FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Compilers

Bert De Saffel

Master in de Industriële Wetenschappen: Informatica Academiejaar 2018–2019

Gecompileerd op 23 februari 2019

UNIVERSITEIT
GENT

Inhoudsopgave

Hoofdstuk 1

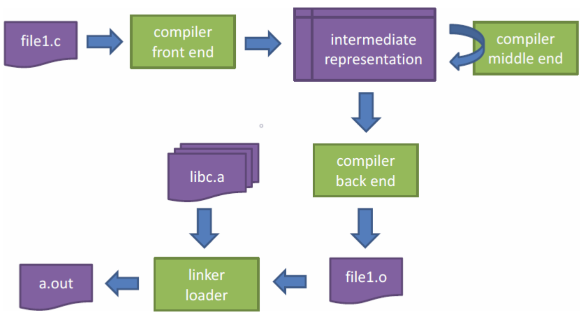
Inleiding

1.1 Compilers

Voorbeelden van functies die een statische compiler moet bevatten:

- Broncode omzetten in uitvoerbare fouten:
  - met dezelfde semantiek
  - zo snel mogelijk
  - en/of zo compact, debugbaar, portable, veilig, ... mogelijk
  - en linkbaar.
- Syntaxfouten moeten herkent worden.

1.2 Basiswerking compilers



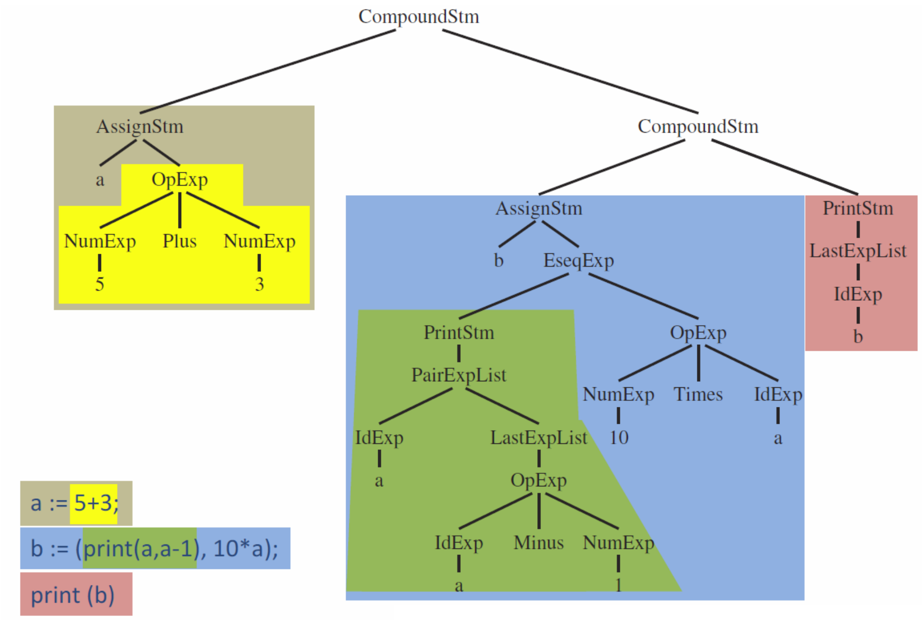
Figuur 1.1: De basiswerking van een compiler.

Op figuur 1.1 is de vereenvoudigde basiswerking van een compiler te zien. Een **C** bestand wordt eerst door de compiler front end gestuurd, die het bestand zal omvormen tot een intermediaire representatie. Deze representatie wordt dan door de compiler back end gestuurd om zo assembly of objectcode te genereren. De linker loader zal deze objectcode samenvoegen met eventuele andere libraries om zo een uitvoerbaar programma te hebben.

Q: Waarom wordt de front end en back end opgesplitst?
A: Op die manier is de compiler modulair: Enerzijds moet bij een andere programmeertaal enkel de front end aangepast worden en anderzijds moet bij het wijzigen van de architectuur (de onderliggende processor) enkel de back end aangepast worden.

1.3 Abstract Syntax Tree

De eerste stap van elke compiler is het omvormen van de broncode naar een **Abstract Syntax Tree (AST)**. Veronderstel volgende code, en de daarbijhorende AST die te zien zijn op figuur 1.2. Elke knoop van een AST stelt een bepaalde geldige operatie voor, die onafhankelijk is van de gekozen programmeertaal.



Figuur 1.2: De boomvoorstelling van een eenvoudig, lusloos programma. De gekleurde deelbomen komen overeen met de gekleurde segmenten in de code zelf. Als toekenningsoperator wordt er gekozen voor := dat vanaf nu als één geheel moet beschouwd worden.

1.3.1 Contextvrije grammatica's

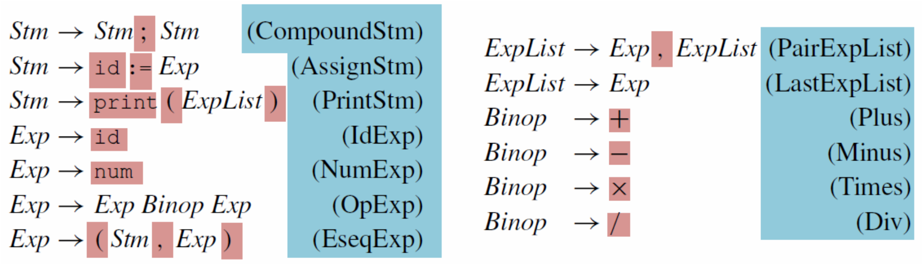
Om een AST op te stellen moet de notie van tokens ingevoerd worden. Een token is eenvoudig gezien een bepaald symbool dat een betekenis heeft. De tokens van de code uit figuur 1.2 zijn te zien in tabel 1.1 Uit de theorie van de generatieve grammatica's weten we dat er zowel terminale als niet-terminale tokens bestaan:

