A Common Model for Language Grammars

First published 16th June 2013 and last updated September 2016

Essentials

- In language theory grammars define the rules for parsing and generating text
- Grammar notations vary, but have common concepts and principles
- This paper compares the core concepts of the most popular grammar notations
- Finally, this paper extends these core concepts with features useful to real-world parser generators

Abstract

In language theory grammars are used to define a set of production rules for strings in a formal language. In computer science grammars are applied to the need to recognize whether a given sequence of characters belongs to the language or is grammatically correct. In other words one is generally used for output and the other for input respectively.

The formal theory of grammars is well established (Chomsky, 1956) and recent contributions to theory (Ford, 2004) concern parser expression grammars. In computer science grammars have been used in a wide range of applications including as a notation for the definition of computer languages beginning with Algol (Backus, 1957).

Since Backus Naur Form popular grammar notations have been developed by Wirth (Wirth, 1977) Extended Backus Naur Form (EBNF), “BS6154 – A Standard Syntactic Meta-language” (BSI, 1981), ISO 14977 (ISO, 1997) and regular expressions (Kleene, 1956) (Thomson, 1968).

This paper examines three aspects of language grammars relevant to parsers. The first section reviews common grammar notations. The second section introduces a common model for their features expressed using EBNF and XML notations. The final section examines grammar extensions that are practical considerations for parsers and generators.

A Comparison of Language Grammars

In order to compare grammars we first need to understand their features expressed through notation. Features defined in this section include grammars themselves, rules, terminal strings (for production and consumption), non-terminals (rule references), sequences, groups, options, repetitions, choices, pipelines, predicates, empty sequences and entities. We compare the notations and terminology of (Chomsky, 1956), (Ford, 2004), (Backus, 1957), (Wirth, 1977), (ANSI, 1989) and (Thomson, 1968).

Grammar

A grammar defines a collection of rules, which form the definition of a language. A grammar is incomplete when it contains a rule reference that is not defined by syntax rules in the grammar.

Grammars are identified by name, however, none of the notations outlined in this paper provide a way of defining a grammar's identity. The absence of an identifier limits the ability to cross reference and reuse rules in different grammars.

Example Notation	Source
None given	(Chomsky, 1956)
None given	(Ford, 2004)
None given	(Backus, 1957)
None given	(Wirth, 1977)
None given	(ANSI, 1989)
None given	(ISO, 1997)
None given	(Thomson, 1968)

Rules

Chomsky coined the term production rules because in his view of the world text was produced from the grammars. In computer science rules are usually referred to as syntax rules and serve to recognize or parse text. Rules have a unique name that can be referenced in other parts of the grammar. Some notations permit spaces in rule names and some have an explicit terminator. Rules are followed by a defining expression.

Example Notation	Source
$name \rightarrow$	(Chomsky, 1956) denotes production
$name \leftarrow$	(Ford, 2004) denotes recognition or parsing, no terminator
<name>::=	(Backus, 1957) – no terminator
name= .	(Wirth, 1977) – dot terminator
name:	(ANSI, 1989) – no terminator
name= ;	(ISO, 1997) – names may have spaces; semicolon or dot are permitted alternative terminators
None given	(Thomson, 1968)

Primary

A syntactic primary, (ISO, 1997), includes an optional sequence, repeated sequence, grouped sequence, rule reference, terminal string or empty sequence. A primary is an abstract concept and is useful for describing features common across syntactic primaries. Pipelines are also a primary.

Terminal

Terminals are literal symbols (or strings of characters) which may appear in the production or parsing rules of a formal grammar and cannot be changed using the rules of the grammar. Often surrounded by quotes, terminals can define both parsed input (consumption) and produced output (production), however, the following notations cannot show which type of terminal is being defined.

Consumption Terminal

Example Notation	Source
None given	(Chomsky, 1956)
name <- t	(Ford, 2004)
<name>::=t	(Backus, 1957)
name="t".	(Wirth, 1977)
name: **t**	(ANSI, 1989) – in bold
name='t';	(ISO, 1997) – either single or double quotes
t	(Thomson, 1968)– regular expressions include characters, hexadecimal representations of characters (character ranges and predefined ranges more properly are pseudo non-terminals)

Production Terminal

Example Notation	Source
$name \to t$	(Chomsky, 1956) where 't' is not defined by a rule
None given	(Ford, 2004)
None given	(Backus, 1957)
None given	(Wirth, 1977)
None given	(ANSI, 1989)
None given	(ISO, 1997)
None given	(Thomson, 1968)

Non-Terminal

Non-terminal symbols are those symbols which can be replaced by rules. They are rule references. In some notations non-terminals may refer to predefined rules ie. specific characters or character ranges.

Example Notation	Source
$name \to a$	(Chomsky, 1956) where a is defined by a rule
name <- name2 or [A-Z] or .	(Ford, 2004) – character classes; dot means any character
<name>::=<name2>	(Backus, 1957)
name=name2.	(Wirth, 1977)
name: *name2*	(ANSI, 1989) – in italics
name=name2, first-quote-symbol;	(ISO, 1997) – a non-terminal ending "-symbol" is pre-defined by the standard
[:upper:] or \u or [A-Z] or .	(Thomson, 1968)– regular expressions do not have non-terminals, however, some predefined rules and pseudo rules exist. Dot means any character (outside of square brackets).

Sequences

A sequence is a series of one or more expressions evaluated in order as if a boolean AND was between the items. Expressions include terminals, non-terminals, sequences, choices, groups, iterations, options and the empty sequence. BS-6154 and ISO-14977 refer to sequences as a single definition. Sequences are usually implicit and are concretely implemented in syntax rules, grouped sequences, repetitions and options.

