppen – reducera test och hopp.

<i>Före</i>	<i>Efter</i>
<pre>i = 1; while (i<=50) { a[i] = b[i]; i = i + 1; }</pre>	<pre>i = 1; while (i<=50) { a[i] = b[i]; i = i + 1; a[i] = b[i]; i = i + 1; }</pre>

Användandet av lokala optimeringar

- Varje lokal optimering ger inte så mycket.
- Men tillsammans kan de göra en hel del!
- Typiskt så leder en optimering till att en annan optimering är möjlig.
- Svårt att veta i vilken ordning dessa ska utföras.
- Vanligt att upprepa optimeringar tills ingen förbättring sker.

Globala optimeringar

- Optimeringar som utförs på en hel funktion.
- Interprocedurell analys hanterar hela programmet.
- Kräver dataflödesanalys.
- Dataflödesanalysen kan upptäcka
 - variabler som aldrig används.
 - beräkningar som aldrig används.
 - kod som inte nås (död kod).
 - variabler som inte initierats.

Dataflödesanalys

Koncept:

- *def*: När en variabel får ett värde.

`a = 5;`

- *use*: När en variabel används.

`b = a*c;`

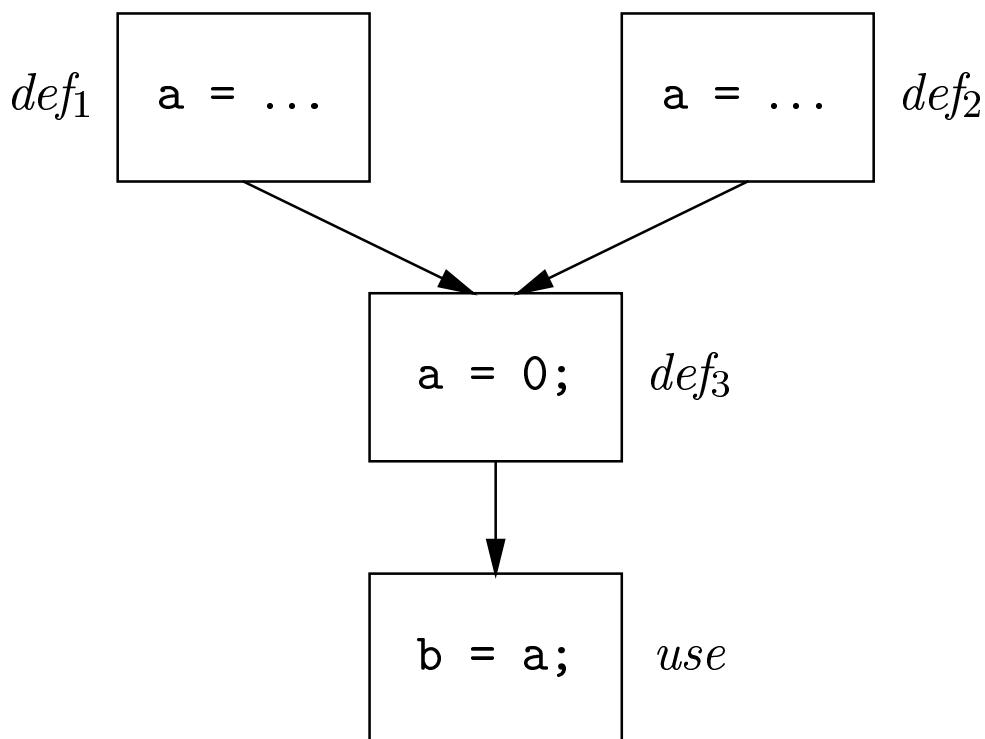
Två typer av dataflödesanalys:

1. Framåtanalys.
2. Bakåtanalys.

Framåtanalys

- Exempel: *reaching definitions.*

Vilka *def*:s når en viss punkt i flödesgrafen?



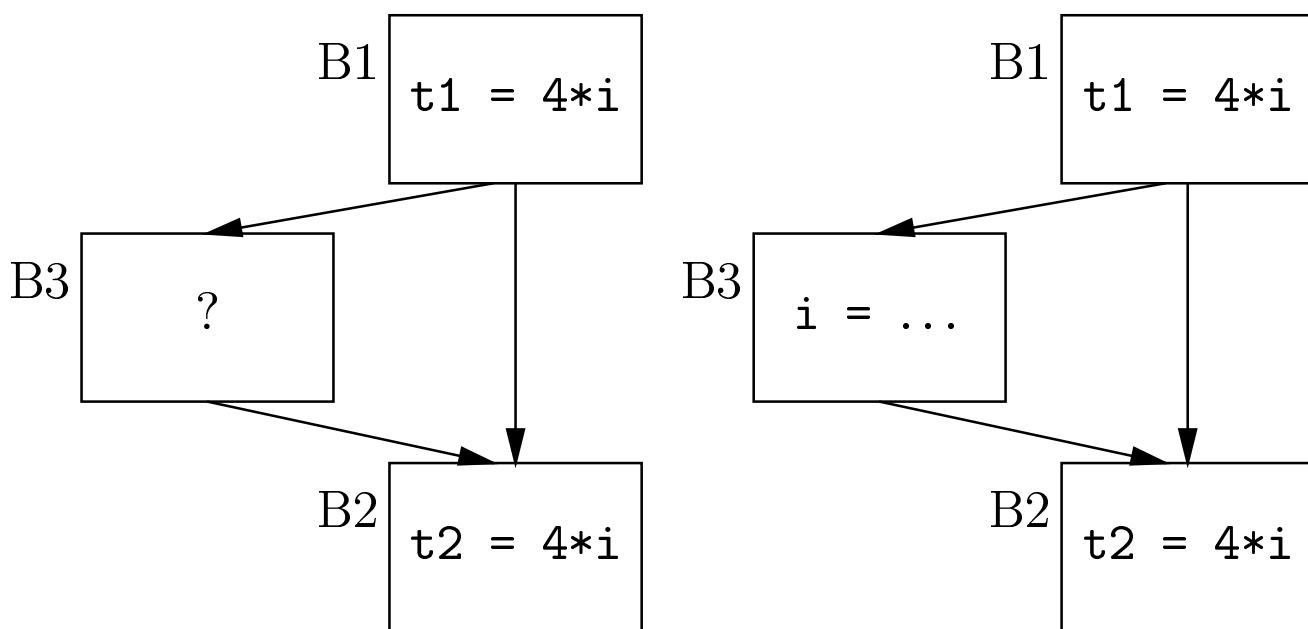
def_3 ”dödar” def_1 och def_2 av a .

def_3 är enda definitionen som når $b = a$.

Låter oss ersätta $b = a$ med $b = 0$.

- Exempel: *available expressions*.

För att eliminera gemensamma deluttryck över blockgränser.



Om B3 inte definierar i så är $4*i$ tillgängligt i B2 och behöver inte räknas ut igen.

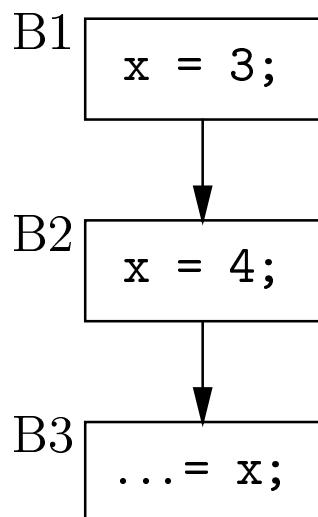
Om B3 definierar i så måste B3 också räkna ut $4*i$ för att B2 ska slippa göra det.