- **Terminale tokens** zijn symbolen die een blad voorstellen in de AST. Deze tokens hebben als eigenschap dat ze geen verdere tokens kunnen genereren en vormen dan ook het alfabet van het programma.
- **Niet-terminale tokens**, kortweg niet-terminalen genoemd, zijn de regels die de taal definiëren en zijn de niet-bladeren van de AST. Niet-terminalen hebben als eigenschap dat ze letters van het alfabet kunnen genereren.

symbolen(ascii)	token	waarde
a	id	string a
:=	:=	
5	num	integer 5
+	+	
3	num	integer 3
;	;	
b	id	string b
(	(	
print	print	
-	-	
*	*	
	whitespace	

Tabel 1.1: De tokens die voorkomen uit het programma van figuur 1.2

Op figuur 1.3 zijn een aantal terminalen en niet-terminalen te zien. De niet-terminale token *CompoundStm* bestaat bijvoorbeeld uit twee *Stm* tokens, gescheiden door een punt komma. Deze twee *Stm* tokens kunnen in deze vereenvoudigde programmeertaal enkel een *AssignStm* of *PrintStm* zijn. Bij *AssignStm* wordt er een terminale token verwacht in de vorm van een variabele identifier, gevolgd door de toekenningsoperator en een *Exp* token. Enkel deze *Exp* kan nog vier vormen aanneemen: *IdExp*, *NumExp*, enz... Dit wordt uitgewerkt voor de eerste toekenningsoperatie uit figuur 1.2 en



Figuur 1.3: De rood omkaderde symbolen zijn terminalen terwijl de blauw omkaderde niet-terminalen zijn.

is te zien op figuur 1.4.

1.3.2 Opbouw AST

Een AST kan nu **bottom-up** opgemaakt worden door volgende procedure uit te voeren:

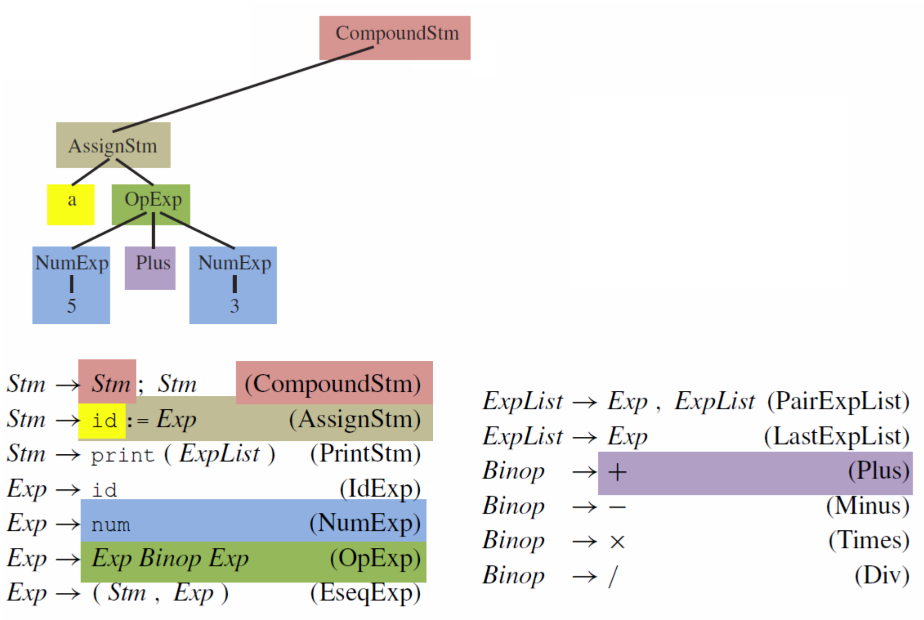
1. Voor elke mogelijke knoop moet er een struct gemaakt worden zoals bijvoorbeeld:

   `A_stm_`   `A_exp_`   `A_expList_`

2. Elke struct moet bestaan uit

   - een enum voor het precieze token te bepalen,
   - een union voor de verschillende combinaties van tokens in het rechter lid en,
   - pointers naar kindknopen.

   Dit wordt geïllustreerd in code 1.1.



Figuur 1.4: Illustratie van contextvrije grammatica op de eerste toekenningsoperatie uit figuur 1.2.

```
typedef char * string;
typedef struct A_stm_ * A_stm;
typedef struct A_exp_ * A_exp;
typedef struct A_expList_ * A_expList;

struct A_stm_ {
        enum {A_compoundStm, A_assignStm, A_printStm} kind;
        union {
                struct {A_stm stm1, stm2;} compound;
                struct {string id; A_exp exp;} assign;
                struct {A_expList exps;} print;
        } u;
};
```

Code 1.1: Voorbeeld van een struct voor een AST.

3. In de constructor worden de knopen aangemaakt, zoals te zien in code 1.2.

```
A_stm A_CompoundStm(A_stm stm1, A_stm stm2){
        A_stm s = malloc(sizeof(*s));
        s->kind = A_compoundStm;
        s->u.compound.stm1 = stm1;
        s->u.compound.stm2 = stm2;
        return s;
}
```

Code 1.2: Voorbeeld van een constructor voor een AST.

Op deze manier zou de boom uit figuur 1.2 hardgecodeerd kunnen worden, wat natuurlijk geen goede manier is. Het is de taak van een lexer en parser om de constructie van een AST te automatiseren, die respectievelijk in hoofdstuk 2 en 3 behandelt worden.

1.3.3 Interpreter

Uit een AST kan een eenvoudige interpreter geschreven worden. Dit stuk is informatief, en wordt niet gevraagd op het examen.

