Meindert Meindertsma

BNF's en GEN's

Ideaal metaproza — een queeste

Inhoud

Voorwoord

```
(
((grammar)°((rules)°))
((rules)(rule))
((rules)(rules)(rule))
((rule)°((nonterminal)(chain)°))
((nonterminal)°((name)°))
((chain)(atom))
((chain)(chain)(atom))
((atom)(nonterminal))
((atom)(pseudoterminal))
((atom)(terminal))
((pseudoterminal)°(°((name)°)°))
((name)((namecharacter)))
((name)(name)((namecharacter)))
((terminal)(character))
((terminal)(terminal)(character))
((character)°°°()
((character)°°°))
((character)°°°°)
((character)((nonmetacharacter)))
)
```

M.M.

1 Inleiding

Al is mijn voorwoord dan alleen maar met zichzelf bezig, het onthult wel dat schrijver dezes soms tot een compromis bereid is. Ik kan daaraan toevoegen, dat ik het voorwoord niet erg vriendelijk vind jegens de voorbijtrekkende lezer. Dat suggereert dat lezersvriendelijkheid een aandachtspunt voor me is, vooral wanneer lezers voorbij willen trekken. En inderdaad, zo is het.

De metataal die gebruikt werd om de syntax van Algol 60 te specificeren [1] is bekend geworden als de Backus-Naur Form (BNF) [2]. Een elegante en compacte manier van formuleren, maar niet zonder tekortkomingen. Er zijn dan ook talloze Extended BNF's (EBNF's) opgedoken, allemaal een beetje anders [3]. Pogingen van Niklaus Wirth om een wereldwijde EBNF-standaard in het leven te roepen zijn mislukt. Dat wil zeggen, er is in 1996 wel een ISO/IEC-standaard 14977 voor EBNF [4, 5] uitgebracht, maar iedereen is zijn eigen EBNF blijven schrijven. Ter onderscheid duid ik de goedbedoelde maar algemeen genegeerde standaardversie in het vervolg aan met 'EBNF-14977'.

Nu is er ook wel een en ander op deze standaard aan te merken [6, 7]. Zo'n twintig jaar geleden ontdekte ik het bestaan ervan toen ik zelf in ontwerpdocumenten, beheerhandleidingen en cursusmateriaal syntactische aanwijzingen begon te schrijven. Een van mijn teleurstellingen was, dat EBNF-14977 geen handzame methode had voor het beschrijven van een lijst van nul of meer door een scheidingsteken gescheiden elementen. En anderzijds werd het lezen van EBNF-formules door de woekering van metasymbolen al gauw een kwelling. Zodoende bleef ik me vijftien jaar lang behelpen met steeds weer opnieuw uitgevonden ad hoc notaties van eigen makelij.

Toen probeerde ik het nog een keer, en stuitte op het bestaan van Augmented BNF (ABNF). Dat was tenminste een standaard — eigenlijk een evoluerende reeks van standaards [8] — die wèl werd nagevolgd. Toch was ook deze vorm niet wat ik zocht. Het werd me duidelijk dat verder naar een heilige graal zoeken in de uitbundige overvloed aan huisvlijt-BNF's geen zin had en dat ik er maar beter zelf een kon smeden. Vanwege sommige aantrekkelijke principes die erin zaten, begon ik met EBNF-14977, paste aan wat zich niet met mijn eigen ideeën verdroeg en voegde toe wat ik miste.

Het ideaal bereikt? Mijn graal voldeed uitstekend in het beschrijven van naamgevingsconventies ten behoeve van een nieuw in te richten Linux-omgeving. Daarnaast viel ik echter toch weer herhaaldelijk terug op allerlei geïmproviseerde notaties, omdat die soepeler in de omgang waren. Ik moest maar onder ogen zien dat *one size fits all* op dit terrein niet werkt. *Two sizes* dan misschien. Aan de slag dus met een canonieke vorm voor een tweede metataal, die een slankere gedaante heeft maar in wezen nog steeds de wezenlijke BNF-principes volgt. Ik geloof dat ik nu wel iets bruikbaars heb.

2 BNF in 1960

In veel programmeertalen ziet een functie-aanroep eruit als in deze voorbeelden van een aanroep zonder argument, met één argument respectievelijk met twee argumenten:

```
f()   g(x)   h(x,y)
```

Laten we stellen dat de functienaam met een letter moet beginnen, verder letters en cijfers mag bevatten, maar niet langer mag zijn dan acht posities. In het oorspronkelijke BNF van 1960 zou de specificatie er zo uit kunnen zien:

```
⟨function designator⟩ ::= ⟨identifier⟩⟨argpart⟩
⟨argpart⟩             ::= ()|(⟨arglist⟩)
⟨arglist⟩             ::= ⟨argument⟩|⟨arglist⟩,⟨argument⟩

⟨identifier⟩          ::= ⟨letter⟩⟨next⟩⟨next⟩⟨next⟩⟨next⟩⟨next⟩⟨next⟩⟨next⟩
⟨next⟩                ::= ⟨empty⟩|⟨next character⟩
⟨empty⟩               ::=
⟨next character⟩      ::= ⟨letter⟩|⟨digit⟩
⟨letter⟩              ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
                          |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
⟨digit⟩               ::= 0|1|2|3|4|5|6|7|8|9
```

De non-terminal ⟨function designator⟩ is hier het lokale startsymbool. De non-terminal ⟨argument⟩ heb ik niet verder uitgewerkt, maar ⟨identifier⟩ wel, tot op de terminals voor de individuele letters en cijfers. Fraai, maar verre van ideaal:

