Fungsi Dasar Kompilator

Grammar
Analysis Leksikal
Analysis Syntaks
Pembangkitan Kode

Bahasa Pemrograman Tingkat Tinggi

- Bahasa pemrograman tingkat tinggi dideskripsikan dalam grammar, yang menspesifikasi sintaks statement yang legal.
  - Statement assignment:
    - nama variabel + operator assignment + ekspresi

```
1   PROGRAM STATS
2   VAR
3        SUM,SUMSQ,I,VALUE,MEAN,VARIANCE : INTEGER
4   BEGIN
5        SUM   :=  0;
6        SUMSQ := 0;
7        FOR I := 1 TO 100 DO
8            BEGIN
9                READ(VALUE);
10               SUM := SUM + VALUE;
11               SUMSQ := SUMSQ + VALUE * VALUE
```

Kompilator

- Kompilasi: statement yang sesuai (ditulis oleh programmer) dengan struktur (didefinisikan oleh grammar) dan membuat object code yang sesuai
  - Analisis leksikal (scanning)
    - Scanning source statement, mengenali dan mengklasifikasi berbagai token, termasuk keyword, nama variabel, tipe data, operator, dsb.
  - Analisis syntaks (parsing)
    - Mengenali setiap statement sebagai suatu konstruksi bahasa yang dideskripsikan oleh grammar
  - Semantik (pembangkitan kode)
    - Pembangkitan object code

Grammar

- Grammar merupakan deskripsi formal dari sintaks.
- BNF (Backus-Naur Form):
  - Notasi yang sederhana dan banyak digunakan untuk menulis grammar yang diperkenalkan oleh John Backus dan Peter Naur pada sekitar tahun 1960.
  - Meta-symbols BNF:
    - **::=** “didefinisikan sebagai"
    - **|** "or"
    - **< >** kurung siku digunakan untuk mengapit simbol non-terminal symbols
  - Aturan BNF yang mendefinisikan nonterminal mempunyai bentuk: nonterminal **::=** urutan alternatif yang terdiri dari strings terminal (token) atau nonterminal dipisahkan oleh meta-symbol **|**

Grammar Pascal yang Disederhanakan

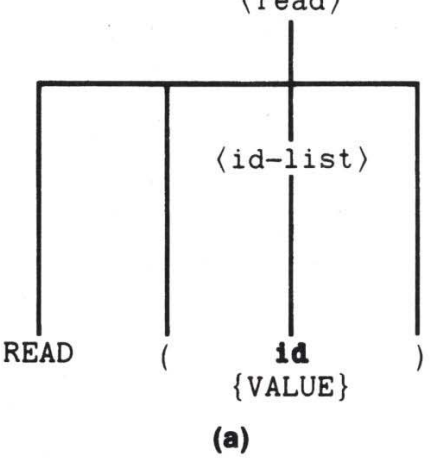
```
1  <prog>          ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END
2  <prog-name>     ::= id
3  <dec-list>      ::= <dec> | <dec-list> ; <dec>
4  <dec>           ::= <id-list> : <type>
5  <type>          ::= INTEGER
6  <id-list>       ::= id | <id-list> , id
7  <stmt-list>     ::= <stmt> | <stmt-list> ; <stmt>
8  <stmt>          ::= <assign> | <read> | <write> | <for>
9  <assign>        ::= id := <exp>
10 <exp>           ::= <term> | <exp> + <term> | <exp> - <term>
11 <term>          ::= <factor> | <term> * <factor> | <term> DIV <factor>
12 <factor>        ::= id | int | ( <exp> )
13 <read>          ::= READ ( <id-list> )
14 <write>         ::= WRITE ( <id-list> )
15 <for>           ::= FOR <index-exp> DO <body>
16 <index-exp>     ::= id := <exp> TO <exp>
17 <body>          ::= <stmt> | BEGIN <stmt-list> END
```

Aturan rekursif

Figure 5.2 Simplified Pascal grammar.