- A syntax rule is a named sequence
- A grouped sequence is an unnamed sequence
- An option is sequence with zero or one occurrences
- A repetition is a sequence with zero or more occurrences
- An empty sequence has no terminals

Example Notation	Source
$name \rightarrow abc$	(Chomsky, 1956)
name <- a “b” c	(Ford, 2004) – space between items
<name>::=<a> b <c>	(Backus, 1957) – space between items
name=a “b” c.	(Wirth, 1977) – space between items
name: *a* ***b*** *c*	(ANSI, 1989) – space between items
name=a, “b”, c;	(ISO, 1997) – comma between items
abc	(Thomson, 1968)– list of items without separators

Group

A group is an unnamed sequence.

Example Notation	Source
None given	(Chomsky, 1956) – optional b
None given	(Ford, 2004) - question mark
None given	(Backus, 1957)
None given	(Wirth, 1977)
None given	(ANSI, 1989)
name=a, (“b”, c);	(ISO, 1997)
None given	(Thomson, 1968)

Option

An option is a sequence that can occur zero or one times.

Example Notation	Source
$name \rightarrow a\ b\ c$ $name \rightarrow a\ c$	(Chomsky, 1956) – optional b
name <- a “b”? c	(Ford, 2004) - question mark
<name>::= <a> “b” <c> | <a> <c>	(Backus, 1957)
name=a [“b”] c.	(Wirth, 1977)
name: *a* [***b***] *c*	(ANSI, 1989)
name=a, [“b”], c;	(ISO, 1997) – “(/” and “/)” may be used as alternative repeat start and end symbols
ab?c	(Thomson, 1968) – question mark

Repetition

A repetition is a sequence that can occur zero or more times. It is also sometimes referred to as iteration.

Example Notation	Source
$R \rightarrow \varepsilon$ $R \rightarrow a\ b\ c\ R$	(Chomsky, 1956) – epsilon indicates the empty string
rule <- a “b”* c	(Ford, 2004) – star symbol
<rule>::= | <a> “b” <c> <rule>	(Backus, 1957)
rule=a {“b”} c.	(Wirth, 1977)
rule: *a* {***b***} *c*	(ANSI, 1989) –
rule=a, {“b”}, c;	(ISO, 1997) - “(:” and “:)” may be used as alternative repeat start and end symbols
ab*c	(Thomson, 1968) – star symbol

A repetition that occurs 1 or more times can be expressed using the following notations.

Example Notation	Source
$R \rightarrow a\ b\ c\ R$ $R \rightarrow \varepsilon$	(Chomsky, 1956) – epsilon indicates the empty string
rule <- a “b” “b”* c	(Ford, 2004) – star symbol
<rule>::= <a> “b” <c> <rule> |	(Backus, 1957)
rule=a “b” {“b”} c.	(Wirth, 1977)
rule: *a* ***b*** {***b***} *c*	(ANSI, 1989)
rule=a, ”b”, {“b”}, c;	(ISO, 1997) – “(:” and “:)” may be used as alternative repeat start and end symbols
abb*c	(Thomson, 1968) – star symbol

A repetition that occurs n times can be expressed using the following notations.

Example Notation	Source
$R \rightarrow a\ a\ a\ a$	(Chomsky, 1956)
rule <- a a a a	(Ford, 2004)
<rule>::= <a> <a> <a> <a>	(Backus, 1957)
rule=a a a a.	(Wirth, 1977)
rule: a a a a	(ANSI, 1989)
rule=4*a;	(ISO, 1997) – “(:” and “:)” may be used as alternative repeat start and end symbols

Choice

A choice is a sequence of one or more items evaluated in order until one is found to be valid as if a boolean XOR was between the items. More than one item in the series may be valid, but only the first valid choice is used. They are sometimes known as alternates.

Example Notation	Source
$R \rightarrow a$ $R \rightarrow b$ $R \rightarrow c$	(Chomsky, 1956)
rule <- a / “b” / c	(Ford, 2004) – forward slash
<rule>::=<a> | b | <c>	(Backus, 1957)

rule=a | “b” | c.	(Wirth, 1977)
rule: *a* **|** ***b*** **|** *c*	(ANSI, 1989)
rule=a | “b” | c;	(ISO, 1997) – forward slash and exclamation mark are permitted symbols
[abc] or a | b | c	(Thomson, 1968)

Pipeline

A pipeline (Nelms, 2016) is a sequence of two or more expressions evaluated simultaneously, with the production from an expression in the chain sent to its successor in the pipeline sequence. Pipelines are common operating system features and are significant in many grammars. Notably pipelines are not a feature of the grammar-languages examined here.

The colon (:) symbol is suggested (Nelms, 2016) as a notation to separate expressions in a pipeline.

Example Notation	Source
rule = a : b : c;	(Nelms, 2013)

Empty Sequence

The empty sequence consists of an empty sequence of terminals. The empty sequence always evaluates to true.

Example Notation	Source
$name \rightarrow \varepsilon$	(Chomsky, 1956)
None given	(Ford, 2004) – forward slash
None given	(Backus, 1957)
None given	(Wirth, 1977)
None given	(ANSI, 1989)
name=a, ,c;	(ISO, 1997)
None given	(Thomson, 1968)

Predicate

Syntactic predicates (Ford, 2004) add characteristics to primaries that simplify grammars. The two predicates defined by Ford are ‘not’ and ‘and’. BS6154 includes a form of predicate known as an exception.