- De taal die beschreven wordt, mag geen scherpe haakjes, verticale strepen of **::=**-sequenties bevatten. BNF is dus niet in staat zijn eigen syntax te beschrijven. In de praktijk is het nog een graadje erger, omdat de kleiner-dan- en groter-dan-tekens worden gebruikt om de non-terminals af te bakenen.
- De status van spaties en regelovergangen is niet geheel duidelijk: onderdeel van een terminal of betekenisloos? Bij een spatie-agnostische taal als Algol 60 kwam men er wel mee weg.
- Wanneer de functienaam maximaal 255 posities lang mag zijn, moeten er flink wat non-terminals ⟨next⟩ worden afgepast (of met verschillende coupures worden gewerkt).
- Alle letters, cijfers (en buiten dit voorbeeld ook andere karakters) moeten uitputtend worden opgesomd. Dat wordt, nu we bijna allemaal aan de Unicode zijn, wel wat bezwaarlijk.
- Herhaling wordt aangegeven door middel van recursiviteit. Dat is niet ieders kop thee.

3 BNF-extensies

Het eerste in het vorige hoofdstuk genoemde bezwaar van BNF is eenvoudig te ondervangen door terminals tussen aanhalingstekens te zetten. Dan hebben non-terminals geen afbakening meer nodig, mits er geen spaties in hun naam worden toegelaten. Spaties buiten de terminals hebben geen betekenis meer en kunnen worden gebruikt om de leesbaarheid te vergroten. Ze zijn nu zelfs vereist om non-terminals te scheiden:

```
function_designator ::= identifier argpart
argpart             ::= "()" | "(" arglist ")"
arglist             ::= argument | arglist "," argument

identifier          ::= letter next next next next next next next
next                ::= empty | next_char
next_char           ::= letter | digit
letter              ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
                      | "J" | "K" | "L "| "M" | "N" | "O" | "P" | "Q" | "R"
                      | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
                      | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
                      | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
                      | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
digit               ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Eén klein nadeel misschien toch: in het Algol-60-rapport [1] komt deze constructie voor:

```
comment ⟨any sequence not containing ;⟩;
```

Hier sjoemelt het rapport een beetje, want wat vormgegeven is als een non-terminal, is dat niet, aangezien er geen regel is gegeven die tot een terminal leidt. Moeten we daar dan in de nieuwe situatie zoiets als hieronder van maken?

```
"comment " any_sequence_not_containing_; ";"
```

Feitelijk hebben we te maken met iets tussen een terminal en een non-terminal in. We zouden van *pseudoterminal* kunnen spreken. In EBNF-14977 is dit fenomeen geformaliseerd tot wat daar *special sequence* heet. Dat is wel zo eerlijk.

Al die aanhalingstekens maken de metataal toch wat 'harig'. Wanneer de typografische mogelijkheden dit toelaten, zijn er andere, minder opdringerige middelen, zoals het vet drukken van de terminals (is echter bij veel niet-alfanumerieke symbolen vaak slecht of niet te onderscheiden van normaal!), het cursiveren van non-terminals en het gebruik van kleur. De syntaxbeschrijving van Lua 5.4 is een aardig voorbeeld van een hybride oplossing, zie Appendix A. Terminals zijn daar vet gezet. De alfanumerieke terminals zonder aanhalingstekens, de niet-alfanumerieke met. De namen van non-terminals beginnen met een kleine letter, die van de pseudoterminals met een hoofdletter.[1] Zo blijft de specificatie visueel vrij schoon.

Andere voorzieningen doen mogelijk wel enige afbreuk aan een rustgevend uiterlijk. De meeste EBNF's ondersteunen iteratie-operatoren om niet op recursie te hoeven leunen. De waarschijnlijk meest gebruikelijke zijn **[**…**]** voor facultatieve componenten, **{**…**}** voor componenten die nul of meer keer kunnen voorkomen, en **(**…**)** voor het groeperen van componenten. Andere veel toegepaste operatoren zijn …***** voor nul of meer keer, en …**⁺** voor één of meer keer.

[1] 'The complete syntax of Lua', meldt de reference trots omdat deze op één pagina past. Maar toch niet helemaal, getuige de drie pseudoterminals die er opgewekt 'terminals' worden genoemd. Bovendien ontbreekt de beschrijving van commentaar in deze syntax. Is ook wel een lastig onderwerp om in een formele grammatica te vangen.

4 EBNF-14977

Aandachtspunten bij het bedenken van een gemeenschappelijke EBNF die al het goede van de reeds bestaande EBNF's zou moeten verenigen, waren precisie, eenduidigheid, uitbreidbaarheid en de mogelijkheid de specificaties op een historische schrijfmachine uit te tikken. Dat laatste is geen strikt verbod op typografische fratsen, maar het moest allemaal toch ook zonder zulke technieken uitvoerbaar zijn.

Pakken we de in de vorige hoofdstukken gegeven definities van een function designator erbij, dan zouden deze er in EBNF-14977 zo uitzien:

```
function designator = identifier, argpart;
argpart             = "(", [arglist], ")";
arglist             = argument, {",", argument};

identifier          = letter, 7 * [letter | digit];
letter              = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I"
                    | "J" | "K" | "L "| "M" | "N" | "O" | "P" | "Q" | "R"
                    | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
                    | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i"
                    | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
                    | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
digit               = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

Een aantal zaken vallen hierin op:

- Elke regel wordt afgesloten met een puntkomma.
- Er wordt gebruik gemaakt van de komma als expliciete concatenatie-operator. Daardoor kunnen non-terminals uit meer woorden bestaan, wat op zich niet zo veel meerwaarde heeft. Maar het maakt ook andere metataaluitbreidingen mogelijk die anders misschien op omslachtiger wijze zouden moeten worden gerealiseerd. Ik kom daar in het volgende hoofdstuk op terug. Nadeel is dat de specificaties er 'hariger' van worden.
- De bovengrens van acht posities in een identifier kan kort en op intuïtieve wijze worden aangegeven.