- Door de boom postorder diepte-eerst te overlopen, wordt de boom in de juiste manier behandelt.
- Het bijhouden van de waarden van variabelen kan via een gelinkte lijst:

```
typedef struct table * Table_;
struct table {string id; int value; Table_ tail;};
Table_ Table(string id; int value; Table_ tail) {
        Table_ t = malloc(sizeof(*t));
        t->id = id;
        t->value = value;
        t->tail = tail;
        return t;
}
```

- Stel nu dat dit de eerste drie regels van een programma zijn:

```
a := 2;
b := 3;
a := 3;
```

- Voor de eerste toekenning bevat de gelinkte lijst slechts één knoop met als sleutel *a* en waarde *2*.
- Bij de tweede toekenning wordt de originele gelinkte lijst meegegeven via de variabele *tail*. Na deze constructor zal de gelinkte lijst twee knopen bevatten.
- Na deze constructor bevat de gelinkte lijst drie knopen. Merk op dat er twee knopen zijn met sleutel *a*, maar dat ze elk een verschillende waarde hebben. Aangezien een nieuwe knoop vooraan wordt toegevoegd, zal de interpreter enkel de meest recentste waarde opvragen.

Hoofdstuk 2

Lexicale Analyse

2.1 Lexicale tokens

- Herkennen van een reeks opeenvolgende karakters die een geheel vormen volgens de syntax van een programmeertaal, zoals o.a:
  - sleutelwoorden: int, float, for, new, ...
  - identifiers: foo, n14, variabelenaam
  - getallen: -37, 0x16L, 10.4, ...
  - operatoren: +, -, *, &, &&, ...
  - andere tokens: { } "; /* */ / ( ) [ ]
- Veronderstel volgende code:

```
float match0(char * s) {/* find a zero */
        if(!strncmp(s, "0.0", 3))
                return 0.;
}
```

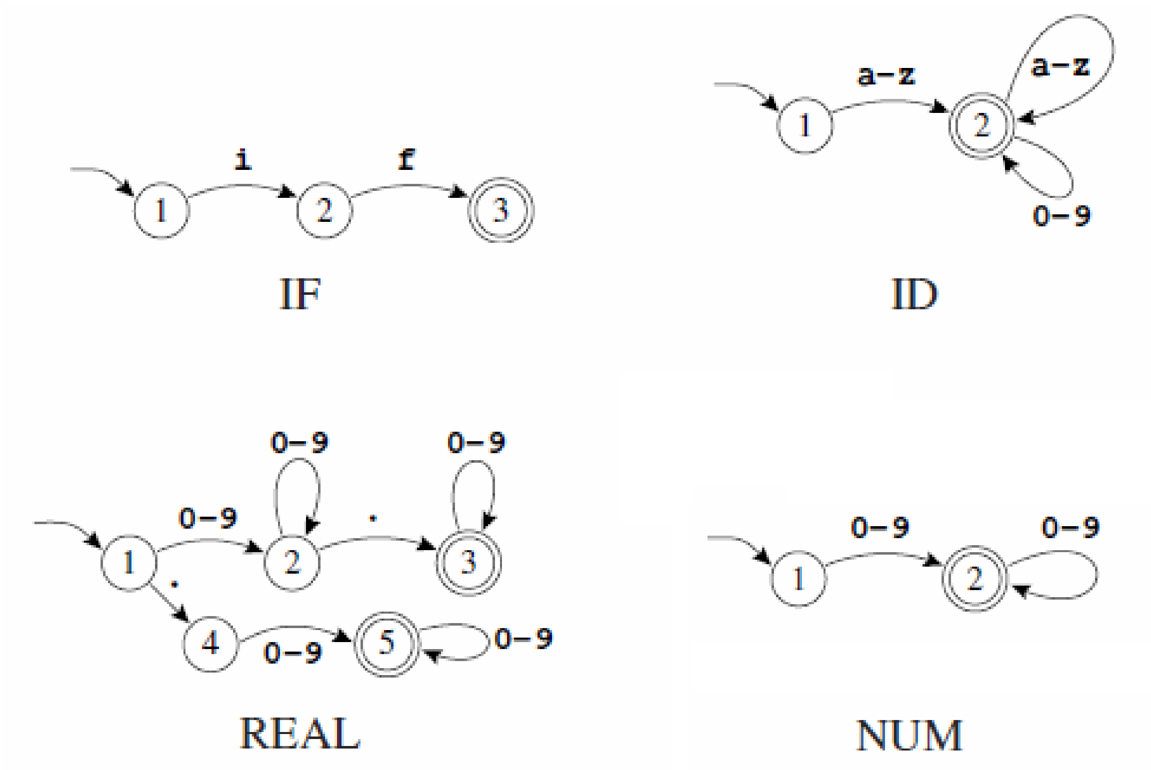
, dan worden volgende tokens gegenereerd, waarbij dat sommige tokens een attribuut hebben:
`FLOAT ID(match0) LPAREN CHAR START ID(s) RPAREN LBRACE IF LPAREN BANG ID (strncmp) LPAREN ID(s) COMMA STRING(''0.0'') COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF`

2.2 Eindige automaten

- Er wordt met reguliere expressie gewerkt om te omschrijven welke karaktersequentie met een bepaald token overeenstemmen:

if	{return IF;}
$[a-z][a-z0-9]*$	{return ID;}
$[0-9]+$	{return NUM;}
$([0-9]+"."[0-9]*)\|([0-9]*"."[0-9]+)$	{return REAL;}

Tabel 2.1: Reguliere expressies voor een aantal tokens.

