

Berekening van de address environments ρ

We definiëren functies *elab_...* voor 3 gevallen: parameter specificaties, variabele declaraties en procedure declaraties. Deze functies berekenen ρ voor een rij van dergelijke declaraties. De vorm van de definities reflecteert de manier waarop het environment ρ stapsgewijs opgebouwd wordt.

$\text{Addr_Env} = \text{Name} \longrightarrow \text{Addr} \times \text{ST}$ (defining occ afgebeeld op **adres** en **nesting diepte**)

Vorm van de definities:

elab_specs ($\text{spec}_1; \text{spec}_2; \text{spec}_3$) $\rho \quad n_a \quad st = \text{elab_specs}$ ($\text{spec}_2; \text{spec}_3$) $\rho' \quad n_a' \quad st'$
waar $\rho' \quad n_a' \quad st'$ gewijzigde versies zijn van $\rho \quad n_a \quad st$,
verkregen door spec_1 in rekening te brengen.

Uitbreidingen:

- voor namen van variabelen geeft ρ ook de nesting diepte st van de defining occurrence.
- procedurenamen worden gebonden aan een symbolisch label en een nesting diepte.
- geheugenallocatie: aanpassen van n_a , het eerstvolgende beschikbare adres

Eerste geval: parameter specificaties

($f[b/a]$ staat voor: f , gewijzigd door a op b af te beelden)

$$\text{elab_specs} : \text{Spec}^* \times \text{Addr_Env} \times \text{Addr} \times \text{ST} \rightarrow \text{Addr_Env} \times \text{Addr}$$
$$\text{elab_specs} (\text{var } x : t; \text{specs}) \rho \ n_a \ st =$$
$$\text{elab_specs specs } \rho[(n_a, st)/x](n_a + 1) \ st$$
$$\text{elab_specs} (\text{value } x : \text{array}[u_1..o_1, \dots, u_k..o_k] \text{ of } t'; \text{specs}) \rho \ n_a \ st =$$
$$\text{elab_specs specs } \rho'(n_a + 3k + 2) \ st \text{ where}$$
$$\rho' = \rho[(n_a, st)/x][(n_a + 2i + 1, st)/u_i]_{i=1}^k [(n_a + 2i + 2, st)/o_i]_{i=1}^k$$
$$\text{elab_specs} (\text{value } x : t; \text{specs}) \rho \ n_a \ st =$$
$$\text{elab_specs specs } \rho[(n_a, st)/x](n_a + \text{size}(t)) \ st \quad \text{for static types } t$$
$$\text{elab_specs } () \rho \ n_a \ st = (\rho, n_a)$$

De keywords **var** en **value** duiden op de soort parameters

Voor arrays is hier het gebruik van een descriptor verondersteld.

x komt overeen met de eerste plaats van de descriptor, en ook de locaties van u_i en o_i zijn bekend omdat we de layout van de descriptor kennen

Tweede geval: variabele declaraties

$$elab_vdecls : Vdecl^* \times Addr_Env \times Addr \times ST \rightarrow Addr_Env \times Addr$$

$$elab_vdecls (\text{var } x : t; vdecls) \rho \ n_a \ st = \quad \text{for non-array types } t$$

$$elab_vdecls \ vdecls \ \rho[(n_a, st)/x] \ (n_a + size(t)) \ st$$

$$elab_vdecls (\text{var } x : \text{array}[u_1..o_1, \dots, u_k..o_k] \text{ of } t; vdecls) \rho \ n_a \ st =$$

$$elab_vdecls \ vdecls \ \rho[(n_a, st)/x]$$

$$\left(\underbrace{n_a + 3k + 2}_{\text{Space for the descriptor}} + \underbrace{\sum_{i=1}^k (o_i - u_i + 1) \cdot size(t)}_{\text{Space for the array components}} \right) st$$

Space for
the descriptor

Space for the
array components

if x is a static array.

Enkel voor statische
arrays; descriptor en array
componenten achter
elkaar opgeslagen!

$$elab_vdecls () \rho \ n_a \ st = (\rho, n_a)$$

Derde geval: procedure-declaraties

We voorzien de mogelijkheid van wederzijdse recursie.

Procedure-namen worden gebonden aan een symbolisch label en een nesting diepte. De labels kunnen maar bepaald worden als de code gegenereerd is, daarom bevat het resultaat nu ook de code.

$$elab_pdecls : Pdecl^* \times Addr_Env \times ST \rightarrow Addr_Env \times Code$$
$$elab_pdecls (\text{proc } p_1(\dots); \dots;$$
$$\vdots$$
$$\text{proc } p_k(\dots); \dots;) \rho \ st =$$
$$(\rho', \quad l_1 : code(\text{proc } p_1(\dots); \dots) \ \rho' \ st + 1;$$
$$\vdots$$
$$l_k : code(\text{proc } p_k(\dots); \dots) \ \rho' \ st + 1)$$
$$\text{where } \rho' = \rho[(l_1, st)/p_1, \dots, (l_k, st)/p_k]$$

Nesting diepte verhoogt

Vertaling van procedures en procedure calls

Dit omvat uiteraard ook het mechanisme voor parameter passing mechanism.

Geheugenorganisatie voor procedures: “macro” stack met een stack frame voor elke incarnatie. Locale stack op de top van de frame.

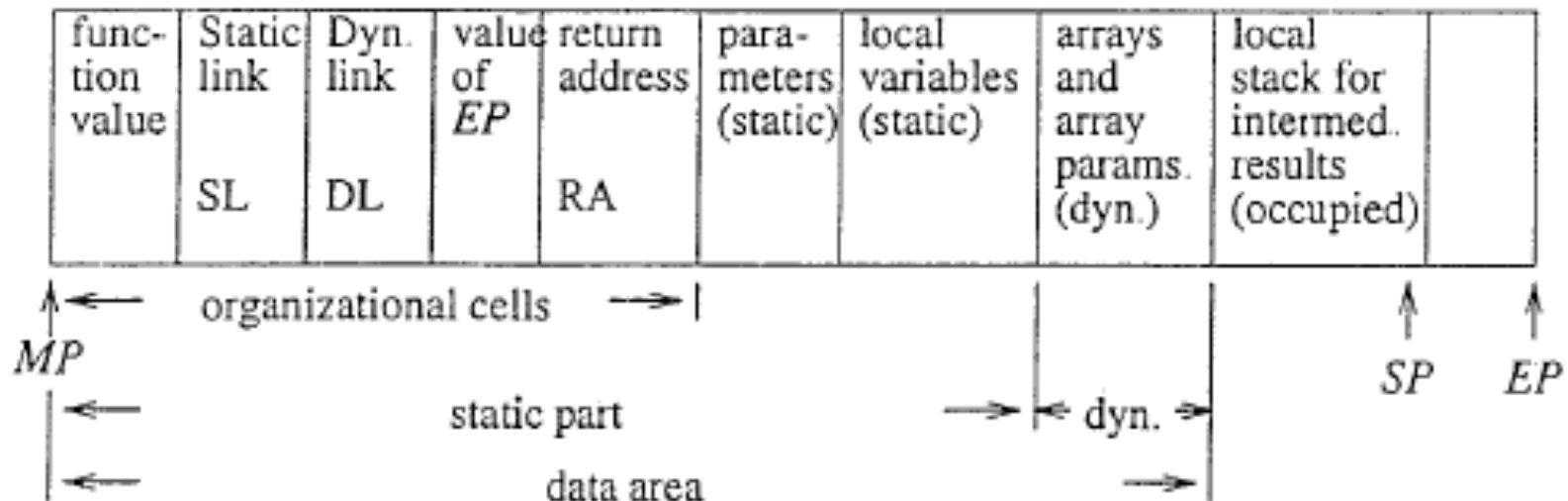


Figure 2.10 The highest stack frame.