De opvallendste buitenissigheid in EBNF-14977 is deze formulering van 'één of meer':

```
one or more letters = {letter}-;
```

Lees dit als 'nul of meer letters behalve niets'. Tja, het is wel korter dan:

```
one or more letters = letter, {letter};
```

Grootste tekortkoming is, dat de verzameling van letters nog steeds letter voor letter bij elkaar moet worden geraapt. En mijn persoonlijke teleurstelling:

```
list of zero or more comma separated elements = [element, {",", element}]
```

Dat moet beter kunnen. De conventie voor het benoemen van non-terminals buiten de EBNF-regels, zoals 'function-designator' (zonder de aanhalingstekens), houdt ook niet over: een uit één woord bestaande naam zoals 'letter' onderscheidt zich niet van het woord 'letter'.

Kortom, ik dacht er wel wat aan te verbeteren viel. Niettemin vind ik EBNF-14977 niet te min om de originele BNF te beschrijven. Zelfs uitgekleed tot welhaast BNF-achtige schamelheid is het capabel, zoals Appendix B laat zien.

5 Een reeks overgehaalde BNF's

Het product van mijn verbeterinspanningen heb ik getooid met de naam 'Overhauled BNF' ('OBNF'). Ondersteuning van oude schrijfmachines heb ik laten vallen, maar dat alles in pure ASCII (32 t/m 126) te vatten zou moeten zijn, heb ik in ere gelaten. De operatoren **[**…**]**, **{**…**}** en …*****… van EBNF-14977 heb ik verenigd tot een vermenigvuldigingsoperator in gedaanten die ik van de reguliere expressies à la Perl heb afgekeken. Het enige verrassende dat ik eraan heb toegevoegd, is dat ik er binaire infixoperatoren van heb gemaakt. Ten koste van de vanzelfsprekendheid van de notatie weliswaar, maar met een meer gecondenseerde en consistente schrijfwijze van menigvouden als winst (zie de derde en vierde regel hieronder):

```
function designator = identifier, argpart;
argpart             = "(", arglist?, ")";
arglist             = argument + ",";

identifier          = letter, (letter | digit) {0..7};
letter              = ["A..Za..z"];
digit               = ["0..9"];
```

Verder heb ik, ook weer geïnspireerd door reguliere expressies, zogeheten *character classes* aan de metataal toegevoegd (vijfde en zesde regel). Dat is een krachtig middel.

Tal van andere voorzieningen heb ik in- en aangebouwd. In korte tijd heb ik honderdduizenden OBNF-varianten gecreëerd. Dat was minder moeilijk dan het lijkt, zie Appendix C, waar slechts een bescheiden 4096-tal vluchtig is gekarakteriseerd.

Buiten de eigenlijke OBNF-regels, dat wil zeggen in de omringende tekst waarin de regels zijn ingebed, maar ook binnen pseudoterminals en commentaar, geldt de conventie dat verwijzingen naar non-terminals tussen de symbolen '<' en '>' worden geplaatst. Daarmee is '<letter>' dus zichtbaar iets anders dan 'letter'.

Zo. Klaar.

Zag ik nu in voldoening terug op het werk?

Hmmm…

Ja, ik vind ik wel een verbetering ten opzichte van EBNF-14977. De vrijheid om als schrijver zelf een specifieke OBNF-versie te kiezen en dat in een handomdraai te kunnen verantwoorden is ook wel een sympathiek. Maar qua 'harigheid' zijn we met OBNF nauwelijks beter af. Misschien was het handhaven van de komma als concatenatie-operator een vergissing. Ik ga toch maar eens bekijken welke consequenties een kommaloze uitvoering zou hebben voor andere OBNF-mechanismen. Uit meer woorden bestaande non-terminals zullen wel uitgesloten zijn. Bij de vermenigvuldigingsoperator zullen de operator en zijn operanden eveneens aaneengesloten moeten worden geschreven, lijkt me. In hoofdstuk 7 zal ik in elk geval de uiterlijke verschijningsvorm tentoonstellen.

Hoe dan ook, ik constateer met enige teleurstelling dat ik OBNF zelden toepas en toch steeds weer teruggrijp naar andere notaties. Er is iets fundamenteel mis met de uitgebreide BNF's die ik heb bekeken. Ze richten zich vooral op de formele correctheid en expressiviteit van de specificatietaal. Maar niet voldoende op hoe gemakkelijk de boodschap bij de lezer binnenkomt, op de ergonomie van de taal. Wat als ik nu eens primair uitga van het lezersperspectief, en pas in tweede instantie probeer de verworvenheden van het BNF-principe te behouden? Alle typografische middelen zijn toegestaan.