Bakåtanalys

- Exempel: *live variables*

En variabel v är levande i en viss punkt p om

- Det finns en punkt p' som använder (*use*) v .
- Det finns en väg från p till p' .
- På den vägen finns inga *def* av v .



- x är levande i B2, men inte i B1.
- Om v ligger i ett register men inte lever så kan registret frigöras.

Andra maskinoberoende optimeringar

- Array-referenser.

$$c = a[i, j] + a[i, j+1]$$

Om elementen ligger bredvid varandra i minnet behöver inte adressen för $a[i, j+1]$ räknas ut.

- Inline-expansion för små funktioner.

```
int sqr(int x) { return x*x; }
```

$y = \text{sqr}(x)$ kan bytas ut mot $y = x*x;$

- Utnyttja algebraiska identiteter.

$$k = -c*(b-a) \Rightarrow k = c*(a-b).$$

- Eliminera multiplikation med 1 och addition med 0.
- Vänsterskifta istf att multiplicera med 2.

Maskinberoende optimeringar

- Peephole-optimering är en effektiv teknik för att förbättra assemblerkod.
- Använd ett fönster på 3–4 instruktioner.

```
load r0,M[a]
add r0,1
store r0,M[a]
```

⇒

```
inc M[a]
```

- Matcha innehållet i fönstret mot sekvenser som har kortare/snabbare sekvenser.
- Flytta sedan fönstret framåt.
- Flera pass kan vara nödvändigt.

Typiska peephole-optimeringar

Optimeringarna kan formuleras som regler

$$i_1, \dots, i_n \rightarrow j_1, \dots, j_m$$

där vänstersidan är originalsekvensen.

- $\text{load } \$a, \$b \quad \text{store } \$a, \$b \rightarrow \text{load } \$a, \$b$
(Kan inte utföras om `store`-instruktionen är destinationen för ett hopp.)
- $\text{add } \$a, i \quad \text{ADD } \$a, j \rightarrow \text{add } \$a, (i+j)$
där i och j är konstanter.
- $\text{add } \$a, 0 \rightarrow$
Dvs, denna instruktion elimineras.

Övningsuppgift

För följande program

```
s = 0;  
for (i = 0; i < n; i++)  
    s += x[i] * y[i];
```

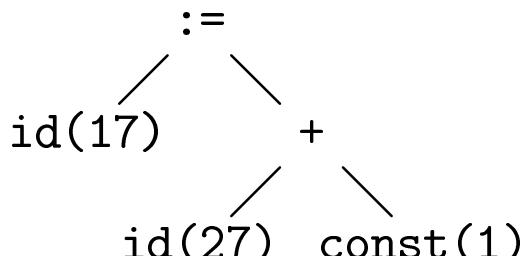
1. Visa motsvarande kod i treaddressform.
2. Konstruera sedan en kontrollflödesgraf med basic blocks.
3. Vilka loopar finns det i kontrollflödesgrafen?

7 Kodgenerering

- Indata: (optimerad) internform.
- Utdata: objektkod.
- Kodgenerering omfattar följande problem:
 1. **Minneshantering**
Var lagras variabler?
 2. **Instruktionsval**
Välja lämpliga instruktioner hos målmaskinen.
Genererar kod med ∞ antal *virtuella* register.
 3. **Registerallokering**
Avbilda virtuella register på fysiska register.
- Viktigast är givetvis att producera *korrekt* kod.

Minneshantering

- Internformen har referenser till symboltabellen:



Symboltabell	
17	
	x: int
27	
	y: int

- Uppgift:

1. Avgör var variabler ska lagras i minnet under körningen.

Dvs, vad är deras *run-time* adress?

2. Låt dessa adresser vara explicit i internformen.

Organisation av minnet

1. Koden och statiska variabler

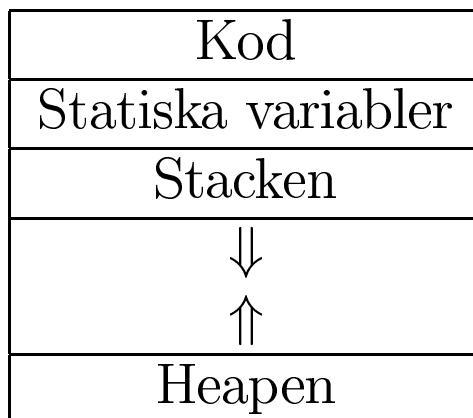
Storleken på kod och statiska variabler kan bestämmas vid kompileringstillfället.

2. Stacken

I stacken lagras lokala variabler och parametrar som skickas till funktioner.

3. Heapen

Heapen är en minnesarea för data som allokeras under körningen, t ex med `malloc()`.



Statisk eller dynamisk minneshantering?

- Variabler som lagras i statiska arean kan refereras med en fast minnesadress.
- För att klara rekursion måste dock lokala variabler lagras på en stack.
- Tidiga FORTRAN-versioner klarade inte rekursion.

Stacken

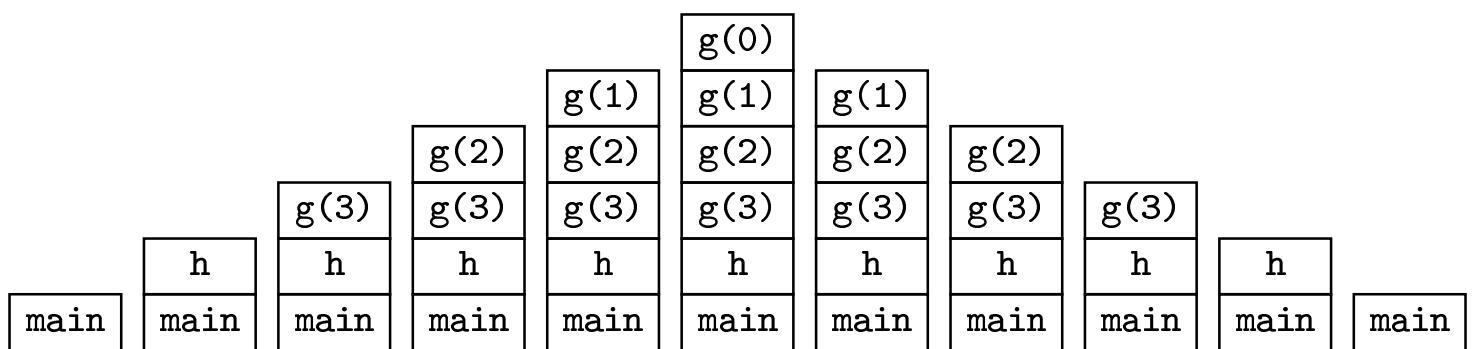
- Stacken består av *aktiveringspostar*.
- Varje funktionsanrop skapar en ny aktiveringspost.
- En aktiveringspost lagrar bl a
 1. Lokala variabler
 2. Parametrar till funktionen
 3. Återhopsadress

