

SYNTEX

Compiler's Compiler

release 2.0(b) - ShareWare version

Documentation and Tutorial File

This program is by no means 'public domain'. Furthermore, it does represent a great deal of sweat and work, whatever you may think about it. I don't ask for any license in counterpart, but I would greatly appreciate any feedback from you - the User -, especially if you decide to add SYNTEX to your Turbo Pascal development tool box.

So feel free to send me anything you may think of, that you think is appropriate - phone call, post card, money, unused computer material, software,...). Those of you that are at least occasional programers will sure understand what I mean.

I will then be glad to send you the source code as well as a fully functionnal version.

John

To the Newcomers	2
Overview	3
Example	4
BNF Syntax - GRM Files	5
User Interface	7
menu	7
File menu	7
Edit menu	7
Generate menu	7
Debug menu	7
Options menu	8
Windows menu	8
How to write a LL(1) compliant grammar	9
Quick overview of a compiler's structure	9
Descendant recursive syntax analysis with one prevision symbol	10
LL(1) Grammars	11
First Try	12
Changes	12
Misceleanous considerations	13
Version History	15
How to contact the author	16
§	

To the Newcomers

This file is both SYNTEX's documentation file and a tutorial file to compilation techniques.

Chapter 2 - Example - is probably the best introduction to SYNTEX for anyone who is puzzled by the 'Compiler's Compiler' subtitle - it shows on a quick example what SYNTEX really DOES.

Chapter 5 goes through the basic vocabular of compilation and compiler's technique, introducing concepts like Descendant compilation, syntax tree, production, terminal, and LL(1) grammar.

Those two chapters should be enough to get you started using SYNTEX in your own projects. For those of you that might be interested in gaining deeper insight on compilation and compilers, I strongly recommend reading Aho, Sethi, and Ullman's book on the subject.

Chapter 1 - Overview - is not designed to be understood by a novice, so if you feel frustrated, I suggest you read chapters 2 and 5 first.

Chapters 3 and 4 contain a reference of the BNF syntax and all menu options available during execution time.

Overview

SYNTEX is a syntax analyser's generator based on a series of articles by Alix Alix, published in issues 29, 31 and 35 of the French Pascal programming magazine 'Pascalissime'.

Given a text file containing a BNF description of the targeted grammar, SYNTEX automatically generates a pascal source for the syntax analysis part of the targeted compiler.

Described grammars must be LL(1) compliant, in order for SYNTEX to be able to generate a descendant recursive syntax analyser. The Wirth Pascal grammar is a classic example of such grammars.

If so, SYNTEX generates the source code.

**** Caution**** in the current version,

SYNTEX does ***NOT*** perform any checking on the grammar description. That is it does not make sure that the targeted grammar is non-ambiguous, non-contextual, and so on. A fortiori, SYNTEX does ***NOT*** perform any optimisation of the grammar description.

Only the body of the lexical analyser is generated. You will have to flesh it out later in order to get a functioning compiler.

Only the body of the error handling procedure is generated. Here again, it is up to you to fill it in. A simple WriteLn statement is suggested as a quick make up.

Semantic analysis instructions are to be inserted manually in the syntax analysis procedures.

Example

Start SYNTEX and File|Open the Test.grm file.

Select Generate|Generate.

Press any key when prompted - the generation phase has ended.

Look at the source code produced by File|Open Test.pas

BNF Syntax - GRM Files

Every grammar description file is tagged with a .GRM extension. The syntax used for writing descriptions is derived from the Backus-Naur Form (BNF) where

<u>The Sign</u>	<u>Stands for</u>
=	Symbol - symbol definition separator.
	Alternatives indicator.
[]	Zero or one appearance indicator.
{ }	Zero or many appearances indicator.
.	End of definition mark.

NB.: {} and [] symbols have priority upon the | symbol.