De acties die uitgevoerd moeten zijn vóór een door p opgeroepen procedure q actief mag worden:

1. De statische link moet gezet worden: hij moet verwijzen naar het frame van de meest recente incarnatie van de procedure die q direct omsluit
2. De dynamische link moet gezet worden: hij moet verwijzen naar het stackframe van p
3. De actuele waarde van EP moet bewaard worden
4. De waarden en adressen van de actuele parameters moeten bepaald en bewaard worden. I.h.b. moeten descriptors gemaakt worden voor formele “value” array parameters (de kopieën van de overeenkomstige actuele array parameters worden later gecreëerd, zie verder)
5. Het MP register moet verhoogd worden zodat het verwijst naar het nieuwe stackframe
6. Het returnadres moet bewaard worden
7. Er moet gesprongen worden naar de eerste instructie van de code voor q
8. SP moet wijzen naar het einde van de ruimte voor statische parameters en locale waarden
9. De kopieën van de actuele “value” array parameters moeten gecreëerd worden; hier wordt de grootte van een dergelijke parameter berekend met behulp van zijn descriptor en wordt de waarde van SP daarmee verhoogd
10. De nieuwe waarde van EP moet berekend worden om de ruimte nodig voor de lokale stack in rekening te brengen. Er wordt getest of er een “collision” is van de stack met de heap.

Table 2.12 Instructions for calling and entering procedures, **mst** (mark stack), **cup** (call user procedure), **ssp** (set stack pointer) and **sep** (set extreme stack pointer). $base(p, a) =$ if $p = 0$ then a else $base(p - 1, STORE[a + 1])$ fi.

Instr.	Meaning	Comments
mst p	$STORE[SP + 2] := base(p, MP);$ $STORE[SP + 3] := MP;$ $STORE[SP + 4] := EP;$ $SP := SP + 5$	Static link Dynamic link Save EP The parameters can now be evaluated starting from $STORE[SP + 1]$
cup $p\ q$	$MP := SP - (p + 4);$ $STORE[MP + 4] := PC;$ $PC := q$	p is the storage requirement for the parameters Save return address Branch to procedure start address q
ssp p	$SP := MP + p - 1$	p size of static part of data area
sep p	$EP := SP + p;$ if $EP \geq NP$ then <i>error</i> ('store overflow') fi	p max. depth of local stack Check for collision of stack and heap

1. De statische link moet gezet worden: hij moet verwijzen naar het frame van de meest recente incarnatie van de procedure die q direct omsluit
2. De dynamische link moet gezet worden: hij moet verwijzen naar het stackframe van p
3. De actuele waarde van EP moet bewaard worden

mst

4. De waarden en adressen van de actuele parameters moeten bepaald en bewaard worden. I.h.b. moeten descriptors gemaakt worden voor formele “value” array parameters (de kopieën van de overeenkomstige actuele array parameters worden later gecreëerd, zie verder)

caller

5. Het MP register moet verhoogd worden zodat het verwijst naar het nieuwe stackframe
6. Het returnadres moet bewaard worden
7. Er moet gesprongen worden naar de eerste instructie van de code voor q

cup

8. SP moet wijzen naar het einde van de ruimte voor statische parameters en locale waarden

ssp

9. De kopieën van de actuele “value” array parameters moeten gecreëerd worden; hier wordt de grootte van een dergelijke parameter berekend met behulp van zijn descriptor en wordt de waarde van SP daarmee verhoogd

callee

10. De nieuwe waarde van EP moet berekend worden om de ruimte nodig voor de lokale stack in rekening te brengen. Er wordt getest of er een “collision” is van de stack met de heap.

sep

Het compilatieschema voor *call* $p(e_1, \dots, e_k)$:

```
code  $p(e_1, \dots, e_k) \rho \ st = \mathbf{mst} \ st - st';$   
       $\textit{code}_A \ e_1 \ \rho \ st;$   
       $\vdots$   
       $\textit{code}_A \ e_k \ \rho \ st;$   
cup  $s \ l$ 
```

where $\rho(p) = (l, st')$ and s is the space requirement for the actual parameters.

\textit{code}_A : code voor actuele parameters, zie later

Compilatieschema voor procedures:

code (**procedure** p ($specs$); $vdecls$; $pdecls$; $body$) ρ st =

ssp n_a'' ;	Storage requirement of static part
<i>code_p</i> $specs$ ρ' st ;	Storage requirement of dynamic part
<i>code_p</i> $vdecls$ ρ'' st ;	Create and initialize
sep k ;	k max. depth of the local stack
ujp l ;	
<i>proc_code</i> ;	Code for the local procedures
l : <i>code</i> $body$ ρ''' st ;	Code for procedure body
retp	

where

(ρ', n_a') =	<i>elab_specs</i> $specs$ ρ 5 st
(ρ'', n_a'') =	<i>elab_vdecls</i> $vdecls$ ρ' n_a' st
$(\rho''', proc_code)$ =	<i>elab_pdecls</i> $pdecls$ ρ'' st

Table 2.13 Return from function procedures and proper procedures.

Instr.	Meaning	Comments
retf	$SP := MP;$	Function result in the local stack
	$PC := STORE[MP + 4];$	Return branch
	$EP := STORE[MP + 3];$	Restore EP
	if $EP \geq NP$	
	then <i>error('store overflow')</i>	
retp	fi	
	$MP := STORE[MP + 2]$	Dynamic link
	$SP := MP - 1;$	Proper procedure with no results
	$PC := STORE[MP + 4];$	Return branch
	$EP := STORE[MP + 3];$	Restore EP
	if $EP \geq NP$	
	then <i>error('store overflow')</i>	
	fi	
	$MP := STORE[MP + 2]$	Dynamic link

Code voor actuele parameters

$\text{code}_A \ x \ \rho \ \text{st} = \text{code}_L \ x \ \rho \ \text{st}$ indien de formele parameter die met x overeenkomt een “var” parameter is (adres doorgeven)

$\text{code}_A \ e \ \rho \ \text{st} = \text{code}_R \ e \ \rho \ \text{st}$ indien de formele parameter die met x overeenkomt een “value” parameter is (waarde doorgeven)

Instructies voor het kopiëren van blokken geheugen (voor arrays, al dan niet met gebruik van een descriptor):