Aktiveringsposters livstid

```
int g(int i) {
    if (i > 0)
        g(i-1);
}
```

```
int h() {
    g(3);
}
```

```
int main() {
    h();
}
```



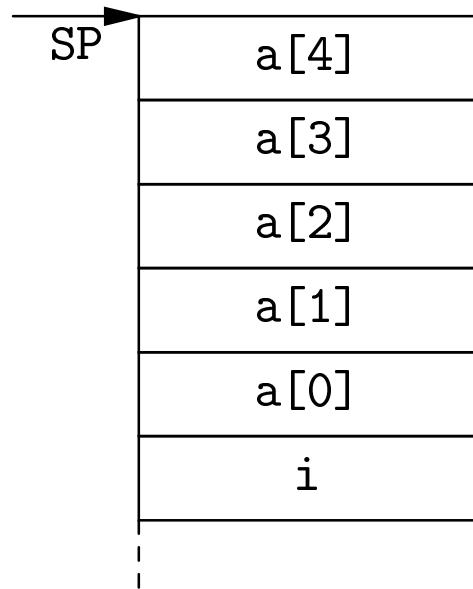
Minneslayout för lokala variabler

Pascal, C, etc: storleken på alla variabler är känt vid kompileringstillfället.

- ⇒ storleken på aktiveringsposten kan bestämmas vid kompileringstillfället.
- ⇒ adressen till en lokal variabel kan beskrivas som ett offset till stackpekaren.

Exempel på aktiveringspost (förenklat)

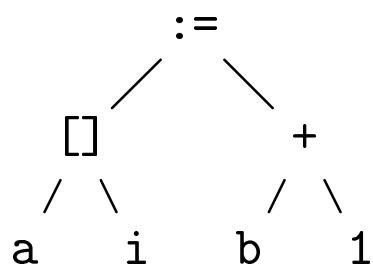
```
int b;  
  
void f() {  
    int a[4];  
    int i;  
    for (i = 0; i < 4; i++)  
        a[i] = b+1;  
}
```



- b är en statisk variabel, kan ges en fast adress.
- a[] och i lagras i f:s aktiveringspost.
- Antag att SP är stackpekaren, ett speciellt register.
- i:s adress är då SP-24 (om `sizeof(int)==4`).
- a:s startadress (a[0]) är då SP-20.

Explicita adresser i internformen

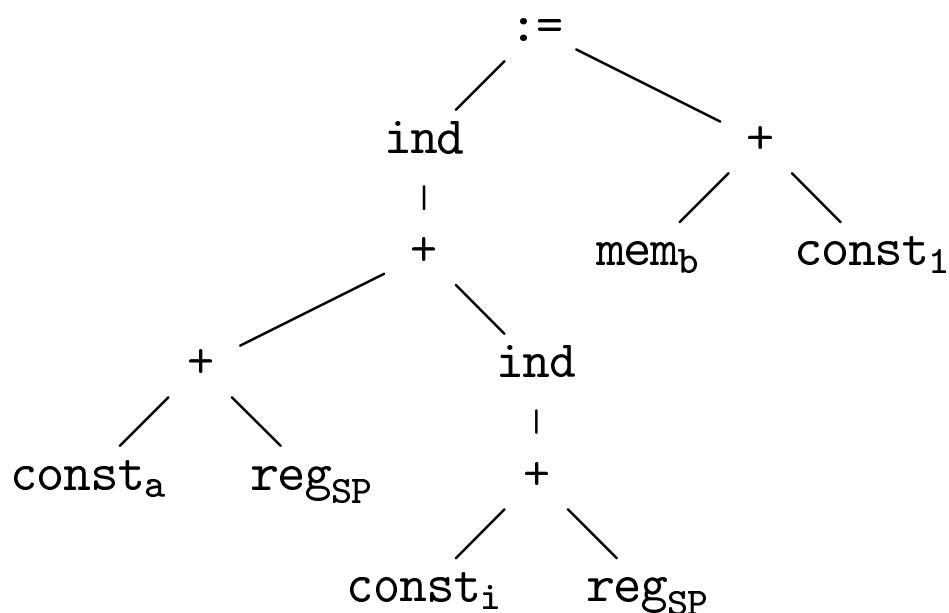
Abstrakt syntaxträd för $a[i] = b+1$:



Låt nu alla adresser och minnesaccesser vara explicita i internformen:

- Markera statiska variabler x med fast adress mem_x .
- Markera konstanter c med const_c .
- Markera register r_i med reg_i .
- Alla referenser via minnet markeras med en speciell `ind`-nod.

- Leder till en internform med explicita minnesreferenser:



- const_i är offset från SP till i .
- const_a är offset från SP till a :s startadress.

Denna förfinade internform underlättar instruktionsval eftersom alla accesser till register och minne görs explicit.

Målarkitektur

1. Stackmaskin.

- Oftast en *virtuell* maskin, t ex Java.

Källkod	Stackmaskinkod
$x = y + z * v$	pusha x push y push z push v mul add assign

- Register saknas.
- Aritmetiska uttryck översätts till RPN.

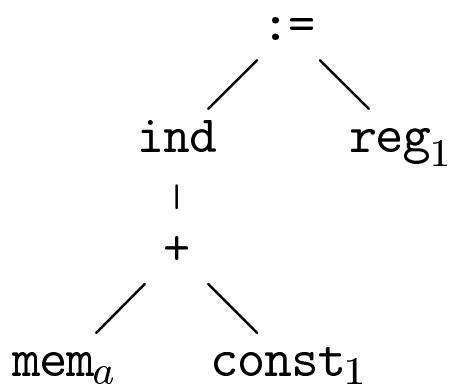
2. Registermaskin.

- ”Riktig” hårdvara, t ex MIPS eller Pentium.
- Två typer: RISC och CISC.
- RISC: *Reduced Instruction Set Computer*
 - Instruktionerna är ”enkla”, 32 bit långa.
 - Av formen $r_1 \leftarrow r_2 \oplus r_3$ eller load/store.

```
load  r1, M[r1+r2]
load  r2, M[r3+disp]
mult  r3 = r1,r2
store r3, M[r1+r2]
```
 - Många register, oftast 32.
 - Kodgenereringen förenklas.
- CISC: *Complex Instruction Set Computer*
 - Färre register (16, 8 eller 6).
 - Avancerade adresseringsmoder.
$$M[r1+r2] = M[r1+r2]*M[r3+disp]$$
 - Svårt att utnyttja alla instruktioner.

Instruktionsval

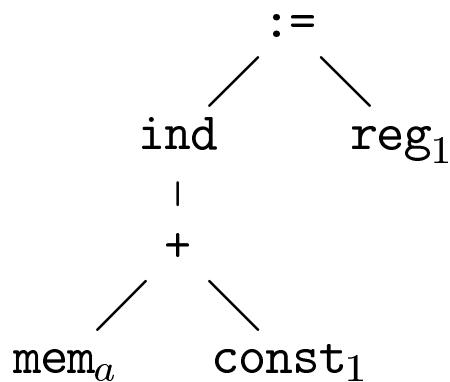
- Internformen (IR) består av *enkla* instruktioner:
Minnesaccess, addition, villkorligt hopp, etc.
- En (CISC-)processor har oftast avancerade adresseringsmoder och kan utföra flera IR-instruktioner med en instruktion.
- Exempel: många processor kan utföra följande med en instruktion `store r1, M[a+1]`



- Att hitta den kortaste/snabbaste sekvensen av instruktioner är en uppgift för *instruktionsväljaren*.