Parse Tree (Syntax Tree)

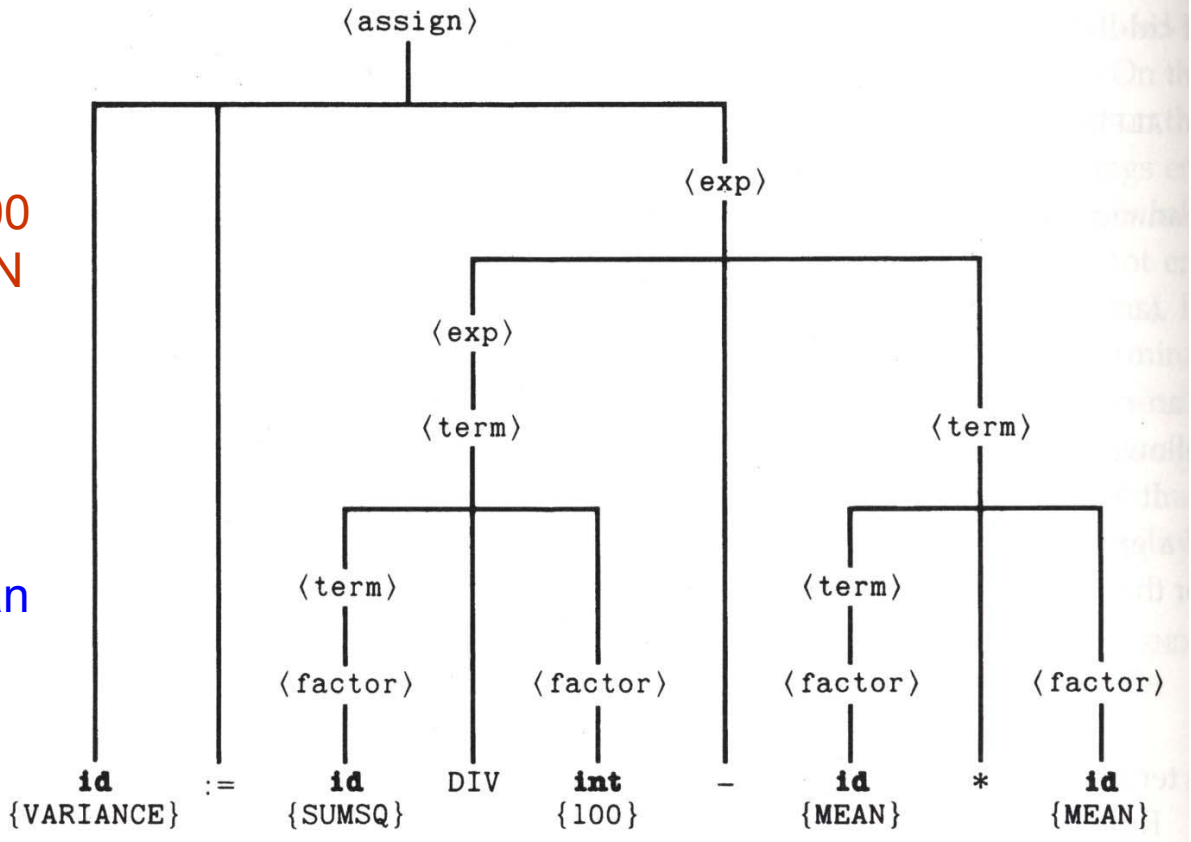
READ(VALUE)