6 Generic Example Notations

Mensen zijn gebaat bij voorbeelden. Of iedereen de dingen precies zo waarneemt zoals ik, weet ik niet zo zeker. Eigenlijk behoor ik in deze situatie proeven met echte mensen te doen. De omstandigheden zijn er niet helemaal naar. Ik hoop daarom dat ik zelf echt genoeg ben om dan maar te varen op mijn op eigen ondervindingen gebaseerde vermoedens over ergonomie, zoals de gedachte dat mensen gebaat zouden zijn bij realistische voorbeelden. Zodoende ben ik tot een aantal *generieke voorbeeldnotaties* gekomen, of meedobberend op de vaart der volkeren in het Engels gezegd: tot Generic Example Notations (GENs). Hoofdlijnen hiervan:

1. Het rechterlid van een herschrijfregel moet eruitzien als programmacode. De terminals moeten prominent aanwezig zijn, niet overschaduwd door allerlei metasymbolen. Een manier om dit te bereiken is: terminals vet drukken en niet met aanhalingstekens aankleden.
2. Spaties (individueel of als een aaneengesloten reeks) gelden als afzonderlijke terminals.
3. Non-terminals mogen wat bescheidener ogen en dienen evenals terminals vrij te zijn van versieringen zoals afbakeningstekens. Het onderscheid met terminals kan nog wat sterker worden aangezet door de non-terminals te cursiveren en/of een ander lettertype toe te passen.
4. De metataal kent een IS-GEDEFINIEERD-ALS-operator (of als het geen BNF is, een PRODUCEERT-operator).
5. Als de metataal een BNF is, kent deze een OF-operator. In andere gevallen kan het ook zonder. Ik prefereer de BNF-vorm.
6. De metataal kent een aantal operatoren waarmee afzonderlijk of in combinatie de volgende modaliteiten kunnen worden weergegeven: FACULTATIEF, NUL-OF-MEER en ÉÉN-OF-MEER.
7. De taal kent mogelijk een GROEPEER-operator. Dat zal een circumfixoperator moeten zijn, die zijn operanden insluit. Indien de in het vorige punt genoemde operatoren ook van het circumfixtype zijn, is de GROEPEER-operator minder hard nodig (bij afwezigheid van een OF-operator wellicht zelfs geheel overbodig — zie het volgende punt).
8. Het moet mogelijk zijn expliciet een lege string te specificeren, bijvoorbeeld als scheiding tussen twee aaneensluitende non-terminals. Dat kan door middel van een speciaal meta-symbool hiervoor en/of met behulp van de GROEPEER-operator (die daartoe een lege ruimte insluit).
9. De meta-operatoren mogen niet in conflict kunnen komen met de symbolen in de doeltaal. Er zal in de Unicode-tabel altijd wel iets te vinden zijn dat buiten de doeltaal valt, en met typografische middelen is eveneens onderscheid aan te brengen. Een van de ideeën hiervoor is de aanleg van een twee-verdiepingenstructuur, waarin de meta-operatoren zich in de vorm van superscripts 'op zolder' bevinden.
10. De meta-operatoren moeten zo min mogelijk aandacht trekken (maar natuurlijk nog wel zichtbaar zijn) of in de weg staan, zodat de structuur van het gepresenteerde doeltaalfragment goed zichtbaar blijft.

Een GEN-voorbeeld:

```
functionDesignator ⟶ identifier•argpart
argpart            ⟶ (arglist°)
arglist            ⟶ argument,···

identifier         ⟶ firstchar•continuation
continuation       ⟶ nextchar°nextchar°nextchar°nextchar°nextchar°nextchar°nextchar°
letter             ⟶ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
                     |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
digit              ⟶ 0|1|2|3|4|5|6|7|8|9
```

Dit is weer dezelfde definitie van function designator als in de vorige hoofdstukken. De metataal in dit voorbeeld heet om alleen voor mij vanzelfsprekende reden Welsh Rarebit (WR). Als ideaal had ik de manpages van Unix en Linux voor ogen, maar dan zo volwaardig dat de uitdrukkingsmogelijkheden en precisie niet voor die van BNF en flinke delen van EBNF-14977 en OBNF onderdoen. Dat lijkt aardig gelukt, al is er vooral ten opzichte van OBNF wel behoorlijk aan expressiviteit en diversiteit ingeleverd.

De drie puntjes zullen ook zonder kennis van alle WR-subtiliteiten al gauw worden geïnterpreteerd als 'enzovoort'. Dat zit inderdaad heel dicht in de buurt. Maar het kan wel nauwkeuriger worden uitgelegd. Het betreft hier een postfixoperator met twee operanden en drukt uit: 'één of meer keer de eerste operand gescheiden door de tweede'. Het gradenteken is een unaire postfixoperator die zijn operand facultatief maakt. De combinatie van drie puntjes met het gradenteken leidt dan vanzelf tot de betekenis 'nul of meer keer'. Als een identifier oneindig lang mag zijn, kan deze worden gedefinieerd als:

```
identifier ⟶ firstchar•nextchar•···°
```

WR kent verschillende manieren om 'niets' uit te drukken. Groeperingshaakjes zijn een middel dat altijd en op verschillende manieren kan worden ingezet. Alternatieve definities van de function designator zijn bijvoorbeeld:

```
functionDesignator ⟶ identifier()argpart
functionDesignator ⟶ identifier(argpart)
```

Omwille van een 'gladder' uiterlijk prefereer ik echter een afzonderlijk metasymbool • hiervoor, ook al is dit redundant. In de praktijk zul je het betrekkelijk weinig gebruiken. Zo is bijvoorbeeld de function designator veel bondiger te definiëren:

```
functionDesignator ⟶ identifier(argument,···°)
```

Wel zo duidelijk.