The two following rules are here to help SYNTEX know whether a symbol is terminal:

1. 'Punctuation' symbols, ie. every symbol containing non-alphabetic or non-numeric signs must be listed in the beginning of the GRM file, with the symbol being quoted. An alphanumeric equivalent is then given.

by ex. '=' = Egal.
'<>' = Different.

2. 'Non-punctuation' terminals must begin with a capitalized letter. As a consequence, no non-terminal symbol is allowed to contain capitalized letters.

Besides the first rule concerning punctuation symbols, production rules can be entered in any order that seems more convenient to you.

A single production can extend over 80 characters, but this is the limit for any symbol name.

You can add commentaries in GRM files, providing you type them between @ marks.

Example

\\Example\\

Refer also to the example GRM files included in this package.

The previously stated GRM syntax rules can be seen as a grammar for GRM files, and thus be written using BNF notation. One might get for instance:

Remark1:

It is a real pleasure to watch the computer automatically generate hundreds or thousands of lines of code that would have been previously typed by hand.

Remark2:

remark1 is especially important to me since it took me quite a long time to write WIZZ's interpreter manually. - WIZZ is another personal product, designed to draw and manipulate mathematical functions, with a special attention to their definition domain. WIZZ also includes function libraries facilities and a complete scientific calculator.

User Interface

The interface has greatly improved since the first versions - cf; Version History -, thanks to Turbo Vision.

It is then supposedly user friendly and "intuitive", as one says. Anyway, here is a quick review of every menu option available:

= menu

Program identification box and program registration box. The later beeing removed in the registered version.

File menu

The Editor object used in SYNTEX is TV2's editor - as is. It is thus a simple but quick editor, allowing extensive usage of the mouse pointer. It has be defaulted to auto-indenting for better grammar description texts lisibility.

The only unusual feature is the automatic closing of debugging windows related to a .grm window when it is closed.

Edit menu

Classic text edition function, using TV's clipboard.

Generate menu

Generate generates (yes) TP source code corresponding to the current grammar description, providing it is invoqued from within the window containing the target grammar - otherwise, an error message is displayed.

Options allows the user to generate Unit- or Program-formatted source code.

Make compiles the produced source code. In the current version, this option is not activated. This is not part of a crippled version scheme, but simply an open question to you - the User -: do you wish you could compile your code from within SYNTEX?

Debug menu

NB. In order for the following menu options to be actives, a grammar file needs to be successfully compiled first.

Punctuation opens a window containing the list of all current grammar's declared punctuation symbols.

Key Words does the same with declared terminal symbols.

Syntax Tree opens a window displaying the syntax tree of the current grammar. productions can be developped or reduced, as in classic directory browsing boxes.

Options menu

Video toggles from the 25 lines - default setting - to the 43 (EGA) or 50 (VGA) lines setting.

Save Desktop saves desktop configuration (settings, open windows, etc...). This option is not implemented on non-registered versions. Please note that this is the only functional limitation of the non-registered version, together with the following option.

Load Desktop restores a previously Save Desktop-ed configuration.

Windows menu

General and classic options for windows management - opening, closing, tiling, etc... -.