En enkel, men oacceptabel lösning

- Beräkna alla intermediära resultat
- Exempel



```
load r2,a
load r3,1
add r2,r3
load r4,M[r1]
store r1,M[r4]
```

Hur genererar man bra kod?

- Undvik onödiga beräkningar.
Återanvänd ”common subexpressions”.
- Undvik att läsa/skriva minnet.
Håll registren sysselsatta.
- Undvik förflyttningar mellan registren.
Förutse framtidiga registerbehov.
- Välj lämpliga instruktioner.
Studera hela instruktionsuppsättningen.

Kodgenerering är ett svårt problem, det gäller att veta vad som räknats ut och vilka beräkningar som kommer att göras i framtiden!

Instruktionsval med trädmönster

- Målmaskinen beskrivs med en samling mönster.
- Varje mönster är ett träd med motsv. instruktion.

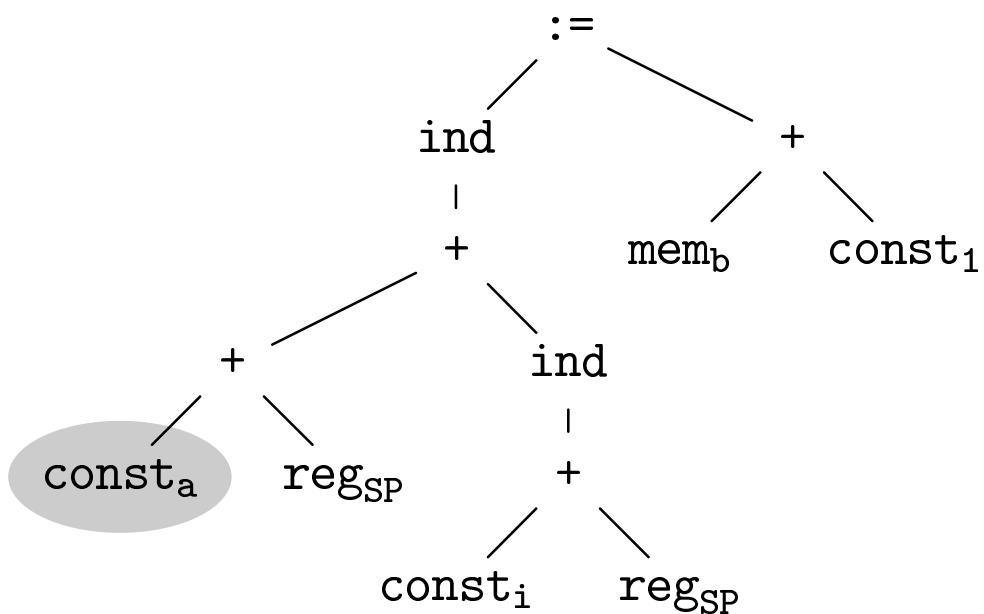
Mönster	Instruktion
$\text{reg}_i \leftarrow \begin{array}{c} + \\ \diagup \quad \diagdown \\ \text{reg}_i \quad \text{reg}_j \end{array}$	add r_i, r_j

- Trädet med internformen matchas top-down:
 - Om ett mönster matchar hela trädet, generera instruktionen.
 - Annars, matcha barnen rekursivt.
 - När ett mönster $l \leftarrow r$ matchar, byt ut r mot l och mata ut instruktionen.

(1)	$\text{reg}_i \leftarrow \text{const}_c$	load ri, c
(2)	$\text{reg}_i \leftarrow \text{mem}_a$	load $ri, M[a]$
(3)	$\text{mem} \leftarrow$ <pre> graph TD mem[mem] --> assign[:=] assign --> mem_a[mem_a] assign --> reg_i[reg_i] </pre>	store $ri, M[a]$
(4)	$\text{mem} \leftarrow$ <pre> graph TD mem[mem] --> assign[:=] assign --> ind[ind] assign --> reg_j[reg_j] ind --- reg_i[reg_i] </pre>	store $rj, M[ri]$
(5)	$\text{reg}_i \leftarrow$ <pre> graph TD reg_i[reg_i] --> assign[:=] assign --> ind[ind] ind --- plus[+] plus --- const_c[const_c] plus --- reg_j[reg_j] </pre>	load $ri, M[c + rj]$
(6)	$\text{reg}_i \leftarrow$ <pre> graph TD reg_i[reg_i] --> assign[:=] assign --> plus[+] plus --- reg_i_2[reg_i] plus --- ind[ind] ind --- plus_2[+] plus_2 --- const_c[const_c] plus_2 --- reg_j[reg_j] </pre>	add $ri, M[c + rj]$
(7)	$\text{reg}_i \leftarrow$ <pre> graph TD reg_i[reg_i] --> assign[:=] assign --> plus[+] plus --- reg_i_2[reg_i] plus --- reg_j[reg_j] </pre>	add ri, rj
(8)	$\text{reg}_i \leftarrow$ <pre> graph TD reg_i[reg_i] --> assign[:=] assign --> plus[+] plus --- reg_i_2[reg_i] plus --- const_1[const_1] </pre>	inc ri

Exempel på matchning

- Indata: internform för $a[i] = b + 1$.

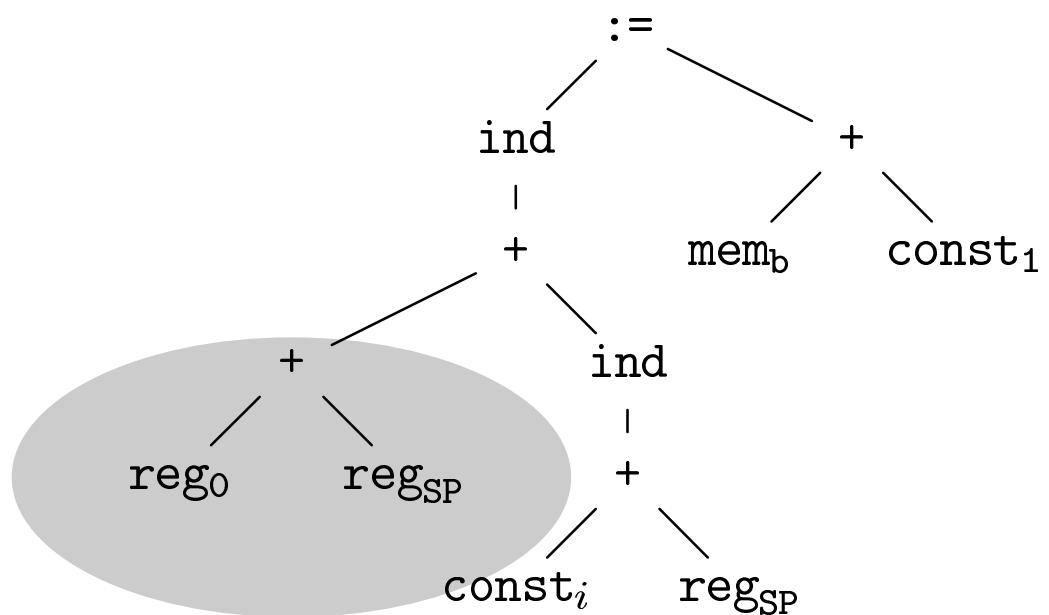


- Den skuggade delen matchar mönster 1.

