

P65Pas 0.8.9

Documentation – Fundamentals and Analysis

By Tito Hinostroza

Contents

REFERENCES	5
1 INTRODUCTION	6
1.1 Summary	6
1.2 Structure of Compiler	6
1.3 Implementation	9
2 The LANGUAGE	11
2.1 Differences with the Pascal	11
2.2 Blocks	11
2.3 Sentences	12
2.3.1 sntAssigment	12
2.3.2 sntExpress	12
2.4 Operand	12
2.5 Expressions	13
2.6 Operands	13
2.7 Structural blocks	14
3 ELEMENTS OF SYNTAX	15
3.1 Declaration elements	16
3.2 Structure elements	17
3.3 Elements of expression	18
3.4 Elements related to instructions	18
4 CONSTANTS	19
4.1 Declarations of constants	19
4.1.1 Structured Constants	21
4.1.2 Evaluation of constants	22
4.2 Life cycle of constants	23
4.3 Constant Expressions	24
5 HANDLING TYPES	26
5.1 The TEleType classDec	26
5.2 Basic Types	27
5.3 Type categories	29
5.4 User-defined types	30
5.5 Copy Types	31
6 BASIC AND STRUCTURED TYPES	32
6.1 Type Boolean	32
6.2 AccumStatInZ flag	33
6.3 Structured Types	35
7 EXPRESSIONS	36
7.1 Pascal Expressions	36
7.2 Expressions in P65Pas	36
7.3 Operators and precedence	38
7.4 Objects and Methods	40
7.5 Special Operators	41
7.6 Functions in Expressions	43
7.7 Tree diagrams	43
7.8 Operands and Expressions	45

7.9	Assignments	46
8	FUNCTIONS AND PROCEDURES	48
8.1	Procedures and functions	48
8.2	Representation of functions	48
8.3	The declaration/implementation call problem	50
8.4	Functions as methods of objects.	51
8.5	User and system functions	52
8.5.1	System Functions	52
8.5.2	User Functions	53
8.6	Implementation of functions for operators	53
10	SYNTAX TREE	56
10.1	Fundamentals	56
10.2	Structure	58
10.3	Duplication of names	60
10.4	Name resolution	61
10.4.1	FindFirst() and FindNext()	61
10.4.2	Identifier Comparison	63
10.4.3	Identifier scanning	64
10.4.4	Calls from TEleBody	64
10.4.5	Calls from Declarations	66
10.5	Filling the syntax tree	66
10.5.1	Considerations	67
10.6	Calls or Referrals	68
10.6.1	Calls from TEleConsDec	69
10.6.2	Calls from TEleVarDec	69
10.6.3	Calls from TEleTypeDec	69
10.6.4	Calls from TEleBody	70
10.6.5	Implementation of calls	70
10.6.6	Code	72
10.6.7	“Callers” and “called”.	73
10.6.8	Calledelements	74
10.7	Program Frames	75
10.8	Final state of AST	76
11	SENTENCES	78
11.1	Types of sentences	78
11.2	Procedure Call Statements	79
11.3	Assignment Statements	79
11.3.1	Expressions	81
11.4	Assembler Blocks	81
11.5	Conditionals	81
11.5.1	Structure of a conditional	83
12	ASSEMBLY BLOCKS	87
12.1	Lexical analyzer	87
12.2	Types of ASM Instructions	88
12.3	Common Instructions	89
12.3.1	Direct operands	89

12.3.2	Operands element	91
12.3.3	Label	91
12.3.4	Operating "\$"	94
12.3.5	Operations	94
12.4	Instructions Label	96
12.5	ORG Directive	97
12.6	DB96 directive	98

REFERENCES

For a better understanding of this documentation, it is recommended to have knowledge of the following topics:

- Object Pascal
- P65Utils library.
- CPU 6502.
- 6502 assembly language.

1 INTRODUCTION

1.1 *Summary*

The P65Pas project is a large project like any project that aims to develop a working compiler for a high-level language like Pascal.

The project consists of the development of a cross-platform Pascal compiler that compiles binary code for the CPU6502, without external dependencies.

The project, including all the libraries used, is developed using the Lazarus IDE, with the Free Pascal compiler.

This project was born as an extension of the PicPas project, a compiler for PIC microcontrollers, but it has taken a separate course and has allowed the author to test new designs regarding the internal structure of the compiler. In other words, this compiler was born from the PicPas compiler but uses new design techniques to improve the compiler's functionalities and fix bugs found in PicPas development.

The main objectives of this project were:

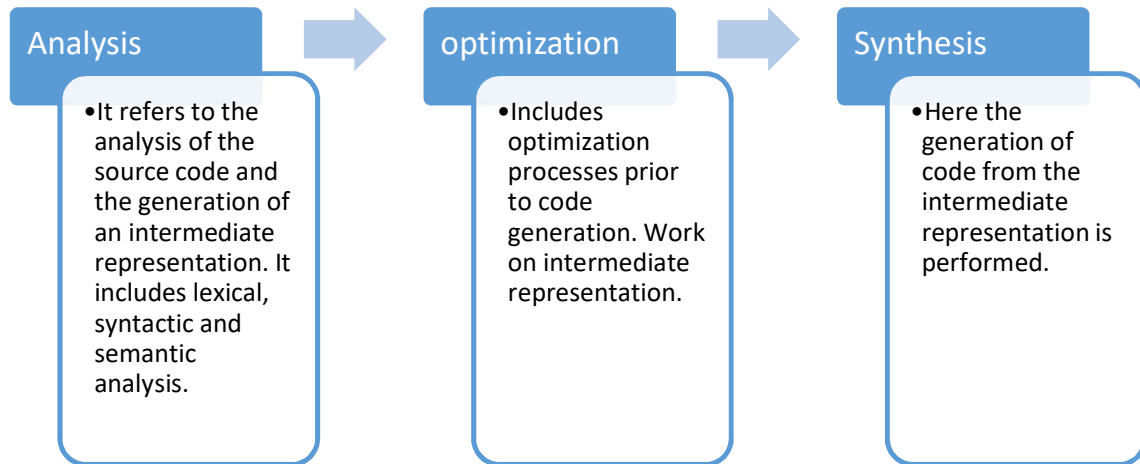
1. Reuse all the engineering developed for the PicPas compiler, in a new compiler project.
2. Test the developed libraries (of my authorship) in a new project in order to improve them and make them more general.
3. Provide a working compiler as an alternative to developing on the 6502 CPU, using the Pascal language.
4. Try new compiler design techniques on a new project, with little risk, because it is a new project.

Although this compiler is intended to be considered a Pascal compiler, the Pascal language implementation is different from the ISO or Turbo/Borland Pascal implementations. For more information on the differences, see Section 2.1.

1.2 *Compiler Structure*

You could say that the compiler follows the more or less standard design of most compilers.

In summary, the compiler could be divided into three main processes:



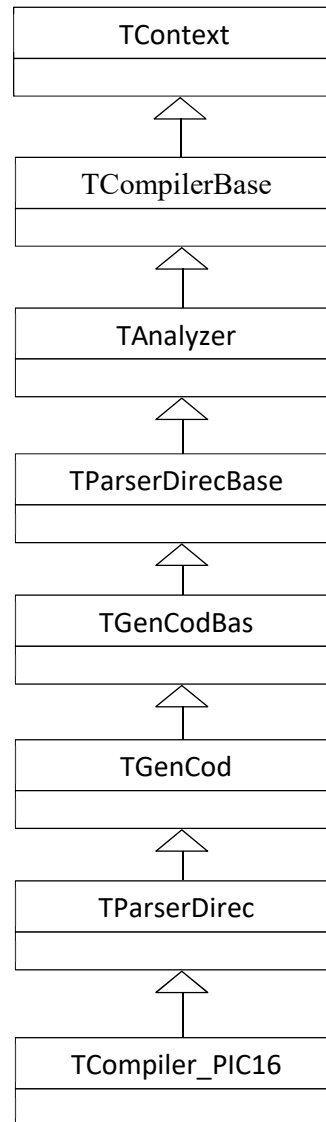
The intermediate representation is the abstract syntax tree. Here the product of the analysis is written and it is also the workspace for the optimization.

One point where P65Pas differs from classical compilers might be in the fact that parsing is minimal and it is tied very tightly to the semantic analyzer. That is to say that a pure syntactic analysis is not carried out, but rather it is carried out while doing the same semantic analysis of the language.

By not having a separate parsing stage, it is difficult for the compiler to generate multiple errors in the same source code, since this would require jumping to a next statement, when the current statement is detected to have errors, and to For this, the sentences would have to be identified first (pure syntactic analysis).

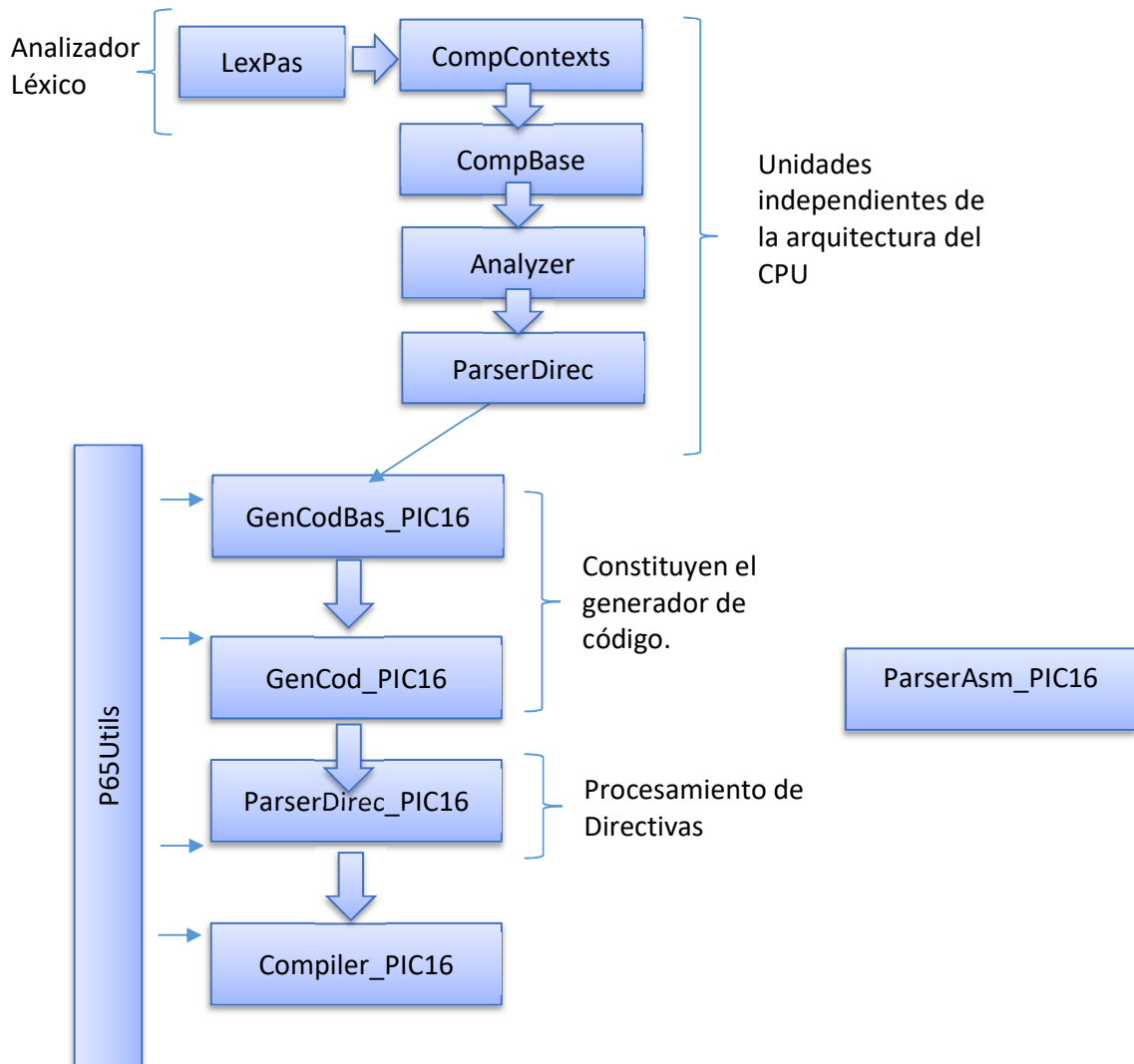
The compiler, at the global level, is seen as a more or less extensive class that is built as a cascade of various classes that scale in a nested inheritance relationship.

In a class diagram, we would see that the compiler follows the following structure:



1.3 Implementation

The units (files) that define the compiler, in hierarchy, are:



The innermost units: CompContexts, CompBase, Analyzer, and ParserDirec, are designed not to be exclusive to the P6502 compiler, but to be able to serve as a base for other compilers or interpreters (as, in fact, is done in the P65Pas project) and contain the lexical analyzer routines, parser, the TCompilerBase compiler object definition, and most of the directive processor.

The outermost units, which make use of the P65Utils library, are specific to the CPU6502, and implement code generation and assembly functions (for ASM blocks). In addition, they contain the higher-level routines, such as those that perform unit compilation and linking.

Although this separation of units, into CPU dependent and non-CPU dependent, may not seem very useful in this compiler (except for the modular issue), it is important if you plan to

reuse code for other compilers. An example of this is best seen in the PicPas compiler, which includes 3 different compilers using the same base units.

2 The LANGUAGE

For the implementation of the compiler, we start from the definition of the Pascal language syntax, in the form in which it will be implemented in this compiler.

The particular implementation of Pascal, which is done in this compiler, is simplified (to work with resource-constrained devices), but this implementation also contains modern features such as basic object support.

For the generation of the diagrams, the tool is being used:
<https://tabatkins.github.io/railroad-diagrams/>

2.1 Differences with the Pascal language

The Pascal implementation, which defines the compiler, differs from the Pascal language, which is used by most compilers, mainly in:

- The syntax of the IF and WHILE blocks correspond to the way they are defined in Modula-2. That is, they do not generate BEGIN ... END blocks.
- Only simple types, such as Char, Boolean, Byte, and Word, are implemented. The string type is also not implemented.
- The syntax sticks to OOP notation, rather than typical Pascal structured programming. So instead of using "length(a)", it is preferred to use "a.length".
- Almost all data types, including basic types like Byte or Boolean, include methods that can be called in the common OOP form: <variable>.<method>
- There are no operators, in the strict sense. Operators, within this language, are just shorthand for calling methods. So for example, the expression "x+y" can be written as "x._add(y)". For more information on handling expressions see section 3.3.

2.2 blocks

The recognized blocks are:

CONST
TYPE
VAR
PROCEDURE
FUNCTION (Only in Pascal mode)
START

Note that unlike standard Pascal, no LABEL tag blocks are recognized.

BEGIN...END blocks are recognized as a set of statements.