Een aantal specifieke concepten uit OBNF zijn in WR overgenomen. Het belangrijkste is de 'nooduitgang' oftewel *special sequence* alias *pseudoterminal* **⟨…⟩**. Hiermee kun je in eigen woorden uitleggen wat je bedoelt. Langs deze weg zou je andere voorzieningen van OBNF of een willekeurige taal naar binnen kunnen smokkelen, bijvoorbeeld:

```
letter ⟶ ⟨ OBNF: ["A..Za..z"] ⟩      ⩾ OBNF character class ⩽
letter ⟶ ⟨ PCRE: /[A-Z]/i ⟩          ⩾ Perl-compatible regular expression ⩽
```

Een extraatje dat niet in OBNF zit, is de *default*-operator:

```
routineDirective ⟶ ::ROUTINE routinename (PRIVATE× | PUBLIC) (EXTERNAL spec)°
```

Het kruisje achter **PRIVATE** geeft aan dat dit woord mag worden weggelaten zonder dat de betekenis van de expressie in de doeltaal verandert. Een semantische aanwijzing bovenop de syntaxdefinitie zogezegd.

Repeterende spatie-terminals gelden uiteindelijk als een ongespecificeerde hoeveelheid witruimte. Bovenstaande definitie schrijft dus niet voor dat er maar één spatie tussen **::ROUTINE** en de routinenaam mag voorkomen en dat er tussen de routinenaam en **PUBLIC** twee spaties moeten staan. Er zijn weinig talen waarin de hoeveelheid verschil maakt. Met regelovergangen kan het wel lastiger worden, want veel talen vatten deze op als beëindiging van een statement.

7 De proef op de som

Welsh Rarebit heb ik vooral bedacht om informele en minder tot op het bot doorgespecificeerde syntaxbeschrijvingen te kunnen opstellen die in een spreekwoordelijke oogopslag zijn te doorgronden. Maar het moest wel kloppen en zich ertoe lenen om als dat zo uitkomt toch tot het uiterste te gaan. Dit hoofdstuk is echter primair een schoonheidswedstrijd. Het oog telt.

7.1 EERSTE RONDE

Het speelveld is onderstaande definitie van een lambda-expressie in Scheme, zoals te vinden is in R5RS [9].

```
⟨lambda expression⟩  →  (lambda ⟨formals⟩ ⟨body⟩)
⟨formals⟩            →  (⟨variable⟩*)
                        |  ⟨variable⟩
                        |  (⟨variable⟩+ . ⟨variable⟩)
⟨body⟩               →  ⟨definition⟩* ⟨sequence⟩
⟨definition⟩         →  (define ⟨variable⟩ ⟨expression⟩)
                        |  (define (⟨variable⟩ ⟨def formals⟩ ⟨body⟩)
                        |  (begin ⟨definition⟩*)
⟨def formals⟩        →  ⟨variable⟩
                        |  ⟨variable⟩* . ⟨variable⟩
⟨sequence⟩           →  ⟨command⟩* ⟨expression⟩
⟨command⟩            →  ⟨expression⟩
```

De notatie wordt in het rapport aldus toegelicht:

> This section provides a formal syntax for Scheme written in an extended BNF. All spaces in the grammar are for legibility. Case is insignificant; for example, **#x1A** and **#X1a** are equivalent. ⟨empty⟩ stands for the empty string. The following extensions to BNF are used to make the description more concise: ⟨thing⟩* means zero or more occurrences of ⟨thing⟩; and ⟨thing⟩+ means at least one ⟨thing⟩.

Anders dan gesuggereerd, zijn spaties wel degelijk relevant in Scheme. Soms kun je ze inderdaad weglaten, maar vaak pakt dat verkeerd uit. In een constructie als ⟨variable⟩* moeten alle ⟨variable⟩s toch echt onderling met witruimte worden gescheiden. Ook de punt in S-expressies kan maar beter wat wit om zich heen hebben. Jammer verder dat de ronde Scheme-haakjes niet meer zo goed opvallen te midden van de scherpe metahaakjes waarmee de non-terminals zijn afgezet. Afgezien van deze kritiekpunten oogt de specificatie wel overzichtelijk en transparant.[2]

Achtereenvolgens zullen hier voorbijtrekken: EFNB-14977, OBNF, OBNF zonder komma's, Slumberland (de directe voorloper van WR), en tot slot WR.

[2] In het volgende rapport R6RS [10] hebben de opstellers het wijselijk over een andere boeg gegooid. Wat de verdiensten van hun nieuwe metataal zijn, kan ik niet meer beoordelen. Om er iets van te begrijpen zou ik eerst vier in het rapport aanbevolen boeken moeten lezen.

EBNF-14977:

```
lambda expr = "(lambda ", formals, " ", body, ")";
formals     = "(", [variable, {" ", variable}], ")"
            | variable
            | "(", {variable, " "}-, ". ", variable, ")";
body        = {definition, " "}, sequence;
definition  = "(define ", variable, " ", expression, ")"
            | "(define (", variable, " ", def formals, ") ", body, ")"
            | "(begin", {" ", definition}, ")";
def formals = variable
            | {variable, " "}, ". ", variable;
sequence    = {command, " "}, expression;
command     = expression;
```

OBNF (met komma's):

```
Lambda expr = "(lambda ", formals, " ", body, ")";
formals     = "(", variable*" ", ")"
            | variable
            | "(", variable+" ", " . ", variable, ")";
body        = (definition, " ")*, sequence;
definition  = "(define ", variable, " ", expression, ")"
            | "(define (", variable, " ", def formals, ") ", body, ")"
            | "(begin", (" ", definition)*, ")";
def formals = variable
            | variable*" ", " . ", variable;
sequence    = (command, " ")*, expression;
command     = expression;
```

OBNF zonder komma's:

```
lambdaExpr = "(lambda " formals " " body ")";
formals    = "(" variable+" " ")"
           | variable
           | "(" variable+" " " . " variable ")";
body       = (definition " ")* sequence;
definition = "(define " variable " " expression ")"
           | "(define (" variable " " defFormals ") " body ")"
           | "(begin" (" " definition)* ")";
defFormals = variable
           | variable*" " " . " variable;
sequence   = (command " ")* expression;
command    = expression;
```