$\text{reg}_i \leftarrow \text{const}_c$	load r_i, c
--	---------------

- Vi kan välja att placera resultatet i register 0.

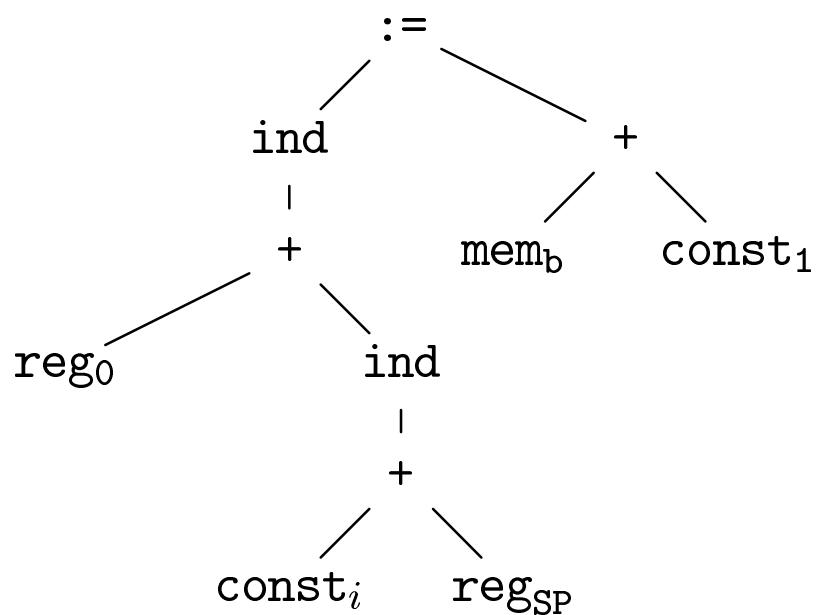
- Nytt träd



- Nu matchar regel 7 det markerade delträdet.

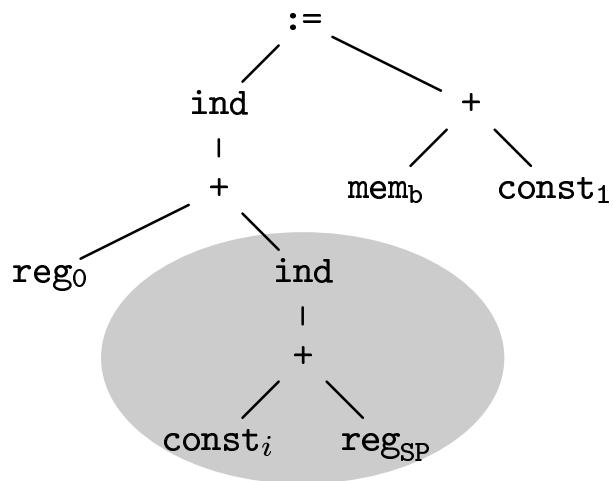
$\text{reg}_i \leftarrow$	$\begin{array}{c} + \\ \text{reg}_i \quad \text{reg}_j \end{array}$	add r_i, r_j
---------------------------	---	----------------

- Nytt träd

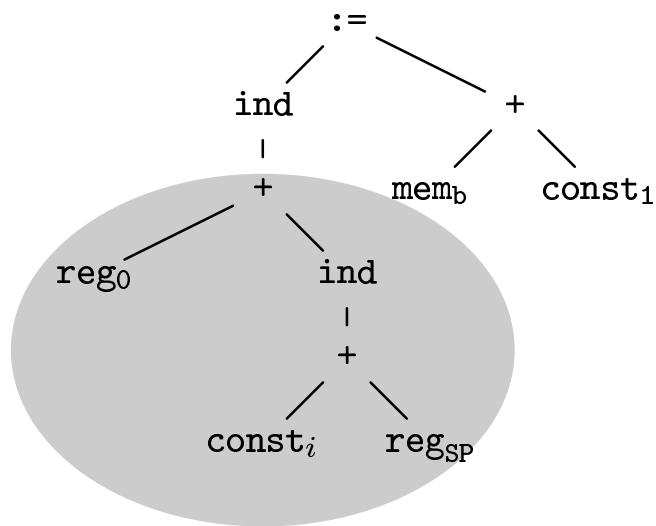


- Nu finns två möjligheter ...

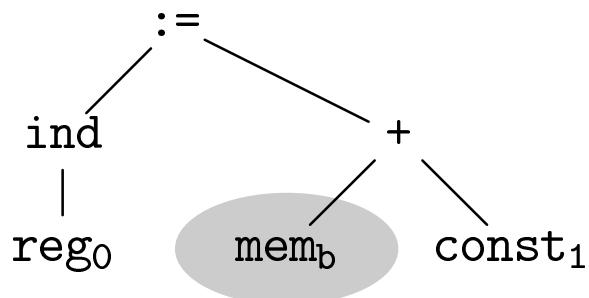
- Vi kan matcha med antingen regel 5,



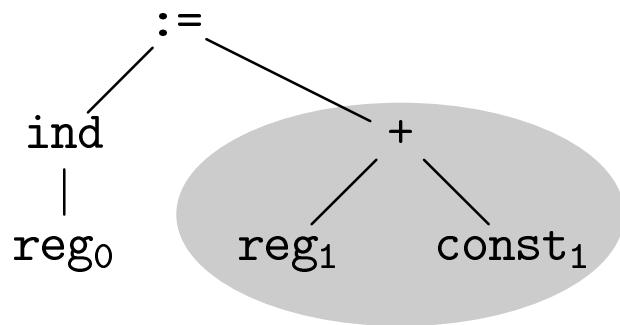
- eller använda regel 6 och matcha ett större delträd:



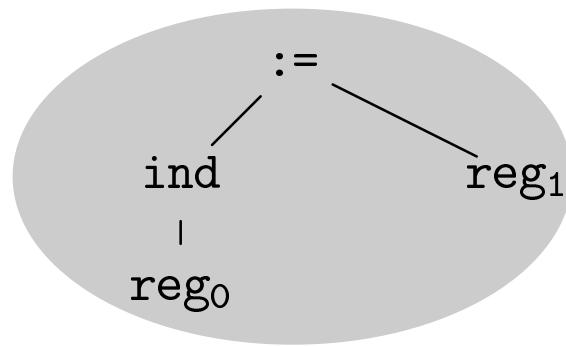
- Om vi använder regel 6 får vi följande träd:



- I det högra delträdet matchar regel 2.



- Nu kan vi använda regel 8 för att generera en inc.



- Det sista trädet matchas slutligen av regel 4 och vi har genererat följande instruktioner.

```
load  r0,a
add   r0,SP
add   r0,M[i+SP]
load  r1,b
inc   r1
store r1,M[r0]
```

- Om flera mönster matchar, använd det mönster som har lägst ”kostnad”.
- Vi har inte diskuterat vad som händer när inget mönster matchar eller hur man undviker att omskrivningsprocessen loopar.
- Verktyg finns (i stil med Yacc) som genererar en kodgenerator från en beskrivning, t ex BURG.