2.3 Sentences

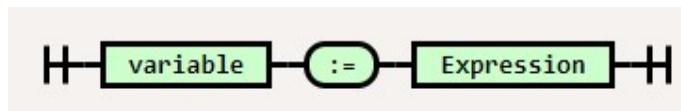
The various types of sentences are determined by the enumeration TxpSentence, defined in the XpresAST unit.

The recognized sentences are:

2.3.1 sntAssignment

Simple assignment statement:

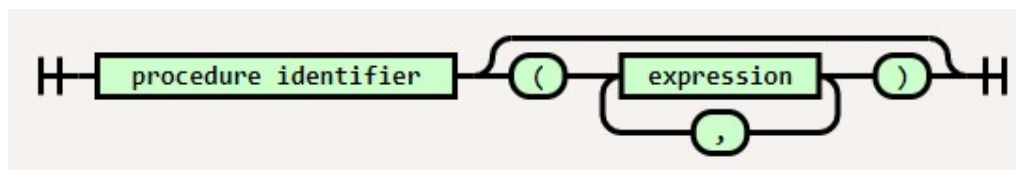
```
Diagram(Sequence(NonTerminal('variable'),Terminal(':='),NonTerminal('Expression')))
```



2.3.2 sntExpres

Expression, or operand.

```
Chart(
  NonTerminal('procedure identifier'),
  Choice(1,
    Skip(),
    Sequence(Terminal('('), OneOrMore(NonTerminal('expression'),''), Terminal(')'))
  ),
)
```



This case also includes functions.

```
sntBEGIN...END
sntIF
sntREPEAT
sntWHILE
sntFOR
sntCASE
```

2.4 operand

2.5 expressions

Expressions are defined a bit differently than common Pascal compilers:

2.6 Operands

They are considered as expressions with the highest precedence. That is, indivisible.

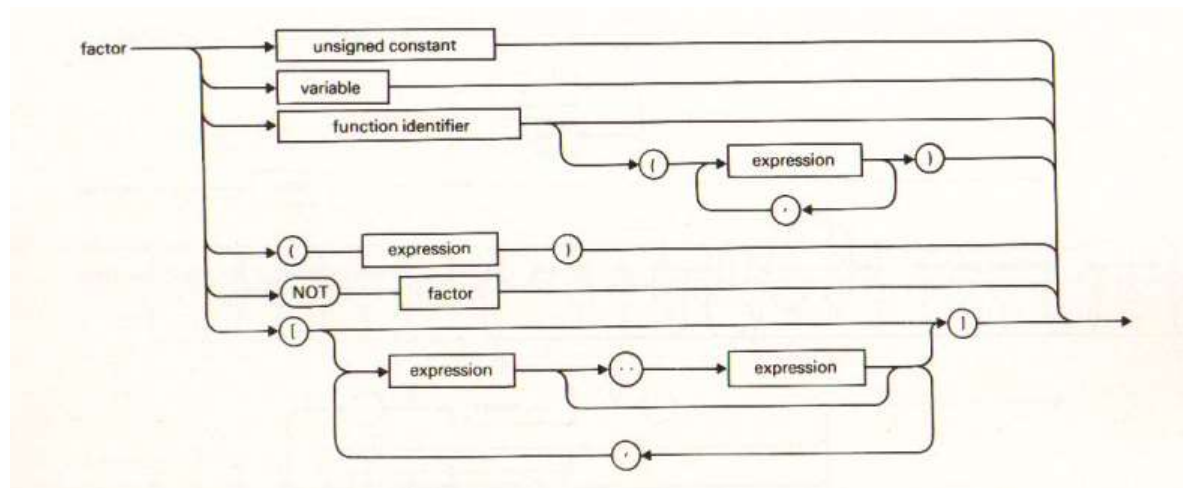
They are classified in:

Constant -> Can be literal or constant expression.

Variable

FunOperand

Brackets -> Expression in parentheses.



2.7 *structural blocks*

This section describes how the compiler processes the building blocks of a program.

These blocks are:

- programs
- statements
- Main body
- Units
- Interface Section
- Implementation Section
- Initialization Section
- Completion Section

A program

Sentences

Statements are of type:

- common sentences.
- assembler blocks

3 ELEMENTS OF SYNTAX

To perform source code analysis, the compiler models:

- The expressions
- variables
- Constants
- Procedures
- etc.,

as objects called "elements" (short for "syntax elements" or "elements of a program"), and this is how they are identified within the compiler's source code.

The elements are objects that derive from the TxpElement class, declared in the XpresElemP65 unit:

```
{ TxpElement }
// Base class for all syntactic elements
TxpElement = class
private
  FName      : string; //Element name
  Funame     : string; //Upper case name. Used to accelerate searchings.
  procedure Setname(AValue: string);
public //Callers management
  lstCallers: TxpListCallers;
  function nCalled: integer; virtual;
  function IsCalledBy(callElem: TxpElement): boolean;
  function IsCalledByChildOf(callElem: TxpElement): boolean;
  function IsCalledAt(callPos: TSrcPos): boolean;
  function IsDeclaredAt(decPos: TSrcPos): boolean;
  ...
public //Location in the source code.
  srcDec: TSrcPos; //Where the element starts.
srcEnd : TSrcPos ; // Where the element ends.
  ...
}
```

The attributes of this class represent the common properties that elements must have. Thus, for example, all elements such as variables, constants or the body of the main program itself have a location in the source code, where they are declared or implemented. This location must be saved in the "srcDec" field.

The various elements of the program are derived from this class, adding the fields and methods required by each element.

Since all element classes have a common ancestor, to identify the specialized classes, a numeric identifier is used that is initialized in the constructor of each specialized class:

```
constructor TElVarDec.Create ;
begin
  inherit ;
```

```
idClass := eleVarDec ;
end ;
```

Classes derived from TxpElement are recognized by their identifier, rather than using Free Pascal's RTTI (to identify classes) options, to gain efficiency.

The different types of elements that are handled in the current version (Declared in the XpresElemP65 unit) are listed below:

```
// Element types for the language.
TxpIDClass = (//Declaration
    eleNone,      //No type
    eleVarDec,    //Variable declaration
    eleConsDec,   //Constant declaration
    eleType,      //Type declaration
    eleFuncDec,   //Function declaration
    eleFunc,      //Function
    //Structural
    eleProgFrame, //Code container
    eleProg,      //Main program
    eleUnit,      //Unit
    eleBody,      //Body procedure/program
    eleBlock,     //Block of code
    eleFinal,     //FINALIZATION section
    //Instructions relative
    eleSenten,    //Sentence/Instruction
    eleAsmInstr,  //ASM line
    eleAsmBlk,    //ASM block
    eleCondit,    //Condition
    //Of Expressions
    eleExpress,   //Expression
    eleAsmOperand, //Operand
    eleAsmOperat  //Operator
);
```

The TxpElement objects are important because they are precisely the objects that are added to the syntax tree (See section 10) and that are precisely the nodes.

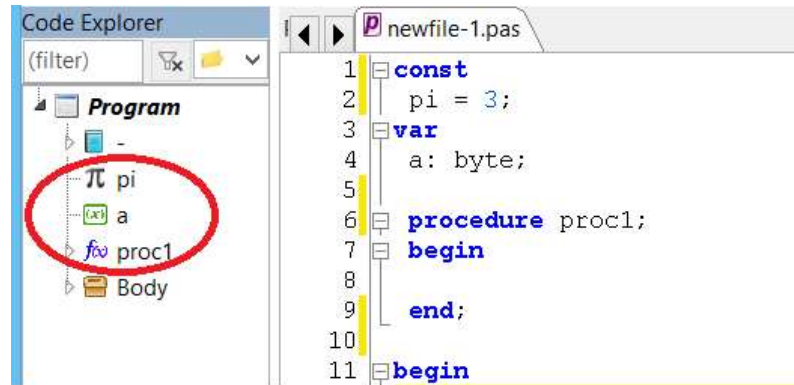
3.1 declaration elements

These elements are created from the declaration sections of programs, functions, or units.

identifier	Class	Description
eleNone	TxpElement	It is an element that has not yet been specialized with a type.
eleVarDec	TEleVarDec	They represent a variable, local or global. Includes objects.
eleConsDec	TeleConsDec	They represent a local, or global, constant. Includes objects.
eleType	TEleTypeDec	They represent a type declaration.

eleFuncDec	TeleFunDec	They represent the declaration of a function or procedure.
eleFunc	TeleFun	They represent the implementation of a function or procedure.

These elements are graphically reflected in the code browser, before the body of a program, function, or unit:



3.2 structure elements

These elements represent the broader blocks that define the structure of programs.

identifier	Class	Description
eleBody	TEleBody	It represents the main body of the program, seen as a single block. Does not include statements. It also represents the body of procedures and the INITIALIZATION section of units.
eleBlock	TEleBlock	Represents a block of code as the body of an IF or a section delimited by BEGIN ... END.
eleProgFrame	TEleProgFrame	They represent a program, unit or function. For more information see Section 10.7
eleProg	TeleProg	It is the main node of the syntax tree, and therefore contains all the elements of the syntax tree.
eleUnit	TeleUnit	They represent the identifier of a unit in the USES section.
eleFinal	TeleFinal	Represents the FINALIZATION section of units.

There is an additional element: TxpEleDIREC which is used to represent the conditional nodes in the directive tree. TxpEleDIREC elements are not stored in the main syntax tree.

3.3 *expression elements*

Expression elements represent all the elements that are usually found in a Pascal expression, such as operands and methods.

identifier	Class	Description
eleExpress	TeleExpress	They represent an expression or operand. For more information see Section 10.8
eleAsmOperat	TeleAsmOperat	Represents an operator within ASM blocks.

The eleAsmOperand and eleAsmOperat types are exclusively reserved for handling ASM blocks, which handle expressions in the classical sense, unlike how they are handled in P65Pas Pascal (See Section 7.2).

Within the compiler (without considering ASM blocks) expressions, of any type, will be represented by an element of the TeleExpress class, and will have the same form as the diagrams shown in section 7.7.

For more information on the representation of expressions within the Syntax Tree, see section 11.2,

3.4 *Items related to instructions*

These elements model the sentences or instructions.

identifier	Class	Description
eleSenten	TeleSentence	They represent a sentence.
eleAsmInstr	TEleAsmInstr	Represents an assembly instruction within ASM blocks.
eleAsmBlock	TEleAsmBlock	Represents an ASM block. ASM-delimited ... END

Statements are all the instructions that a program can contain, including ASM blocks. For more information on sentences, see Section 10.8.

4 CONSTANTS

Semantically we understand Constants as values that, once declared and initialized, do not change during the entire program.

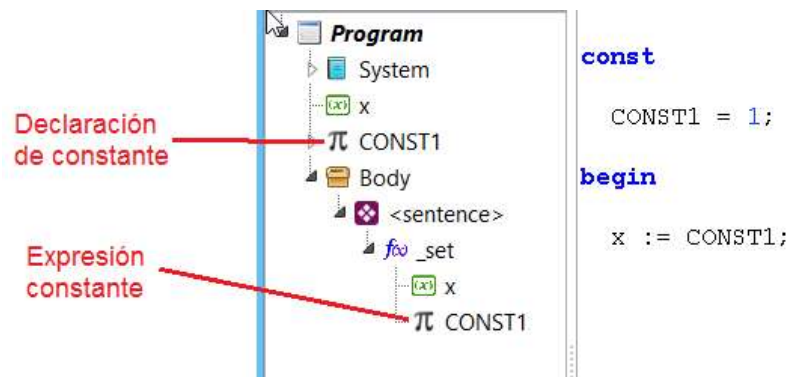
The declaration of a constant is done in the form:

```
Const
CONST1 = 2 ; // Declaration of a constant
...
```

And to reference a constant, just use its name:

```
...
X := CONST1 ; // Reference to a constant
...
```

Syntactically constant declarations and constant references generate different elements in the Syntax Tree:



Therefore, we will study two completely different types of elements:

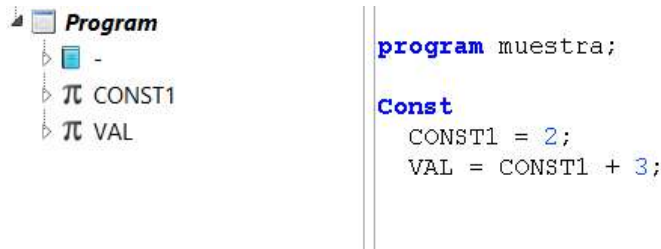
- Declarations of constants
- constant expressions

4.1 Declarations of constants

Constant declarations appear at the beginning of a program:

```
program shows ;
Const
CONST1 = 2 ; // Declaration of a constant
CONST2 = CONST1 + 3 ; // Reference to another constant
...
```

You can verify that these statements generate syntax elements by viewing the code explorer:



The “constant declaration” elements are modeled with the `TEleConsDec` class, which derives from the `TxpEleCodeCont` class that defines code containers (For more information on code containers, see section 10.6.6).

The actual value of the constants is stored in the "value" field of `TEleConsDec`, which is an object that allows values of all data types supported by the compiler to be stored.

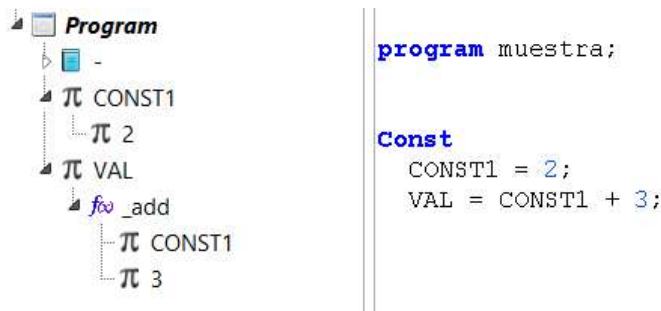
```

TEleConsDec = class ( TxpEleCodeCont )
  // Element type
  typ : TEleTypeDec ;
  evaluated : boolean ;
  // constant-value
  value : TConsValue ;
  Create constructor ; override ;
end ;
  
```

The `TEleConsDec` elements represent the constants in the declaration section, but they do not represent the constants when they are referenced from the statements (the `TxpEleExpres` element is used there).

Literals such as 123, \$FF, or 'A' are also syntactically constant, but are not modeled as a `TxpEleCodeCont` object within the compiler. These literals are read the moment they are encountered and loaded into a `TEleExpress`, assigning the value to it and setting it to type `otConst`.

