Extending Grammars

Going beyond the simple grammar defining notations for Chomsky type 2 languages such as BNF (Backus Naur Form) and extended BNF, and Chomsky type 3 languages such as regular expressions, there are mechanisms for defining Chomsky type 1 and type 0 languages. The Chomsky classification of languages and grammars was covered earlier in the course.

One of the most straightforward notations are W-grammars named after (Prof Aad van Wijngaarden) and covered in the book “Grammars for Programming Languages” by J Craig Cleaveland and Robert C Uzgalis, (North Holland, 1977), unfortunately now out of print and unobtainable. The capabilities of such a notation and how it may be used to extend the scope and clarity of language definition mechanisms will be outlined.

**W-grammar notation.**

Consider a simple type-2 language, which can easily be defined in BNF. In W-grammar notation it would be written:

```
program: stmt;
    program, stmt.

stmt   : id, becomes symbol, number.

id     : letter; id, letter.

letter : letter a symbol;
    letter b symbol;
    letter c symbol;
    letter d symbol;
    letter e symbol;
    letter f symbol;
    letter g symbol;
    letter h symbol;
    letter i symbol;
    letter j symbol;
    letter k symbol;
    letter l symbol;
    letter m symbol;
    letter n symbol;
    letter o symbol;
    letter p symbol;
    letter q symbol;
```

;  rule separator
.  rule terminator
,  word separator

whitespace is ignored
This simple grammar defines a language which recognises a sequence of assignment statements. Such as the following:

\[ a := 44 \]
\[ hello := 1234 \]

The grammar contains a lot of repetitive rules, which could be shortened. The W-grammar notation permits grammar rules to contain be specified by rules themselves, and by using this mechanism we can write the above grammar in a much shorter form.

\[ SEQ ::= stmt; digit; letter. \]
\[ ALPHA ::= a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z. \]
\[ DIG ::= one; two; three; four; five; six; seven; eight; nine. \]
The combination of Meta-Productions and Hyper-Rules create a set of production rules that are used to parse the language. In this case the notation is used to write the grammar in a more compact way than might be done with a Type 2 grammar, and still only represents a simple Type 2 language. The abstract concepts such as sequencing, lists and repetition are just clearly shown. Using the same notation, however, it is possible to define languages for which the Meta-productions and Hyper-rules expand to non-finite set of production rules, which could never be written in a Type 2 grammar.

Consider the following examples which are used to define any arbitrary application of sequencing, listing and nesting:

```
program: stmt sequence .
stmt : letter sequence, becomes symbol, 
     digit sequence .
letter : letter ALPHA symbol.
digit : DIG symbol.
SEQ sequence : SEQ; SEQ sequence, SEQ.
```

The combination of Meta-Productions and Hyper-Rules create a set of production rules that are used to parse the language. In this case the notation is used to write the grammar in a more compact way than might be done with a Type 2 grammar, and still only represents a simple Type 2 language. The abstract concepts such as sequencing, lists and repetition are just clearly shown. Using the same notation, however, it is possible to define languages for which the Meta-productions and Hyper-rules expand to non-finite set of production rules, which could never be written in a Type 2 grammar.

Consider the following examples which are used to define any arbitrary application of sequencing, listing and nesting:

```
ALPHA :: 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.
NOTION :: ALPHA; NOTION ALPHA.
EMPTY :: .
NOTION option : NOTION; EMPTY .
NOTION sequence : NOTION ;
     NOTION sequence, NOTION.
NOTION list : NOTION ;
     NOTION list, comma symbol, NOTION .
NOTION pack : left symbol, NOTION, right symbol .
```
Context Sensitivity

In substituting the values for a meta-notion, such as SEQ, NOTION, ALPHA and so on, W-grammars have a rule that the same symbol must represent the same value in any single expansion of the same hyper-rule. It cannot represent a different thing on the left hand side and the right hand side, for example. Without this constraint the above examples would not work! This feature, called **uniform replacement**, enables more powerful grammars to be constructed.

Consider the problem of a grammar for a language of a sequence of N letter a’s followed by N letter b’s. This could not be done in a Chomsky Type 3 grammar, but can be done in a Type 2:

```
s : letter a symbol, letter b symbol ;
    letter a symbol, s, letter b symbol.
```

This can be re-written using Meta-Notions:

```
N :: n; Nn .
S : Na, Nb .                 - sequence of N a’s and N b’s

nNa : letter a symbol, Na.  - mapping to symbols
nNb : letter b symbol, Nb.
na  : letter a symbol.
nb  : letter b symbol.
```

This grammar illustrates some new points. It uses the **Uniform Replacement** mechanism to ensure the number of A’s matches the number of b’s. It also expands to a potentially infinite number of production rules, whereas earlier examples showed finite numbers of rules. It also uses a counting mechanism followed by rules that map from the concrete representation into the abstract notions of the rules. These mechanisms are very powerful and can be used for much more complex problems. One example, would be to extend the language to three letters a,b,c but still retain identical numbers of symbols of each. This example becomes a simple extension of the one above:

```
N :: n; Nn .
ABC :: a; b ; c.
S : Na, Nb, Nc .             - sequence of N a’s + N b’s + N c’s

nN ABC : letter ABC symbol, N ABC. - mapping to symbols
n ABC : letter ABC symbol.
```

In this grammar we have added one further meta-notion ABC to make it generic, and simplify the hyper-rules even further.
**Arithmetic**

The previous examples utilised a simple counting system. This can be expanded as a mechanism for representing all arithmetic as grammar constructs, and thereby formally define those language features without further semantic explanation, showing how Type-1 grammars can define more than Type-2 grammars.