- The ‘not’ predicate continues evaluation of the sequence when the expression evaluates to false; no input is consumed and no output produced. It is equivalent to a BS6154 exception.
- The ‘and’ predicate continues evaluation of the sequence when the expression evaluates to true; no input is consumed and no output produced
- The ‘again’ predicate (Nelms, 2016) continues evaluation of the sequence when the expression evaluates to true; no input is consumed, but output is produced. It permits repeated evaluation of the same input. The && syntax is suggested as a notation to indicate again predicates in grammars

Not predicate (not b)	And predicate (and b)	Again predicate (again b)	Source
None given	None given	None given	(Chomsky, 1956)
name=!b a;	name=&b a;	None given	(Ford, 2004)
None given	None given	None given	(Backus, 1957)
None given	None given	None given	(Wirth, 1977)

None given	None given	None given	(ANSI, 1989)
name=a-b;	name=a-b;	None given	(ISO, 1997)
None given	None given	None given	(Thomson, 1968)
None given	None given	name=&&b, a;	(Nelms, 2016)

Entity and Entity Reference

Entity is a term used to describe re-usable text with an XML or HTML document. XML/HTML entity references use the &apos; notation, or &#0A; indicating a hexadecimal character value. In languages like C the back-slash character prefixes either a predefined entity such as newline (\n) or a character specified in hex (\x0A). In Unix shells ${name} is used to reference environment variables. We use the term entity to describe these types of reference.

Entities and entity references are not a part of the meta-languages described here, but we believe them to be a useful concept in grammars for consumption of input and production of output.

Entity references placed in terminals (Nelms, 2016) could use the syntax described for XML/HTML, C or shell to reference entities. A rule with a single terminal defines an entity.

Example Notation	Source

a = “a\n”;
a = “${newline}”;
<terminal>&apos;</terminal>
	(Nelms, 2013)

Summary

A table showing a comparison of the grammars described is shown in the appendix. The comparison in this section forms the basis for a common model for grammars defined in the next section.

A Common and Extensible Model for Grammars

A Comparison of Language Grammars (Nelms, 2013) described the common features of grammars expressed in a variety of notations. This section describes a common model and semantics for grammars with the following features:

- Grammar
- Rule
- Rule reference (non-terminals)
- Entity
- Entity reference
- Terminal string (parsing and production)
- Sequence
- Option
- Repetition
- Group
- Choice
- Pipeline
- Predicates
- Empty sequence
- Comment

Characteristics of the Common Grammar Model

In addition to the concepts introduced in a comparison of language grammars the common grammar model also introduces the following:

- Attributes – a means of specifying and extending grammar characteristics
- Identity – an attribute used for rules, but now extended to supported arbitrary addressing of other parts of the grammar
- Echo – attribute used to redirect parser input to parser output
- Predicates – a new predicate 'again' is added to the 'and' and 'not predicates
- Encoding –for terminals
- Input/Output – a unified model for expressing the input and output of grammars and primaries that includes abstract syntax trees (ASTs) and linear storage such as files
- Parent-child relationships – the unified model defines the parent-child relationship that can exist between rules, sequences, groups, options, iterations and other parts of the model