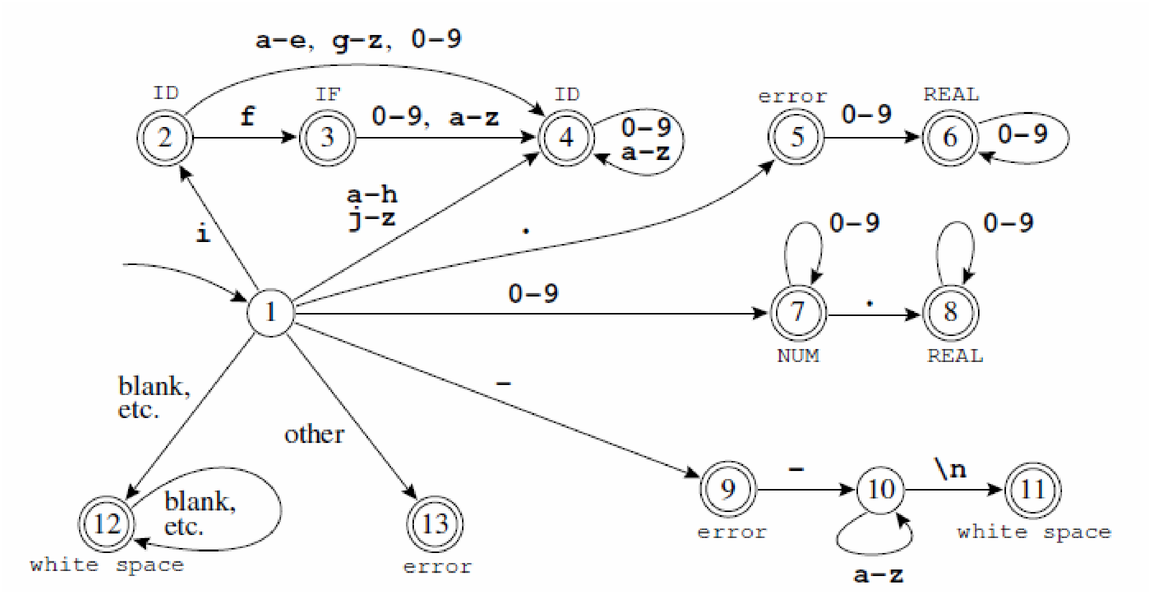


Figuur 2.1: Eindige automaten voor lexicale tokens.

- Met behulp van de constructie van Thompson kan een **niet-deterministische automaat (NFA)** opgebouwd worden uit een reguliere expressie. Op figuur 2.1 zijn de eindige automaten te zien van de reguliere expressies uit tabel 2.1.
- Deze individuele automaten kunnen samengevoegd worden tot een gecombineerde automaat, te zien op figuur 2.2

  In dit geval is de gecombineerde automaat al een **deterministische eindige automaat (DFA)** aangezien elke mogelijke staat slechts één transitie heeft voor elke input. Het doel van lexicale analyse is om een DFA op te stellen zodat de tokens op een efficiënte manier kunnen bepaald worden. Een DFA wordt doorgaans geïmplementeerd als een transitietabel:

```
int edges[][256] = { /* ... 0 1 2 ... - ... e f g h i j ... */
/* state 0 */      {    ... 0 0 0 ... 0 ... 0 0 0 0 0 0 ...},
/* state 1 */      {    ... 7 7 7 ... 9 ... 4 4 4 4 2 4 ...},
/* state 2 */      {    ... 4 4 4 ... 0 ... 4 3 4 4 4 4 ...},
/* state 3 */      {    ... 4 4 4 ... 0 ... 4 4 4 4 4 4 ...},
/* state 4 */      {    ... 4 4 4 ... 0 ... 4 4 4 4 4 4 ...},
/* state 5 */      {    ... 6 6 6 ... 0 ... 0 0 0 0 0 0 ...},
/* state 6 */      {    ... 6 6 6 ... 0 ... 0 0 0 0 0 0 ...},
/* state 7 */      {    ... 7 7 7 ... 0 ... 0 0 0 0 0 0 ...},
/* state 8 */      {    ... 8 8 8 ... 0 ... 0 0 0 0 0 0 ...},
/* ... */
};
```



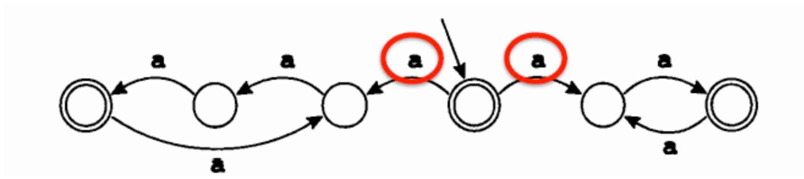
Figuur 2.2: Combinatie van eindige automaten.

2.3 Opbouw deterministische eindige automaat

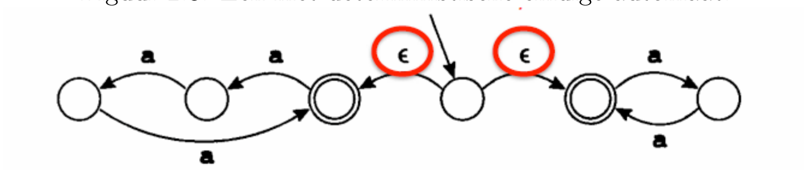
- We starten met een reguliere expressie, die een bepaald token voorstelt:

  `(aaa)*|(aa)*`

- Zoals vermeld zal de constructie van Thompson een niet-deterministische automaat aanmaken van een bepaalde reguliere expressie. Er bestaat de kans dat deze automaat deterministisch is, maar dat is niet altijd zo. In het geval van bovenstaande reguliere expressie ziet de automaat er uit zoals op figuur 2.3 of figuur 2.4.



Figuur 2.3: Een niet-deterministische eindige automaat.



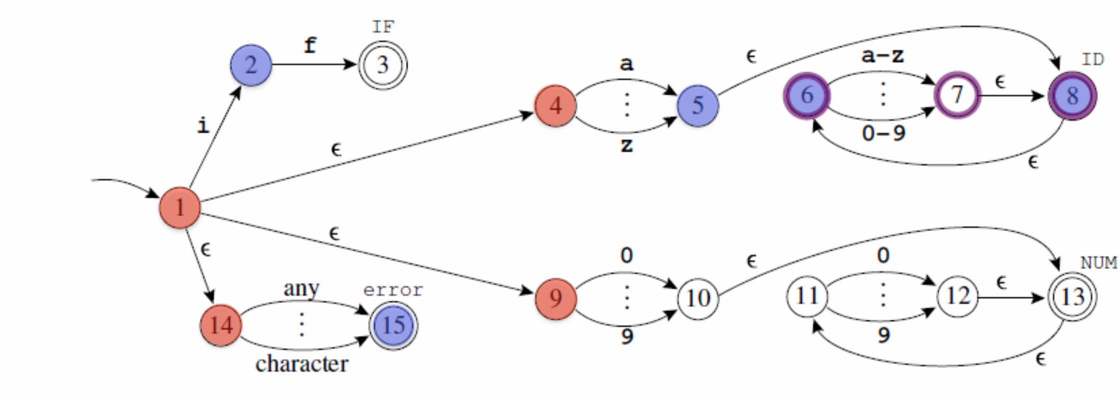
Figuur 2.4: Een niet-deterministische eindige automaat waarbij de eerste transitie kan gebeuren zonder een symbool te verwerken.