Slumberland:

lambdaExpr → **(lambda** *formals body***)**
formals → **(**⌈*variable* ···⌉**)**
| *variable*
| **(***variable* ··· **.** *variable***)**
body → ⌈*definition* ···⌉ *sequence*
definition → **(define** *variable expression***)**
| **(define (***variable defFormals***)** *body***)**
| **(begin** ⌈*definition* ···⌉**)**
defFormals → *variable*
| ⌈*variable* ···⌉ **.** *variable*
sequence → ⌈*command* ···⌉ *expression*
command → *expression*

Welsh Rarebit:

```
lambdaExpr  → (lambda formals body)
formals     → (variable ...°)
            | variable
            | (variable ... . variable)
body        → definition ...° sequence
definition  → (define variable expression)
            | (define (variable defFormals) body)
            | (begin definition ...°)
defFormals  → variable
            | variable ...° . variable
sequence    → command ...° expression
command     → expression
```

7.2 TWEEDE RONDE

De definitie van het if-statement in Lua. De documentatie van Lua 5.4 [11] verwoordt het ongeveer zo:

```
ifstmt ::=
    if expr then block {elseif expr then block} [else block] end
```

EBNF-14977:

```
ifstmt =
    "if ", expr, " then ", block, {" elseif ", expr, " then ", block}, [" else " block], " end ";
```

OBNF (met komma's):

```
ifstmt =
    "if ", expr, " then ", block, (" elseif ", expr, " then ", block)*, (" else ", block)?, " end ";
```

OBNF zonder komma's:

```
ifstmt =
    "if " expr " then " block (" elseif " expr " then " block)* (" else " block)? " end ";
```

Slumberland:

```
ifstmt →
    if expr then block ⌈elseif expr then block⌉ ... ⌈else block⌉ end
```

Welsh Rarebit:

```
ifstmt →
    if expr then block ⟨elseif expr then block⟩ ...° ⟨else block⟩° end
```

7.3 HOE REPRESENTATIEF WAREN DEZE TESTS?

Het weglaten van de komma's uit OBNF geeft weinig verlichting, het lijkt zelfs af en toe schadelijk uit te pakken. Nu vormen Scheme en andere verwanten uit de Lisp-familie misschien niet zo'n representatief speelterrein, omdat deze talen nauwelijks met komma's werken. Voor de kwestie 'wel of geen komma's?' zou je beter kunnen kijken hoe beide alternatieven zich gedragen bij een functieaanroep in Algol-achtige stijl en zo'n zelfde aanroep in Lisp-stijl. Laten we dat eens proberen:

OBNF (met komma's):

```
algol call = "smurf(", tic, ",", tac, ",", toe, ")"
lisp call  = "(smurf", " ", tic, " ", tac, " ", toe, ")"
```

OBNF zonder komma's:

```
algolCall = "smurf(" tic "," tac "," toe ")"
lispCall  = "(smurf" " " tic " " tac " " toe ")"
```

Slumberland en Welsh Rarebit:

```
algolCall → smurf(tic, tac, toe)
lispCall  → (smurf tic tac toe)
```

In de tweede ronde scoort de notatie van de Lua-documentatie zo goed omdat het if-statement geen komma's, ronde haakjes, teksthaakjes of accolades bevat. Dat scheelt wel de nodige metaquotes. In oudere Lua-publicaties die ik ken, zijn de non-terminals ook nog eens gecursiveerd, wat deze notatie nog beter maakt. De kommakwestie is ook in deze ronde niet echt beslist, want zonder de komma's is het lastiger te zien welke gedeelten tussen aanhalingstekens staan en welke juist daarbuiten. Met gekrulde aanhalingstekens zou dat duidelijk te maken zijn, maar dat is strijdig met het beginsel dat het allemaal in ASCII te gieten zou moeten zijn. Om een idee te geven, hier toch nog een presentatie met de alternatieve aanhalingstekens:

OBNF (met komma's):

```
ifstmt =
    “if ”, expr, “ then ”, block, (“ elseif ”, expr, “ then ”, block)*, (“ else ”, block)?, “ end ”;
```

OBNF zonder komma's:

```
ifstmt =
    “if ” expr “ then ” block (“ elseif ” expr “ then ” block)* (“ else ” block)? “ end ”;
```

Maar als we toch die kant opgaan, kunnen we net zo goed ander typografisch geschut in stelling brengen. Denk in eerste instantie aan het vet zetten van terminals en het cursiveren van non-terminals. Zoals:

```
ifstmt =
    “if ” expr “ then ” block (“ elseif ” expr “ then ” block)* (“ else ” block)? “ end ”;
```

De lezer mag zo langzamerhand wel zijn oordeel geven of heimelijke keuzes maken. En ook zelf situaties, oplossingen en complete metatalen bedenken natuurlijk.

8 Nog wat over formuleringsstijlen

Het is me nog niet helemaal duidelijk wat in het rechterlid van een regel de beste volgorde van componenten is. Neem nu eens de volgende regel in de oorspronkelijke BNF:

```
⟨list⟩ ::= ⟨element⟩ | ⟨list⟩,⟨element⟩
```

In eerste instantie gaat het om de keuze tussen ⟨element⟩ enerzijds en ⟨list⟩**,** ⟨element⟩ anderzijds. Het is een recursieve definitie. Het komt me inderdaad logisch voor om in zo'n geval te beginnen met het stopcriterium ('Hebbes! Ik heb hier een geschikte lijst van één element') alvorens te kijken naar een combinatie met ander materiaal dat ook op te rapen valt.