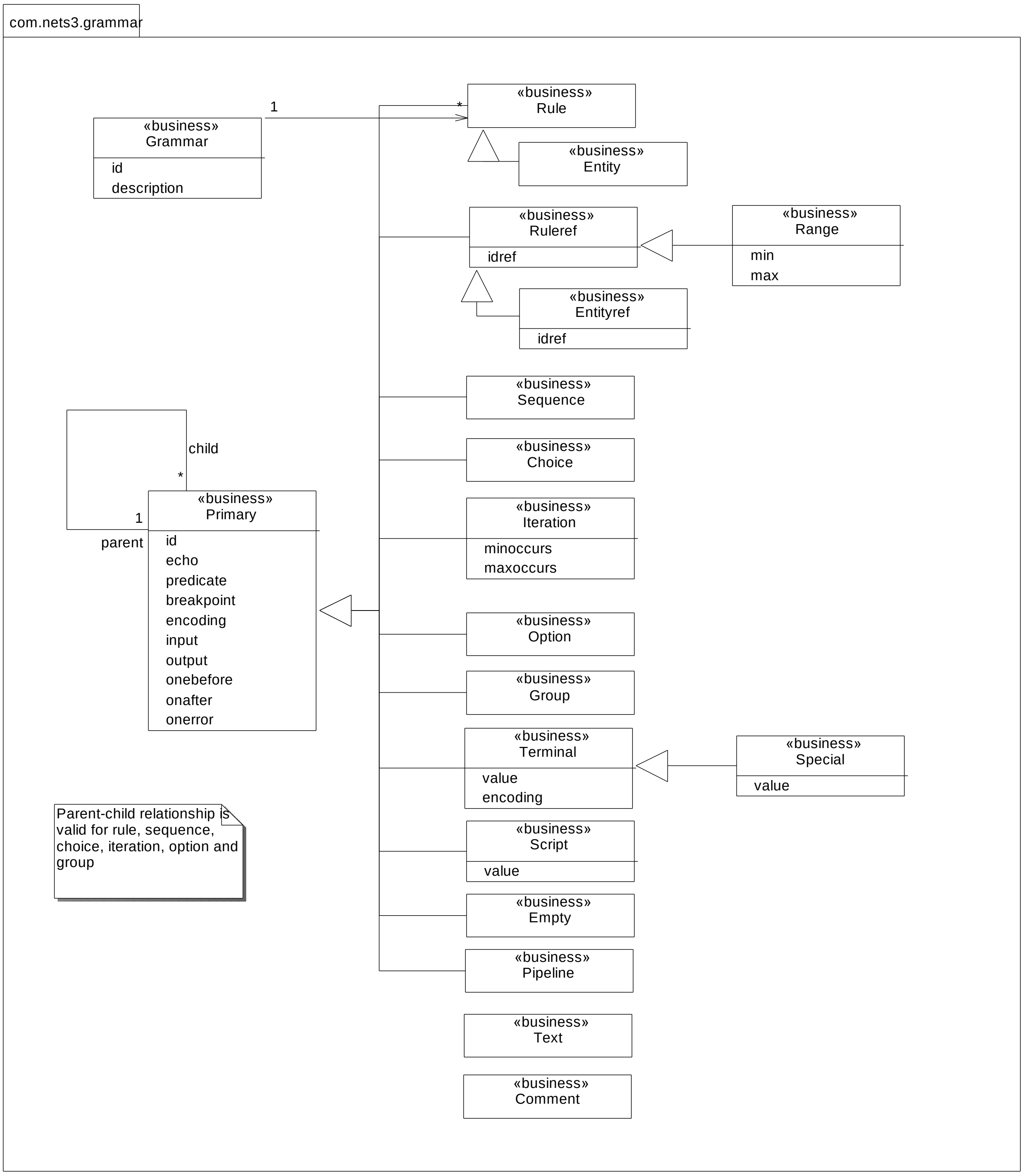

Figure 1: A Common Model for Language Grammars

An EBNF Representation of The Common Model of Language Grammars

The following table describes a notation for grammars using EBNF.

Concept	Description	Grammar XML Notation

Grammar	A collection rules identified by a unique name and with an optional description.	`File name`
		`eg. myfile.g`

Primary	An abstract concept that includes rulerefs, entityrefs, terminals, sequences, choices, options, repetitions, groups and empty sequences.	`None`

Attributes	A primary may include attributes to define common characteristics including identity, echo, predicate, breakpoint, encoding, input, output and actions. These characteristics may be used with any primary including Empty. These should be defined after the primary with any custom attributes added at the end of the attribute list.	`rule = ruleref breakpoint=”true”;`
		`rule = “Hello” breakpoint=”true” custom=”here”;`

Rule	The definition of a parsing or production rule identified by a unique name	`rule = … ;`
Rule Reference	A reference to a parsing or production rule or entity	`rule = rulearef;`
Entity	The definition of an entity with a single terminal.	`Entity = “string”;`

Entity Reference – C	A reference to an entity from a terminal in the C style	`rule = “\n”; (*newline*)`
		`rule = ?\x010?; (*hex 10*)`

Entity Reference - Shell	A reference to an entity from a terminal in the shell style	`rule = “${input}”; (*input reference*)`
		`rule = ?${entityref}?;`

Terminal String	A literal string used for parsing or production whose content is a C99 string	`None`
Parse String	Parses and compares the input to string	`rule = “literal”;`
Production String	Produces the string as output	`rule = ?output literal?;`

Encoding	Defines either multi-byte or wide character strings, using the C style L predicate to strings; any entity references use the declared encoding	`rule = L”literal”; (*Wide literal*)`
		`rule = ”literal”; (*MBCS literal*)`
		`rule = L?literal? (*Wide literal*)`

Sequence	A sequence of primaries evaluated in order.	`rule = rulea, ruleb, rulec;`
Group	A sequence of primaries evaluated in order. Provided for traceability with traditional notations.	`rule = (rulea, ruleb);`
Option	An optional sequence of primaries evaluated in order	`rule = [rulea, ruleb];`
Repetition	A sequence of primaries evaluated in order zero or more times. Frequency can be controlled with minoccurs and maxoccurs as follows	`rule = {rulea, “literal”};`
Fixed Repetition	A sequence of primaries evaluated n times.	`rule = 4*rulea;`
Range Repetition	A sequence of primaries evaluated at least n times and no more than m times.	`rule = {ruleref} minoccurs=”n” maxoccurs=”m”;`
Minimum Repetition	A sequence of primaries evaluated at minimum n times.	`rule = {ruleref} minoccurs=”n”;`
Maximum Repetition	A sequence of primaries evaluated a maximum n times.	`rule = {ruleref} maxoccurs=”m”;`
Choice	A sequence of primaries evaluated in order of which the first valid primary is chosen	`rule = rulea|ruleb|rulec;`
Pipeline	A sequence of two or more expressions evaluated simultaneously, with the production from one primary in the chain sent to its successor in the pipeline sequence.	`rule = asciitoUTF-8 : bas64encode;`

Exception	An exception follows a primary using a "-" symbol. If the exception evaluates to false, reset the parser state and continue.	`rule = rulea - ruleb;`
		`Equivalent to:`
		`Rule = rulea, !ruleb;`

Predicate	Either a not or and condition linked to any primary.	
And Predicate	If the primary evaluates to true, reset the parser state and continue	`rule = &rulea, rulea;`
Not Predicate	If the primary evaluates to false, reset the parser sate and continue	`rule = !rulea, ruleb;`
Again Predicate	If the primary evaluates to false, reset the parser sate and continue; keep the output	`rule = &&rulea, ruleb;`
Empty	The empty sequence.	`Rule = rulea, ;`
Comment	Comment to the grammar.	`(* Comment here *)`

An XML Representation of The Common Model of Language Grammars

The following table describes a notation for grammars using XML.