How to write a LL(1) compliant grammar

Quick overview of a compiler's structure

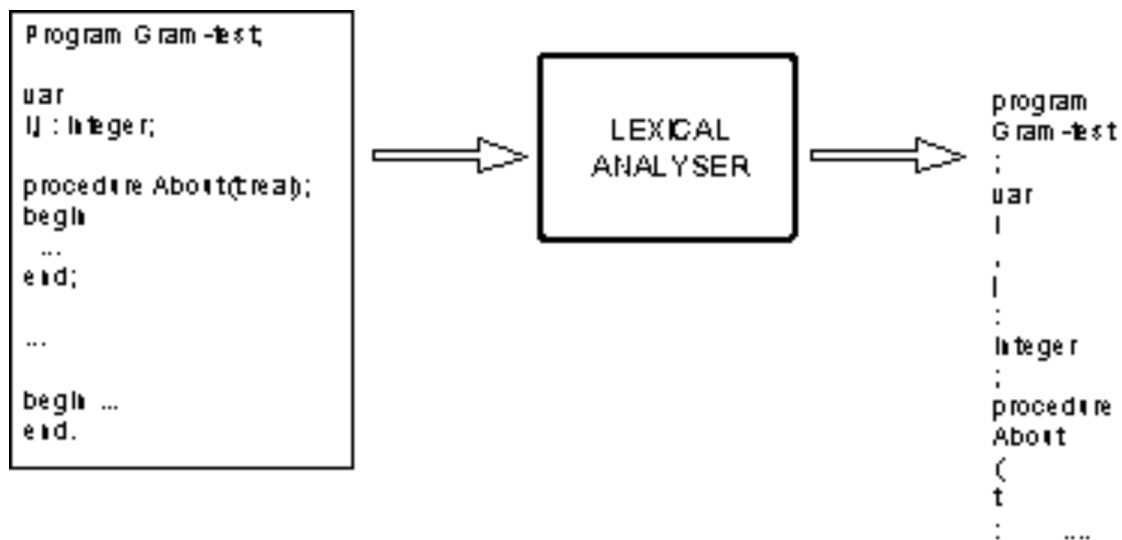
A compiler is a translation program - given a source text, written in language A, it produces a new text, written in language B.

The source text being considered as a whole by a compiler, one needs to make a distinction with interpreters, which read and translate source texts one sentence at a time.

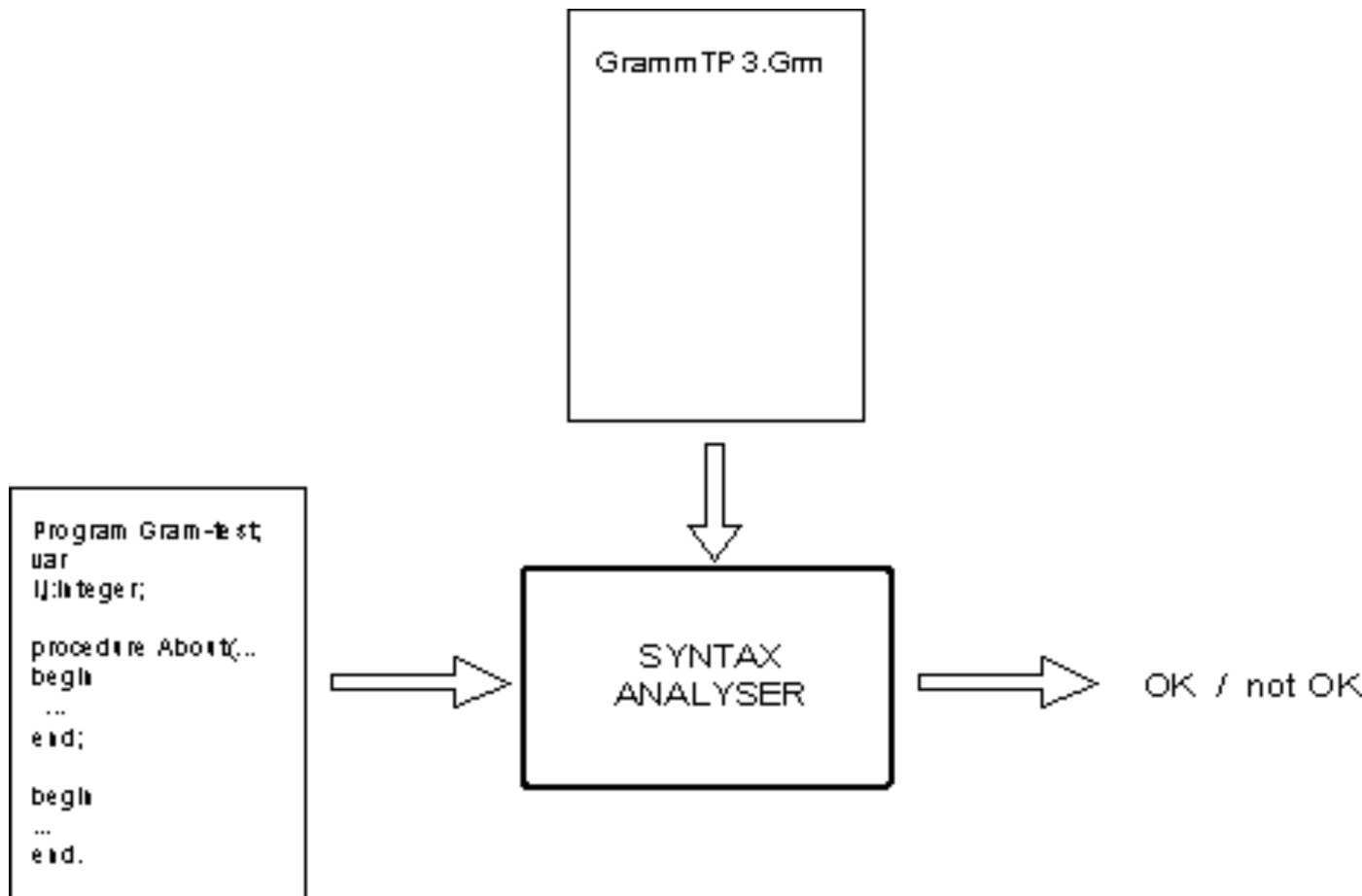
Most of the time, compilers are used to produce machine executable code. The Turbo Pascal package is basically nothing more than a Pascal to Intel 80x86 machine code translator. Providing a .Pas file, you get an .Exe file.