Instr.	Meaning	Cond.	Result
movs q	for $i := q - 1$ down to 0 do $STORE[SP+i] := STORE[STORE[SP] + i]$ od; $SP := SP + q - 1$	(a)	
movd q	for $i = 1$ to $STORE[MP + q + 1]$ do $STORE[SP + i] :=$ $STORE[STORE[MP + q]$ $+ STORE[MP + q + 2] + i - 1]$ od; $STORE[MP + q] := SP + 1 - STORE[MP + q + 2]$ $SP := SP + STORE[MP + q + 1]$		

Echt startadres =
som van
1e en 3e element
van de descriptor

movs : move blok van grootte q

movd : move blok met gebruik van een descriptor op relatief adres q

$\text{code}_A \times p_{st} = \text{code}_L \times p_{st}$ als de formele parameter die met x overeenkomt
movs g van een gestructureerd type is met statische
grootte $\text{size}(t) = g$

$$\text{code}_p(\text{value } x : \text{array}[u_1..o_1, \dots, u_k..o_k] \text{ of } t) \rho \text{ st} = \text{movd ra}$$

kopieert de array, waar $\rho(x) = (ra, st)$
 en de caller heeft de descriptor van de actuele array
 parameter al gekopieerd met behulp van **movs**

code_p voor lokale dynamische arrays (cf. schema voor procedures, lijn 4):
analoog

De L-value van zowel lokale als niet-lokale variabelen kan op een uniforme wijze verkregen worden:

$code_L(x\ r)\ \rho\ st =$ **lda** $d\ ra$;
 $code_M\ r\ \rho\ st$,
 where $\rho(x) = (ra, st')$ and $d = st - st'$, if x is a
 variable or a formal value parameter,

$code_L(x\ r)\ \rho\ st =$ **lod** $a\ d\ ra$;
 $code_M\ r\ \rho\ st$
 where $\rho(x) = (ra, st')$ and $d = st - st'$, if x is a
 formal var parameter

Main Program

Veronderstel dat alle registers geïnitieerd zijn op 0, enkel PC staat op -1.
De uitvoering begint dus met de instructie in Code[0].

```
code (program vdecls; pdecls; stats) 0 =  
    ssp n_a;  
    codep vdecls ρ 1;  generates code to fill  
                           array descriptors  
    sep k;                max. depth of local stack  
    ujp l;  
    proc_code;  
l: code stats ρ' 1;  
    stp  
    where   (ρ, n_a) = elab_vdecls vdecls 0 5 1  and  
            (ρ', proc_code) = elab_pdecls pdecls ρ 1
```


Dit schema bevatte een aantal vereenvoudigingen; bv we hebben geen rekening gehouden met het feit dat de benodigde constanten te groot kunnen zijn om in te vullen als parameters in de VM instructies.

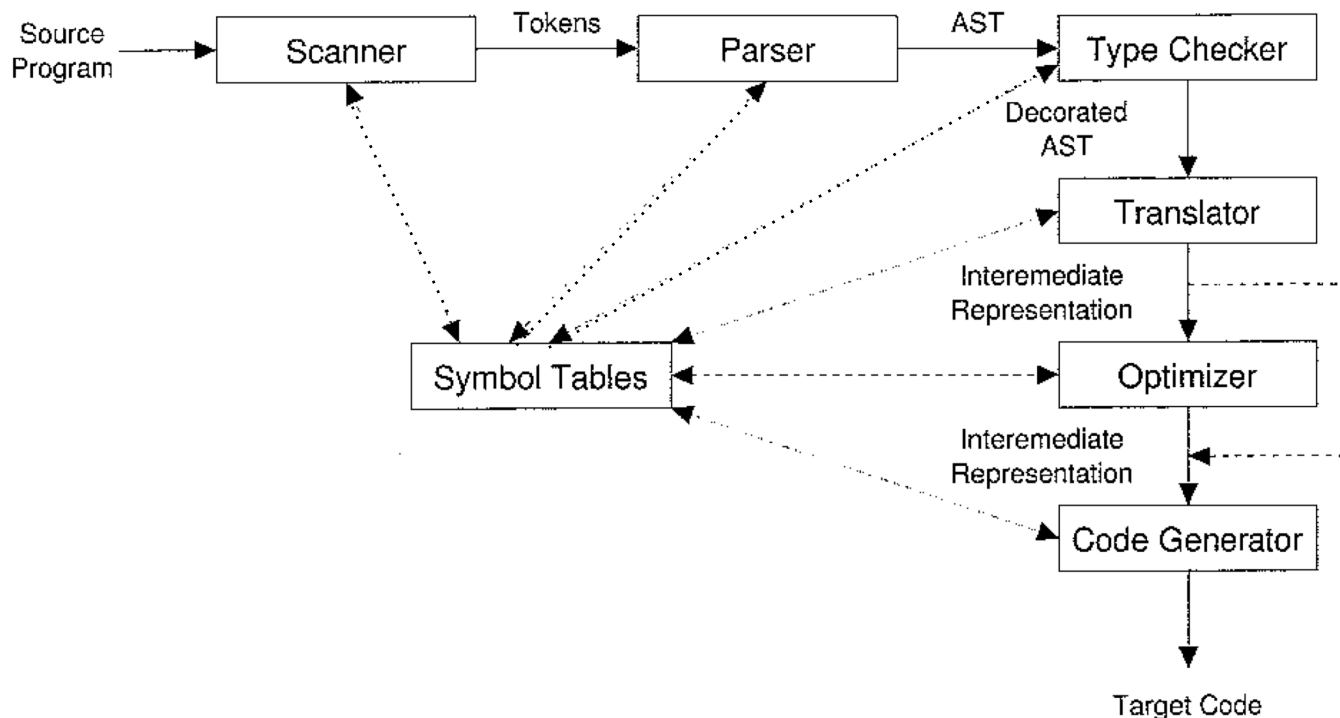
Verder gingen we ervan uit dat de verschillende argumenten bekend waren, en dat we zonder meer kunnen bepalen welke van de code – functies er telkens moet opgeroepen worden.

Alle type-informatie was verondersteld beschikbaar te zijn

...

Werking van een compiler: de fazen van het compilatieproces

Nu we in principe hebben vastgelegd hoe de gewenste vertaling eruit ziet, gaan we na hoe ze tot stand komt. Het werk wordt verdeeld over een aantal fazen met elk een welomschreven doel. Dat betekent niet dat de fazen in de tijd strict op elkaar volgen, het is slechts een logische verdeling van het werk.