Concept	Description	Grammar XML Notation
Grammar	A collection rules identified by a unique name and with an optional description.	`<grammar id="name" description="My Grammar"> … </grammar>`
Primary	An abstract concept that includes rulerefs, entityrefs, terminals, sequences, choices, options, repetitions, groups and empty sequences.	`None`

Attributes	A primary may include attributes to define common characteristics including identity, echo, predicate, breakpoint, encoding, input, output and actions. These characteristics may be used with any primary including Empty. These should be defined after the element name with any custom attributes added at the end of the attribute list.	`<rule>`
		`<ruleref id="name" breakpoint="true"> … </ruleref>`
		`</rule>`

Identity	Identity is required for Grammars and Rules using the id attribute. Optionally Identity may be applied to any primary, which allows referencing from Rule References.	`<iteration id="it1"> … </iteration>`
Rule	The definition of a parsing or production rule identified by a unique name	`<rule id="name"> … </rule>`
Rule Reference	A reference to a parsing or production rule	`<ruleref idref="name">`
Terminal String	A literal string used for parsing or production whose content is text and entity references	`<terminal>` *`literal`* `</terminal>`
Parse String	Parses and compares the input to string	`<terminal>` *`literal`* `</terminal>`
Production String	Produces the string as output	`<special>` *`literal`* `</special>`

Encoding	Defines either multi-byte or wide character strings using the encoding attribute; any entity references use the declared encoding	`<terminal encoding="wchar">` *`literal`* `</terminal>`
		`<special encoding="char">` *`literal`* `</special>`

Entity Reference - Shell	A reference to an entity from a terminal in the shell style	`<terminal>${input}</terminal>`
		`<special>${entityref}</special>`

Entity Reference - XML	A reference to an entity from a terminal in the XML style	`<terminal>&entityref;</terminal>`
		`<special>&#10;</special>`

Sequence	A sequence of primaries evaluated in order.	`<sequence> … </sequence>`
Group	A sequence of primaries evaluated in order. Provided for traceability with traditional notations.	`<group> … </group>`

Option	An optional sequence of primaries evaluated in order.	`<option> … </option>`
Repetition	A sequence of primaries evaluated in order zero or more times. Frequency can be controlled with minoccurs and maxoccurs as follows	`<iteration> … </iteration>`
Fixed Repetition	A sequence of primaries evaluated n times.	`<iteration minoccurs=”n” maxoccurs=”n”> … </iteration>`
Range Repetition	A sequence of primaries evaluated at least n times and no more than m times.	`<iteration minoccurs=”n” maxoccurs=”m”> … </iteration>`
Minimum Repetition	A sequence of primaries evaluated at minimum n times.	`<iteration minoccurs=”n” > … </iteration>`
Maximum Repetition	A sequence of primaries evaluated a maximum n times.	`<iteration maxoccurs=”n”> … </iteration>`
Choice	A sequence of primaries evaluated in order of which the first valid primary is chosen.	`<choice> … </choice>`

Pipeline	A sequence of two or more expressions evaluated simultaneously, with the production from one primary in the chain sent to its successor in the pipeline sequence.	`<pipeline>`
		`<ruleref idref= asciitoUTF-8”/>`
		`<ruleref idref=”bas64encode”/>`
		`</pipeline>`

Predicate	Either a ‘not’, ‘and’ or ‘again’ condition linked to any primary.	None
And Predicate	If the primary evaluates to true, reset the parser state and continue	`<ruleref predicate=”and”/>`
Not Predicate	If the primary evaluates to false, reset the parser sate and continue	`<ruleref predicate=”not”/>`
Again Predicate	If the primary evaluates to false, reset the parser sate and continue; keep the output	`<ruleref predicate=”again”/>`
Comment	Comment to the grammar.	`<!-- Comment here -->`

Summary

A common and extensible model for grammars allows grammars to be more readily defined and reused. This section defines an XML grammar language that offers a grammar model that can be extended to support the practical needs of parsers. In addition the model has been designed as a superset of grammars from (Chomsky, 1956), (Backus, 1957), (Wirth, 1977), (Ford, 2004), (Thomson, 1968), (BSI, 1981) and (ANSI, 1989) so they can be mapped to this common model and back again if required.

Practical Considerations for Parsers

This final section looks at practical considerations for grammars necessary for building parsers and generators.

Parsers and Grammars

Parsers are the implementation of grammars. They can be either interpretive or compiled. Interpretive parsers such as those used for regular expressions read the grammar and use it to parser some input and usually generate an output. Compiled parsers are written in a language such as C or Java and are often generated from grammars. They also read some input and generate an output, but only for one grammar. GREP is an example of an interpreter for grammars and LEX/YACC is an example of a parser generator for grammars.

Grammars are rarely portable between interpretive and compiled parsers.

Input and output

Parsers act upon some input and generate an output. This may be as simple as a one-to-one mapping (an echo) between source and destination or it may involve a complex domain specific mapping. It is not unusual for the mapping between the source domain model and the target domain model to be a multi-stage process resulting in output from the resulting domain model.

Parsers and interpreters therefore need to read and write from common sources like files, pipes and memory but also complex forms like the elements & attributes in an AST. A common model for addressing these types of input and output remains elusive and defining the input and output of a parser is usually implementation specific concern. A model for defining input and output that addresses file, pipe, memory and AST’s would help make parsers grammars more useful and potentially more readily transferable to new platforms.

Abstract Syntax Trees

In computer science an abstract syntax tree represents the structure of the input ie. if the source represents contact data the AST may have nodes corresponding to name, address and telephone number. In many respects ASTs have much in common with Document Object Models found in web browsers having nodes with a hierarchy of parents and children.

The relationship between the document object model and the grammar is usually implementation specific and often requires coding.

Given the practical use of grammars to build AST’s it is surprisingly difficult to trace the relationship between a grammar and its corresponding AST. A declarative approach to linking ASTs to grammars is important to making complex multi-stage parsers easy to create.

Encoding

The encoding of the input and output require grammars to identify the encoding of parsing and production strings. Alternatively all input and output must be converted into some uniform intermediate encoding. Defining the internal representation of parsing and production strings is essential to handle legacy multi-byte encoded data and more modern Unicode encoded data.

Character Classification

Practical consideration must be given to character classification, for example the common need to express alphabetic characters, upper case and lower and numbers as ranges rather than a choice. Regular expressions use a number of notations to express this [:upper:] or \u or [A-Z] which are either a reference to a pseudo rule or a range descriptor. Both are requirements for practical purposes.