Compilers functions are divided into 4 modules:

The Lexical analysis module reads the source text and slices it into "words", also called lexical units.



Then comes the Syntax Analysis module, which is in charge of checking the 'grammatical correctness' of the previously read sentences. Most of the time, it belongs to this module to request new lexical units from the lexical analysis module as needed.



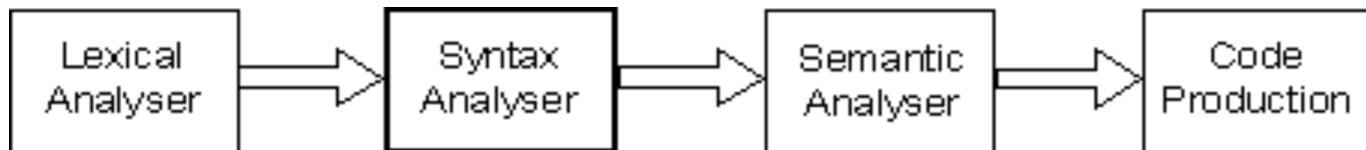
Third, grammatically correct sentences are checked for meaning in a Semantic Analysis module. In Pascal, for instance, Identifiers used in expressions must have been previously declared in a previous declarative section; if not, this sentence has 'no' meaning, according to the compiler's point of view, since it does not know what the text is talking about.

Thus, most of this module's job is to check for contextual informations - that is informations that are not contained within the current sentence's boundaries, but must be localised in some pre-defined place. Type checking and visibility are also common checks performed by semantic analysis modules.

Those first tree modules share a data structure called the Symbol Table, in which every new symbol is stored, together with relevant information such as type, value if applicable, ...

Last comes the Code Production module, that is in charge with the actual translation, on the basis of the checked correctness of words and sentences, plus the info contained in the symbol table.

This last module is called the end module of a compiler, since it is strongly machine dependent, as opposed to the front part, that includes the first three modules, and that is to a lesser extent dependant on the hardware and low level software implementation.



To sum up, one can say that SYNTEX is a Syntax Analyser Generator - hence its name -. The 'Compiler's Compiler' claim being used as a hint for people wanting a quick explanation of SYNTEX's overall functionality.