Parse tree (afleidingsboom)

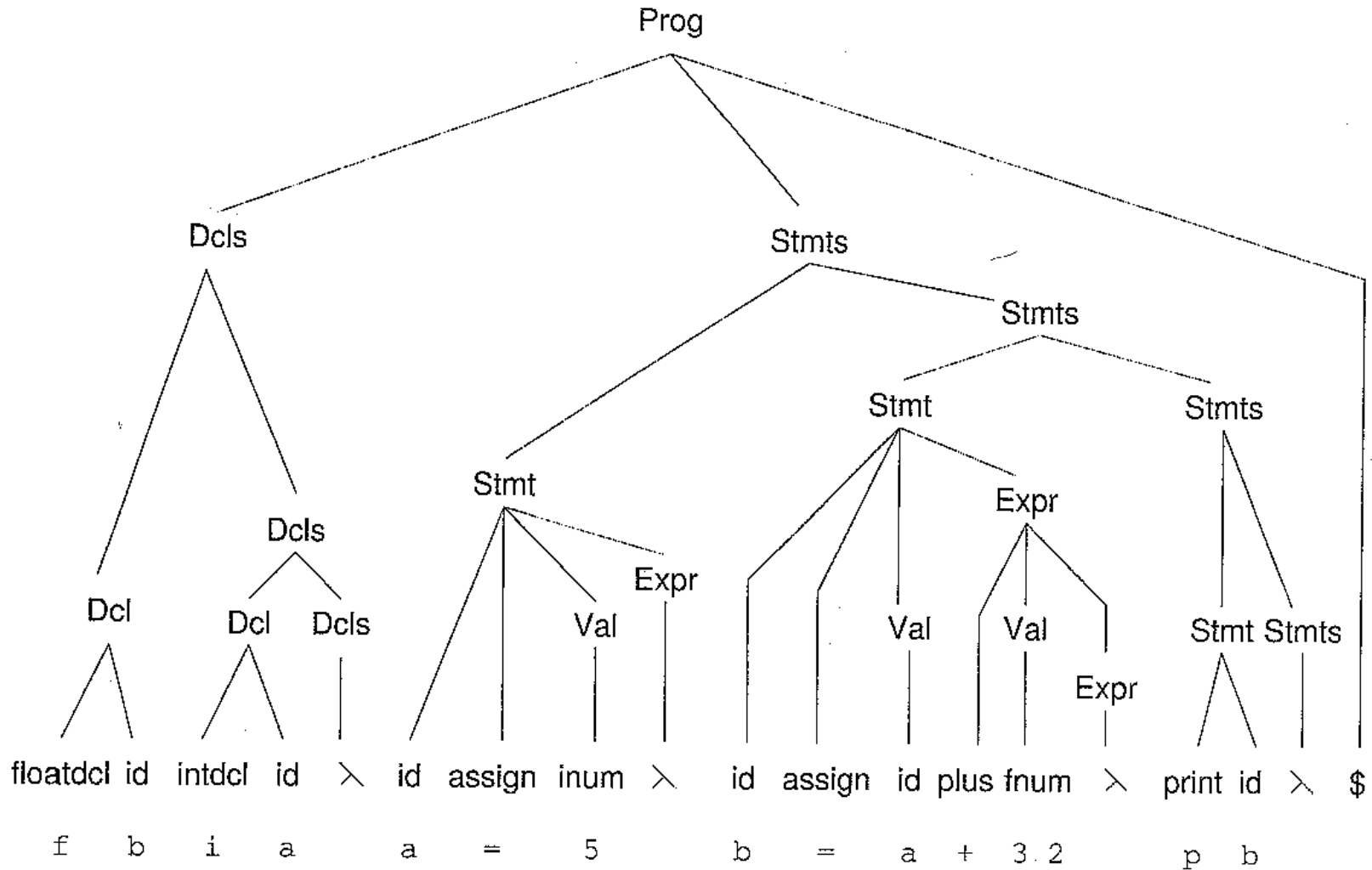
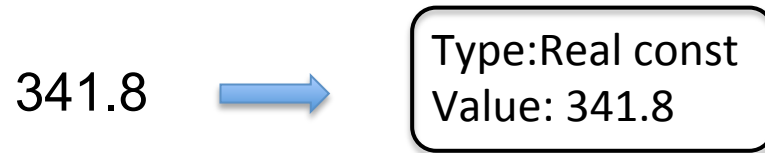


Figure 2.4: An ac program and its parse tree.

Scanning (Lexicale analyse)

- Taak:
- Groepeer karakters in “tokens”
 - Vul waarden in



Rij van karakters

token (terminaal symbool in de syntax)

De lexicale analyse is vrij eenvoudig automatisch te genereren door het gebruik van reguliere expressies en eindige automaten

We hebben, voor elk token, een precieze beschrijving nodig van zijn “lexemes” (d.w.z. de rijen karakters die ermee overeenkomen)

bv. getallen: 10 en 10.6 zijn OK, maar wat met .6 of 45. ??

strings: abcc is OK, maar wat met de lege string?

Gewoonlijk wordt dit gedaan door een reguliere expression te geven voor elke soort van token

Reguliere expressies over een alfabet Σ :

Φ voor de lege taal

Λ voor de taal die enkel het lege woord bevat

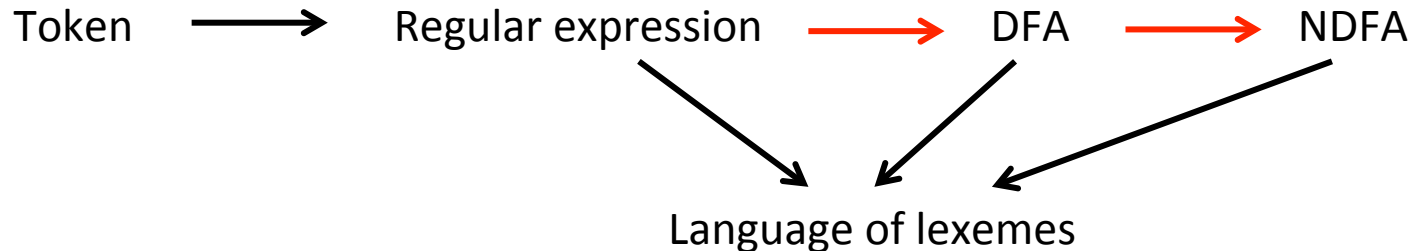
Voor elke a in Σ , a voor de taal die enkel het woord a bevat.

Als A en B reguliere expressies zijn, dan staat $A \mid B$ voor de unie,

$A.B$ of AB voor de concatenatie, en A^* voor de iteratie van hun talen.

Voordeel: elke reguliere expressie **e** kan omgezet worden in een deterministische eindige automaat (DFA) die de woorden herkent van de taal van **e**

De stappen → kunnen automatisch



hulpsymbolen: **+** (positieve iteratie)
A^k (itereer k keer)
[] (optioneel deel)
{ } (nul of meer keren herhalen)
NOT() (complement t.o.v. het alfabet)

Voorbeelden

Single line comment beginnend met //

Fixed decimal
literal, zoals 12.345

Integer literal met optioneel teken

Commentaar afgebakend met ## waarbij een enkele # toegelaten is in de commentaar

Voorbeelden


Single line comment beginnend met// $// \text{NOT(Eol)}^* \text{Eol}$

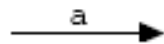
Fixed decimal
literal, zoals 12.345 $D^+.D^+$ (waar D staat voor een digit)


Integer literal met optioneel teken $(+|-|\lambda)D^+$


Commentaar afgebakend met ## waarbij
één enkele # toegelaten is in de commentaar $##((\#|\lambda)\text{NOT}(\#))^*##$