Actions

A parsing or generating process will need to perform actions (in addition to populating the AST) during processing. Formal language grammars do not define actions, but many tools provide extensions that allow actions to be defined, however, there is usually no declarative way of linking grammars with their actions without mixing code fragments with grammar.

Making a clear separation between the grammar and the action code is necessary if grammars are to be reused for parsers built using different languages. But having a clear relationship between grammar and the action code through declaration is also desirable.

A common approach for declaring actions in grammars is needed, whilst delivering code optimized for the target environment.

Debugging

A further desirable feature for parsers is to provide debugging and error reporting. The most common needs are to be able to define places in the grammar where such actions are to occur and error message required.

A Grammar XML Representation of Practial Parser Features

The following table describes a notation for grammars using Grammar XML. The same attribute tags may be used with NETS3 EBNF extensions.

Concept	Description	Grammar XML Notation

Ignore Case	Ignore the case of a string when comparing the input of a primary.	`<terminal ignorecase=”true”/>`
		`<group ignorecase=”true”> … </group>`

Input and Output	Input or output for a primary may be file, pipe, memory, element, attribute or property.	`<ruleref input=”test.in”>`

Echo	Indicates whether a primary is a parsing rule, a production rule or both. When “false” or “in” indicates input only – when “true” or “inout” indicates echoing input to output, when “out” indicates only output.	`<ruleref echo=”false”/> or <ruleref echo=”in”/>`
		`<ruleref echo=”true”/> or <ruleref echo=”inout”/>`
		`<ruleref echo=”inout”/>`

File Input/Output	Input or output of a primary to a file	`<ruleref input=”test.in”> or`
		`<ruleref input=”file:test.in”>`

Pipe Input/Output	Input or output of a primary to a pipe – may be either stdin, stdout or stderr or a filename.	`<ruleref input=”pipe:stdin”> or`
		`<ruleref input=”pipe:test.in”>`

Memory Input/Output	Input or output of a primary to memory – the string following the “mem:” specifier is used.	`<ruleref input=”mem:some text here”>`
		`<ruleref input=”m:some text here”>`

Abstract Syntax Trees	ASTs use elements and attributes to define the tree structure. In addition properties are global resources.	`None`

Element Input/Output	Input or output of a primary to an element in an AST. By default the parent element is assumed to be the last element.	`< ruleref input=”element:name”>`
		`< ruleref input=”e:name”>`

Attribute Input/Output	Input or output of a primary to an attribute of an element in an AST. By default the parent element is assumed to be the last element.	`< ruleref input=”attribute:name”>`
		`< ruleref input=”a:name”>`

Text node Input/Output	Input or output of a primary to a text node in the AST. Text nodes are named #text	`< ruleref output=”textnode:”>`
		`< ruleref input=”textnode:”>`

Entity Reference Input/Output	Input or output of a primary to an entity reference in the AST.	`< ruleref output=”entityref:”>`

Property Input/Output	Input or output of a primary to a grammar entity..	`< ruleref input=”property:name”>`
		`< ruleref input=”p:name”>`

Character Classification	Character classification comes in two forms. Pseudo rules define common character classifications and can be referenced using Rule References. Classifications are implementation defined but at a minimum should include alnum, alpha, ascii, char, cntrl, digit, graph, lower, print, punct, space, upper and xdigit. The “char” pseudo rule indicates any character.	`<ruleref idref=”alpha”/>` `<ruleref idref=”digit”/>`
Character Ranges	A ruleref may optionally define a character range using the max and min attributes. These are only used when the “range” pseudo rule is referenced.	`<ruleref idref=”range” min=”48” max=”57”/>` Equivalent to `<ruleref idref=”digit”/>`
Actions	The actions associated with a primary including before execution, after successful execution and after erroneous execution.	`<ruleref onbefore=”execute_before” onafter=”execute_after” onerror=”execute_error_handler”/>`
Script	To define Scripts or code	`<script>My Script</script>`
Debugging and Breakpoints	An action to pause the running of a parser when this part of the grammar is reached.	`<ruleref breakpoint=”true”/>`

Grammar Libraries

Grammar libraries define reusable rules and entities provided by inbuilt or third party libraries. Libraries provide the opportunity to not only reuse declarative grammars but also introduce algorithmic processes and procedures into grammars that are opaque in their operation.

For example the libraries in the core NETS3 parser include nets-ctype for character classification and nets-XML for entity definitions.

C-Type Library

The nets-ctype grammar library defines common character classifications and entity references. This library also supports custom character ranges using a minimum to a maximum value. Variants are provided for multi byte and wide character strings.

Rules

Concept/Rule	Description	Usage
alnum	A union of the alpha and digit classifications	`<ruleref idref=”alnum”/>`
alpha	Any alphabetic character in the current locale	`<ruleref idref=”alpha”/>`
ascii	Any asci 7 bit character in the current locale	`<ruleref idref=”ascii”/>`
char	Any multibyte character in the current locale	`<ruleref idref=”char”/>`
cntrl	Any control character in the current locale	`<ruleref idref=”cntrl”/>`
digit	Any decimal digit in the current locale	`<ruleref idref=”digit”/>`
lower	Any lower case letter in the current locale	`<ruleref idref=”lower”/>`
print	Any printable character in the current locale	`<ruleref idref=”print”/>`
punct	Any punctuation character in the current locale	`<ruleref idref=”punct”/>`
space	Any space character in the current locale	`<ruleref idref=”space”/>`
upper	Any upper case character in the current locale	`<ruleref idref=”upper”/>`
xdigit	And hexadecimal digit in the current locale	`<ruleref idref=”xdigit”/>`
range	And character with the min and max range defined in the current locale	`<ruleref idref=”range” min=”10” max=”20”/>`
min attribute	Minimum character value	
max attribute	Maximum character value	
walnum	A union of the walpha and wdigit classifications	`<ruleref idref=”walpha”/>`
walpha	Any alphabetic wide character in the current locale	`<ruleref idref=”walpha”/>`
wascii	Any asci 7 bit wide character in the current locale	`<ruleref idref=”wascii”/>`