  Welke richting moeten we nu uit bij `aaaaaaa` voor de eerste `a`? Bij een niet-deterministische automaat moeten we gokken welke de juiste zal zijn.

- Gelukkig kan ook een DFA opgebouwd worden uit een NFA via de deelverzamelingconstructie. Op die manier kan een DFA opgebouwd worden door (i) enkel de reguliere expressies handmatig te definiëren, (ii) algoritmisch deze reguliere expressies om te vormen tot een NFA, en (iii) algoritmisch deze NFA om te vormen tot een DFA.

- Veronderstel de reguliere expressies en de daarbijhorende NFA in figuur 2.5.

```
if                  {return IF;}
[a-z][a-z0-9]*      {return ID;}
[0-9]+              {return NUM;}
.                   {error();}
```

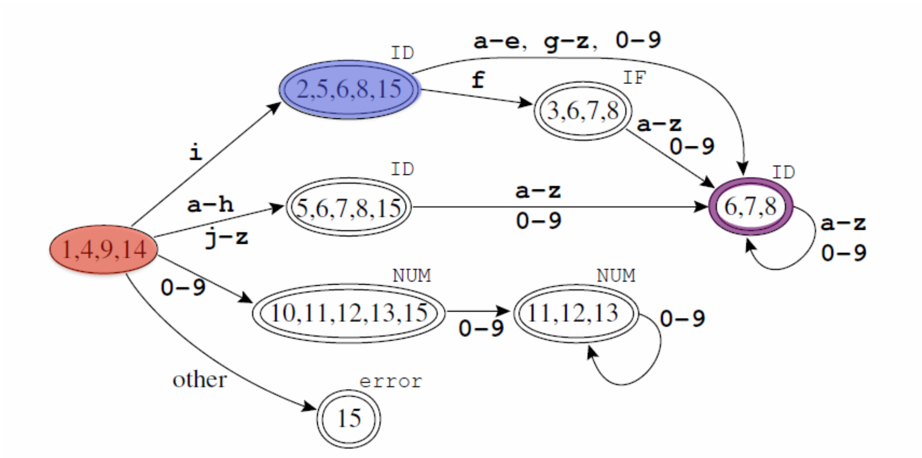


Figuur 2.5: Een aantal reguliere expressies en de daarbijhorende NFA.

Stel dat we nu de string *in* moeten checken:

1. Zonder teken op te eten kunnen we in 1 komen en in zijn ϵ-closure: {1, 4, 9, 14}.
2. Vanuit {1, 4, 9, 14} kunnen we voor i naar {2, 5, 6, 8, 15}.
3. Vanuit {2, 5, 6, 8, 15} kunnen we voor n naar {6, 7, 8}.
4. Daarvan is 8 een aanvaardingstoestand voor `ID`.

Op die manier bekomen we de DFA uit figuur 2.6.



Figuur 2.6: De NFA uit figuur 2.5 geconverteerd naar een DFA.

2.3.1 Conversie NFA naar DFA

- Drie functies:
  1. **edge(s, c)** = alle NFA staten bereikbaar uit toestand s over pijlen met transitiesymbool c.
  2. **closure(S)** = de kleinste verzameling T voor een subset S waarvoor geldt:

     $$T = S \cup \left(\bigcup_{s \in T} \mathbf{edge}(s, \epsilon) \right)$$

     Q: Waarom moet dit de kleinste verzameling zijn?
     A: De volledige verzameling van toestanden voldoet ook aan deze vergelijking, en dat is een triviaal geval.

     Via iteratie kan T berekent worden:

     $T \leftarrow S$
     repeat
     $\quad T' \leftarrow T$
     $\quad T \leftarrow T' \cup (\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon))$
     until T = T'

     - Dit is een voorbeeld van een fixpoint algoritme. Dit wil zeggen dat uiteindelijk $f(x) = x$ geldig is. In het voorbeeld van de functie **closure(S)**, hier genoteerd als **F(x)**, is dit zeker waar:

       $$\begin{aligned} F(\epsilon) &= S \\ F(S) &= ... \\ F(F(S)) &= ... \\ &... \\ F(F(F(...))) &= T \\ F(T) &= T \end{aligned}$$

     - Aangezien dat uiteindelijk $F(T) = T$ en dat er maar een eindig aantal staten zijn zal het algoritme zeker stoppen.
  3. Veronderstel dat we ons bevinden in een set $d = \{s_i, s_k, s_l\}$ van NFA staten s_i, s_k en s_l. Startend vanuit d en het symbool c, bekomen we een nieuwe set van NFA staten:

     $$\mathbf{DFAedge(D,\ c)} = \mathbf{closure}\left(\bigcup_{s \in D} \mathbf{edge}(s, c) \right)$$

     Via deze functie, de startstaat s_1 en input string $c_1, ..., c_k$ kan de NFA simulatie als volgt geschreven worden:

     $d \leftarrow \mathbf{closure}(\{s_1\})$
     for$i \leftarrow 1$ **to** k
     $\quad d \leftarrow \mathbf{DFAedge}(d, c_i)$