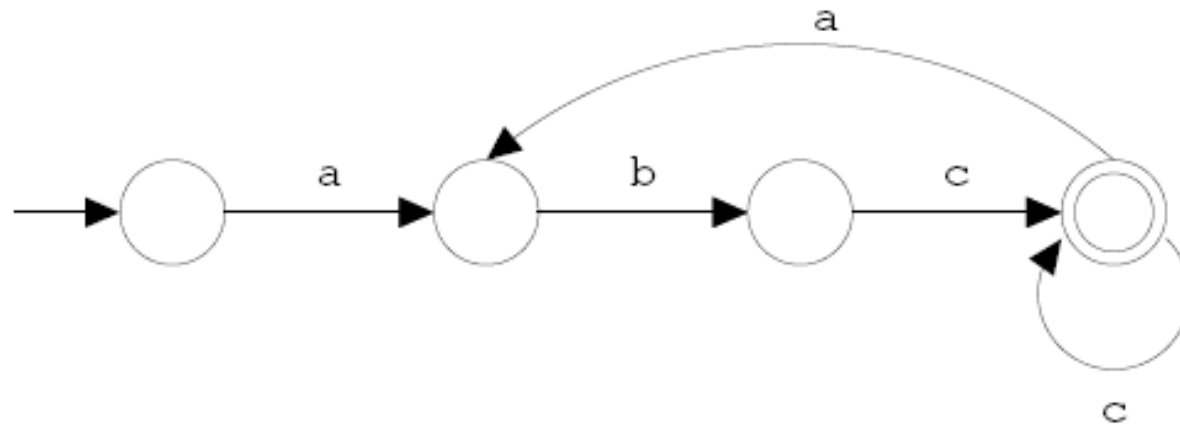
Diagram-voorstelling van automaten

 is a state

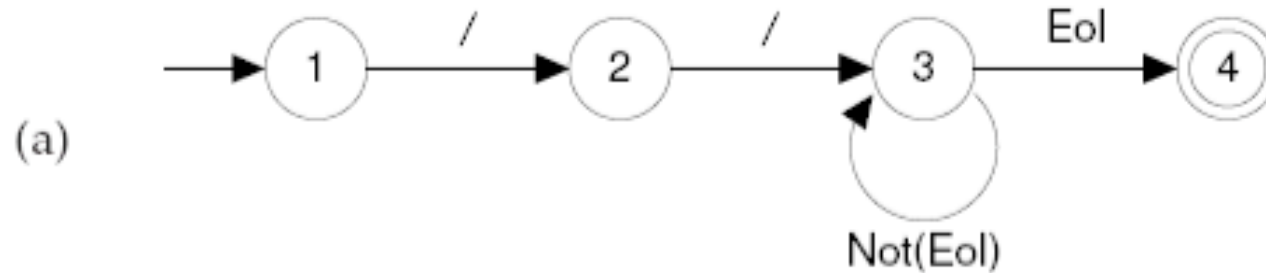
 is a transition on $a \in \Sigma$

 is the start state

 is an accepting state



Tabel-voorstelling



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Implementatie van de automaten

“Driver” programma: gebruikt de tabel om de automaat te simuleren – en dus, woorden (lexemes) te herkennen.

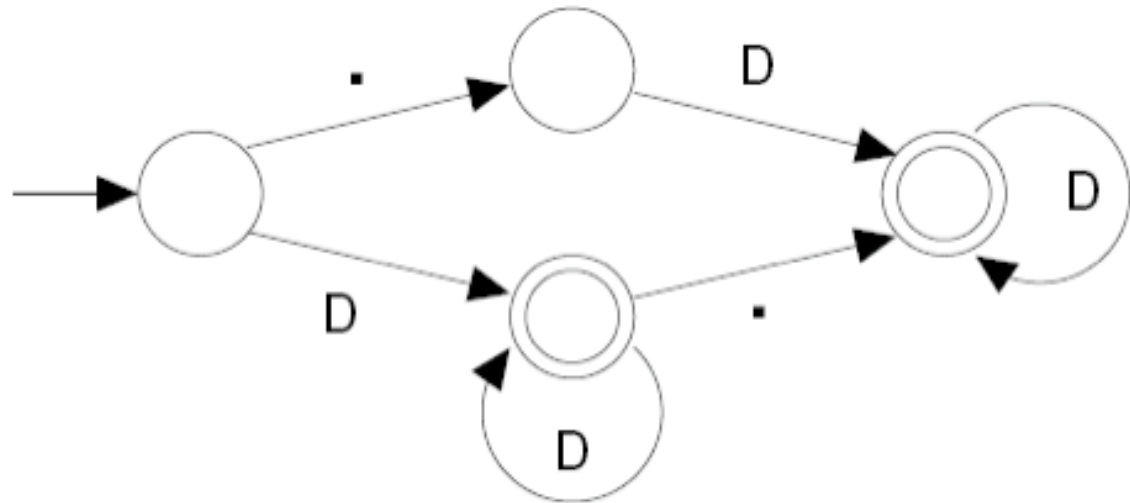
```
/* Assume CurrentChar contains the first character to be scanned */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← Read()
if State ∈ AcceptingStates
then /* Return or process the valid token */
else /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

Geschikt voor generatie van scanners!

$(D^+(\lambda | .)) \mid (D^*.D^+)$

(a)



$L(L \mid D)^*(- (L \mid D)^+)^*$

(b)

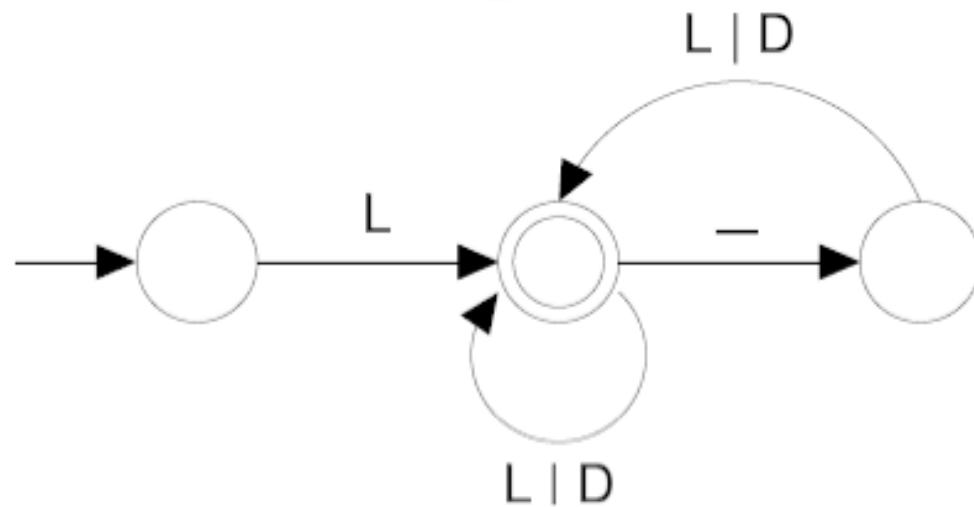


Figure 3.5: DFAs: (a) floating-point constant; (b) identifier with embedded underscore.

Automatische generatie van de scanner

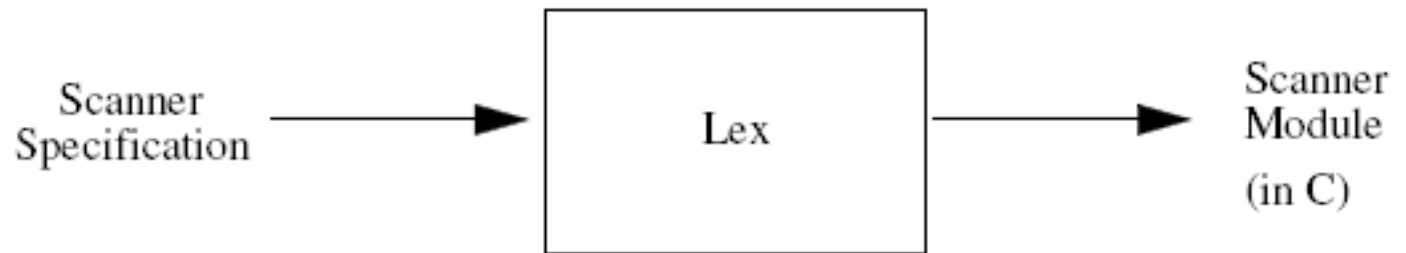


Figure 3.6: The operation of the Lex scanner generator.