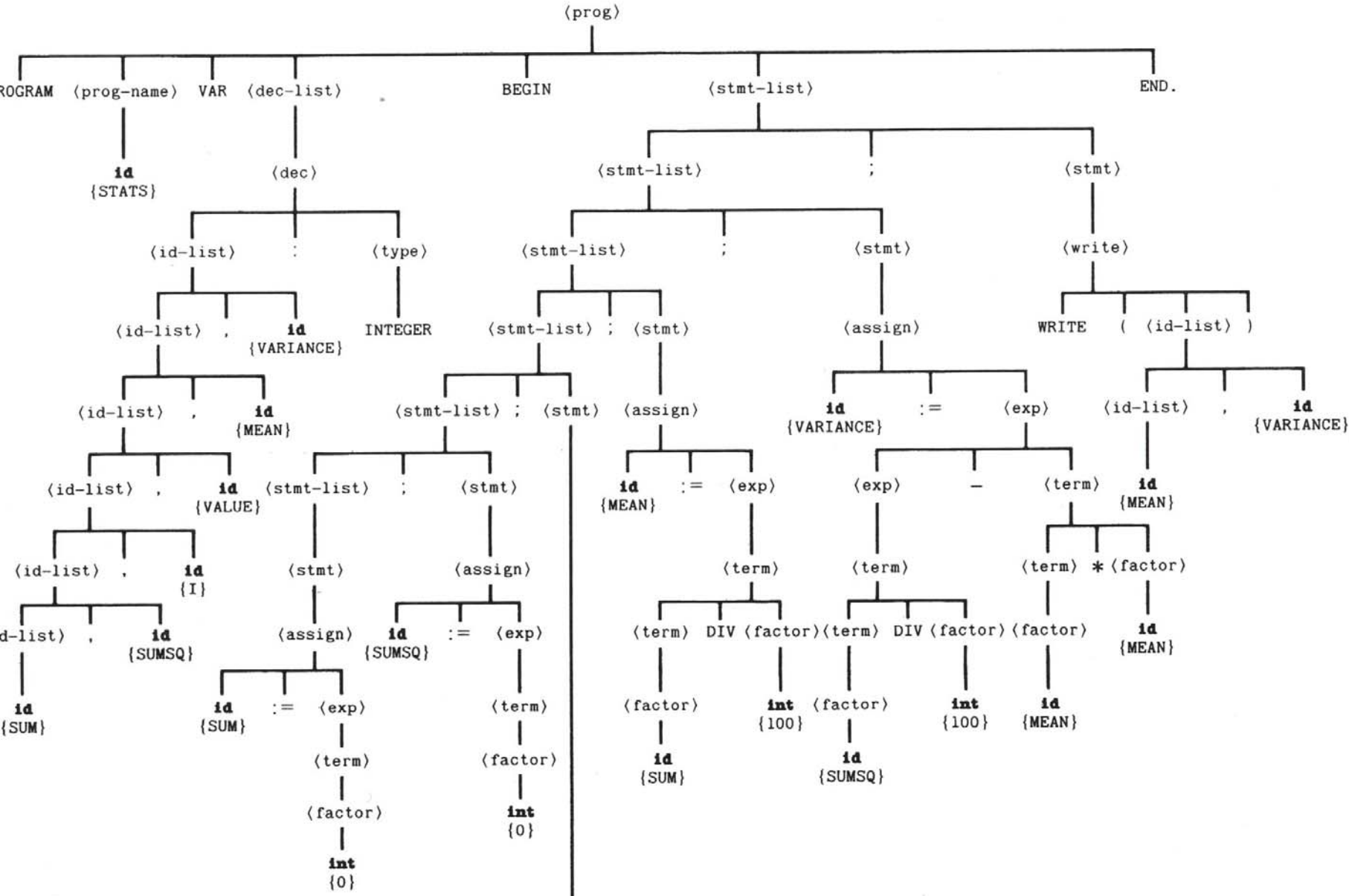
(a)