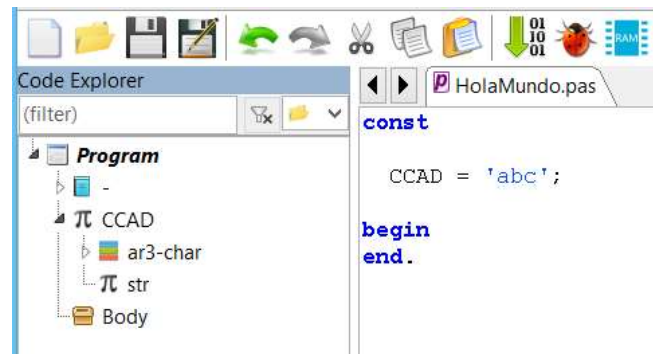
Constants can also contain expressions in their definition. In the parsing process, the expressions inside the constant elements are included:



4.1.1 Structured constants

Constant declarations may be initialized with complex values such as arrays, pointers, or objects.

Consider for example the following case:

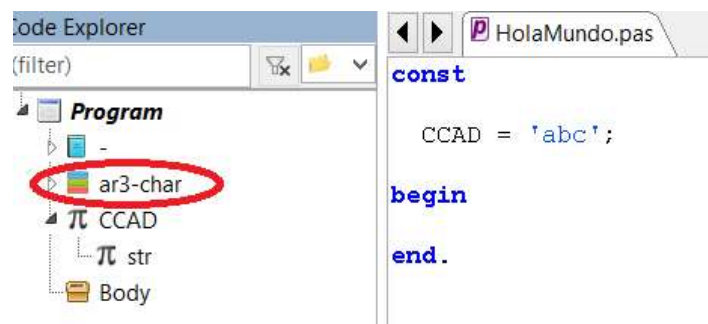


This constant declaration is initialized with a string that, within the language, is interpreted as an array of CHAR elements.

Due to the way `TAnalyzer.AnalyzeConstDeclar()` works, we see that in the code explorer, the constant declaration containing the type “ar3-char” has been generated, which is the type that has been created automatically in order to create the literal constant. 'abc'

However, this type has been created inside the “CCAD” declaration (because that's how `GetExpression()` works), but to be consistent with Pascal's syntax, it's best to create the type outside of CCAD, at the same level.

This is why `TAnalyzer.AnalyzeConstDeclar()` does extra work and checks to see if the evaluation of the seed value for the constant has created new types (it is only expected that a top-level one and possibly more nested ones can appear, if the expression is complex) and moves them (only to top-level types) in a way that emulates a previous type declaration.



In this case, the CCAD constant has generated the type “ar3-char”, which is an array of 3 characters, and has taken that type to contain 'abc'. Don't forget that the compiler interprets all strings as character arrays.

4.1.2 Evaluation of constants

Processing constants might seem simple, because you can read them directly with their value and update them in your "value" ¹. And this is true for type constants:

```
CONST CONST1 = 2;
```

But constants that have expressions as their definition require a previous evaluation to obtain their final value, as in the following case:

```
VAL = CONST1 + 3;
```

The use of this type of expression requires a previous evaluation, using an expression analyzer, to obtain the final value.

Now, if an expression evaluator is needed, we would also need to define the capabilities of this expression evaluator, such as what level of complexity would be supported in expressions and how data types would be handled.

To avoid having to create a special expression parser, it was decided to use the same parser that is used for code generation (based on TCompilerBase.GetExpression). But the processing of the constants, and their expressions, is done in the analysis process, so the calculation of the constant expressions is left for the synthesis process (actually it is done in the optimization).

Using the same expression parser used for synthesis (code generation), in evaluating constant expressions, has the following advantages:

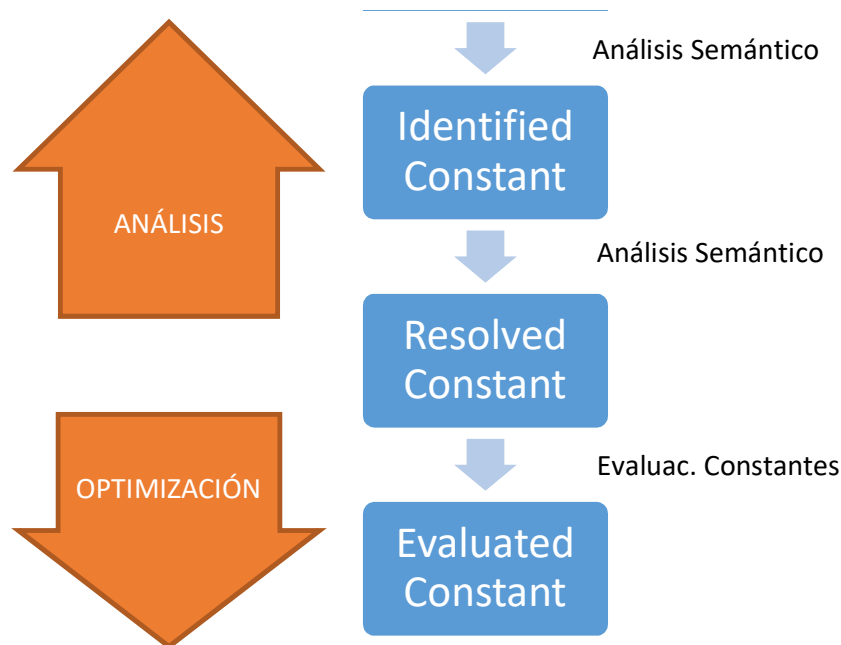
- You don't need to create a whole expression parser, or an additional error processing system.
- The syntax of the expressions and the data type is similar to that of the Pascal language of the entire program. Thus it is not necessary to manage (and maintain) separate syntaxes.
- Expressions can be assigned a data type consistent with Pascal types.

On the other hand, using the code generator's expression parser has the disadvantage that expressions cannot be evaluated in Parsing. So it must be postponed until optimization, to be able to obtain the final value of constants that have been defined as expressions.

¹This way of working was used in previous versions of the compiler and it simplified the compilation process considerably, but since the same expression evaluator was used as for the compilation, it was necessary to include code generator routines in the parsing process, which was against adequately separating the process of analysis from that of synthesis.

4.2 Life cycle of constants

Like variables and functions, constants also handle a life cycle that goes through the following stages:



Does it apply to TElConsDec and TElExpress?

This process applies only to constants with identifiers, because literal constants are identified and computed (read) in the same process and do not require resolution.

Constants are identified and resolved in the parsing process. The calculation or evaluation is done in the optimization process, which also includes the folding of constants and other additional optimizations, described in the document “Documentation – Optimization and Synthesis”.

Identifying a constant implies taking its identifier and concluding that it is the identifier of a constant.

Solving for a constant implies knowing which constant declaration it refers to. This job is executed with a simple comparison of the constant identifier with all declared constants. The code that resolves, not just constants, is found in TCompilerBase.GetOperand():

```

function TCompilerBase.GetOperand (): TElExpress ;
...
ele := TreeElems.FindFirst ( token ); // Identify element
findState := TreeElems.curFind ;
if ele = nil then begin
  //Unidentified element.
  GenError(ER_UNKNOWN_IDE_, [token]);
  
```

```

    exit;
end;
if ele.idClass = eleConsDec then begin //Is constant
    xcon := TTeleConsDec(ele);
AddCallerToFromCurr ( ele );
    Op1 := OpenExpression ( ele.name , xcon.Typ, otConst, GetSrcPos );
...

```

Name resolution is explained in section 10.4.

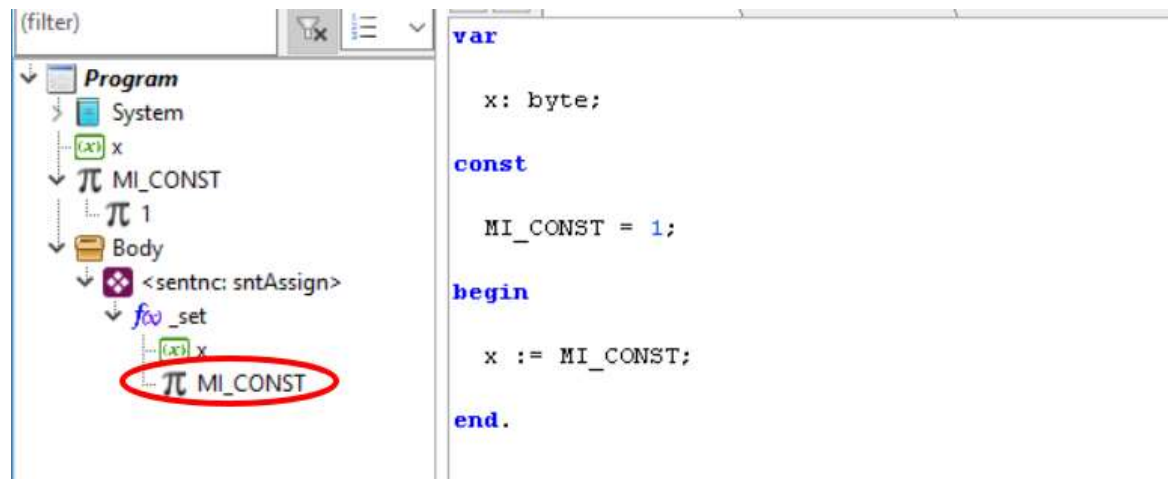
As the resolution of elements is done in the Analysis process, it is expected that when the optimization starts (See section 1.2 for more information on the processes) all the identifiers have already been resolved.

4.3 constant expressions

Constant expressions (or constant operands), are syntactic elements that represent:

1. A constant literal (of any type) within an expression.
2. An identifier referencing a constant declared in the CONST section.

Constant expressions, like all expressions, are modeled with elements of type TTeleExpress, but of type otConst.



Note the difference of the constant operands with the constant declaration, that although semantically they can represent the same thing, within the AST they are differentiated objects.

The characteristics of constant expressions are defined in the following table

Access:	Read only
Element	TeleExpress
Guy:	otConst
Storages:	stConst, stRamFix

The storage of a constant refers to the way its value is read and stored in the compiler or executable program. For more information on storage, review the document “Documentation – Optimization and Synthesis”.

5 HANDLING OF TYPES

From the beginning, the Pascal language has been characterized by the power that type management offers, beyond the simple structures that the C language offered in its beginnings.

The implementation of Pascal types of P65Pas is not complete, but it is still under development and it is expected to cover as much as possible, with the specifications, of at least Turbo Pascal, always considering the hardware limitations of the 6502 CPU.

Within P65Pas, the types are divided into:

- Basic Types. Like Char, Byte or Word.
- User-defined types.

The difference between basic types, and user-defined types, is similar to the difference between system functions and user functions.

BASIC TYPES	USER DEFINED TYPES
They always exist, from the beginning of the compilation, in the drive System.	They are created by code, according to the compiled program.
They are always available and from anywhere in the code.	Its access follows the scope rules of all syntactic elements.

Despite the difference, all types are represented as instances of the same `TEleTypeDec` class.

5.1 The *TeleTypeDec* class

Within P65Pas, all data types, both basic and user-defined types, have an instance of the `TEleTypeDec` class associated with them:

```
TEleTypeDec = class ( TxpEleCodeCont )
private
fSize : SmallInt ;
internalTypes : TEleTypeDecs ; // Container for types recursively defined.
function getSize : smallint ;
procedure setSize ( AValue : smallint ) ;
public // Events
OnSaveToStk : procedure of object ; // Save data in reg. stack
OnLoadToWR : TProcLoadOperand ; {Used when requesting to load a
operand (of this type) on the stack. }
OnGlobalDef : TProcDefineVar ; {It is called every time the
declaration of a variable (of this type)
at the global level.}
public
copyOf : TEleTypeDec ; // Indicates this type is copy of other
group : TTypeGroup ; //Type group (numéric, string, etc)
catType : TxpCatType ; //Categoría del tipo
```

```

    property size: smallint read getSize write setSize; //Tamaño en bytes del
tipo
    function groupStr: string;
    function catTypeStr: string;
public //Arrays and pointers
    consNitm: TEleConsDec;
    itmType : TEleTypeDec;
    isDynam : boolean;
    ptrType : TEleTypeDec;
    function nItems: integer;
public
    procedure SaveToStk;
public //Identificación
    function IsByteSize: boolean;
    function IsWordSize: boolean;
    function IsDWordSize: boolean;
    function IsArrayOf(itTyp: TEleTypeDec; numIt: integer): boolean;
    function IsPointerTo(ptTyp: TEleTypeDec): boolean;
    function IsEquivalent(typ: TEleTypeDec): boolean;
public
    Create constructor ; override ;
    destroyer Destroy ; override ;
end ;

```

Various events are included in the class, to perform tasks of the type. That is why, when creating a type, various routines must be implemented. Let's take as an example the creation of the byte type:

```

// single-byte numeric type
typByte := CreateSysType ( 'byte' ,t_uinteger,1 ); // 1-byte
...
typByte.OnLoadToRT := @byte_LoadToRT ;
typByte.OnDefRegister := @byte_DefineRegisters ;
typByte.OnSaveToStk := @byte_SaveToStk ;

```

The OnLoadToWR event generates the code required to load the value of a data byte into the appropriate RTs (which in this case is just W).

The OnDefRegister event ensures that the RTs associated with the type (See section **Error! Reference source not found.**), are assigned to memory. In this case it is not necessary.

The OnSaveToStk event generates the code to save the content of the RTs associated with the type on the stack.

5.2 Basic Types

The basic types, in P65Pas, are the ones that always exist, and do not need to be created to use them inside the compiler, like the byte or char types.

Currently, the following basic types are created:

```

////////// System Types //////////
typBool  : TEleTypeDec ;
typByte  : TEleTypeDec ;
typChar  : TEleTypeDec ;
typWord  : TEleTypeDec ;

```

They represent the types boolean, byte, char, and word, respectively.

In addition to these types, there is the typNull type (created in the XpresElemP65 unit), which is a type that indicates that no value is returned. Something like the VOID of the C language, only you don't need to specify it, within the Pascal syntax.

The basic types are represented as an instance of the TEleTypeDec object, and are created in TGenCod.CreateSystemElements, inside the System unit:

```

procedure TGenCod.CreateSystemElements;
...
begin
...
uni := CreateUnit('-'); //System unit
TreeElems.AddElementAndOpen(uni); //Open Unit
////////// System types //////////
typBool := CreateEleType('boolean', srcPosNull, 1, tctAtomic);
typBool.location := locInterface;
TreeElems.AddElementAndOpen(typBool); //Open to create "elements" list.
TreeElems.CloseElement; //Close Type
typByte := CreateEleType('byte', srcPosNull, 1, tctAtomic);
typByte.location := locInterface;
TreeElems.AddElementAndOpen(typByte); //Open to create "elements" list.
TreeElems.CloseElement;
typChar := CreateEleType('char', srcPosNull, 1, tctAtomic);
typChar.location := locInterface;
TreeElems.AddElementAndOpen(typChar);
TreeElems.CloseElement;
typWord := CreateEleType ('word', srcPosNull, 2, tctAtomic );
typWord.location := locInterface ;
TreeElems.AddElementAndOpen ( typWord );
TreeElems.CloseElement ;
...

```

Type creation is done in a similar way to defining a new type by code.

Each type is associated with RTs defined for that type. Thus, for example, the types byte and char are associated with record A. For more information on the records associated with the types, see Section 6.