Registerallokering

- Bekvämt att låta instruktionsvälvjaren använda obegränsat många *virtuella* register.
- Uppgift: avbilda virtuella register på fysiska.
- Vissa register i målmaskinen är reserverade, t ex stackpekaren.
- Andra register kan användas av oss.
- Mål: håll registren så sysselsatta som möjligt.
- Men *vilka* variabler skall placeras i register?

$a = c + d;$	$r1 = r2 + r3$
$e = a + b;$	$r4 = r1 + r5$
$f = e - 1;$	$r1 = r4 - 1$

Observera att $r1$ används av både a och f .

Registerallokering med graffärgning

- En enkel och effektiv algoritm [Chaitin, 1980].
- Kräver dataflödesanalys (se ⟨Kodoptimering⟩).
- Terminologi:
 - **Variabel:** En programvariabel eller temporär.
 - **Def:** En variabel *definieras* när den får ett värde.
 - **Use:** En variabel *används* om dess värde refereras.
 - **Live:** En variabel är *levande* på ett ställe om dess värde kommer att användas längre fram.

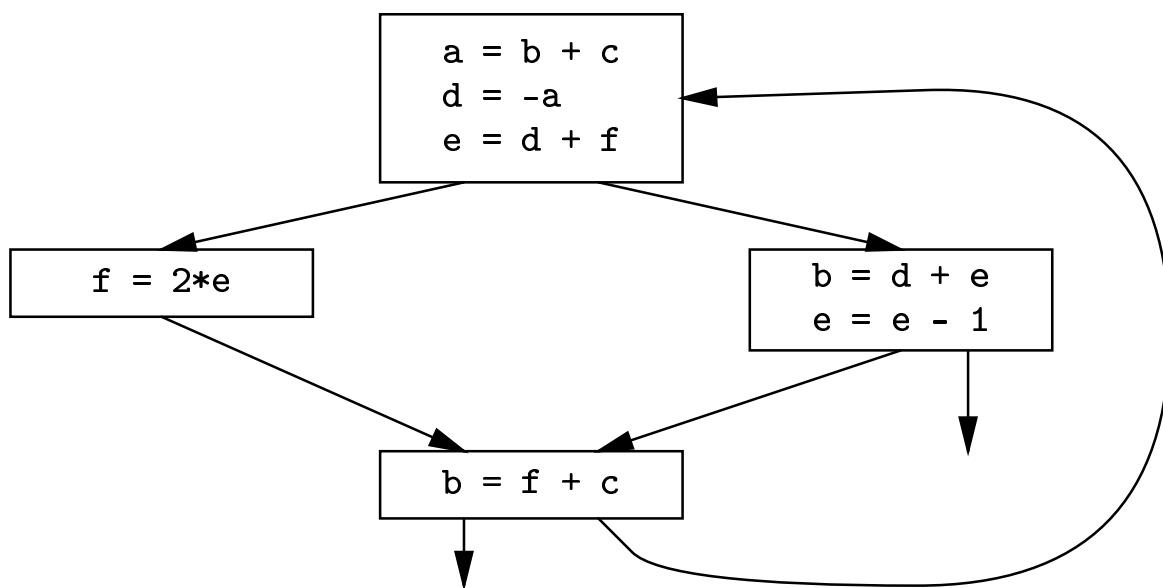
Insikt

Två variabler kan inte dela på ett register om de lever samtidigt.

Exempel

- Betrakta följande flödesgraf.

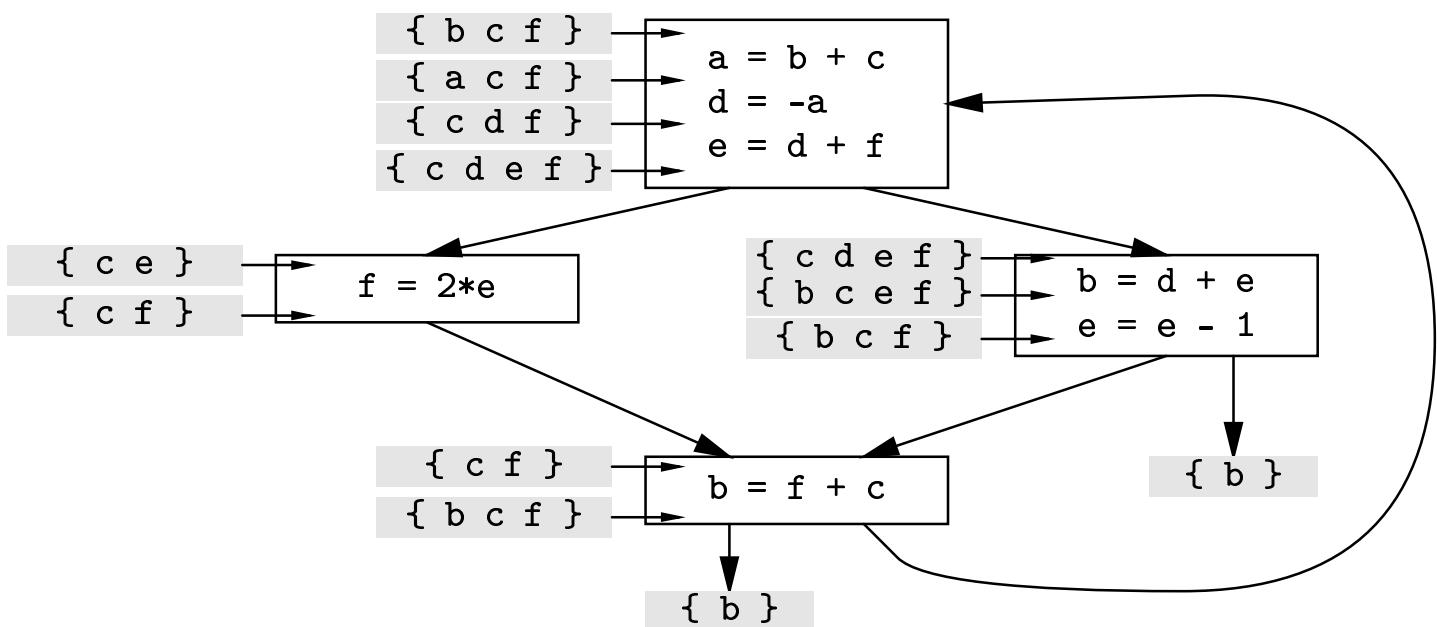
(Test och hopp är inte med av pedagogiska skäl)



- Antag att vi vill hålla f i ett register.
- Vilka andra variabler lever samtidigt med f ?

Livstidsinformation

- Antag att b behövs när loopen avslutas.
- Bilden visar levande variabler i olika punkter.

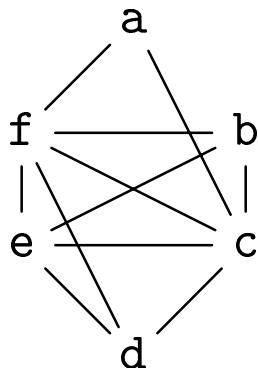


- Livstidsinformation beräknas *bakåt*.
- Betrakta en tilldelning $x = y + z$.
- Antag att y och z lever före (och efter).
- Efter tilldelningen vill vi hålla x , y och z i register.