In representing numbers in the arithmetic system, a monadic notation is used. It is a simple tally system using a sequence of tokens to represent a number. For example, in such a system:

- i would be one
- ii would be two
- iii would be three
  and so on

We can represent this in the meta-productions:

- **N :: Ni; .** – Natural Numbers
- **P :: Ni** – Positive Numbers

**Adding**

Adding can be achieved in this system by placing counters next to each other.

- **NN’ : N, N’**.

Or

- **NN’ : N plus N’**.

**Decimal**

We can count in decimal (based 10) by grouping into tens:

- **D :: E; EF.**
- **E :: i; ii; iii; iiii; .**
- **F :: iiii .**

**Sequences**

We can manage several numbers at once, but separating them by another symbol, such as a j.
Multiplication

We can use sequences to manage a count, and use it to perform iterated adding, by the following method:

\[
\text{N times } N' : N \text{ times } N'. \\
N : i \text{ times } N . \\
N N' : i \text{ times } N p N' .
\]

Subtraction and Division

These can be dealt with reversing the left and right hand sides of addition and multiplication methods.

Powers

This can be performed as sequences of multiplications.

Prime Numbers

Even prime numbers can be defined, and an example grammar for this is on the web page.

Contextual use of Invisible Productions

Another mechanism that can be used with 2-level grammars is that of Invisible Productions. These do not map to concrete symbols in the final language but act in the hyper-rules to provide some extra context sensitivity. To illustrate this, consider an example grammar that specifies that letters in a symbol can only occur in alphabetic order. This could not be achieved in a Type-2 grammar.

\[
\text{ALPHA :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.}
\]

\[
\text{NOTETY :: EMPTY; NOTETY ALPHA.}
\]

\[
\text{EMPTY :: .}
\]

\[
\text{S : letter ALPHA’ symbol, letter ALPHA’’ symbol, where ALPHA’ precedes ALPHA’’ in abcdefghijklmnopqrstuvwxyz .}
\]
where ALPHA’ precedes ALPHA’’ in NOTETY’ ALPHA’ NOTEY’’
ALPHA’’ NOTETY’’’ : EMPTY.

Note the empty production in the last rule.

**Putting it all together**

A final example, which puts all the features together, is a grammar that enforces Fermat’s Last Theorem. It only accepts those numbers which adhere to the theorem! It includes counting, arithmetic, powers, context rules and so on.

\[
\begin{align*}
N &:: i ; Ni . \\
EMPTY &:: . \\
NETY &:: N; EMPTY. \\
RADIX &:: iii iii iii I . \\
EXP &:: iiN.
\end{align*}
\]

\[
\begin{align*}
\text{fermat} &: \text{EXP pr } N1, \text{equal symbol,} \\
&\quad \text{EXP pr } N2, \text{plus symbol,} \\
&\quad \text{EXP pr } N3, \\
&\quad \text{where } N1 \text{ is } N2N3.
\end{align*}
\]

\[
\begin{align*}
N1 \text{ pr } N2 &: \text{N3 number, power symbol, } N1 \text{ number,} \\
&\quad \text{where } N2 \text{ is } N3 \text{ to the } N1 \text{ power.} \\
&\quad \text{where } N1 \text{ is } N2 \text{ to the } NETY \text{ i power :} \\
&\quad \quad \text{where } N3 \text{ is } N2 \text{ to the } NETY \text{ power,} \\
&\quad \quad \quad \text{where } N1 \text{ is } N3 \text{ times } N2; \\
&\quad \quad \quad \text{where } N1 \text{ is } N2, \text{ where } NETY \text{ is EMPTY.}
\end{align*}
\]

\[
\begin{align*}
&\quad \text{where } NETY1 \text{ N is } N \text{ is } N \text{ time } NETY2 \text{ i :} \\
&\quad \quad \text{where } NETY1 \text{ is } N \text{ times } NETY2 ; \\
&\quad \quad \text{where } NETY1 \text{ NETY2 is EMPTY.}
\end{align*}
\]

\[
\begin{align*}
&\quad \text{where } NETY \text{ is } NETY : \text{EMPTY.}
\end{align*}
\]

\[
\begin{align*}
N1 \text{ number} &: N1 \text{ token; } N2 \text{ number, NETY token,} \\
&\quad \text{where } N3 \text{ is } N2 \text{ times } RADIX, \\
&\quad \text{where } N1 \text{ is } NETY \text{ N3.}
\end{align*}
\]

**Conclusions**

- A Grammar can describe all computer functions and capabilities
- All functions of a language can be defined in a grammar
- Full formal and correct language specifications
- Avoidance of language faults
- Avoidance of ambiguity caused by natural language descriptions