- De combinatie van deze drie functies leiden tot het algoritme om een NFA om te zetten naar een DFA:

  states[0] $\leftarrow \{\}$;
  states[1] $\leftarrow \mathbf{closure}(\{s_1\})$;
  $p \leftarrow 1$; $\quad j \leftarrow 0$;

while $j \leq p$
    foreach $c \in \Sigma$
    $e \leftarrow$ **DFAedge**(states[j], c)
    if $e ==$ states[i] for some $i \leq p$
        then trans[j, c] $\leftarrow i$
        else $p \leftarrow p + 1$
            states[p] $\leftarrow e$
            states[j,c] $\leftarrow p$
$j \leftarrow j + 1$

De gegenereerde DFA is suboptimaal: vanuit sommige toestanden worden identiek dezelfde strings aanvaard. Volgende optimalisaties kunnen nog doorgevoerd worden:

- Knopen samenvoegen waarvoor geldt dat

$$\forall c \in \Sigma : \text{trans}[s_1, c] = \text{trans}[s_2, c]$$

- Staten s_1 en s_2 zijn equivalent als ze beiden niet finaal of finaal zijn voor dezelfde tokens.

Hoofdstuk 3

Parsing

3.1 Inleiding

Het basisidee van parsing is om een string van tokens te analyseren en kijken of deze syntactisch geldig zijn.

Q: Waarom gaan we context-vrije grammatica gebruiken in plaats van reguliere expressies om de tokens van een lexer te parsen?
A: Reguliere expressies kan geen recurise uitdrukken. Ook kan de eis voor gebalanceerde haakjes niet uitgedrukt worden met reguliere expressies.

3.2 Context-vrije grammatica

- Een **taal** is een verzameling **strings**.
- Een **string** is een eindige sequentie **symbolen** uit een **alfabet**.
- Analogie met een parser:
  - De broncode levert de strings op via lexicale analyse.
  - De lexicale tokens zijn de symbolen.
  - Het alfabet is de verzameling tokentypes die gegenereerd worden door de lexicale analyzer.
- De taal van een context-vrije
- Context-vrije grammatica definieert de **syntax** van de taal.

3.2.1 Afleiden van een zin

Grammatica 3.1 toont een voorbeeldsyntax voor lusloze programma's. Een voorbeeld van een zin is:

$$\texttt{id := num ; id := id + (id := num + num, id)} \tag{3.1}$$

die bijvoorbeeld afgeleidt is door de lexer van:

```
a := 7; b := c + (d := 5 + 6, d)
```

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \mathrm{id} := E$
3 $S \rightarrow \mathrm{print}\ (\ L\ )$
4 $E \rightarrow \mathrm{id}$
5 $E \rightarrow \mathrm{num}$
6 $E \rightarrow E\ +\ E$
7 $E \rightarrow (S\ ,\ E\ )$
8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

Grammatica 3.1: Een syntax voor een lusloos programma.

Het afleiden van een zin start altijd met een **startsymbool**, die twee vormen kan aannemen:

1. Het startsymbool kan enerzijds het eerste symbool zijn.
2. Anderzijds wordt het startsymbool expliciet aangeduid zoals bijvoorbeeld $P \rightarrow S\$$, met \$ het stopsymbool.

$$\begin{array}{l}
\underline{\mathrm{S}}\\
\mathrm{S}\ ;\ \underline{\mathrm{S}}\\
\underline{\mathrm{S}}\ ;\ \mathrm{id}\ :=\ \mathrm{E}\\
\mathrm{id}\ :=\ \underline{\mathrm{E}}\ ;\ \mathrm{id}\ :=\ \mathrm{E}\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{\mathrm{E}}\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{E} + \underline{\mathrm{E}}\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \underline{\mathrm{E}} + (\mathrm{S},\ \mathrm{E})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\underline{\mathrm{S}},\ \mathrm{E})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\mathrm{id}\ :=\ \underline{\mathrm{E}},\ \mathrm{E})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\mathrm{id}\ :=\ \mathrm{E} + \mathrm{E},\ \underline{\mathrm{E}})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\mathrm{id}\ :=\ \underline{\mathrm{E}} + \mathrm{E},\ \mathrm{id})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\mathrm{id}\ :=\ \mathrm{num} + \underline{\mathrm{E}},\ \mathrm{id})\\
\mathrm{id}\ :=\ \mathrm{num}\ ;\ \mathrm{id}\ :=\ \mathrm{id} + (\mathrm{id}\ :=\ \mathrm{num} + \mathrm{num},\ \mathrm{id})
\end{array}$$

Code 3.1: Het afleidingsproces.