The basic types are organized into groups:

```

TTypeGroup = (
t_integer, // integers
t_uinteger, // unsigned integers
t_float, // floating point

```

```
t_string, // character string
    t_boolean, // boolean
t_enum // enumerated
);
```

Not all of these groups are used in the current version of P65Pas, but they have been defined for future implementations.

The following table describes each of the groups:

GROUP	DESCRIPTION	TYPES INCLUDED
t_integer	Signed integer.	<none>
t_uinteger	Unsigned integer.	Byte, Word, DWord, Char
t_float	Floating point number.	<none>
t_string	Character string.	<none>
t_boolean	boolean types.	boolean
t_enum	Enumerated types.	<none>

The idea of this field was to categorize types and provide additional information indicating compatibility between types. But so far, it is seen that the usefulness of the group field is scarce and should be analyzed if it is necessary to maintain it.

5.3 Type Categories

The first type categorization is done through the catType field, of the enumerated type TxpCatType:

```
// type categories
TxpCatType = (
tctAtomic, // Basic type as ( byte, word, char )
    tctArray, // Array of some other type.
tctPointer, // Pointer of another type.
tctObject // Record multiple fields.
);
```

The tctAtomic types are the basic types like Byte or Char. But it is also used on types that are copies of the base types (See section 5.5). These would be the atomic types, those that cannot be divided into simpler types.

The other categories of types are no longer atomic types, but are formed from other types. They can be considered as structured types.

The tctPointer types are the simplest structured types. They are references to another type of variable. Its implementation is somewhat complex, but it serves as the basis for the implementation of fixes.

The `tctArray` types are arrays of a basic type or another non-basic type. In the current version of P65Pas, only basic type arrays are allowed.

The `tctObject` types are records that group different types in a single structure.

5.4 User-defined types

User-defined types are also handled as instances of the `TEleTypeDec` class. But they are created on demand, by code.

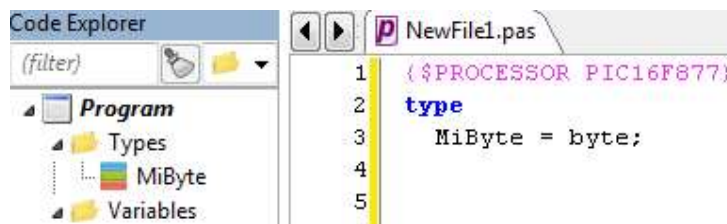
Initially, the basic types belonged to a class different from that of the user-defined types, but they were unified in the same class, to simplify handling and to give the compiler more flexibility, because since all the types are syntactic elements, they are you have more control over them, from the source code.

Consider the case of a type declaration:

```
type
MyByte = byte ;
```

This declaration will create a new `TEleTypeDec` element, and add it to the syntax tree, so that it can be recognized later in the variable declaration.

You can easily check that this type is recognized by the compiler by looking at the code browser.



However, the idea of creating user-defined types is to be able to create complex structured types from basic types. Among them:

- Pointers to basic types.
- Basic type arrays
- Registers combined basic types.

And you should not only be able to create types from basic types, but also from other user-defined types.

Flexibility in Pascal's type declaration also means having a flexible definition for the `TEleTypeDec` class.

5.5 Types Copy

A simple type declaration like:

```
type  
MyByte = byte ;
```

Generates a new type, which is (or at least should be) similar to the byte type. The only differences would be the name of the type, and that this new type does have a declaration in the source code.

Declaring a type in this way is what is called a copy type, and internally it is saved as a user-defined type, but putting the “copyOf” field pointing to the basic type typByte.

Such a declaration actually generates a copy of the basic object type. And although you may think that it is not necessary to create a copy, but to use the same object, there are a couple of good reasons for this:

- Although a copy type is functionally similar to a basic type, the copy type has other scoping rules.
- A basic type is unique and has the same name. Many copy types can be created, pointing to the same basic type, using different names.

A copy type is identified by its “copyOf” field, which is the first field that must be read from a type before trying to work with it, because otherwise unused fields can be accessed.

To avoid errors when accessing copy types, the TELeVarDec class considers in its “TELeVarDec.typ” property, if it is a copy type, and refers to the copy type, if so. Then it could be said that access to the type, in the TELeVarDec class, is safe, considering that accessing its “typ” field does not represent the original type of the variable, but rather its copy.

For the other cases, in which TELeTypeDec is accessed, outside of TELeVarDec, its “copyOf” field should always be verified.

6 BASIC AND STRUCTURED RATES

This section describes the particular implementation of the basic types within P65Pas.

Unlike the other sections, which are more general to the compiler itself, this section describes how the compiler is implemented to support this CPU architecture.

6.1 Boolean type

Value range	false, true
type size	8 bit
Main Storage in RT (Boolean Expression Storage)	Record A
alternative storage	Z-flag, C
binary representation	Variable : \$00 -> Interpreted as FALSE \$FF -> Interpreted as TRUE Other value -> Undetermined. Record A: \$00 -> Interpreted as FALSE \$FF -> Interpreted as TRUE Other value -> Undetermined.
Important flags, associated with the operand.	<None>
Important flags, associated with the compiler.	lastASMcode -> Gives information about the ASM code generated by the last BOR or UOR. lastASMaddr -> Complements "lastASMcode", and indicates the position in memory where the last code fragment indicated by "lastASMcode" begins. AccumStatInZ -> Indicates that the value stored in register A is consistent with what the Z flag indicates. This is common.
Implemented operations	:=, =, <>, NOT, AND, OR, XOR

Boolean values are handled in a byte, instead of a bit, because the 6502 lacks bit-access instructions, so it would be very laborious to work with individual bits.

The binary values (\$00 and \$FF) chosen to represent the boolean type are applicable to all operand stores. The value \$FF is chosen ²and not one, for convenience to perform the logical operations on all the bits.

The "lastASMcode" variable is used extensively for the boolean type, and helps with optimizations. It is of type TLastASMcode, defined as follows:

²In previous versions, the value \$02 was used to facilitate operations with bit 1 of the SR register. Having both bits in the same position simplified some operations.


```

TLastASMcode = (
    lacNone, // No special code generated.
    lacLastIsTXA, {Last instruction is TXA, and it's used only
to move result from X to accumulator. }
    //Flags applied to boolean expression results.
    lacCopyZtoA, {Last ASM code is for obtaining boolean expression in regA
from Z.}
    lacInvZtoA, {Last ASM code is for obtaining boolean expression in regA
from Z (inverted).}
    lacCopyCtoA, {Last ASM code if for obtaining, boolean expression in regA,
using the bit C and copied to A. }
    lacInvCtoA, {Last ASM code is for obtaining boolean expression in regA
from C (inverted).}
    lacInvAtoA {Value of regA is inverted in all bits to regA}
);

```

These optimizations, when applied to boolean types, only work with otFuncnt operands (Stores stRegister, stRegistA, stRegistX, or stRegistY). Thus, every time you have a Boolean expression, you must check the status of "lastASMcode", in order to optimize the code.

The "lastASMcode" values of interest for optimization are lacCopyZtoA, lacInvZtoA, lacCopyCtoA, lacInvCtoA, and lacInvAtoA. The idea is that when you want to process a boolean operand of type otFuncnt, you first check the last generated code to see if it can be dispensed with, thus reducing the amount of generated code. Typically, the last generated code, like a "lacCopyZtoA" for example, is just code to move the value of the Z flag to register A, so you can remove this part of the code and generate code with BEQ or BNE directly. This form of optimization can be seen in the routines used to evaluate the conditions of IF statements.

The following table summarizes the above:

FLAG	APPLICABLE TO	PROSECUTION
lastASMcode	Boolean expressions or other types of expressions.	Optional to process.
lastASMaddr	Boolean expressions or other types of expressions.	It is complemented by lastASMcode.
AccumStatInZ	Only to boolean expressions	Optional to process.

6.2 AccumStatInZ Flag

The AccumStatInZ flag provides additional information about the compiled code in a ROB or ROU.

If its value is TRUE (its default value), it indicates which value of the accumulator is reflected in the Z flag of the SR register, that is, for example, if A=0 then Z=1. This is the most common case, because it is assumed that if a value has been set in A, with instructions such as LDA, TXA or PLA, the value of the status register flags is automatically changed.

If the value of `AccumStatInZ` is `FALSE`, it means that the value of the Z flag does not necessarily reflect the value stored in register A. This can happen in very few situations, such as when executing some other instruction (which affects status flags). , after the statement that changes A. The following code illustrates the case:

;ROB example code

...

LDA #1 ;After this instruction, `AccumStatInZ` reflects the value of A

LDX #0 ;After this instruction, no more.

;ROB end

6.3 *Structured Types*

As noted above, structured types are types that don't fall into the `tctAtomic` category. These are:

- Pointers (`tctPointer`).
- Arrays (`tctArray`).
- Objects (`tctObject`).

7 EXPRESSIONS

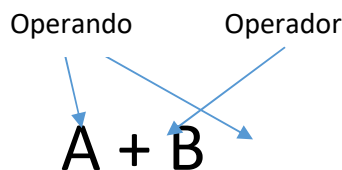
The handling of expressions in this compiler is very different from the classic way they are handled in Pascal. This section explains this particular handling of expressions that is more like modern languages (such as Ruby) that follow the OOP paradigm.

7.1 *Pascal expressions*

Expressions in Pascal are handled as mathematical expressions in Infix notation, as in the following examples:

```
1 + 2
a * b + c
2 - (a mod b)
```

In this notation, operators are clearly distinguished from operands.



In a classic Pascal compiler implementation, code generation focuses on operators, considering the data types of the operands, ie the operators are what determine how the code is generated.

7.2 *Expressions in P65Pas*

The expressions handled by P65Pas although they seem similar to the standard Pascal (and can be used in the same way), internally they are handled more in the style of methods and objects, of the OOP paradigm where the methods are called in the form:

`<object>.<method>`

Thus, each operator (such as +, -, or *) is actually a shorthand way of calling the corresponding method.

The methods associated with the operators are diverse. Some of them are:

Operator +	<code>_add()</code> method
Operator -	<code>_sub()</code> method
Operator *	<code>_mul()</code> method
Operator /	<code>_div()</code> method

MOD operator	<code>_mod()</code> method
--------------	----------------------------

Common operation method names start with “_”, although this is just a rule used for predefined operators, actually any name can be used when implementing a new operator (Not supported yet, in current version).

The following table shows the equivalence between common Pascal expressions and the notation on objects used by this compiler:

Common Pascal expression	Equivalent expression in OOP
variable	variable
$a + b$	<code>a._add(b)</code>
$a + b + c$	<code>a._add(b)._add(c)</code>
$a + b * c$	<code>a._add(b._mul(c))</code>
$a * b + c * d$	<code>a._mul(b)._add(c._mul(d))</code>
<code>to[index]</code>	<code>a._item(index)</code>
a^b	<code>a._ptrto()</code>

OBSERVATIONS:

1. Operators are shorthand ways of calling their respective methods.
2. The methods, associated with operators, are functions that always return values³. They cannot be used as a destination for assignments. They can be considered “getters”.
3. Method parameters are read-only (CONST parameters) and are never modified.
4. The precedence rules of the operators are respected when generating their equivalent versions in OOP.

As operators are nothing more than methods associated with object types (such as byte or char). It is possible to define these operators to new types of objects, but one cannot speak of operator overloading in this language, rather that all operators need to be defined in order to be used.

The creation of new operators is not fully implemented in the current version of the compiler, but their syntax will be very similar to the creation of new methods within classes.

The fact of defining the operators in this way has the following objectives:

- Modernize the compiler by making it object-oriented from the ground up.
- Simplify the implementation of the code generator, so that the syntax is simplified.
- Allow defining custom operators on any data type. This feature is not implemented yet.

³ These values are usually stored in internal registers of the CPU or in temporary variables, as explained in the document “Documentation – Optimization and Synthesis”. Sometimes they can also return constant values, but in both cases the goal is read only.

- Allow a more secure implementation of the code generator, since it is forced to implement all the necessary methods. And in case some method/operator is not implemented, the compiler will simply give the corresponding message.

The equivalent version in OOP is how expressions are understood internally by the compiler. But it's not just an internal representation, it's also a completely valid way of writing the expressions. Therefore there will be no difference between writing: “a+b” and “a._add(b)”. Both will give the same result. The first form can be considered a syntactic sugar to maintain compatibility and make the expression easier to read.

Note that the expression “a + b” is equivalent to “a._add(b)” and that the _add() method has the ability to return a result.

At this level, the way in which the operands will be stored in memory is not discussed, since that corresponds to the job of the code generator.

The assignment operator “:=” (and its associates like +=, -=) is considered as one more method, with low precedence. However, unlike the other operators, this one does not return values (unlike the C language).

As we have seen, expressions are defined as simple method calls. The only condition for a syntactic element to be considered as an expression would be that it returns a value.

For example, the following examples are expressions:

```
123          // numeric literal
'a'          // Literal character
Object1      // Variable object
function1 () // Function call
to. attr ()  // Access an attribute of an object.
a._add ()    // Method call that returns value
```

But the following code does not:

```
procedure1 () // Procedure call
a.method ()   // Call a method that does not return a value
if a = 0 then a = 1 end ; // conditional structure
```

7.3 Operators and precedence

The way an infix expression is translated into OOP form also considers the precedence of the operators. That is why the expression:

a + b * c

It translates as: “a._add(b._mul(c))” since the “*” operator has higher precedence than the “+” operator. Otherwise, it would translate to “a._add(b)._mul(c)”

The precedence defines how the calls to the methods associated with the operators will be generated.

We know that within our compiler language we can (or will) create methods associated with operators. What we cannot create are new operators and there is only a fixed number of them. These are:

OPERATORS	PRECEDENCE
~, NOT, “+” sign, “-” sign, @, **	6
*, /, DIV, MOD, AND, SHL, SHR	5
, !, +, -, OR, XOR	4
=, <>, <, <=, >, >=, IN	3
:=, +=, -=, *=, /=	two

Note that some of these operators (the “~”, the “|” and the “!”) do not belong to the Pascal standard (and are not used by this compiler either), but are available if you want to implement them for some data type.

Some operators are only unary (NOT), and others can work as unary or binary (“+” and “-”).

One way to define a custom operator (not implemented yet) would be:

```
type TPoint = objects
x, y : byte ;
  procedure add ( dest : TPunt ) operator +;
end ;
```

Although operators can be attached to methods, what you cannot do is change the precedence. Operators always maintain their precedence. This means that the “*” symbol will always have higher precedence than the “+”, no matter what method it is associated with.