wchar	Any multibyte wide character in the current locale	`<ruleref idref=”wchar”/>`
wcntrl	Any control wide character in the current locale	`<ruleref idref=”wcntrl”/>`
wdigit	Any decimal wide digit in the current locale	`<ruleref idref=”wdigit”/>`
wlower	Any lower case wide character in the current locale	`<ruleref idref=”wlower”/>`
wprint	Any printable wide character in the current locale	`<ruleref idref=”wprint”/>`
wpunct	Any punctuation wide character in the current locale	`<ruleref idref=”wpunct”/>`
wspace	Any space wide character in the current locale	`<ruleref idref=”wspace”/>`
wupper	Any upper case wide character in the current locale	`<ruleref idref=”wupper”/>`
wxdigit	Any hexadecimal wide digit in the current locale	`<ruleref idref=”wxdigit”/>`
wrange	Any wide character with the min and max range define in the current locale	`<ruleref idref=”wrange” min=”10” max=”20”/>`

Entities

Concept/Entity	Description	Usage
\a	x07 Beep	`<terminal>&\n;</terminal>`
\b	x08 Backspace	`<terminal>&\b;</terminal>`
\t	x09 Horizontal Tab	`<terminal>&\t;</terminal>`
\n	x0A Newline	`<terminal>&\n;</terminal>`
\v	x0B Vertical Tab	`<terminal>&\v;</terminal>`
\f	x0C Form Feed	`<terminal>&\f;</terminal>`
\r	x0D Carriage Return	`<terminal>&\r;</terminal>`
\”	x22 Double Quote	`<terminal>&\”;</terminal>`
\’	x27 Single Quote	`<terminal>&\’;</terminal>`
\?	x3F Question Mark	`<terminal>&\?;</terminal>`
\\	x5C Backslash	`<terminal>&\\;</terminal>`
\xnn	Any character code nn defined in hexadecimal	`“\x64”`

XML Library

The nets-XML grammar library defines common entity references for grammars defined using XML grammar notation.

Concept/Entity	Description	Usage
lt	Less than	`<terminal>&lt;</terminal>`
gt	Greater than	`<terminal>&gt;</terminal>`
amp	Ampersand	`<terminal>&amp;</terminal>`
apos	Single quote	`<terminal>&apos;</terminal>`
quot	Double quote	`<terminal>&quot;</terminal>`
#nnn	Any character code nnn defined in decimal	`<terminal>&#0064;</terminal>`

Iconv Library

A practical consideration for parsers is how to interpret character streams and ensure that they support a range of uses that include not just the traditional 7 or 8 bit character sets like ASCII, Latin-1 etc... The Unicode standard offers a model for a wide range of characters and its use in grammars ensures that the range of characters and their classification supported by a parser is known and consistent.

A practical consideration for parsers is that legacy systems use a mix of single byte, multi-byte and variable byte encodings. Parsers therefore need to recognize and convert between these encodings consistently for input and output. Not only this but some containers may use one or more encodings in a single stream of input.

Iconv is a utility for converting text between character sets and is defined as part of the Posix standard. The iconv grammar library supports conversion of strings using any combination of source and target character sets using the input_encoding and output_encoding attributes.

Concept	Description	Usage
Iconv rule	iconv	`<rule ruleref=”iconv” input_encoding=”ASCII” output_encoding=”WCHAR_T”>`
Input encoding attribute	The source encoding for the iconv conversion	
Output encoding attribute	The target encoding for the iconv conversion	

Note: Future revisions of this paper will add more declarative notations for practical parser needs.

Summary.

This section reviewed practical considerations for parsers and how declaratively linking grammars with their domain model or AST provides better traceability between the two and has the potential to simplify the process of defining grammars. Furthermore combining AST’s with the input output model simplifies and unifies a key practical consideration for parsers. Finally declaratively addressing encodings, character classification, actions, breakpoints and echo behaviour also has clear benefits for writers of parsers.

Conclusion

A common model for grammars allows grammars to be more readily defined and reused. A Comparison of Language Grammars (Nelms T. , 2013) defines the common characteristics of language grammars through comparison of their notations. “A Common and Extensible Model for Language Grammars” (Nelms T. , 2013) defines a common model through which all common notations could be expressed. “Practical Considerations for Parsers” examines how a common model could be extended to suit the practical needs of parsers and extension through grammar libraries.

References

ANSI. (1989). ANSI X3.159-1989. New York: American National Standard Institute.

Backus, J. W. (1957). The Syntax and Semantic of the Proposed International Algebraic Language. Zurich: Proceedings of the International Conference on Information Processing, UNESCO.

BSI. (1981). *BS-6154:1981 Method of Defining - Syntactic Metalanguage.* London: BSI (British Standards Institution).

Chomsky, N. (1956). *Three Models for the Description of Language.* Cambridge: Massachusetts Institute of Technology.

Ford, B. (2004). Parsing Expression Grammars: A Recognition Based Syntactic Foundation. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM doi:10.1145/964001.964011. ISBN 1-58113-729-X. .

ISO. (1997). ISO 14977 - A Standard Syntactic Metalanguage. www.iso.org.

Jinks, P. (2005). *Peter Jinks BNF/EBNF Variants*. Retrieved from Manchester University: http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html

Kleene, S. C. (1956). Representation of Events in Nerve Nets and Finite Automata. Princeton University Press.

Thomson, K. (1968). Regular Expression Search Algorithm. *Connmunications of the ACM Volume 11 / Number 6*, 419-422.