Naturally, many different approaches can be used to produce a Syntax Analyser, as well as there are many different types of grammars. Furthermore, certain types of grammars ask for certain types of approaches.

For SYNTEX, the analysis is performed by a descendant recursive analysis, which in turn requires the grammar to be of the LL(1) type. This concept is explained and developed in the next chapters.

Descendant recursive syntax analysis with one prevision symbol

The productions - ie. grammar rules - of a given .grm file are simply patterns that source text must follow, in order to be 'grammatically' correct.

Some of these patterns are composed with symbols that are also described in another pattern of the same file. This is a special category of non-terminal symbols.

The sum of all these productions can be drawn as a tree - called the Syntax Tree -, each node being a symbol, and each leaf being a terminal symbol - that is a symbol that is not defined somewhere in the file with a pattern.

To browse through the syntax tree moving from the root to the leaves according to the symbols encountered in a source text statement is called deriving the productions. See an example below.

\\Example\\

Thus, to practice a descendant recursive syntax analysis on a sentence is simply to check that every word encountered in the sentence is the kind of word expected, according to the productions describing the targeted grammar.

Starting from the root, the analyser climbs down the tree, branching at every node - ie. deriving - according to the next lexical unit. This is the reason why this algorithm is called 'descendant', and one talks about 'one prevision symbol'.

There are two basic steps in this algorithm:

1. Given a symbol from the sentence to analyse as input, the current symbol in the syntax tree is derived until all terminal symbols that can be reached from this position are known. This set of possible terminal symbols is also the set of legal symbols for the next word in the sentence.
2. If the input symbol is in the possible terminal symbols set, then everything is OK and one goes on with the analysis, setting the production that begins with this terminal as the new current

syntax tree position and moving forward to the next symbol in the sentence.

If the current symbol in the sentence is not in the possible set of terminals, then the analyser issues an error and stops, because the tested sentence is not grammatically correct.

As you can see now, the ability to be analysed recursively using only one symbol of prevision is a very useful property since it yields a very elegant algorithm, but it is also a very strong one, that defines a whole class of grammars, called the LL(1) class.

LL(1) Grammars

It is possible, thanks to the special property of LL(1) grammars - see above - to automatically build a syntax analyser using recursive nested procedures, which structures are based on the graphical structure of the syntax tree.

\\ Example\\

LL(1) stands for 'Left to right scanning - Leftmost derivation' - '1' standing for the only prevision symbol required.

Thus, the main idea you must keep in mind is that your grammar must be design so that it is always possible to decide which production to invoke next, knowing only the next symbol in the sentence.

More theoretically, the correct definition of a LL(1) grammar is that, for each production of the $A = B \mid C$ form,

For any terminal symbol a , B and C cannot be both derivated into a - ie., starting from two different points, one cannot reach the same destination. Why is it so ?

Imagine that the symbol next to the current one in the sentence is a , and that $A = B \mid C$ is the production to be invoqued. Which is the right alternative to derive: B or C ?

It follows, from the very definition of LL(1) grammars, that one must write down the sets of possible terminal symbols, starting respectively from B and C , and then check whether a is in the B or in the C set.

If a is in both, we cannot decide, the grammar is ambiguous.

B and C must not be derivable into the empty string - ie in no symbol. Put more simply, this means that at least one of both alternatives must lead to somewhere, otherwise, it is still impossible to decide where to go.

First Try

On the more pragmatic side, despite the fact that it is certainly possible to check a .Grm file for LL(1) strict compliance, it is more common to write down a draft version, to test it, and to perform the necessary adjustments, using SYNTAX error messages and debugging facilities.

Moreover, it can be useful to compile a quasi-LL(1) grammar, which is possible because SYNTEX doesn't check for strict compliance.

Some example .Grm files are provided with this package, as a guide to writing LL(1) grammars.