```
%%  
f      { return(FLOATDCL); }  
i      { return(INTDCL); }  
p      { return(PRINT); }  
%%
```

Figure 3.7: A Lex definition for ac's reserved words.

```
declarations  
%%  
regular expression rules  
%%  
subroutine definitions
```

Figure 3.8: The structure of Lex definition files.

Character Class	Set of Characters Denoted
[abc]	Three characters: a, b, and c
[cba]	Three characters: a, b, and c
[a-c]	Three characters: a, b, and c
[aabbcc]	Three characters: a, b, and c
[^abc]	All characters except a, b, and c
[\^-\]]	Three characters: ^, -, and]
[^]	All characters
"[abc]"	Not a character class. This is an example of one five-character <i>string</i> : [abc].

Figure 3.9: Lex character class definitions.

```
%%
[a-eghj-oq-z]      { return(ID); }
%%
```

Figure 3.10: A Lex definition for ac's identifiers.

```
%%
(" ")+              { /* delete blanks */}
f                    { return(FLOATDCL); }
i                    { return(INTDCL); }
p                    { return(PRINT); }
[a-eghj-oq-z]        { return(ID); }
([0-9]+)|([0-9]+ "." [0-9]+) { return(NUM); }
"="                  { return(ASSIGN); }
"+"                  { return(PLUS); }
"_"                  { return(MINUS); }
%%
```

Figure 3.11: A Lex definition for ac's tokens.

%%	
Blank	" "
Digits	[0-9]+
Non_f_i_p	[a-eghj-oq-z]
%%	
{Blank}+	{ /* delete blanks */ }
f	{ return(FLOATDCL); }
i	{ return(INTDCL); }