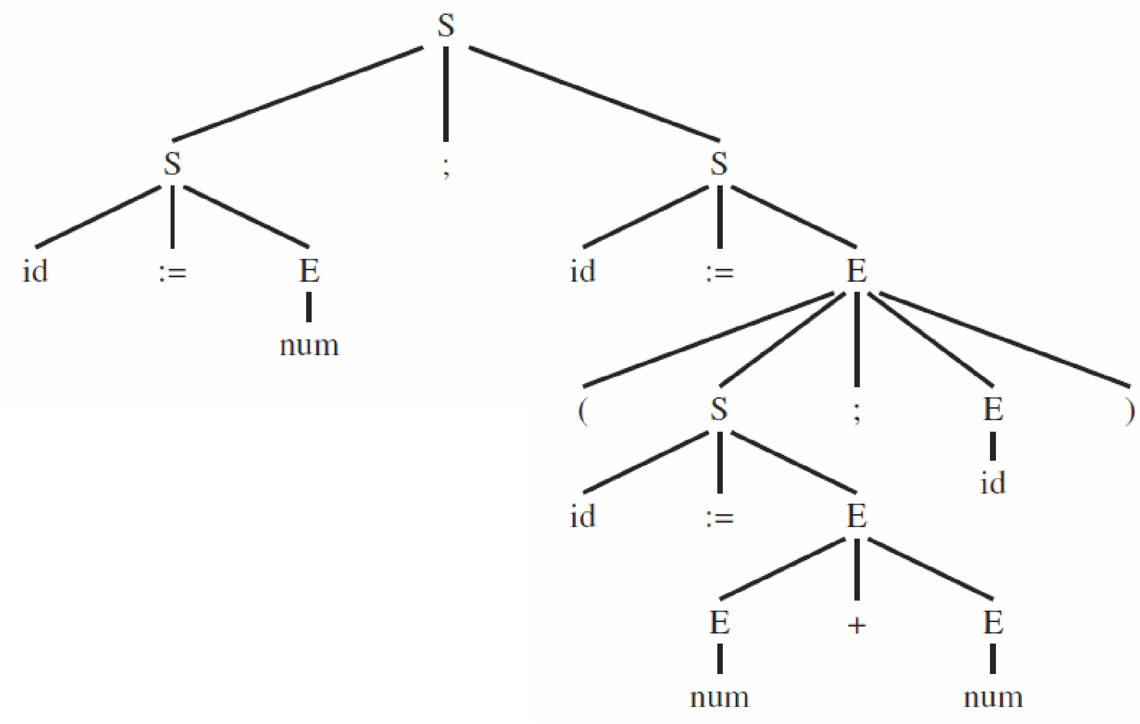
Code 3.1 toont een illustratie van hoe het afleidingsproces te werk gaat, toegepast op voorbeeldzin 3.1. Bij elke iteratie wordt het niet-terminale token dat onderlijnt is verwerkt.

> **Q:** Is dit een linkse of een rechtste afleiding?
> **A:** Geen van beide, omdat er gekozen kan worden om zowel de meest linkse als de meest rechtse token te verwerken.

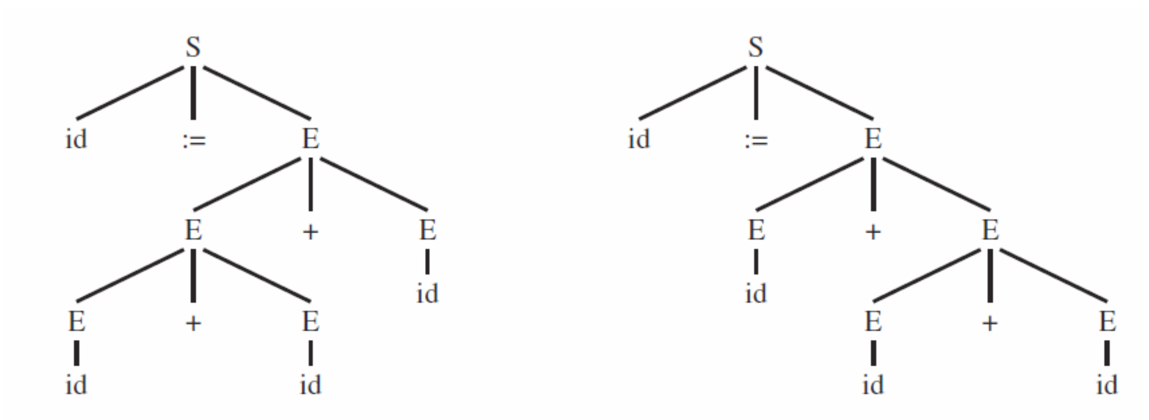
Figuur 3.2 toont de bijhorende parse tree. Hier zijn de bladeren ook een verzameling van terminale tokens. De taak van een parser is om de bijhorende boom op te stellen, uitgaande van enkel de bladeren.

3.2.2 Ambigue grammatica

> **Q:** Stel dat we Grammatica 3.1 hebben. Wat gebeurt er voor het statement `a := (x + (y) + z)`?
> **A:** Dit is een voorbeeld van een ambigue grammatica. Aan de hand van de grammatica is het onmogelijk om slechts één parse tree op te bouwen. Figuur 3.3 toont beide parse trees voor het statement. Bij de linkse boom worden de rode haakjes gebruikt terwijl be de rechtse boom de blauwe haakjes gebruikt worden.



Figuur 3.2: De bijhorende parse tree voor voorbeeldzin 3.1.



Figuur 3.3: Voor Grammatica 3.1 kunnen er twee parse trees opgebouwd worden voor het statement `a := x + y + z`.

Bij een plus-operatie is dit niet heel belangrijk aangezien het toch associatief is, maar bij niet-associatieve operaties is dit duidelijk niet goed.

3.2.3 Grammatica disambiguëren

Een grammatica hoeft niet perse de regels van de wiskunde te volgen. Daarom is het automatiseren ook moeilijk, omdat het afhangt van welke semantiek gewenst is. Er kan bijvoorbeeld gesteld worden dat:

- * en / voorrang heeft op + en -,
- a + b + c = (a + b) + c, dus + is links associatief.

Om dit te realiseren worden er **termen** en **factoren** ingevoerd. Op die manier kan Grammatica 3.4 omgevormd worden tot 3.5.

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

Grammatica 3.4: Een ambigue grammatica. Hier wordt de regel dat * en / voorrang heeft op + en - niet gerespecteerd.

$$
\begin{array}{lll}
E \rightarrow E + T & \quad T \rightarrow T * F & \quad F \rightarrow \text{id} \\
E \rightarrow E - T & \quad T \rightarrow T \,/\, F & \quad F \rightarrow \text{num} \\
E \rightarrow T & \quad T \rightarrow F & \quad F \rightarrow (\, E\,)
\end{array}
$$

Grammatica 3.5: Grammatica 3.4 kan hervormt worden, door termen T en factoren F in te voeren. Deze termen dwingen de volgorde van operaties en associativiteit vast.

3.3 Predictive Parsing

Sommige grammatica's kunnen eenvoudig geparsed worden met een **recursive descent parser**. Voor elke niet-terminal is er een overeenkomstige functie. In elke functie is er een switch clause voor elke productieregel die door de niet-terminal kan gegenereerd worden. Niet-terminals worden recursief aangeroepen terwijl terminals verwerkt worden.