Initially, in the compiler design, the precedence of an operator was defined to be variable and that it could also be defined, but this generated the following problems:

- If you have overloaded methods, there is a danger of associating them to the same operator with different precedence, as in:

```
type TPoint = objects
x, y : byte ;
  procedure add ( dest : TPunt ) operator + precedence 2;
  procedure add ( dest : TPoint2 ) operator + precedence 3;
end ;
```

Then an additional protection rule (which seems very restrictive) would have to be added, otherwise the parsing of expressions would be complicated.

- Ambiguity could arise when a unary operator like NOT acts on an expression like:

```
not data1 + data2
```

Since you would first have to apply NOT to "data1" to know the precedence of that NOT and then be able to compare it with the precedence of "+" (which depends on the types of "not data1" and "data2"), but it can happen that the precedence of "+" is greater than that of NOT and then the evaluation should be extended to NOT (data1 + data2). To further complicate it, it could happen that the precedence of that NOT is lower than that of the "+" in this new case.

- An additional complication is generated if the operators are allowed to handle different precedences with different data types, because it does not allow a general verification of the expressions but would force to review the precedences of those custom operators.

7.4 Objects and Methods

Así como las expresiones clásicas se componen de operandos y operadores, las expresiones en P65Pas, se componen de objetos y métodos.

We have seen that expressions in P65Pas are made up of objects and methods.

Objects	They do not receive parameters. <ul style="list-style-type: none"> - They can be associated with an object that has a data type Tc. - They return a data type To (basic or structured type).
Methods	They must return a value of a specific type. <ul style="list-style-type: none"> - They are associated with an object that has a data type of Tc. - They receive a data type Ti(parameter). - They return a data type To(result).

The types used in this description are:

Ti -> Represents the type that receives the element.

To -> Represents the type that the element returns.


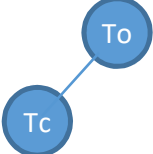
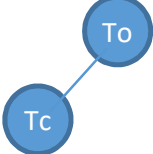
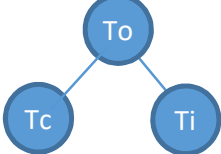
Tc -> Represents the type to which the element is associated.

Note that Ti can be zero or several types, depending on the number of parameters it receives and the operator it emulates:

- Ti nonexistent -> On unary operators, like: "x++" or x._plusplus()
- Ti with single type -> In binary operators, like: "x + y" or x._add(y)

- Ti with two types -> Ternary operators, like “true?1:0” or `_iff(true,1,0)`⁴
- Ti with multiple types -> No application in operators.

To complete our definition of objects and methods, we summarize in the following table:

	OBJECT		METHOD	
	Simple Type or Object	Attribute	Unary Operator	Binary Operator
Example	var_byte	Object.att_byte	a++	a + b
OOP notation	var_byte	Object.att_byte	a._plusplus	a._add(b)
Type Tc	<Does not exist>	The type of "Object"	The type of "a"	The type of "a"
Type Ti	<Does not exist>	<Does not exist>	<Does not exist>	The type of "b"
Type To	The same of the variable.	The type of the attribute	The type returned by <code>_plusplus()</code>	The type returned by <code>_add()</code>
Type Notation	To	Tc.To	Tc.method():To	Tc.method(Ti):To
tree diagram				

It can be identified that the cases "OBJECT-attribute" and "METHOD-Unary Operator", are similar from the point of view of the types. However, we will find decisive differences when analyzed from the point of view of the Code generator.

in the syntax 3.3.

7.5 special operators

There are some special operators that might not be special at first glance, but are also treated as methods of an object.

These are:

OPERATOR	EXAMPLE	OOP VERSION
fix index	<code>to[index]</code> <code>a[index] := value;</code>	<code>a._getitem(index)</code> <code>a._setitem(index, value)</code>
Pointer	<code>a^</code> <code>a^ := value;</code>	<code>a._getptr()</code> <code>a._setptr(value)</code>

⁴ This operator is not implemented in the current version of the compiler.

The index of an array is considered an operator, to standardize and simplify the way the compiler works. In this case, the parameter in square brackets behaves as the second operand.

The methods associated with these operators behave like those described in section 7.2, however, what makes this operator special is that:

- A. Use two separate characters '[' and ']'.
- B. They usually return references to variables ⁵, that is, they can be read/written. Operators like + or - return the result of expressions, which can only be read.
- C. It is used for both reading and writing in the same way. That is to say that it is read with `a[i]` and it is written in `a[i]`.

The C difference is the most problematic, that's why when the operator is converted to a method, two versions are created (getter and setter) depending on whether it is used for reading or writing:

```
_getItem(index)  
_setItem(index, value)
```

Thus, although the expression `a[i]`, can appear in the same way both when it is read and when it is written; the compiler will generate the appropriate call to the method according to its use.

For information on using getter or setter in expressions, see section 7.9.

The case of the Pointer operator is very similar to that of arrays, with the difference that it would be treated as a unary operator.

Consider the definition of the following variable:

```
var ptr : ^byte ;
```

When the expression is used:

```
ptr^
```

You will have to:

- Tc is ^byte
- Ti is not used, and
- To is byte.

The replacement is also done considering a "getter" and a "setter":

```
_getPtr()  
_setPtr(value)
```

⁵ They could also return constant values, when array and index are constants as well.

Again, the use of one or the other method will depend on the use of the expression within a statement.

7.6 *Functions in expressions*

In the definition of expressions, the case of object methods was considered, of the form:

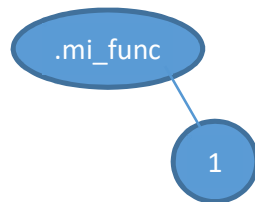
```
object.method(<parameters>)
```

But the case for simple function calls was not discussed as the case:

```
my_function(1)
```

This case is particular because it is not a method associated with an object or class, but rather an independent function.

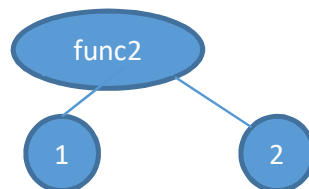
Since the function is not associated with any classes; its corresponding diagram would be:



The same is true for functions defined in units (other than object methods). Thus the function "func2(1, 2)" defined in the "Utils" unit would be defined as:

```
func2(1, 2)
```


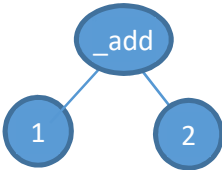
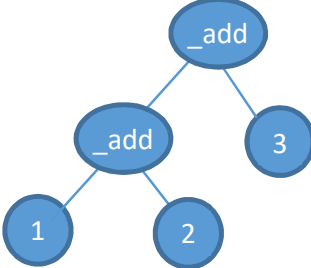
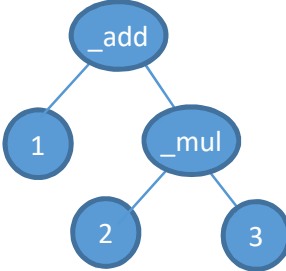
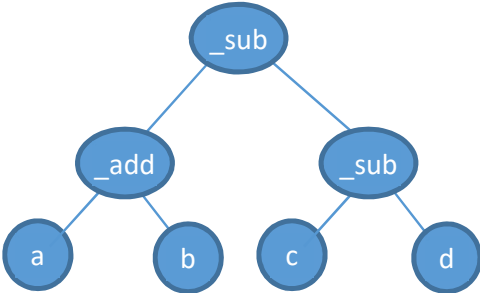
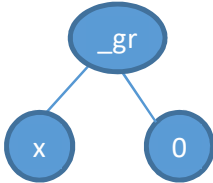
And your expression diagram would be.

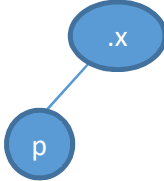
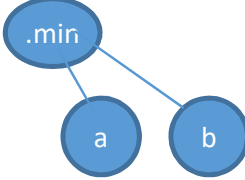
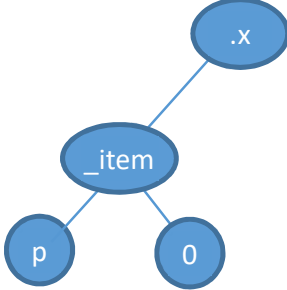


7.7 *tree diagrams*

As seen in the previous sections, it is possible to construct an expression tree diagram for all expressions.

Here is the expression tree for various Pascal expressions:

EXPRESSIONS	TREE OF EXPRESSIONS
1	
1+2	
1 + 2 + 3	
1 + 2 * 3	
(a+b) - (cd)	
x > 0	

px	 <pre> graph TD px((.x)) --- p((p)) </pre>
min(a, b)	 <pre> graph TD min((.min)) --- a((a)) min --- b((b)) </pre>
p [0] .x	 <pre> graph TD x((.x)) --- item((_item)) item --- p((p)) item --- 0((0)) </pre>

Note that all expressions, however complex they may be, always start from a single root node, which can be a:

- Object or simple variable/constant.
- Simple method or function/procedure.

Another detail to highlight is the fact that the only difference between a function/procedure and a method is that the latter receives as its first parameter the object that serves as the basis for the method.

The tree representations shown in the table above are not just graphical representations, but are actually created that way in the syntax tree.

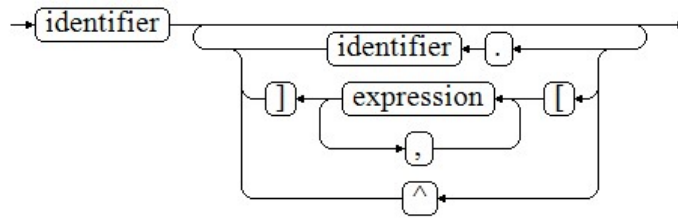
7.8 Operands and Expressions

Syntactically, operands are defined as those expressions that are only joined by the operators “.”, “[]” or “^”. Its notation in BNF would be:

```

operand ::= identifier { "." identifier |
  "[" expression { "," expression } "]" |
  "^" }

```

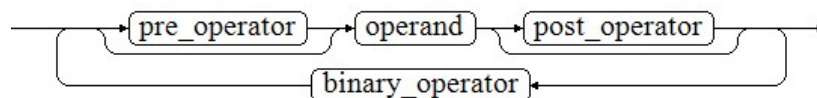


This definition, in standard Pascal, would correspond to what is called a “variable”. However, in our version of Pascal, which follows OOP notation, it could represent a constant or a function (method); that is why we simply call this syntax element “Operand”.

The expressions would be defined as:

```

expression = [pre_operator] operand [post_operator]
             { binary_operator [pre_operator] operand [post_operator] }
  
```



Expressions can be understood as a broader concept than operands, so it encompasses them. This implies that all operands are also expressions, but the converse is not necessarily true.

When studying the processing of expressions, we will see that some operands are also handled as expressions (since the symbols [], ^ are interpreted as if they were operators associated with a method). In this case, a high precedence is assigned to these operators and the concept of operand becomes that of an expression created with operators with precedence greater than a defined one.

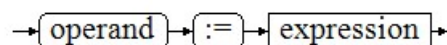
7.9 assignments

As stated above, assignments are special operations within the language, but on a semantic level, they are considered to be just a simple method call.

In the BNF grammar, assignments, within the compiler, are defined as:

```

assignment = operand "[:=" expression
  
```



The term "operand", defined in section 7.8, must return an expression that is writable, that is, a variable. Any other element type will be invalid and will generate a compile-time error.

The conditions that should be met for an assignment to be valid are:

- The operand on the left side must be of type Variable (not to be confused with the data type). This implies that you either have a way to write information to it or you have a setter() method.
- Both the operand and the expression must be of the same data type.

The operand, where the variable is assigned, can be quite complex. Something like:

- variable
- object.attribute
- object.attribute^
- object.attribute^[i]

Simple cases like access to common variables do not require the use of "setters". The write operation is done by the compiler, as simply as copying bytes.

When the operand is special, such as a pointer or array, "setters" are used.

Consider that the "setters" are only used in the final part of the operand to assign. So if the operand is:

```
object.attribute[i].attribute[j] :=
```

The calls to the operand-methods are:

```
object.attribute ._getitem(i) .attribute ._setitem(j)
```

In the expression that goes to the right side of the assignment, getters are always used.

8 FUNCTIONS AND PROCEDURES

We know well that Pascal differentiates between Procedures and Functions, but, for simplicity, when we say “functions”, we will be referring to both functions and procedures.

Considering that P65Pas also has support for objects, the term "function" could also represent a method.

8.1 *Procedures and functions*

Internally, within P65Pas, all procedures are considered functions. The only difference is whether they return any information.

The syntax of the procedures in P65Pas is similar to that used in Modula-2:

```
procedure SoloProc ;
begin
...
    exit ;
...
end ;

procedure Function : byte ;
begin
...
    exit ( value );
...
end ;
```

When you want a procedure to work as a function, you only specify the type that it must return, and you must include the "exit" instruction, with the value to return, passed as a parameter.

When working with the procedure, only as a procedure, the assigned type is typNull, which is a type used to indicate that there is no type. It would be the equivalent of VOID, from the C language.

When the procedure is used, as a function, it is assigned the type indicated in the declaration. This is done in TCompiler.CompileProcHeader(), for user-defined functions.

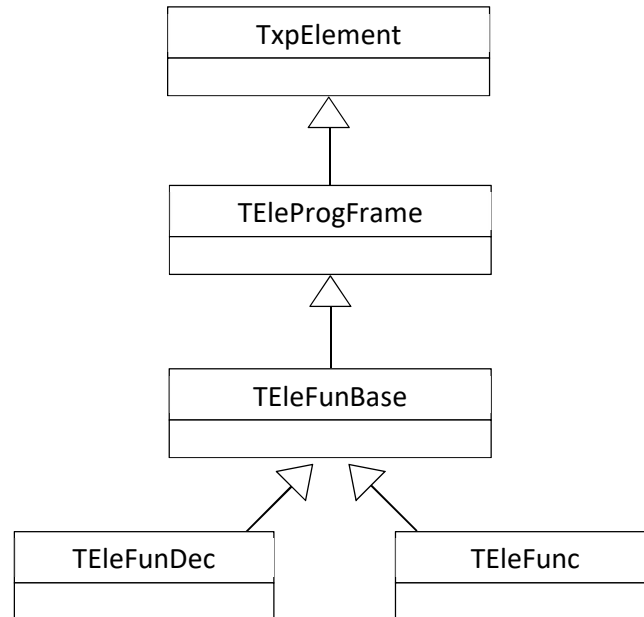
8.2 *Representation of functions*

Inside the compiler, functions are represented as instances of three possible classes:

- TeleFunBase
- TeleFunDec
- TeleFun

All these classes are elements (derived from the `TxpElement` class) specialized to represent both the function declaration and the implementation (in cases where both exist).

The relationship between the above classes is shown in the following class diagram:



The `TEleFunDec` class represents function declarations. Consider two cases:

1. Declarations in the `INTERFACE` section of the units.
2. `FORWARD` statements.

The `TELeFunc` class represents the implementation of the functions. Consider the cases:

1. Implementation in the `IMPLEMENTATION` section of the units.
2. Implementation of `FORWARD` functions.
3. Implementation of functions that do not have a previous declaration.