p	{ return(PRINT); }
{Non_f_i_p}	{ return(ID); }
{Digits} ({Digits}"."{Digits})	{ return(NUM); }
"="	{ return(ASSIGN); }
"+"	{ return(PLUS); }
"_"	{ return(MINUS); }
%%	

Figure 3.12: An alternative definition for ac's tokens.

Wat indien een lexeme een prefix is van een ander, of wanneer een lexeme overeenkomt met meerdere tokens?

Regels vastleggen voor de keuze, bv:

- De **langste** match telt
- De **eerste** mogelijkheid (in de orde van de definitie) wordt verkozen.
De orde van definitie heeft dus belang.

Soms is **lookahead** nodig:

In het volgende geval zou input 10..100 ertoe leiden dat de lexer vast raakt na 3 karakters

Oplossing: ga terug, schakel over naar volgende mogelijkheid.

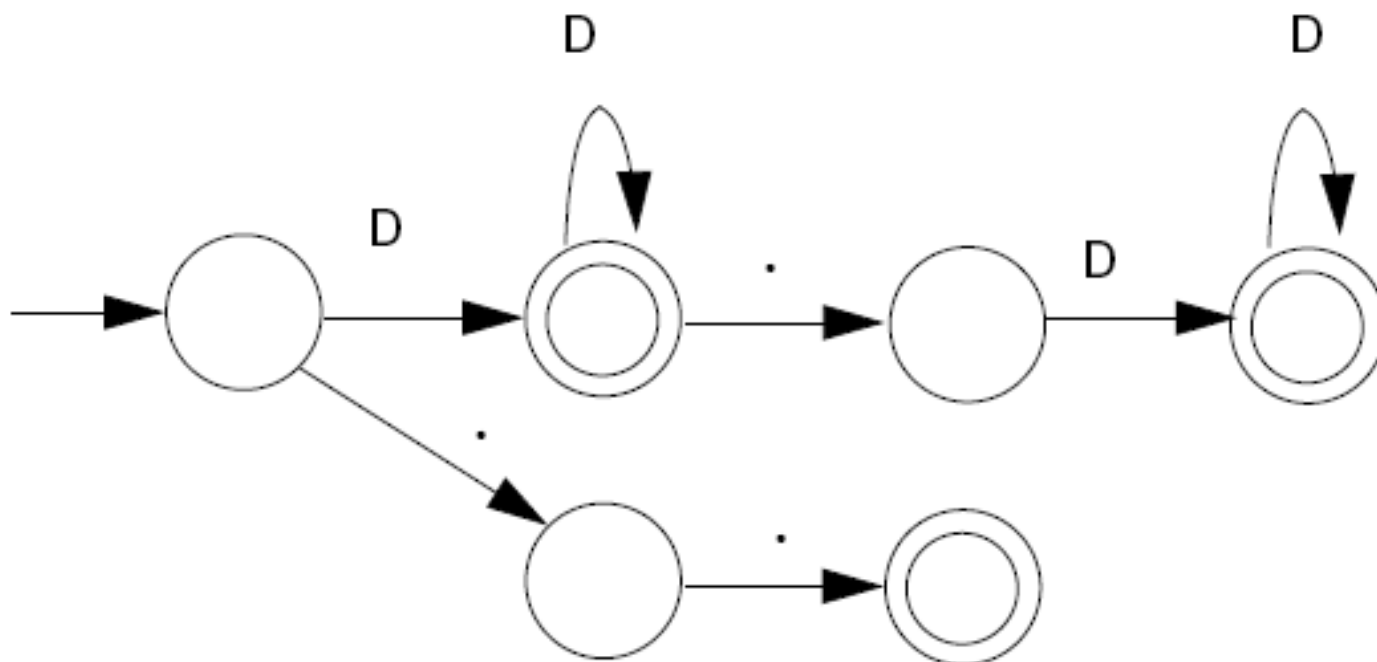


Figure 3.14: An FA that scans integer and real literals and the subrange operator.

Deterministisch maken van een NDFA:
2 bronnen van niet-determinisme

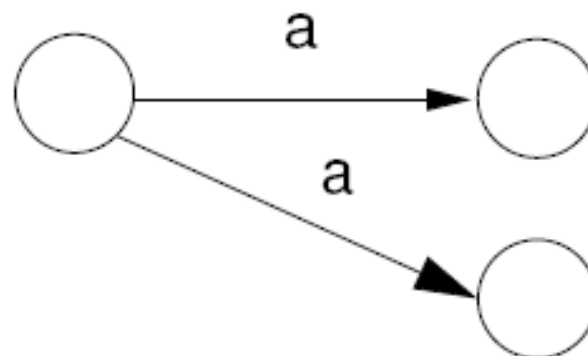


Figure 3.17: An NFA with two a transitions.

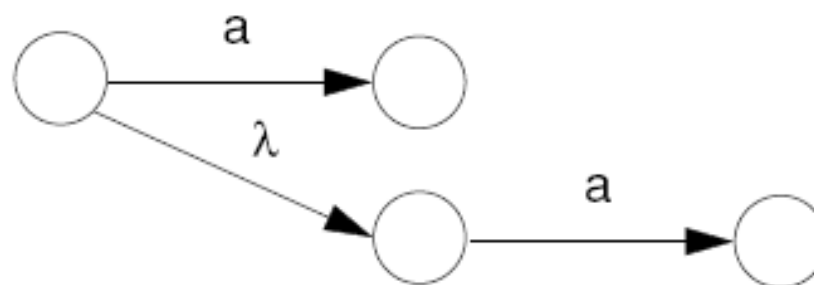


Figure 3.18: An NFA with a λ transition.

Van reguliere expressie tot NDFA

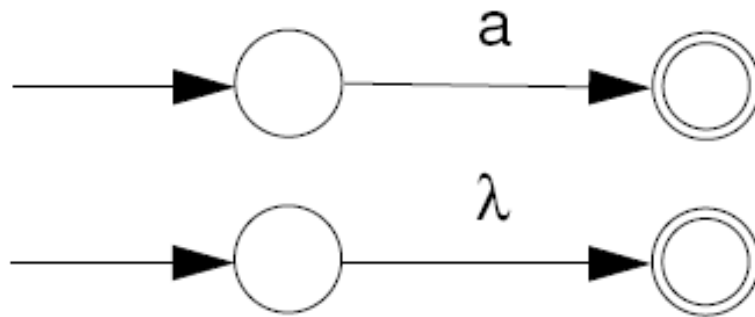


Figure 3.19: NFAs for a and λ .

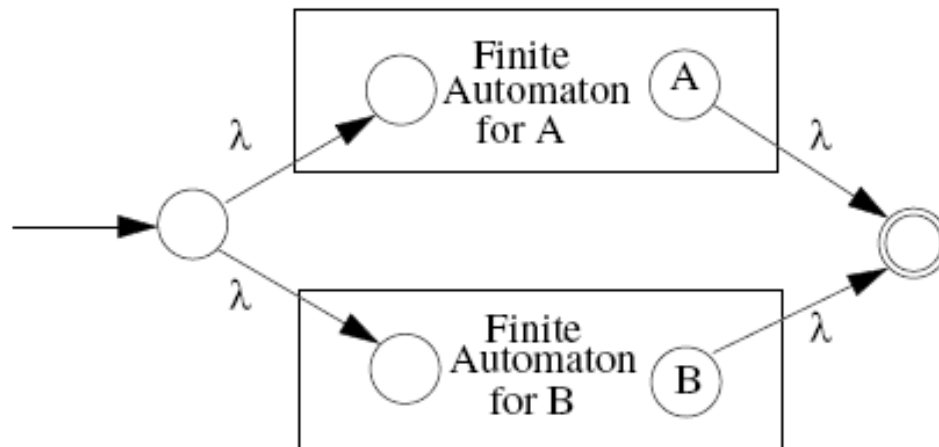


Figure 3.20: An NFA for $A \mid B$.

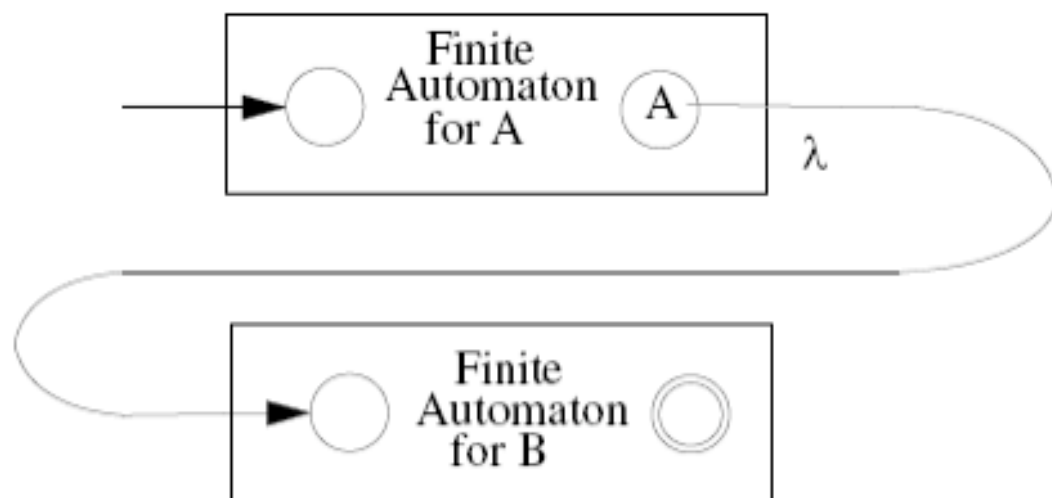


Figure 3.21: An NFA for AB .

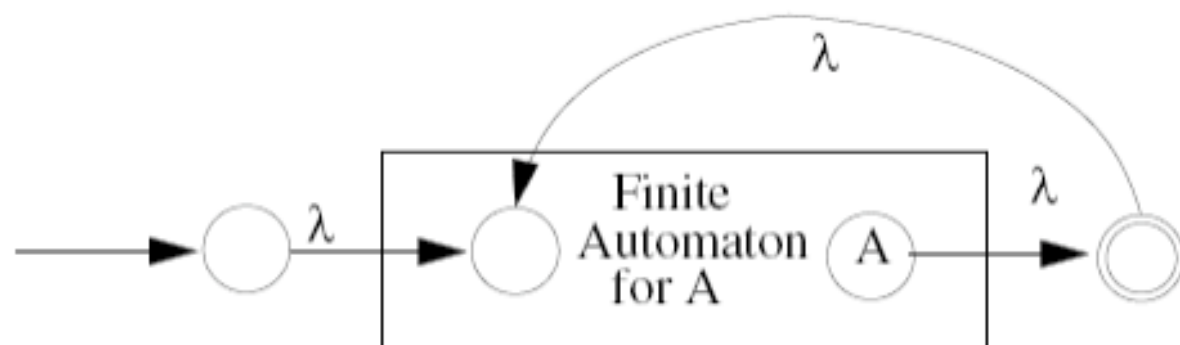


Figure 3.22: An NFA for A^* .