VARIANCE:=SUMSQ DIV 100 – MEAN*MEAN

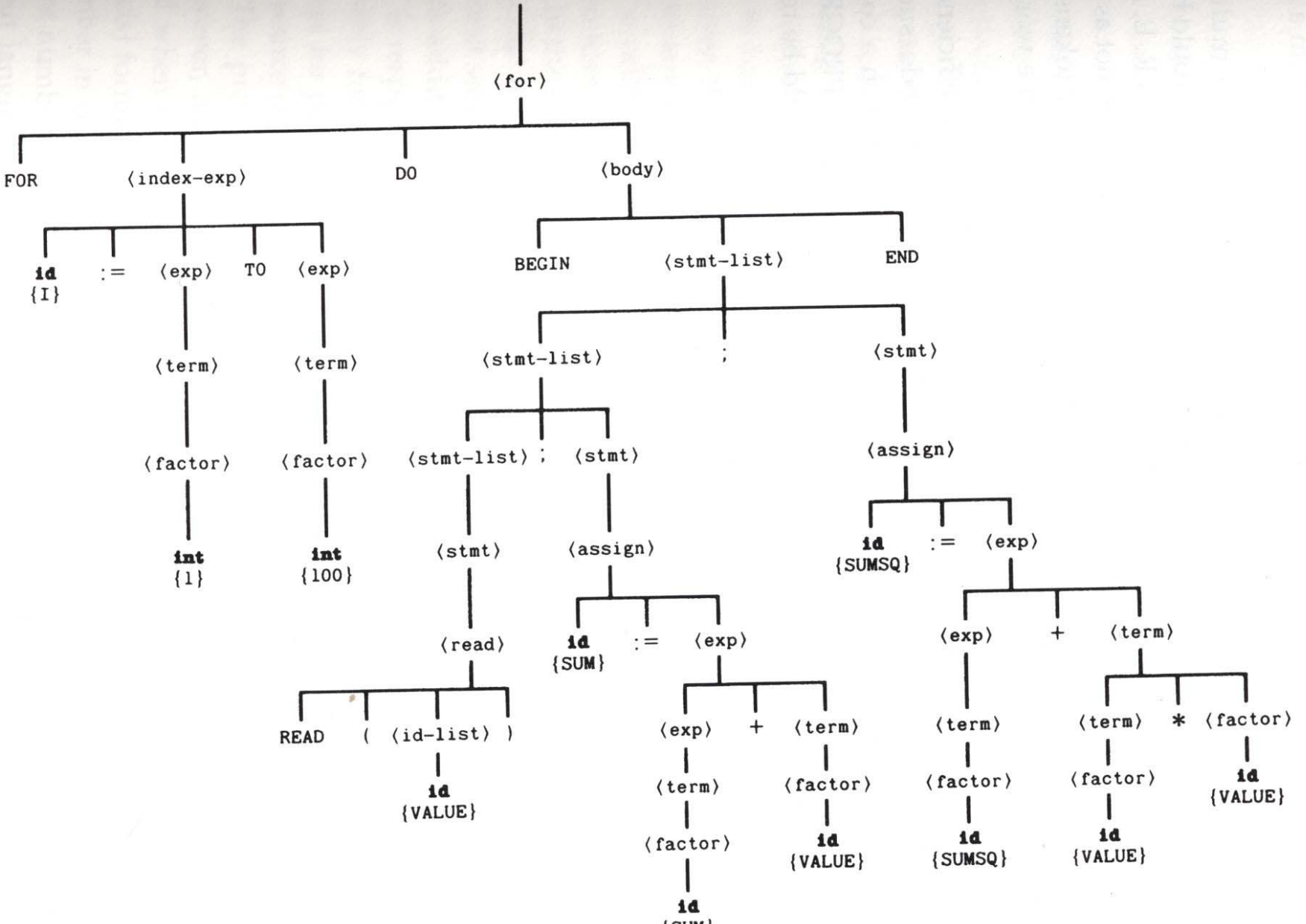
Perkalian dan pembagian mendahului penambahan dan pengurangan



Parse Tree


⟨prog⟩
ROGRAM ⟨prog-name⟩ VAR ⟨dec-list⟩ BEGIN ⟨stmt-list⟩ END.
id {STATS}
⟨dec⟩
⟨stmt-list⟩ ; ⟨stmt⟩
⟨id-list⟩ : ⟨type⟩
⟨stmt-list⟩ ; ⟨stmt⟩ ⟨write⟩
⟨id-list⟩ , id {VARIANCE} INTEGER
⟨stmt-list⟩ ; ⟨stmt⟩ ⟨assign⟩ WRITE (⟨id-list⟩)
⟨id-list⟩ , id {MEAN}
⟨stmt-list⟩ ; ⟨stmt⟩ ⟨assign⟩ id {VARIANCE} := ⟨exp⟩ ⟨id-list⟩ , id {VARIANCE}
⟨id-list⟩ , id {VALUE}
⟨stmt-list⟩ ; ⟨stmt⟩ id {MEAN} := ⟨exp⟩ ⟨exp⟩ − ⟨term⟩ id {MEAN}
⟨id-list⟩ , id {I}
⟨stmt⟩ ⟨assign⟩ ⟨term⟩ ⟨term⟩ ⟨term⟩ * ⟨factor⟩
d-list) , id {SUMSQ}
⟨assign⟩ id {SUMSQ} := ⟨exp⟩ ⟨term⟩ DIV ⟨factor⟩ ⟨term⟩ DIV ⟨factor⟩ ⟨factor⟩ id {MEAN}
id {SUM}
id {SUM} := ⟨exp⟩ ⟨term⟩ ⟨factor⟩ int {100} ⟨factor⟩ int {100} id {MEAN}
⟨term⟩ ⟨factor⟩ id {SUMSQ}
⟨factor⟩ int {0} id {SUM}
int {0}

Parse Tree


⟨for⟩
FOR
⟨index-exp⟩
DO
⟨body⟩
id {I}
:=
⟨exp⟩
TO
⟨exp⟩
BEGIN
⟨stmt-list⟩
END
⟨term⟩
⟨term⟩
⟨stmt-list⟩
;
⟨stmt⟩
⟨factor⟩
⟨factor⟩
⟨stmt-list⟩
;
⟨stmt⟩
⟨assign⟩
int {1}
int {100}
⟨stmt⟩
⟨assign⟩
id {SUMSQ}
:=
⟨exp⟩
⟨read⟩
id {SUM}
:=
⟨exp⟩
⟨exp⟩
+
⟨term⟩
READ
(
⟨id-list⟩
)
⟨exp⟩
+
⟨term⟩
⟨term⟩
⟨term⟩
*
⟨factor⟩
id {VALUE}
⟨term⟩
⟨factor⟩
⟨factor⟩
⟨factor⟩
id {VALUE}
⟨factor⟩
id {VALUE}
id {SUMSQ}
id {VALUE}
id