The `TEleFunDec` and `TELeFunc` classes derive from the common ancestor `TEleFunBase` which is used for polymorphic function references. They are defined as different classes for:

- Being able to differentiate declarations from implementations and thus maintain the position they occupy (declarations and implementations) in the syntax tree, since the correct location of these elements is necessary to properly resolve identifiers.
- Being able to check whether a function declaration includes its implementation, or otherwise raise an error.

However, both classes share a lot of their fields and only `TELeFunc` is specialized, with some fields dedicated to its modeling. The `TEleFunDec` class demonstrates its main identification utility, because it only has one dedicated field:

```
TEleFunDec = class ( TEleFunBase )
public
implement : TEleFun ;    // Reference to implementation item.
public // Initialization
    Create constructor ; override ;
end ;
```

It follows then that the heavy work of the functions is done with the TEleFun class.

References that must point to both declarations and implementations are handled as references to TEleFunBase.

8.3 The declaration/implementation call problem

Having the declaration and implementation of the functions separated generates an additional difficulty, referred to the control of calls or references (See Section 10.6).

The difficulty arises because some function references are resolved with the declaration and others with the implementation (For information on name resolution, see section 10.4), and keeping track of the calls would imply the accumulation of several calls, at the same time. same function, in two different elements.

If you decide to accumulate all the calls in the declaration, the following problems arise:

- Not all functions have a declaration
- It would be going against the philosophy of accumulating all the information in the implementation and leaving the declaration for reference only.

On the other hand, if you decide to accumulate all the calls in the implementation, you would have the following problem:

- Not all statements are going to have an implementation, when referenced, as the implementation could come later.

To remedy this problem, simply add calls to the declaration, when only it exists, and to the implementation, when it exists. When the implementation of a declaration is found, the calls to the implementation are moved and from that moment all the calls in the implementation will be accumulated.

This logic is seen in the TCompilerBase.AddCallerToFrom() function:

```
function TCompilerBase.AddCallerToFrom ( toElem : TxpElement ; callerElem :
TxpElement ) : TxpEleCaller ;
...
begin
    //Creates caller class.
    fc := TxpEleCaller.Create;
    fc.caller := callerElem;
```

```

fc.curPos := GetSrcPos; //Can be changed later if not apply.
if toElem.idClass = eleFunc then begin
    //For implementation of functions, the caller are added directly.
    fun := TEleFun(toElem);
    fun.lstCallers.Add(fc);
end else if toElem.idClass = eleFuncDec then begin
    fundec := TEleFunDec(toElem);
    if fundec.implem = nil then begin
        //Not yet implemented
        fundec.lstCallers.Add(fc);
    end else begin
        //Pass call to the function implementation.
        fundec.implem.lstCallers.Add(fc);
    end;
end else begin
    //Other elements
    toElem.lstCallers.Add(fc);
end;
exit ( fc );
end ;

```

8.4 Functions as object methods.

P65Pas has simple support for object declarations that can also include member functions or methods, as in the following example:

```

type TPoint = objects
x, y : byte ;
    procedure draw ( color : byte );
end ;

```

Methods are represented, within the compiler, as simple function declarations with an additional parameter, similar to how they are handled in Python.

Thus, in the previous case, the draw() method would be saved internally as a function with the following declaration:

```

procedure draw ( p : TPoint ; color : byte );

```

But the call to this function can be done only in the common way:

```

point.draw ( 0 );

```

And not in the form:

```

draw ( point, 0 );

```

Object methods can be declared internally (as is done in TGenCod.CreateSystemElements), such as by code when declaring a variable or object type.

Another characteristic of the methods is that they can be associated with an operator, as is actually done in `TGenCod.CreateSystemElements()`. The use of operators as methods is a feature of expression handling in P65Pas (See section 7).

8.5 User and system functions

Considering the way functions and procedures are declared, they can be classified into two types:

SYSTEM FUNCTIONS	USER FUNCTIONS
They are declared internally in the System unit.	They need to be declared by the user.
They are always accessible from anywhere in the code.	They are accessible, according to the scope rules.

Although system functions are declared internally, in practice both system and user functions will appear similarly in the AST after the parsing process.

8.5.1 System Features

System functions are declared in the “System” unit, so they don't need to be declared. Some of these functions are `inc()`, `word()`, `chr()`, etc.

System functions are accessed through an identifier or an operator, in the source code, as if they were any function.

They are created within `TGenCod.CreateSystemElements()` as methods for predefined types (such as `Byte._add()` or `Byte.sub()`) or as stand-alone functions (such as `Inc()` or `Dec()`).

As an example, let's see the creation of the system function `Inc()`, which increments the value of a variable.

```
// Create system function "inc"
setlength ( pars, 0 ); // Reset parameters
AddParam ( pars, 'n' , srcPosNull, typNull, decNone ); // NULL, allows any type.
f := CreateSystemFunction ( 'inc' , typNull, srcPosNull, pars, @fun_Inc );
AddCallerToFromBody ( H, f.BodyNode );
```

The definition of parameters will be used to validate when performing the semantic analysis and fill the syntax tree. Here only one parameter is created, named “n”.

`AddParam()` defines the parameter to use, including its name and type. The type used in this example, `typNull`, indicates that this function can receive any data type in the parameter. This means that the semantic analyzer will allow any valid expression as a parameter (it would normally do a pre-validation). It will be up to the `fun_Inc()` routine to determine any errors that may be generated from this debauchery.

The `fun_Inc()` routine will be used for code generation when this phase is reached, according to predefined rules. For more information on code generation in system functions, review the document “Documentation – Optimization and Synthesis”.

8.5.2 User Roles

The programmer can create his own procedures or functions, using the typical Pascal syntax:

```
procedure MyProc ;
begin
    // do something
end ;
```

The routine in charge of processing these declarations is `TAnalyzer.AnalyzeProcDeclar()`, which is quite an elaborate routine and is supported by other specialized routines.

`AnalyzeProcDeclar()`'s mission is to recognize all procedure declaration syntax, detect errors, and fill the syntax tree with information about the procedure.

All functions are parsed and semantically parsed and included in the syntax tree, but compilation to binary depends on whether or not the function is used within the program.

For more information on code generation in user functions, review the document “Documentation – Optimization and Synthesis”.

8.6 Implementation of functions for operators

As noted in the Expressions in P65Pas, common operators like `+` or `-` are just syntactic sugar for calls to methods defined on various data types, like `byte` or `word`.

La definición de estos métodos se hace en `TGenCod.CreateSystemElements()`:

```
procedure TGenCod.CreateSystemElements ;
..
    //////////// Byte type ////////////
    //Methods-Operators
    TreeElems.OpenElement (typByte) ;
    CreateBOMethod (typByte,      ':=',      2,      '_set',      typByte,      typNull,
@ROB_byte_asig_byte) ;
    CreateBOMethod (typByte,      '+=',      2,      '_aadd', typByte,      typNull,
@ROB_byte_aadd_byte) ;
    CreateBOMethod (typByte,      '-=',     2,      '_asub', typByte,      typNull,
@ROB_byte_asub_byte) ;
```

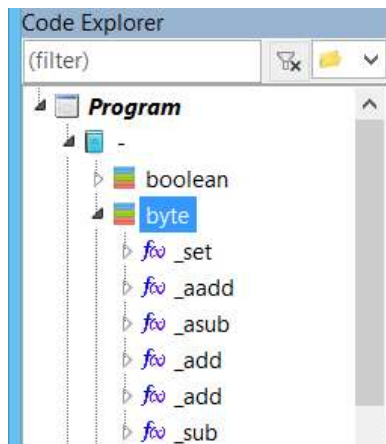
```

    CreateBOMethod(typByte, '+', 4, '_add', typByte, typByte,
@ROB_byte_add_byte);
    CreateBOMethod(typByte, '+', 4, '_add', typWord, typWord,
@ROB_byte_add_word);
    CreateBOMethod(typByte, '-', 4, '_sub', typByte, typByte,
@ROB_byte_sub_byte);
...

```

The creation of these methods is entrusted to the `CreateBOMethod()` routine, which allows you to create a new function, associated with a type, and associate it with an operator, with a given precedence. Internally it calls `AddFunctionUNI()` like when creating a user-defined function. The code was designed in this way to be able to implement, in the future, the creation of custom operators.

All of these operator-methods are created inside the System unit, so they could be viewed with the IDE's Code Explorer:



Until the current version of the compiler, all method-operators of the System unit are defined as `INLINE` functions, but that could change in the future.

`INLINE` functions are used for cases in which the code necessary to implement the operation is reduced. For example, the code for the implementation of Byte addition and subtraction is so small that using a subroutine for this would imply spending more memory on the calling code, and the `JSR` and `RTS` instructions. Not to mention that you would lose clock cycles and a level in the call stack.

In general, implemented subroutines are expected to:

1. Be as small and efficient as possible. If two space- and speed-optimized versions can be handled, that would be better, although only one version is used at the moment.
2. Use as few records as possible. They should preferably not use auxiliary variables, but only the available working records. But if there is a need for more registers, working registers should be used rather than auxiliary variables.
3. That they can integrate the largest number of operations in a single code. For example, if a routine that does division and remainder is implemented in a single code, it will help in optimization.

4. That return values respecting the standard of returned values. For example, if the routines are going to return a Word value, they should return their result, directly, in the H and W registers (avoiding having to add value copy instructions).
5. The input parameters are also work records: A mandatory, and H, or E or U, in that order of priority.

10 SYNTAX TREE

10.1 fundamentals

P65Pas, like most compilers, uses an internal data structure, called a syntax tree, to support compilation ⁶and code generation.

This syntax tree should be considered an AST or abstract syntax tree, because it does not strictly represent the source code, but rather a more digested version that facilitates the code generation task.

As in classic compiler design, the syntax tree sits between the compiler and the code generator:



This container, effectively, has the structure of a tree and allows representing the various syntactic elements (variables, constants, procedures, types, statements, ...) of the syntax of the programming language used.

The syntax tree functions are:

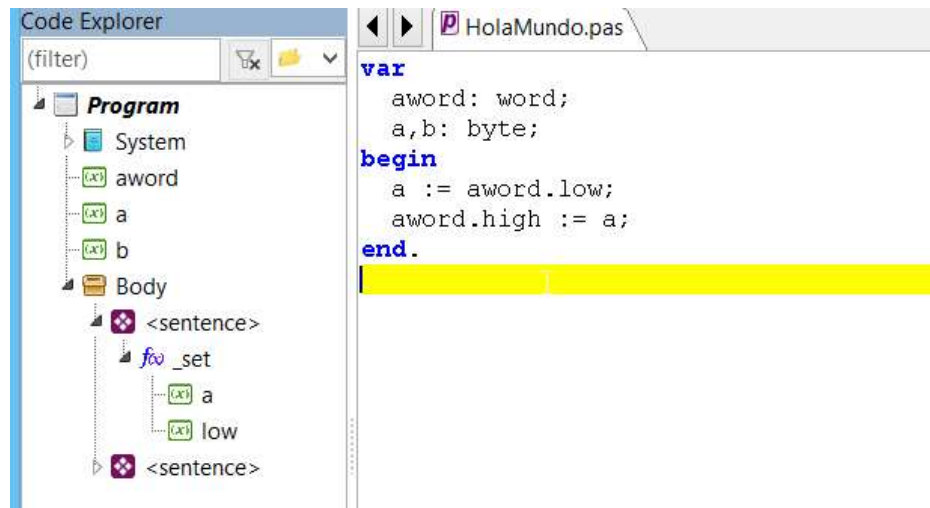
1. Serve as the main container for the syntactic elements of the program.
2. Check the duplicity of names, in the declaration blocks.
3. Resolve names, in code blocks. Model namespaces.
4. Serve as a source for the code generation process. This better separates the parts of the compiler and makes code optimization easier.

As the compiler explores the source code, it builds the syntax tree or element tree. The element tree is an object of the `TXpTreeElements` class, which is defined in the “XpresAST” unit. It consists of a root node, which represents the main program.

The element tree is not only a structure to store the various elements of the program, but also serves as a representation of the namespaces or scopes of all the source code, so that it serves to resolve names (See section 10.4).

We have a graphic representation of the syntax tree in the P65Pas IDE, in the panel named “Code Explorer”:

⁶ In addition to the main syntax tree, P65Pas implements an additional syntax tree for directive processing.

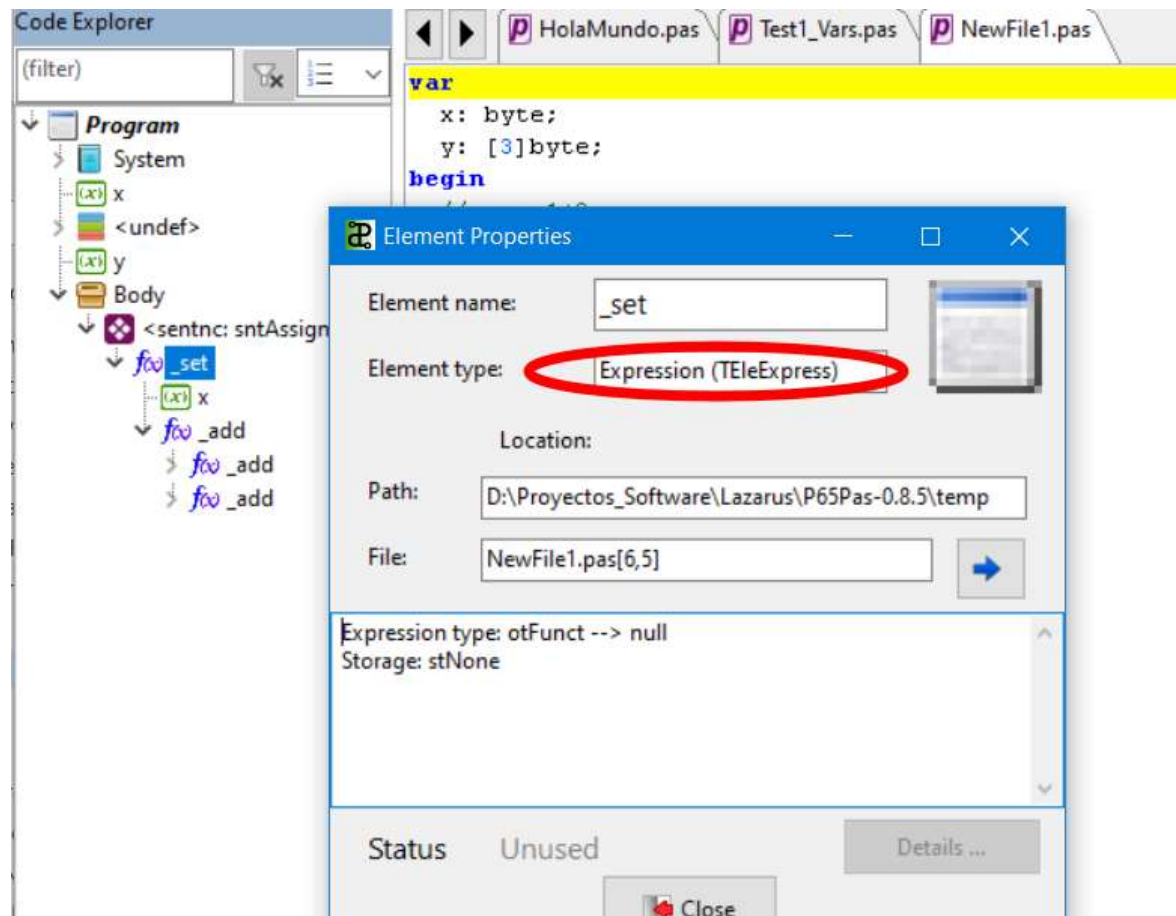


Functions and system variables are also created within the syntax tree, as are statements or expressions.