Code 3.2 toont een voorbeeld van zo een recursive descent parser toegepast op Grammatica 3.6.

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

$$E \rightarrow \text{num} = \text{num}$$

Grammatica 3.6:

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok = getToken();}
void eat(enum Token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok){
        case IF:     eat(IF); E(); eat(THEN); S(); eat(ELSE); S(); break;
        case BEGIN:  eat(BEGIN); S(); L(); break;
        case PRINT:  eat(PRINT); E(); break;
        default:     error();
}}

void L(void) {switch(tok){
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
}}

void E(void) {
        eat(NUM) ; eat(EQ) ; eat(NUM);
}
```

Code 3.2: Een recursive descent parser gebaseerd op Grammatica 3.6

Een recursive descent parser werkt enkel als het eerste terminale symbool van een subexpressie genoeg informatie oplevert.

3.3.1 First and follow sets

Om de begrippen **first set** en **follow set** uit te leggen wordt Grammatica 3.7 gebruikt.

$$Z \rightarrow d \qquad Y \rightarrow \qquad X \rightarrow Y$$
$$Z \rightarrow X\ Y\ Z \qquad Y \rightarrow c \qquad X \rightarrow a$$

Grammatica 3.7:

- **nullable(X)** $\rightarrow$ boolean: true als X de lege string kan afleiden.

We zien dat nullable(Y) zeker waar is voor Grammatica 3.7. We kunnen echter vanuit X ook naar de lege string gaan via $X \to Y \to \epsilon$, maar niet vanuit Z.

	nullable	FIRST	FOLLOW
X	yes		
Y	yes		
Z	no		

- **FIRST(γ)**: verzameling terminals waarmee strings kunnen beginnen die van expressie γ kunnen afgeleid worden.

  Uitgewerkt voor de drie startsymbolen:

  X : Vanuit X zijn er twee mogelijkheden: $X \to a$ en $X \to Y$. We zien dat a een terminal is dus die behoort al zeker tot de FIRST set. Vanuit Y kan ook nog de lege string en c bereikt worden. Hieruit volgt FIRST(X) = $\{a\ c\}$.

  Y : Vanuit Y kan enkel c bereikt worden: FIRST(Y) = $\{c\}$.

  Z : In eerste instantie kan Z direct d bereiken, dus die zit zeker in de FIRST set. Aangezien ook de productieregel $Z \to X\ Y\ Z$ bestaat en zowel X als Y nullable zijn, kan zowel de FIRST set van X als van Y overgenomen worden.
  FIRST(Z) = $\{a\ c\ d\}$

	nullable	FIRST	FOLLOW
X	yes	a c	
Y	yes	c	
Z	no	a c d	

- **FOLLOW(X)**: is de verzameling van terminals t die meteen op X kunnen volgen, dus waarvoor de afleiding X_t bestaat.

Algoritme 3.3 is een fixpoint algoritme die de first, follow en nullable berekent.

```
for each terminal symbol Z
  FIRST[Z] ← {Z}
repeat
  for each production X → Y1Y2...Yk
    for each i from 1 to k, each j from i+1 to k,
      if all the Yi are nullable
        then nullable[X] ← true
      if Y1 ··· Yi−1 are all nullable
        then FIRST[X] ← FIRST[X] ∪ FIRST[Yi]
      if Yi+1 ··· Yk are all nullable
        then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]
      if Yi+1 ··· Yj−1 are all nullable
        then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yi]
until FIRST, FOLLOW and nullable did not change in this iteration
```

Code 3.3: Iteratieve berekening van FIRST, FOLLOW en nullable

3.3.2 Opstellen Predictive Parsing Tabel

Uitbreiden definitie van first naar strings:

- `FIRST(`$W\gamma$`)` $=$ `FIRST(`W`)` als niet nullable(W)
- `FIRST(`$W\gamma$`)` $=$ `FIRST(`W`)` $\cup$ `FIRST(`γ`)`

Er zijn drie gevallen waarbij er twee keuzes zijn. Moeten we $X \to a$ of $X \to Y$ nemen? De string d levert minstens twee parse tree op. De grammatica was zelfs ambigu. Dit kan nooit geparsed worden.

3.3.3 LL(1) Parsers

- Elk vak in de tabel bevat slechts 1 productieregel.
- Left-to-right parse: begin vooraan in broncode en verwerk van links naar rechts.
- Leftmost-derivation:
- 1-symbol lookahead: Er wordt slechts één symbool vooraf bekeken.
- LL(k):
  - k symbolen vooraf bekijken. De first sets bevatten sequenties van k terminals.
- ! Mogelijke problemen:
  - Linkse recursie.
    - Probleem: zekerheid van meerdere productieregels in een vak want `FIRST(T)` $\in$ `FIRST(E - T)`.
    - Oorzaak: E verschijnt links in de rechterkant van een E-productie.
    - Oplossing:
  - Linkse factorisatie.
    - Probleem: De parser kan geen onderscheid maken tussen twee gelijkaardige strings.
    - Oplossing: grammatica herschrijven.
- Error recovery is mogelijk.
- ! Beslissing nemen na k symbolen blijft een zwakte.

3.3.4 Error Recovery

Probleem: pseudocode voor error. We willen geen compiler die geen nuttige foutboodschappen kan geven. Compiler mag ook niet stoppen bij eerste fout, omdat meerdere fouten nog verder kunnen voorkomen.

- Gewoon een print statement = vrij slechte methode aangezien er geen tokens opgegeten worden. De parser doet voort alsof hij F en Tprime al geparsed heeft. De parser komt in foute toestand.
- Print statement combineren met de skipto functie, die tokens zal opeten totdat er een token tegenkomt die in de follow set zit. Alle karakters die niet in de follow zitten, zal nog deel uitmaken van de subexpressie.

3.4 LR(1) parser

- Left-to-right parse
- Rightmost-deviation