In tweede instantie gaat het om hoe je de nieuwe vondst combineert met wat je al had ingezameld. In bovenstaand voorbeeld (en in formulering (*a*) hieronder) komt het nieuwe exemplaar aan het eind. Maar had het niet net zo goed aan het begin van de opgebouwde reeks kunnen worden geplaatst (formulering (*b*) hieronder)? Is het louter een kwestie van conventie, zit er meer achter, of is het pure willekeur? Ik heb grammatica's gezien die in dit opzicht inconsistent waren. Maar slordigheid is menselijk, dus dat zegt ook weer niet zo heel veel.

```
(a)  ⟨list⟩ ::= ⟨element⟩ | ⟨list⟩,⟨element⟩
(b)  ⟨list⟩ ::= ⟨element⟩ | ⟨element⟩,⟨list⟩
(c)  list    = {element, ","}, element;
(d)  list    = element, {",", element};
```

Hetzelfde probleem speelt bij een meer geavanceerde BNF-taal, bijvoorbeeld EBNF-14977: formulering (*c*) of formulering (*d*), maakt het werkelijk enig verschil?

Bij metatalen die geen NUL-OF-MEER-operator kennen, is weer iets anders aan de hand. Je zult deze operatie zelf moeten samenstellen met behulp van de EEN-OF-MEER-operator en de FACULTATIEF-operator. De werking kan op twee manieren worden uitgedrukt. De goede, universele manier is: 'facultatief één of meer keer'. De verwerpelijke, principieel verkeerde manier is: 'één of meer keer facultatief'. Deze uitdrukking pakt verkeerd uit als er een scheidingsteken anders dan een spatie tussen de te herhalen elementen voorkomt. Gaat het om spaties, dan kan men zijn schouders er nog over ophalen, want productie van overbodige spaties kan doorgaans geen kwaad. In Slumberland op pagina 12 kon ik de verleiding niet weerstaan omdat het me daar een paar haakjes scheelde… didactisch heel onverantwoord! Die verleiding is er bij WR niet.

Nu zou je kunnen stellen dat een regel als deze op pagina 8 dus eigenlijk ook onwenselijk is:

```
continuation ⟶ nextchar°nextchar°nextchar°nextchar°nextchar°nextchar°nextchar°
```

Dat dit de betere formulering is:

```
continuation ⟶ (nextchar(nextchar(nextchar(nextchar(nextchar(nextchar(nextchar)°)°)°)°)°)°)°
```

Of anders (let op de volgorde, en belangrijker, onderken de zwakte van deze spitsvondigheid!):

```
continuation ⟶ next4chars°next2chars°nextchar°
next4chars   ⟶ next2chars•next2chars
next2chars   ⟶ nextchar•nextchar
```

Wanneer mag er water bij de wijn?

Appendix A 'The Complete Syntax of Lua'

Onderstaande tekst is overgenomen uit [11]:

Here is the complete syntax of Lua in extended BNF. As usual in extended BNF, {A} means 0 or more As, and [A] means an optional A. (For operator precedences, see §3.4.8; for a description of the terminals Name, Numeral, and LiteralString, see §3.1.)

```
chunk            ::= block
block            ::= {stat} [retstat]
stat             ::= ';'
                   | varlist '=' explist
                   | functioncall
                   | label
                   | break
                   | goto Name
                   | do block end
                   | while exp do block end
                   | repeat block until exp
                   | if exp then block {elseif exp then block} [else block] end
                   | for Name '=' exp ',' exp [',' exp] do block end
                   | for namelist in explist do block end
                   | function funcname funcbody
                   | local function Name funcbody
                   | local attnamelist ['=' explist]
attnamelist      ::= Name attrib {',' Name attrib}
attrib           ::= ['<' Name '>']
retstat          ::= return [explist] [';']
label            ::= '::' Name '::'
funcname         ::= Name {'.' Name} [':' Name]
varlist          ::= var {',' var}
var              ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
namelist         ::= Name {',' Name}
explist          ::= exp {',' exp}
exp              ::= nil | false | true | Numeral | LiteralString | '...'
                   | functiondef | prefixexp | tableconstructor
                   | exp binop exp | unop exp
prefixexp        ::= var | functioncall | '(' exp ')'
functioncall     ::= prefixexp args | prefixexp ':' Name args
args             ::= '(' [explist] ')' | tableconstructor | LiteralString
functiondef      ::= function funcbody
funcbody         ::= '(' [parlist] ')' block end
parlist          ::= namelist [',' '...'] | '...'
tableconstructor ::= '{' [fieldlist] '}'
fieldlist        ::= field {fieldsep field} [fieldsep]
field            ::= '[' exp ']' '=' exp | Name '=' exp | exp
fieldsep         ::= ',' | ';'
binop            ::= '+' | '-' | '*' | '/' | '//' | '^' | '%'
                   | '&' | '~' | '|' | '>>' | '<<' | '..'
                   | '<' | '<=' | '>' | '>=' | '==' | '~='
                   | and | or
unop             ::= '-' | not | '#' | '~'
```

Appendix B BNF gedefinieerd in een uitgeklede EBNF

Onderstaande grammatica beschrijft BNF in EBNF-14977, waarvan hieronder alleen de meta-operatoren "=" en "|" zijn toegepast. De frase **?**…**?** is een zogeheten *special sequence*, waarin de definitie van de desbetreffende component op informele wijze kan worden gespecificeerd.

```
grammar     = rule | grammar, ? whitespace ?, rule;
rule        = placeholder, "::=", fork;
placeholder = "⟨", ? words ?, "⟩";
fork        = chain | fork, "|", chain;
chain       = atom | chain, atom;
atom        = placeholder | terminal;
terminal    = ? anything not being part of the BNF language itself ?;
```