The syntax tree stores all the elements of the syntax, which are necessary for code generation. Comments, or white space, have no representation within the syntax tree.

The elements that constitute the nodes of the tree are objects of the “TxpElement” class (See section 3) and therefore can represent all the syntactic elements of the language.

To see the type of any tree element, simply access the tree properties window, using the context menu:



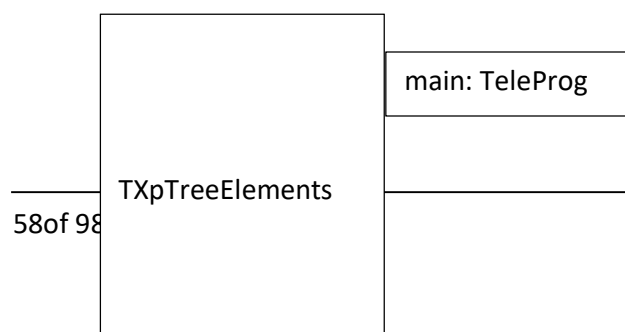
In the figure above, the selected element “_set” is of type TEleExpress, which is normal for elements found within an assignment statement.

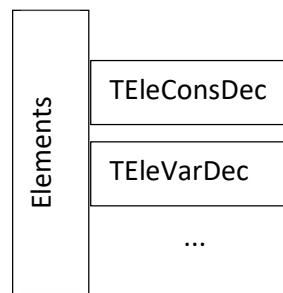
The properties window also shows additional information, according to the type of element selected.

10.2 Structure

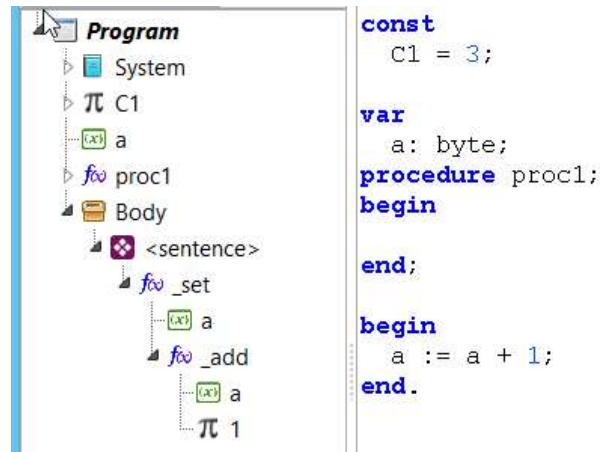
The syntax tree is built from scratch in the code, without using any additional libraries to the standard libraries.

The syntax tree contains TxpElement type nodes, but the tree as a structure is defined with the TXpTreeElements class, which is the main container, and which contains the "main" node, from where the other nodes begin to branch, taking advantage of the fact that the objects TxpElement, include an “elements” list field, which can contain other TxpElement objects.





The following figure shows the AST representation of a simple code:



There are certain nodes that contain other nodes. When a node contains no other nodes, we say that it is a terminal node.

Each element of the syntax tree is also called a "Node", as an analogy to the graphic form of a tree.

The syntax tree follows some general rules. In principle an AST can be generated for a program or for a unit.

1. An AST may contain, at the main level, zero or many type elements: units, constants, variables, types, or procedures.
2. An AST must contain, at the main level, only one element of type "Body" that must always be at the end.
3. Unit elements should always appear at the top of the list, at the top level ⁷.
4. Constant elements, variables, types and procedures can appear multiple times and in no particular order, as long as they respect rules 2 and 3.

The procedures follow similar rules:

1. **A procedure** may contain, at the main level, zero or many type elements: constants, variables, or types.

⁷ This rule could change if the "PROGRAM ..." statement is defined to generate an element that appears in the AST.

2. A **procedure** must contain, at the main level, only one element of type "Body" that must always be at the end.
3. Constant elements, variables, types and procedures can appear multiple times and in no particular order, as long as they respect rule 2.

These rules are primarily determined by the Pascal language definition and may vary in implementations for other languages.

The syntax tree definition is recursive, with respect to procedures, because they could contain other procedures as well, but in the current version of the compiler, nesting of procedures is not allowed.

The syntax tree is an ordered structure, it has a first and last element.

10.3 duplication of names

Element identifiers must be unique within their scope so that the compiler can accurately identify them when they are referenced within code.

When we say “scope”, we mean one level of the syntax tree. For example, a local variable cannot have the same name as another local variable, in the same node as the first.

However, it is completely legal to have a local variable with the same name as a global variable.

The duplication of names must be avoided, regardless of whether they are elements of different types. There cannot be a unit with the same name as a constant.

To prevent name duplication, the `TxpElement.DuplicateIn()` method has been implemented, which performs a linear search in a list of elements.

The duplication of names is always done in the declaration sections:

```
program aaa ; // It is not necessary to see duplicity, it is the first
identifier.
use
drive1, drive2 ; // Duplicity check.
build
a = 1 ; // Duplicity check.
var
x, y : word ; // Duplicity check.

procedure proc ( par1 : bit ); // Duplicity check, except the first.
var
  p1, p2: bit ; // Duplicity check.
begin
p1 := 1;
p2 := par1;
```

```
end ;  
  
var  
x1, x2 : byte ; // Duplicity check.  
  
begin  
    // Program body  
end .
```

Within the code section (Body), there is no duplication check, but name resolution.

Duplication for constants or variables is simple, but for functions, it is different because there can be multiple functions with the same name, in the case of function overloading.

10.4 name resolution

Name resolution is understood as the process that makes it possible to precisely identify which element an identifier refers to in the source code.

Thus, for example, if the instruction appears in a source code:

```
x := 1 ;
```

The compiler must know if "x" refers to a variable, constant, type or function (and thus check the syntax). The simple name "x" can be referring to a local variable, global or one defined in any of the units used.

Within the Pascal philosophy, all names are resolved at compile time. That is, the compiler must accurately resolve all identifiers in order to generate the executable.

Name resolution has two parts:

- Exploration.
- The comparison.

Scanning is done as indicated in section 10.4.3.

10.4.1 FindFirst() and FindNext()

Within the code, there are several methods to resolve names, but the main ones are two:

```
function TXpTreeElements.FindFirst ( const name : string ) : TxpElement ;  
  
function TXpTreeElements.FindNext : TxpElement ;
```

These methods are used separately, to optimize the identifier search process, when overloaded functions (or procedures) are handled, since it is common that before identifying the correct function, other versions of the same function are found.

Having the search functions, separately, allows to continue a search at the same point where it was left off, so the exploration will not be done in the whole tree, but only until the correct version of the function is found, except, of course is, when the function is not found.

To save or restore the state of a search, the field "TxpTreeElements.curFind" of type TxpFindState must be used. An example of using "curFind" to save the state of a find is found in TCompilerBase.GetOperand().

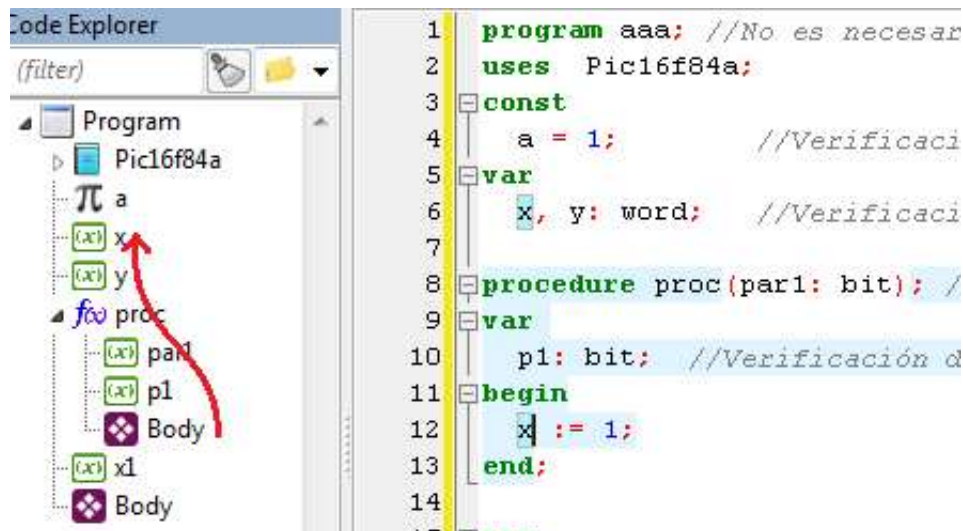
The "curFind" field allows nested searches, using FindFirst() and FindNext().

One of the difficulties in the implementation of FindFirst() and FindNext() is that they must work with the AST filling up, this happens because the creation of the AST is done with all the nodes resolved.

FindNext()'s search algorithm goes something like this:

1. The current search node "curFind.Node" and the current search index "curFind.Idx" are defined.
2. The search is started in "curFind.Node.elements" from position "curFind.Idx-1", backwards.
3. If the searched element is not found within "curFind.Node.elements", it jumps to the parent node of "curFind.Node".
4. The search is repeated in the same way on that parent node.
5. If the parent node is the main program ("Main" node), the drives are searched.

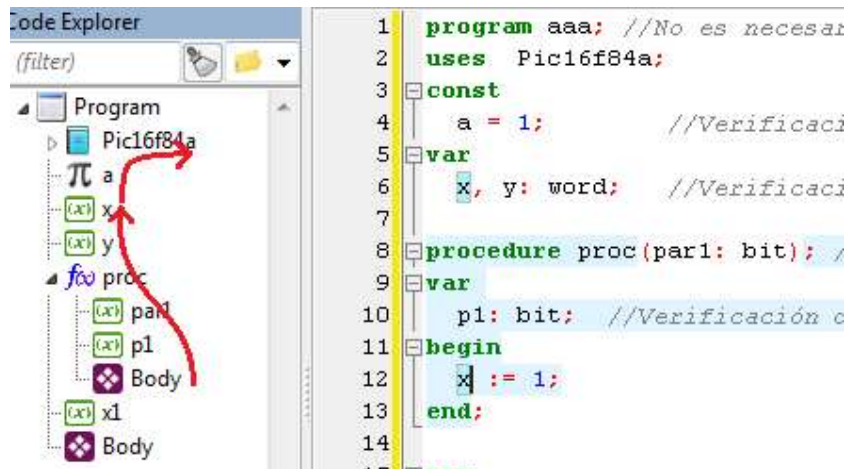
The following tree shows how a name resolution would be done through the AST:



Note that this process is supported by the fact that the elements of the syntax tree are ordered.

Normally, the exploration of names is descending in the level of the tree, thus avoiding considering the local variables of other procedures.

But when it is required to search for the name, within units, an exception is thrown, in the normal flow of exploration:



In this case, you must level up to access the items in the unit, which functions as its own container. The search here is only done at the first level of nesting, and only for elements in the INTERFACE section.

Name resolution is done whenever the compiler needs to execute or evaluate some code. This happens only in the Analysis process (See section 1.2 to see the compiler structure), where the AST is created that contains valid references to the affected syntax elements.

10.4.2 Identifier Comparison

Identifier scanning is performed until a name matching the searched name is found. However functions are an exception.

The following table shows how the comparison is made:

ELEMENT	WAY TO SOLVE
Constants	Identifier match.
variables	Identifier Match
Types	Identifier match.
Procedures	Match identifier and type of parameters.

The need to check parameters in procedures complicates the process, but is necessary to support procedure overload. Without this case, only a simple search function would suffice.

A single, simpler function could be handled that would scan the entire syntax tree, to find the different versions of the function (including the units), and then compare each of them; but this would have to be done, for each function, and it would be too heavy a process in terms of CPU.

10.4.3 Identifier scanning

Identifier scanning refers to the way the AST is scanned to find the elements it refers to.

This exploration is usually bottom-up, leveling up through the AST. That is, from the most nested elements to the most external.

The exploration of identifiers, to resolve them, is related to the Calls or References, described in section 10.6, since resolving an identifier will imply generating a call to the resolved element.

In accordance with the cases in which the calls are generated, for the exploration of identifiers, the following cases are considered:

- A. Calls from the program body/functions (TEleBody).
- B. Calls from declarations (constants, variables and types).

For more information on how these calls are generated, go to section 10.6.

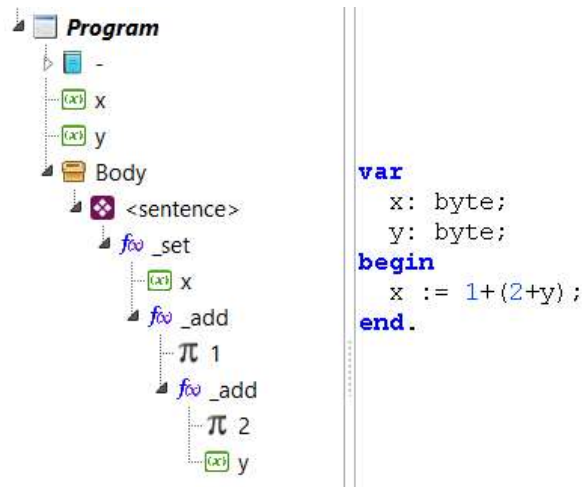
10.4.4 Calls from TeleBody

Most calls occur from the body of the main program or functions (TEleBody).