Interferensgraf

Konstruera en graf där

- noder är variabler
- två noder x och y är sammanbundna med en båge om y lever direkt efter det att x definierats.

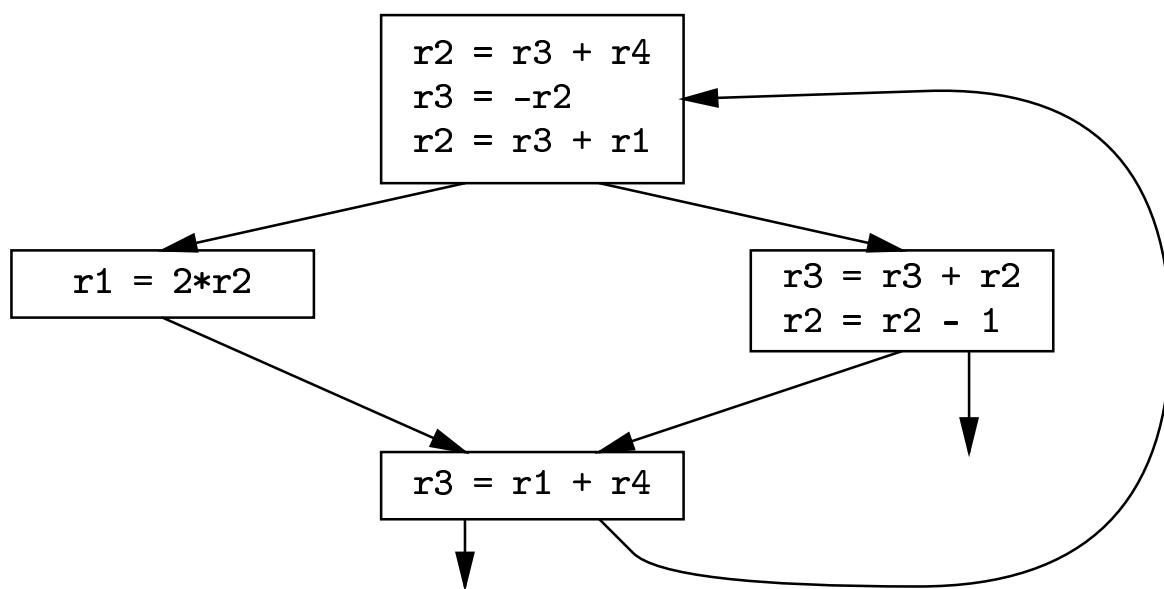


- Intuitivt betyder en båge (x, y) att x och y inte kan ligga i samma register.
- En registerallokering är giltig om alla variabler som är sammanbundna med en båge har olika register.

Graffärgning

- Problemet är ekvivalent med ett klassiskt problem.
- Antag att det finns k färger (register).
- Färglägg grafen så att två noder sammanbundna med en båge har olika färg.
- Det finns en sådan färgläggning av våran graf.

a->r2 b->r3 c->r4 d->r3 e->r2 f->r1



Mer om graffärgning

- Hur avgör man om grafen kan färgläggas med k färger?
 1. Ett mycket svårt kombinatoriskt problem.

Vi kan istället använda en s k *heuristik*.
En heuristik är snabbare men ger kanske inte optimala resultat.
 2. En del grafer kan inte färgläggas med k färger.

Programmet måste då modifieras så att en del värden sparas i minnet.

Heuristik för graffärgning

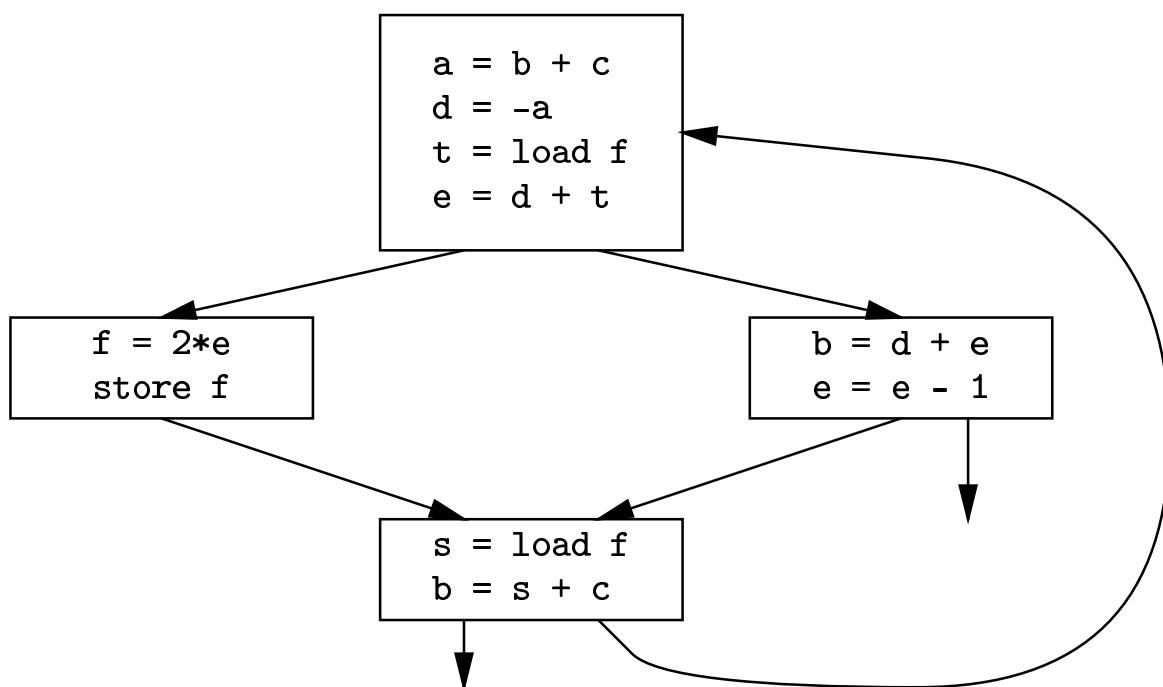
1. Antag att det finns k st färger.
2. Välj en nod x med $< k$ grannar.
3. Placera x på en lista och plocka bort x och alla dess utgående bågar från grafen.
4. Upprepa 2–3 tills grafen är tom.
5. För alla noder x i listan i omvänd ordning:

Välj en färg (register) för x som inte leder till konflikt med x :s grannar.

- Färgläggning i det tidigare exemplet producerades av denna heuristik.
- Heuristiken misslyckas om ingen nod med $< k$ grannar finns.