Wirth, N. (1977). What Can We Do about the Unnecessary Diversity of Notations for Syntax Definitions? *Communications of the ACM* (pp. Pages 822-823). New York: ACM.

Version History

Version 0.3 first published 16th July 2013

Version 0.4 published 3rd October 2013.

Version 0.5 published August 2016

Version 0.6 published September 2016

Contacts

For more information please contact:

Tim Nelms, timnelms@nets3.com, +44 7968 489105

Contact Us

To learn more about nets3 visit us at www.nets3.com

Meta-language	Grammar (name)	Rule (name)	Terminal (t) consumption	Terminal (t) production	Non-Terminal (name2)	Sequence	Option	Group	Choice	Pipeline
Chomsky, 1956	None given	$\mathit{name} \rightarrow$	None given	$\mathit{name} \rightarrow t$	$\mathit{name} \rightarrow a$	$\mathit{name} \rightarrow abc$	$\mathit{name} \rightarrow a\ b\ c$ $\mathit{name} \rightarrow a\ c$	None given	$\mathit{name} \rightarrow a$ $\mathit{name} \rightarrow b$ $\mathit{name} \rightarrow c$	None given
Ford, 2004	None given	$\mathit{name} \leftarrow$	name <- t	None given	name <- name2 or [A-Z] or .	name <- a “b” c	name <- a “b”? c	None given	name <- a / “b” / c	None given
Backus, 1957	None given	<name>::=	<name>::=t	None given	<name>::=<name2>	<name>::=<a> b <c>	<name>::= <a> “b” <c> | <a> <c>	None given	<name>::=<a> | b | <c>	None given
Wirth, 1977	None given	name= .	name=“t”.	None given	name=name2.	name=a “b” c.	name=a [“b”] c.	None given	name=a | “b” | c.	None given
ANSI, 1997	None given	name:	name: **t**	None given	name: *name2*	name: *a* ***b*** *c*	name: *a* [***b***] *c*	None given	name: *a* **|** ***b*** **|** *c*	None given
ISO, 1997	None given	name= ;	name=‘t’;	None given	name=name2, first-quote-symbol;	name=a, “b”, c;	name=a, [“b”], c;	name=a, (“b”, c);	name=a | “b” | c;	
Thomson, 1968	None given	None given	t	None given		abc	ab?c	None given	[abc] or a | b | c	None given
Nelms, 2013	<grammar id=”name”>… </grammar>	<rule id=”name”> </rule>	<terminal>t </terminal>	<special>t </special>	<rulref idref=”name2”>… </ruleref>	<sequence>… </sequence>	<option>… </option>	<group>… </group>	<choice>… </choice>	<pipeline>… </pipeline>

Meta-language	Repetition	Repetition 1+	Repetition n=4	Not predicate (not b)	And predicate (and b)	Again predicate (again b)	Entity (name)	Entity reference (name)	Attributes (name,value)
Chomsky, 1956	$\mathit{name} \rightarrow \varepsilon$ $\mathit{name} \rightarrow a\ b\ c\ R$	$\mathit{name} \rightarrow a\ b\ c\ R$ $\mathit{name} \rightarrow \varepsilon$	$\mathit{name} \rightarrow a\ a\ a\ a$	None given	None given	None given	None given	None given	None given
Ford, 2004	name <- a “b”* c	name <- a “b” “b”* c	name <- a a a a	name=!b a;	name=&b a;	None given	None given	None given	None given
Backus, 1957	<name>::= | <a> “b” <c> <name>	<name>::= <a> “b” <c> <name> |	<name>::= <a> <a> <a> <a>	None given	None given	None given	None given	None given	None given
Wirth, 1977	name=a {“b”} c.	name=a “b” {“b”} c.	name=a a a a.	None given	None given	None given	None given	None given	None given
ANSI, 1997	name: *a* {***b***} *c*	name: *a* ***b*** {***b***} *c*	name: a a a a	None given	None given	None given	None given	None given	None given
ISO, 1997	name=a, {“b”}, c;	name=a, ”b”, {“b”}, c;	name=4*a;	name=a-b;	name=a-b;	None given	None given	None given	None given
Thomson, 1968	ab*c	abb*c	Non given	None given	None given	None given	None given	None given	None given
Nelms, 2013	<iteration>… </iteration>	<iteration minoccurs=”1”> </iteration>	<iteration minoccurs=”4” maxoccurs=”4”> </iteration>	<primary predicate=”not”>… </primary>	<primary predicate=”and”>… </primary>	<primary predicate=”again”>… </primary>	<entity id=”name”>… <terminal>t</terminal> </entity>	<terminal>&name; </terminal>	<primary name=”value”> </primary>

Meta-language	Empty sequence	Character classes	Echo	Case insensitivity	Breakpoint	Encoding (terminal)			
Chomsky, 1956	$\mathit{name} \rightarrow \varepsilon$	None given	None given	None given	None given	None given			
Ford, 2004	None given	None given	None given	None given	None given	None given			
Backus, 1957	None given	None given	None given	None given	None given	None given			
Wirth, 1977	None given	None given	None given	None given	None given	None given			
ANSI, 1997	None given	None given	None given	None given	None given	None given			
ISO, 1997	name=a, ,c;	None given	None given	None given	None given	None given			
Thomson, 1968	None given	[:upper:] or \u or [A-Z] or .	None given	None given	None given	None given			
Nelms, 2013	<empty/>	<rulref idref=”alpha”/> <rulref idref=”range” min=”10” max=”61”/>	<rulref idref=”rule” echo=””/>	<rulref idref=”rule” ignorecase=””/>	<rulref idref=”rule” breakpoint=””/>	<terminal encoding =”wchar”>xyz</terminal>			