Appendix C Abridged 'Summary of OBNF features'

The basic features of OBNF are the same as EBNF according to ISO/IEC 14977, but some are implemented differently:

Feature	EBNF-14977	OBNF-0
Rule with rule terminator	*nonterminal* `=` *definitionlist* `;`	*nonterminal* `=` *definitionlist* `;`
Terminal	`'`*characters*`'` | `"`*characters*`"`	`'`*characters*`'` | `"`*characters*`"`
Alternatives	*definition* `|` ···	*definition* `|` ···
Concatenation	*term* `,` *term*	*term* `,` *term*
Exception	*product* `-` *product*	*product* `-` *product*
Repetition, *n* times	*n* `*` *element*	*element* `{`*n*`}`
Repetition, zero or more times	`{`*definitionlist*`}`	*element* `*`
Optional	`[`*definitionlist*`]`	*element* `?`
Grouping	`(`*definitionlist*`)`	`(`*definitionlist*`)`
Special sequence	`?`*informalSpecification*`?`	`[[`*informalSpecification*`]]`
Block comment	`(*`*comment*`*)`	`#<<`*comment*`>>`

Some extra features in OBNF:

1. The family of postfix multipliers (`{`*n*`}`, `*`, `?`) is supplemented with `{`*min*`..}`, `{`*min*`..`*max*`}`, and `+` — the latter being the equivalent of `{1..}`.
2. Better support for list separators by considering all multipliers to be binary infix operators: *element* `{`*n*`}` *separator, element* `*` *separator, element* `+` *separator*, and so on.
4. Character class: `[`*definitionlist*`]` meaning 'one character from *definitionlist*'.
8. Multi-line terminal symbol: `<<`*characters*`>>`.
16. End-of-line comment: `#` *comment*.
32. Tagging: `<`*tag*`>`.
64. Stretching of composite delimiters by equals signs: `[[`…`]]` to `[=[`…`]=]`, `[==[`…`]==]`, and so on; same for `#<<`…`>>` and `<<`…`>>`.
128. Coupled expansion ('cloning') of non-terminals: within a rule, all occurrences of a specific non-terminal preceded by the metasymbol `%` are forced to expand to the same terminal.
256. Self-defined function with zero or more semicolon-separated arguments: *function*`(`*arglist*`)` — ISO/IEC 14977 mentions the possibility of such an extension as well, but does not elaborate on it.
512. Presence of a substring: `/`*definitionlist*`/` meaning 'contains *definitionlist*'.
1024. Pointing to a character by its sequence number: `"0A"x`, `"10"d`, `"12"o`, and `"1010"b` all represent the newline character (shorter than the feature 4 phrase `["<<0A>>"]`).
Internal concatenation is also supported: `"0D,0A"x`, `"13,10"d`, etc define strings of multiple characters (in this case a CR+LF sequence).
2048. The terminals `"ABC"i` and `"abc"i` are case-insensitive, both representing `(["Aa"], ["Bb"], ["Cc"])`, which is the same as `("ABC" | "ABc" | "AbC" | "Abc" | "aBC" | "aBc" | "abC" | "abc")`.

Add up the desired feature numbers to assemble your own favorite OBNF version (from OBNF-0 = *base* upwards). The power of OBNF-133 (1+4+128) measures up to what regular expressions usually offer (provided that we ignore more advanced facilities such as lookahead and lookbehind assertions — zero-width lookahead is facility 8192, by the way).

At the time of writing this summary, OBNF-1023 is fully specified in OBNF.

Appendix D Referenties

[1] Peter Naur e.a., *Report on the Algorithmic Language ALGOL 60*. CACM, juni 1960. http://www.softwarepreservation.org/projects/ALGOL/report/Algol60_report_CACM_1960_June.pdf

[2] *Backus-Naur form*. Wikipedia. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[3] *Extended Backus-Naur form*. Wikipedia. https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form

[4] *ISO/IEC 14977 : 1996(E)*. International Organization for Standardization en International Electrotechnical Commission, 1996. https://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

[5] R.S. Scowen, *Extended BNF — A generic base standard*. 17 september 1998. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[6] Vadim Zaytsev, *BNF WAS HERE: What have we done about the unnecessary diversity of notation for syntactic definitions*. Amsterdam, Centrum Wiskunde en Informatica, 2012. https://www.grammarware.net/text/2012/bnf-was-here.pdf

[7] David A. Wheeler, *Don't use ISO/IEC 14977 Extended Backus-Naur Form (EBNF)*. 18 februari 2020. https://dwheeler.com/essays/dont-use-iso-14977-ebnf.html

[8] *Augmented Backus-Naur form*. Wikipedia. https://en.wikipedia.org/wiki/Augmented_Backus%E2%80%93Naur_form

[9] Richard Kelsey e.a., *Revised⁵ Report on the Algorithmic Language Scheme*. 20 februari 1998. https://schemers.org/Documents/Standards/R5RS/r5rs.pdf

[10] Michael Sperber e.a., *Revised⁶ Report on the Algorithmic Language Scheme*. 26 september 2007. http://www.r6rs.org/final/r6rs.pdf

[11] Roberto Ierusalimschy e.a., *Lua 5.4 Reference Manual*. 3 maart 2021. http://www.lua.org/manual/5.4/