Analisis Leksikal

- Token dapat didefinisikan oleh aturan grammar agar dikenali oleh parser:

```
<ident>   ::=  <letter> | <ident> <letter> | <ident> <digit>
<letter>  ::=  A | B | C | D | ... | Z
<digit>   ::=  0 | 1 | 2 | 3 | ... | 9
```

- Untuk efisiensi yang lebih baik, scanner dapat digunakan untuk mengenali dan mengeluarkan token dalam suatu deret yang direpresentasikan oleh kode yang panjangnya tetap dan token specifier yang berhubungan

Token	Code
PROGRAM	1
VAR	2
BEGIN	3
END	4
END.	5
INTEGER	6
FOR	7
READ	8
WRITE	9
TO	10
DO	11
;	12
:	13
,	14
:=	15
+	16
-	17
*	18
DIV	19
(	20
)	21
id	22
int	23

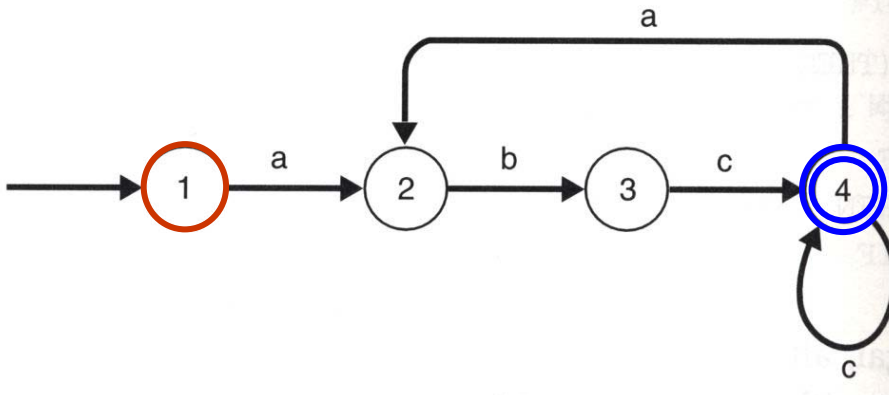
Scan Leksikal

Line	Token type	Token specifier
1	1	
	22	^STATS
2	2	
3	22	^SUM
	14	
	22	^SUMSQ
	14	
	22	^I
	14	
	22	^VALUE
	14	
	22	^MEAN
	14	
	22	^VARIANCE
	13	
	6	
4	3	
5	22	^SUM
	15	
	23	#0
	12	
6	22	^SUMSQ
	15	
	23	#0
	12	
7	7	
	22	^I
	15	
	23	#1
	10	
	23	#100
	11	
8	3	
9	8	
	20	
	22	^VALUE
	21	

Line	Token type	Token specifier
10	22	^SUM
	15	
	22	^SUM
	16	
	22	^VALUE
	12	
11	22	^SUMSQ
	15	
	22	^SUMSQ
	16	
	22	^VALUE
	18	
	22	^VALUE
12	4	
	12	
13	22	^MEAN
	15	
	22	^SUM
	19	
	23	#100
	12	
14	22	^VARIANCE
	15	
	22	^SUMSQ
	19	
	23	#100
	17	
	22	^MEAN
	18	
	22	^MEAN
	12	
15	9	
	20	
	22	^MEAN
	14	
	22	^VARIANCE
	21	

Pemodelan Scanners sebagai Finite Automata

- Token seringkali dapat dikenali oleh finite automata, yang terdiri dari
  - Himpunan status yang tertentu (termasuk status awal dan satu atau lebih status final)
  - Satu set transisi dari satu status ke yang lainnya

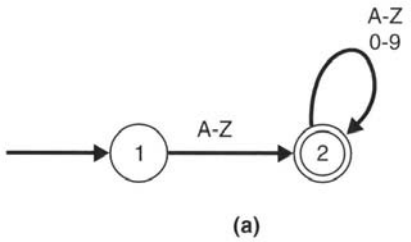


(a)