```

function MAKEDETERMINISTIC(N) returns DFA
    D.StartState  $\leftarrow$  RECORDSTATE(N.StartState)
    foreach S  $\in$  WorkList do
        WorkList  $\leftarrow$  WorkList  $- \{S\}$ 
        foreach c  $\in \Sigma$  do D.T(S, c)  $\leftarrow$  RECORDSTATE( $\bigcup_{s \in S} N.T(s, c)$ )

    D.AcceptStates  $\leftarrow \{S \in D.States \mid S \cap N.AcceptStates \neq \emptyset\}$ 
end

function CLOSE(S, T) returns Set
    ans  $\leftarrow S$ 
    repeat
        changed  $\leftarrow$  false
        foreach s  $\in$  ans do
            foreach t  $\in T(s, \lambda)$  do
                if t  $\notin$  ans
                    then
                        ans  $\leftarrow ans \cup \{t\}$ 
                        changed  $\leftarrow$  true
    until not changed
    return (ans)
end

function RECORDSTATE(s) returns Set
    s  $\leftarrow$  CLOSE(s, N.T)
    if s  $\notin$  D.States
        then
            D.States  $\leftarrow D.States \cup \{s\}$ 
            WorkList  $\leftarrow WorkList \cup \{s\}$ 
    return (s)
end

```

Bereken “sluiting”: voeg alle staten toe die met enkel λ -stappen bereikt kunnen worden

Figure 3.23: Construction of a DFA *D* from an NFA *N*.

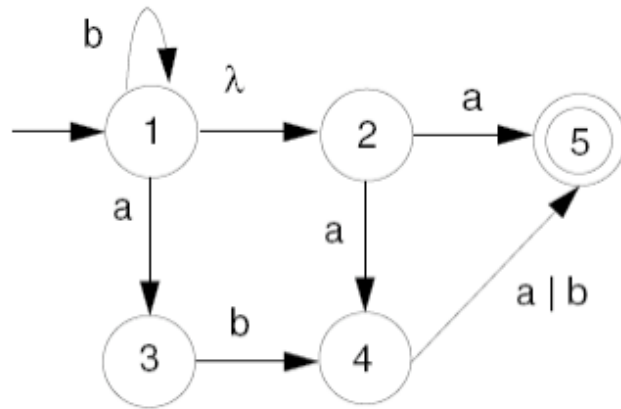


Figure 3.24: An NFA showing how subset construction operates.

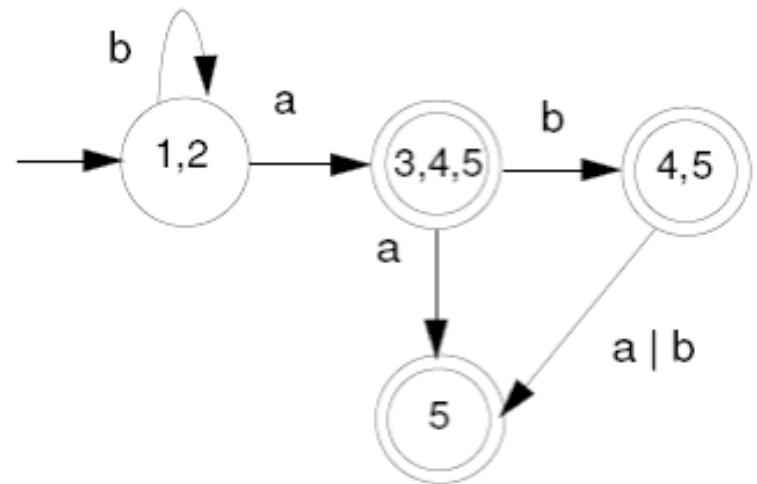


Figure 3.25: DFA created for NFA of Figure 3.24.

Optimizing a DFA (minimizing the number of states)

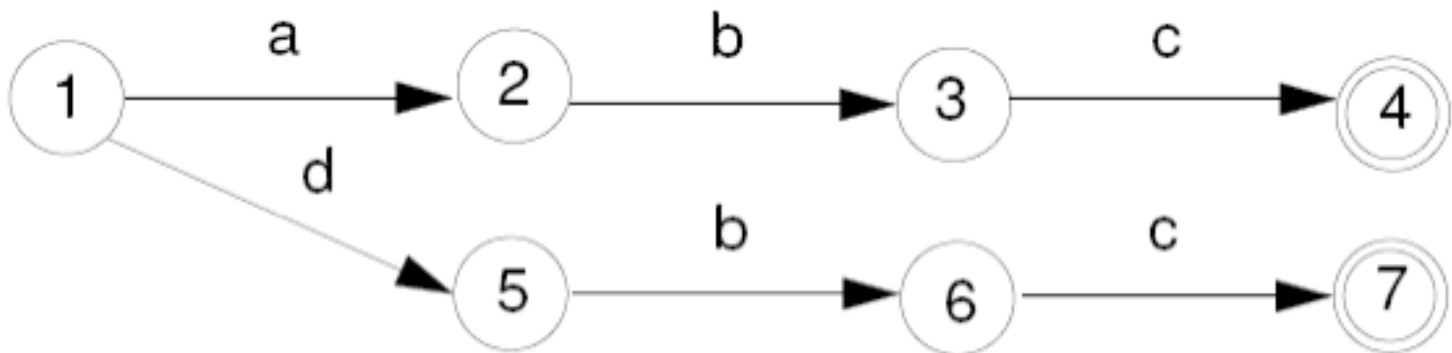


Figure 3.26: Example FA before merging.

Merge accepting/non-accepting states, then split if necessary

```

procedure SPLIT(MergedStates)
  repeat
    changed  $\leftarrow$  false
    foreach  $S \in \text{MergedStates}, c \in \Sigma$  do
      targets  $\leftarrow \bigcup_{s \in S} \text{TARGETBLOCK}(s, c, \text{MergedStates})$ 
      if  $|\text{targets}| > 1$ 
      then
        changed  $\leftarrow$  true
        foreach  $t \in \text{targets}$  do
          newblock  $\leftarrow \{s \in S \mid \text{TARGETBLOCK}(s, c, \text{MergedStates}) = t\}$ 
          MergedStates  $\leftarrow \text{MergedStates} \cup \{\text{newblock}\}$ 
          MergedStates  $\leftarrow \text{MergedStates} - \{S\}$ 
    until not changed
  end

function TARGETBLOCK( $s, c, \text{MergedStates}$ ) returns MergedState
  return  $\{B \in \text{MergedStates} \mid T(s, c) \in B\}$ 
end

```

Figure 3.27: An algorithm to split FA states.

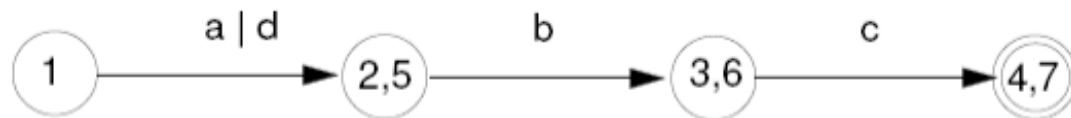


Figure 3.28: The minimum state automaton for Figure 3.26.