Changes

Two types of corrections can be made; corrections on grammar productions, and corrections on the code output.

Concerning grammar productions, it is important to ensure that a few fatal mistakes are avoided, including:

Left Recursion

A production of the $A = Aa$ form is said to be left recursive, providing 'A' is a non-terminal symbol and 'a' is a terminal symbol, because when called, this production issues, first of all, a call to itself.

How come ?

The syntax analyser outputs a request to the lexical analyser, and thus receives a new lexical unit to process.

Suppose now that, in order to treat this lexical unit, the syntax analyser must use production A. A in turn, calls A, WITHOUT requesting a new lexical unit from the lexical analyser. Then follows a new recursive call to A,... and a stack overflow.

To get rid of a left recursion is quite easy, transforming any $A = Aa \mid b$ into $A = bA'$ and $A' = [aA']$.

There might be occasions where left recursions only appear after derivation of a production. Those cases are more complex to eradicate, and need a global level algorithm... No more about this subject in an introductory paper.

Incomplete left Factorisation

Some productions are similar to $A = BC \mid BD$, where 'B', 'C' and 'D' are non-terminal symbols.

Imagine that in the middle of a process, the syntax analyser is to use production A. A set of all possible terminal symbols is available for branching decision, composed with all the terminal symbols derived from B.

In our case, since both alternatives of production A begin with B, it is impossible to decide, knowing that the next symbol must be in the B-derived set, to know whether to branch to B (from BC) or to B (from BD). A run time error is then issued.

You must remember that SYNTEX only works with LL(1) grammars - remember the '1'. This digit

is important, since our left factorisation problem could probably have been solved, using a LL(2) analyser.

Cycles

Cycles problems are close to left recursion problems in their result. They occur when a production can be derived into itself - whether immediately or after many derivations -. They call for a global treatment though, and thus are beyond this paper's purpose.

Empty productions

Empty productions - those that are of the $A = \epsilon$ form must be erased prior to compilation with SYNTEX.

It is important to check for output code correctness. SYNTEX being relatively liberal with the LL(1) theoretical definition, some lines of code MIGHT have to be modified. See below.

Miscellaneous considerations

So far, when SYNTEX builds the possible terminal symbols set, it does not include in this set those terminals that are included in a metasymbol - only the first terminal of the meta-symbol is included. See for yourself the Wizz.pas file.

This limitation can easily be made up with by adding the missing terminals manually - it is on my priority list of missing features, though.

SYNTEX is a compiler itself, since it produces a translation from .Grm to .Pas. As a consequence .Grm files must obey to a specific grammar, which is detailed in the Syntax.Grm file.

Although SYNTEX will generate no compilation error with nested metasymbols, these are not handled correctly by the code production module. This bug can be overcome by dividing those nested metasymbols into productions. See Wizz.Grm as an example of this special style.

Lastly, SYNTEX is not a lexical Analyser Generator. But it is still possible to take advantage of it to write lexical analyser based on recursive procedure calls. See the real numbers grammar given above.

Version History

Version 2.0 (b)

Rewritten and greatly improved User Interface, using Turbo Vision.

Added features:

- Easy editing of Grm files
- Code output formatting as Program or Unit
- Debugging facilities, including Syntax Tree visual representation
- Improved compilation error handling

Version 1.1

- Stronger I/O control added.
- File extension conventions on .Grm et .Pas.
- .Grm files now support comments.
- Improved text buffer handling

Version 1.0

Basically Pascalissime version, slightly adapted to enable compilation using TP4+ unit facilities.

How to contact the author

I would be very glad to receive any feedback on SYNTEX. You can reach me at the following addresses:

Mail: LE TENO Jean Franpois
19,Rue du Docteur Bordier
38100 GRENOBLE
France

Email: Big John @ 3614*TEASER

CompuServe: 100346,3212

EMail leteno@grenoble.cstb.fr