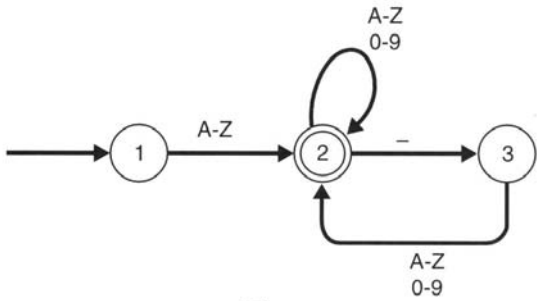
a b c	{recognized}
a b c c a b c	{recognized}
a c	{not recognized}

(b)

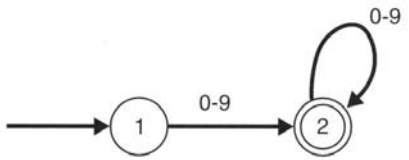
Finite Automata untuk Typical Tokens



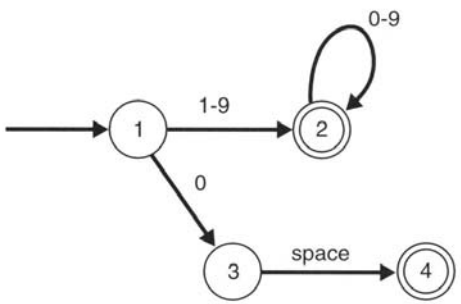
(a)

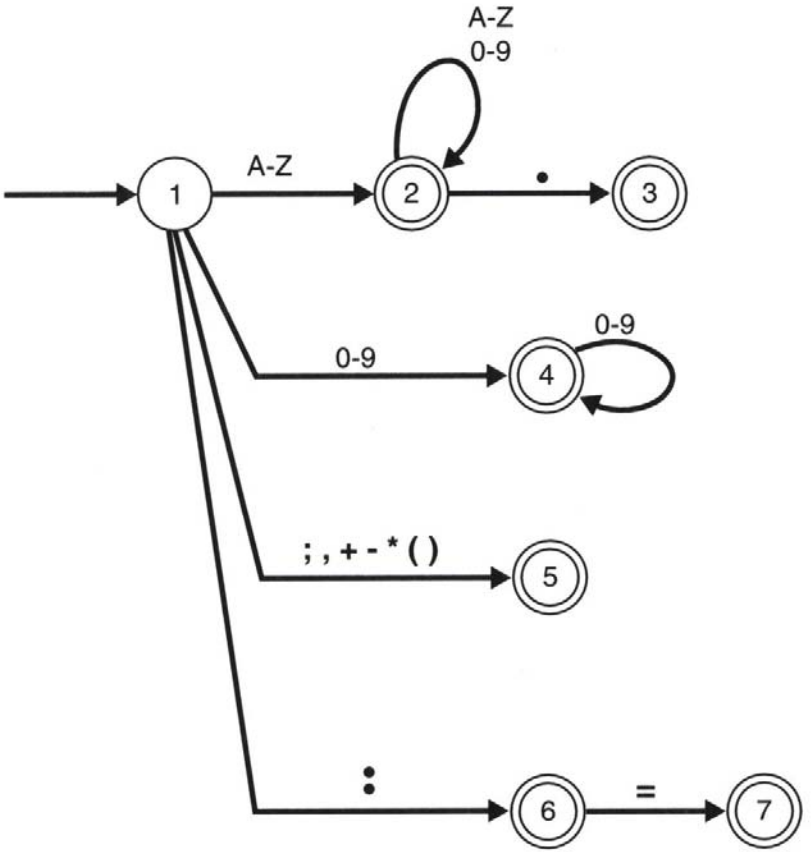


(b)



(c)





Algoritma Pengenalan Token

```
get first Input_Character
if Input_Character in ['A'..'Z'] then
    begin
      while Input_Character in ['A'..'Z', '0'..'9'] do
        begin
           get next Input_Character
           if Input_Character = '_' then
               begin
                  get next Input_Character
                  Last_Char_Is_Underscore := true
               end  {if '_'}
           else
               Last_Char_Is_Underscore := false
        end  {while}
      if Last_Char_Is_Underscore then
        return (Token_Error)
      else
        return (Token_Error)
    end  {if first in ['A'..'Z']}
else
    return (Token_Error)
```

(a)

State	A-Z	0-9	_	
1	2			{starting state
2	2	2	3	{final state}
3	2	2		

(b)