När registren inte räcker till

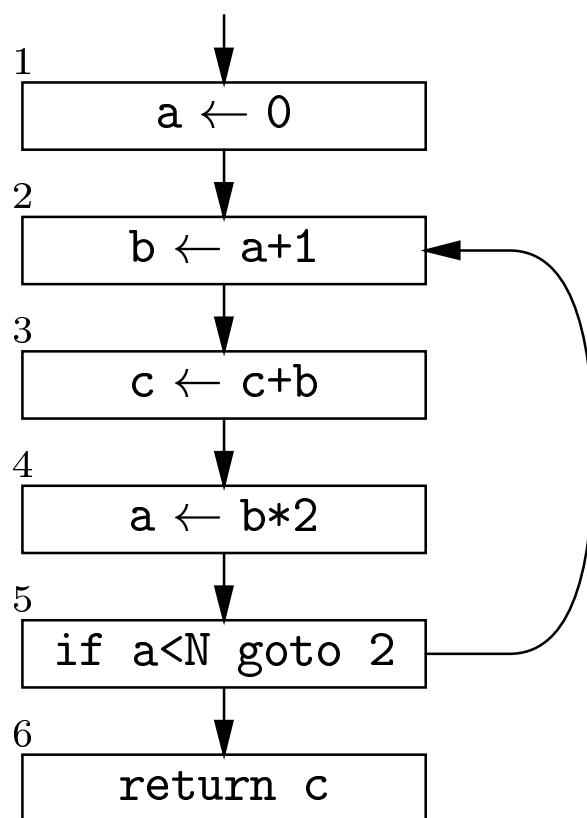
- Försök färglägga grafen med 3 färger \Rightarrow misslyckat.
- Vi kan välja att spara undan f.



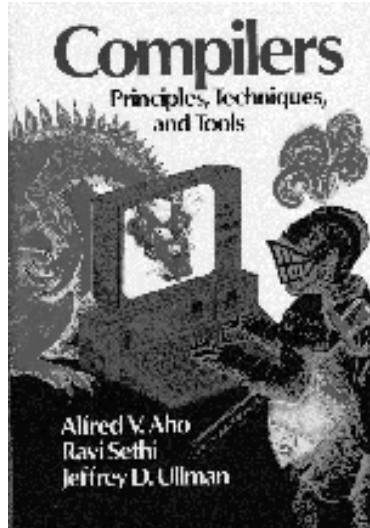
- Livstidsinformation och grafen uppdateras sedan.
- Ger sedan möjlighet för heuristiken att fortsätta.
- Vanligtvis undviker man att spara variabler som ligger i inre loopar.

Övningsuppgifter

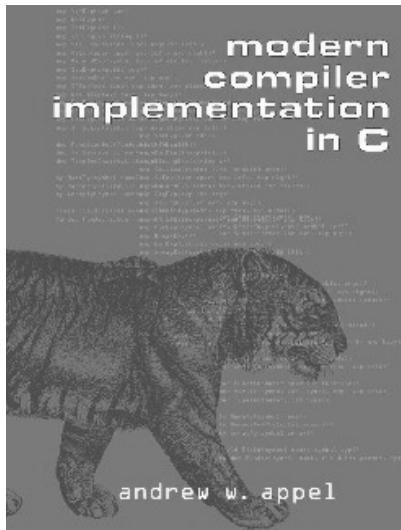
1. Vilka instruktioner produceras om man istället använder regel 5 istället för regel 6 på sidan 152.
2. Beräkna livstider och undersök hur många register som behövs för a , b och c i följande kontrollflödesgraf.



8 Referenser

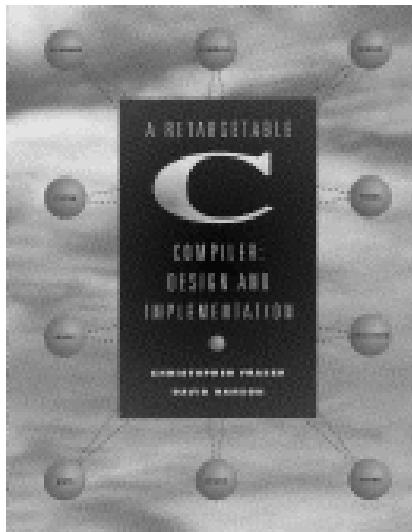


- [ASU85]
 - Klassisk kursbok i mer än 20 år.
 - Behandlar *front-end* väldigt bra.
 - Inte speciellt bra beskrivning av *back-end*.
 - Många exempel.
 - Nästan obligatorisk litteratur.



- [App98]
 - Modern, mer information om *back-end*.
 - Mer betoning på implementation än [ASU85].
 - Finns i tre versioner: C, Java och SML.
 - Visar implementation av ett OO-språk.
 - Mycket erfaren författare.

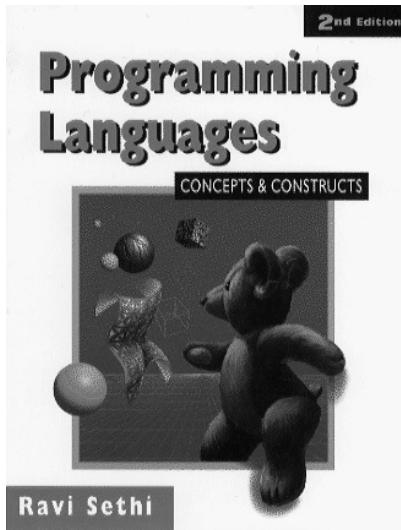
<http://www.cs.princeton.edu/~appel/modern/c/>



- [FH95]
 - Beskriver en fullständig C-kompilator, lcc.
 - Välskriven kod (s k literate programming).
 - Demonstrarer kodgenerator-generatorn lburg.
 - lcc-systemet finns att hämta på Internet.

<http://www.cs.princeton.edu/software/lcc/>

- [SJ85]
 - För den som vill lära sig Lex och Yacc.
 - Bra handledning för nybörjare.
 - Utvecklar en liten C-subset-kompilator.
 - Har dock en del tryckfel.
 - Källkod finns på:
<ftp://a.cs.uiuc.edu/pub/friedman/tar>
- [LMB92]
 - Bra referensmanual för Lex och Yacc.
 - Många exempel.
 - Källkod kan hämtas på [ftp.ora.com](ftp://ora.com).
- [Muc97]
 - Endast för erfarna kompilatorskrivare.
 - Fokuserar på kodgenerering och -optimering.



- [Set96]
 - Presenterar olika programspråksparadigmer.
 - Visar de fundamentala koncepten mycket bra.

<http://cm.bell-labs.com/who/ravi/teddy/>

- [Kam90]

- Visar implementationer av interpretatorer för Lisp, APL, Scheme, SASL, CLU, Smalltalk och Prolog.
- Bra komplement till [Set96].
- Boken skriven för Pascal, men finns portat till C.

<http://www-sal.cs.uiuc.edu/~kamin/pubs/>

- comp.compilers

- En Usenet-grupp för diskussioner om kompilatorer.
- Gamla inlägg finns arkiverade på

<http://compilers.iecc.com/comparch/compsearch>

Referenser

- [App98] A. W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, ISBN 0-521-58388-8, 1998.
- [ASU85] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison Wesley, ISBN 0-201-10088-6, 1985.
- [FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing Company, ISBN 0-8053-1670-1, 1995.
- [Kam90] S. Kamin. *Programming Languages: An Interpreter-Based Approach*. Addison Wesley, ISBN 0-201-06824-9, 1990.

- [LMB92] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly and Associates, ISBN 1-56592-000-7, 1992.
- [Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, ISBN 1-55860-320-4, 1997.
- [Set96] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison Wesley, ISBN 0-201-59065-4, 1996.
- [SJ85] A. T. Schreiner and H. G. Friedman Jr. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, ISBN 0-13-474396-2, 1985.