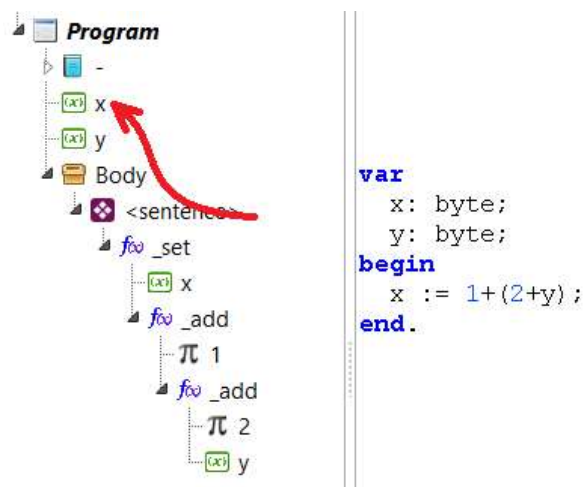
Let's consider the following code:

```
var
  x : byte ;
y : byte ;
begin
  x := 1 +( 2 + y );
end .
```

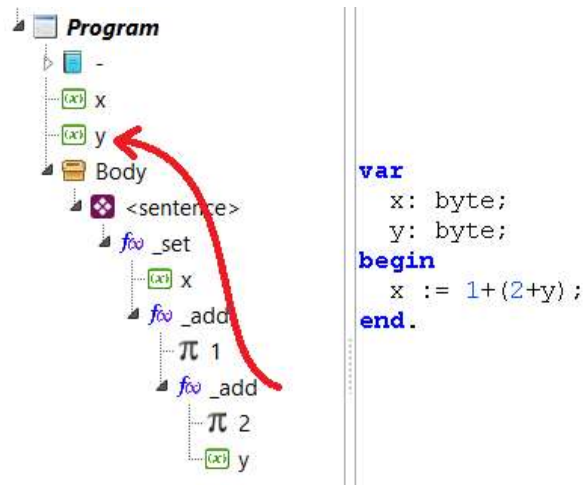
The AST will have the form:



The resolution of "x" (in the analysis), is performed while the current node is the sentence "<sentence>", if the current node is taken as the starting point of the exploration, the exploration flow would be:



Which is not bad. But let us now consider the resolution of "y". This occurs when the current node is the innermost "_add" function. In this case the resolution, from the current node, would be as in the following figure:



As such a search would be inefficient, since the declarations are above the "Body", a search form has been designed that considers the current code container or "Body".

The current code container is stored in "TXpTreeElements.curCodCont" and is kept up to date when a TEleBody element is opened or closed.

10.4.5 Calls from the Statements

Calls from declarations are resolved in the same way as calls from TEleBody, from the point of the declaration, backwards.

The TXpTreeElements.FindFirst() function is prepared to consider this case as well. Because it first checks whether it is inside an "eleBody" node or not:

```
function TXpTreeElements.FindFirst ( const name : string ): TxpElement ;
begin
  curFind. Name := UpCase ( name );
  curFind.inUnit := false ;
  if curCodCont=nil then exit(nil);
  if curCodCont.idClass = eleBody then begin
    ...
    Result := FindNext;
  end else begin
    ...
    Result := FindNext;
  end;
end;
```

10.5 Llenado del árbol de sintaxis

The filling process is done in the Analysis process (See section 1.2), which is executed with `TAnalyzer.DoAnalyzeProgram()` or `TAnalyzer.DoAnalyzeUnit()`.

In this source code scanning process, all accessible elements of the main program, and all units used by the program (including units also used by these units) are included in the syntax tree.

The elements of the syntax tree are not added directly to the lists of the container, but there is a group of special methods to perform the element input. These are:

```
TXpTreeElements = class
...
    public // Functions to fill the tree
        function AddElement(elem: TxpElement; verifDuplic: boolean=true): boolean;
        procedure AddElementAndOpen(elem: TxpElement);
        procedure AddElementToParent(elem: TxpElement; AtBegin: boolean);
        procedure OpenElement ( elem : TxpElement );
        procedure CloseElement ;
...

```

Elements are added sequentially with the option to use the same elements as new element containers.

10.5.1 Considerations

One of the characteristics of the filling routines is that they are oriented to work in the following way:

- Select worker node with `OpenElement()`.
- Do something with the current node.

In other words, it is not usual to work directly with any node, but rather the selection must be made first, at least for tasks such as adding new nodes.

For example, the following code shows how to add a new node to the syntax tree:

```
TreeElems.openElement ( curSent ); // Select worker node
TreeElems.AddElement ( newNode ); // Add a new node to the curSent node.

```

Then, if we wanted to add a new node to “newNode”, we would have to open it first with `openElement()`.

Another important consideration is that for a node to accept new nodes, it must have its list of child elements initialized (Attribute: `TxpElement.elements`).

This is typically done when the element is first added to the tree, using `AddElementAndOpen()`, rather than just using `AddElement()`. Or put another way: If we know that

a node, which we are going to add to the syntax tree, is going to have child nodes, we must add it with `AddElementAndOpen()`⁸.

This somewhat strange feature, which initially complicates the handling of nodes, has remained so since the first versions of the compiler, because it allows you to avoid creating the list of child nodes (`TxpElement.elements`) unnecessarily, improving code performance.

10.6 Calls or Referrals

In the syntax tree, not only are syntactic elements stored in the structure, but also the calling relationships between these elements, such as calls or references made to an element (such as a variable), from another element (such as the body of a procedure) in the source code.

Calls between elements are identified using the `TxpElement.lstCallers` field which is inherited to all element types:

```
TxpElement = class
...
public
  //List of functions that calls to this element.
  lstCallers : TxpListCallers ;
...
end ;
```

This field is a list of all the "callers" of this element. A "caller" element of a base element is another element that calls the base element. A "caller" can only make calls from these points:

- The body of a function or the main program (TEleBody element).
- Declaration of a constant (TEleConsDec element).
- Declaration of a variable (TEleVarDec element).
- Declaration of a type (TEleTypeDec element).

The following diagram shows the points from which calls to other elements are expected.

```
build
<from here>
var
<from here>
type
<from here>
begin
<from here>
end ;
```

Calls from these points are all optional, as explained below.

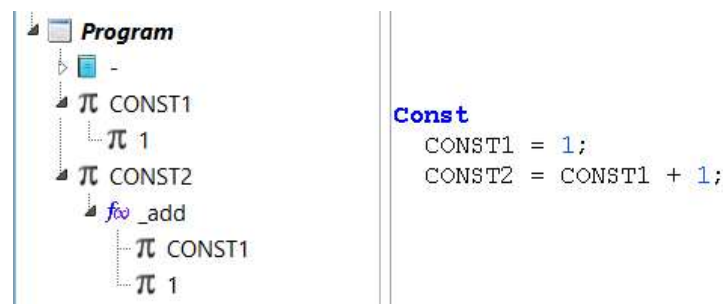
⁸ Of course, it can also be simply added using `AddElement()` with the caveat of executing the statement afterwards:
`added_node.elements := TxpElements.Create(true);`
to create its list of child elements.

10.6.1 Calls from TEleConsDec

The definition of constants can include complex expressions, such as those used in a TEleBody, with the caveat that they must always return constant values:

```
Const
CONST1 = 1 ;           // Declaration of a constant
CONST2 = CONST1 + 1 ; // Reference to another constant
```

The expressions included in the constant definition can be viewed from the code explorer:



The way they appear in the syntax tree is similar to how expressions appear in common statements.

10.6.2 Calls from TEleVarDec

The variable declaration, in some cases, can also include references to other elements:

```
var
x1 : absolute byte CONST1 ; // Reference to a constant
x2 : byte absolute PORTB ; // Reference to another variable
x3 : MyType ;                // Reference to a type
```

References can be to constants, variables, or types.

Reference to other elements, from the variable declaration, is a special case, because most variable declarations will not include references.

10.6.3 Calls from TEleTypeDec

The type declaration can also include references to other types:

```
type Type1 = record
    x : old_type ; // Reference to another type
end ;
type Type2 = other_type ; // Reference to another type
```

The TEleTypeDec element references are used to determine which types are used and which are not.

Initially, in the design, it was not considered to keep track of these references, but it is included for the following reasons:

REASON 1: Declarations like “type t= ARRAY[CONST1] OF BYTE;” they did not update the expression “CONST1”, if the type was not used; which makes sense, given the current way the compiler works, where arrays are handled as classes. Of course the problem can be solved (without too much trouble) by assuming that all types are used, but this leads to REASON 2.

REASON 2: It is expected that, in the future, type declaration may involve the generation of additional code. If so, it is convenient for optimization, to determine the types that are used.

REASON 3: For the sake of order and uniformity, the compiler is expected to keep track of all used element types.

10.6.4 Calls from TeleBody

These calls are the most common. They happen when variables, constants, or functions are used from within the body of the main program or a procedure:

```
var x : byte ;
begin
x := 1 ; // Here add a call to "x"
func1 ; // Here add a call to "func1"
x := 0 ; // Here add another call to "x"
end ;
```

Strictly speaking, calls occur from specific points in a statement; however, only the reference to the TEleBody body is saved, which is sufficient precision for our purposes, which is basically to determine the variables used from within a procedure.

10.6.5 Call Implementation

Calls between syntax elements have two parts:

1. The element called (called)
2. The caller element

Un elemento “caller” se modela con un registro TxpEleCaller, que está definido como:

```
TxpEleCaller = class
  curPos: TSrcPos; //Position from where it is called this element.
  caller: TxpElement; //Element that calls this element.
  function CallerUnit : TxpElement ; // Unit/Program from where it is called
  this element.
```

```
end ;
TxpListCallers = specialize TFPGObjectList < TxpEleCaller >;
```

The “caller” field is the reference to the caller object and could theoretically only point to TEleBody, TEleConsDec, TEleVarDec or TEleTypeDec.

The information of the calls between elements is necessary, to be able to:

- Detect unused elements, so that code can be optimized.
- Implement the procedure parameter reuse mechanism, because to know which parameters can be reused, you need to have the call tree.
- Determine the nesting level of calls between procedures ⁹.

To register a call to an element, one of the following functions must be used:

```
TCompilerBase = class ( TCompOperands )
...
public // Caller methods
    function AddCallerToFrom ( toElem : TxpElement ; callerElem : TxpElement ) :
TxpEleCaller ;
    function AddCallerToFromCurr ( toElem : TxpElement ) : TxpEleCaller ;
...

```

AddCallerToFrom() is the most general function and the idea is that it is only used by AddCallerToFromCurr(), not directly; this is to limit the types of items that can generate calls, because AddCallerToFrom() allows you to log calls from any type of item.

AddCallerToFromCurr() would be the preferred routine for logging calls. It is executed as the source code is scanned, in the AST population, and includes all calls, from all functions or units, regardless of whether these functions or units are used later.

AddCallerToFromCurr() generates a call to an element, from the current point, which must be a code container (See section 10.6.6).

Knowing these dependencies helps to implement the optimizations in the synthesis process.

To determine how many times an element is called, count the number of elements in lstCallers, using the nCalled method:

```
function TxpElement.nCalled : integer ;
begin
Result := lstCallers.Count ;
end ;
```

⁹ Not currently implemented.

10.6.6 code containers

We said that element calls can only occur from TElBody, TElConsDec, TElVarDec or TElTypeDec elements.

These elements are classified as "Code Containers", because they are elements that can contain expressions; and it is these expressions that can ultimately make references to other elements.

Unlike bodies (TElBody), declarations of constants, variables, and types can only contain limited expressions, as shown in the previous sections.

The elements, which are Code Containers, are identified because they derive from the TxpEleCodeCont class, which is simply an empty class, but serves as a common ancestor to all elements that can generate calls to other elements:

```
TxpEleCodeCont = class ( TxpElement )
end ;

TElTypeDec = class ( TxpEleCodeCont )
...
end ;

TElConsDec = class ( TxpEleCodeCont )
...
end ;

TElVarDec = class ( TxpEleCodeCont )
...
end ;

TElBody = class ( TxpEleCodeCont )
...
end ;
```

Code containers open and close as various parts of the source code are explored. The routines in charge of controlling the opening of code containers are:

```
TXpTreeElements.AddElementBodyAndOpen ()
TXpTreeElements.AddElementConsAndOpen ()
TXpTreeElements.AddElementVarAndOpen ()
TXpTreeElements.AddElementTypeAndOpen ()
```

And there are four of them because each one is reserved for a type of code container. These routines are similar to AddElementAndOpen(), but these routines create the code container and update "curCodCont" which is the variable that keeps track of the current code container, so that when they point to an element, it is because it is open. that code container; otherwise they will point to NIL.

To properly control code containers, these functions should be used each time a body, variable or constant declaration, or type declaration is processed, rather than using `TXpTreeElements.AddElement()`.

The current code container is independent of the nesting level we are in when exploring the AST. Thus, for example, when the exploration of a `TEleBody` starts, the container is opened with `AddElementBodyAndOpen()` that updates “`curCodCont`” to the open `TEleBody`, and then it is possible to advance in the exploration of deeper levels of the expressions, while “`curCodCont`” will still point to the initial body.

Only “`curCodCont`” will be cleared when the `TEleBody` is closed with `TXpTreeElements.CloseElement()`. Making use of `TXpTreeElements.CloseElement()` will have no effect on “`curCodCont`”, when an element other than a code container is closed.

```

procedure TXpTreeElements.CloseElement ;
{Close the current node and returns to the parent node.}
var
    isCodeContainer: Boolean;
begin
    isCodeContainer := (curNode=curCodCont);
    if curNode.Parent<>nil then begin
        curNode := curNode.Parent;
    end;
    if isCodeContainer then begin //We are closing a Code container
curCodCont := curNode.codCont ; // Restore last value
    end ;
end ;

```

Only one variable (“`curCodCont`”) is used to control code containers, but nesting of code containers is allowed.

10.6.7 “Callers” and “called”.

Note that the elements save the list of the elements that call them or “callers”, but they do not save the list of elements that they call or “called”.

This information exists and is stored in two lists, of the `TxpElement` class:

```

lstCalled : TxpListCalled ; // Elements called directly
lstCalledAll : TxpListCalled ; // All elements called

```

But unlike the “`lstCallers`” list, these are not populated at compile time, but must be calculated on demand when this information is required, which is after Code Analysis.

The content of this list is taken entirely from the “`lstCallers`” list which is all that is needed. Updating the “`lstCalled*`” lists is done in `TCompilerBase.UpdateFunLstCalled()`:

```

procedure TCompilerBase.UpdateFunLstCalled ;
var

```

```

fun : TElFun ;
itCall : TxpEleCaller ;
whoCalls : TElBody ;
n : Integer ;
begin
  for fun in TreeElems.AllFuncs do begin
    if fun.nCalled = 0 then continue ; // not used
    // Process calls made from other functions, to fill
    // your "lstCalled" list, so you know who you call
    for itCall in fun.lstCallers do begin
      // Add the reference of the function call
      if itCall.caller.idClass <> eleBody then begin
        // By design, itCall.caller should be TElBody
        GenError ( 'Design error.' );
        exit ;
      end ;
    end ;
    whoCalls := TElBody ( itCall.caller );
    // All the calls are added ( even if they are to the same procedure ) but
    // then AddCalled () , filter them out.
    whoCalls.Parent.AddCalled ( fun ); // Add to procedure
  end ;
end ;

{Update the list fun.lstCalledAll with all calls to all
 las funciones, sean de forma directa o indirectamente.}
for fun in TreeElems.AllFuncs do begin
  n := fun.UpdateCalledAll;
  if n<0 then begin
    GenError('Recursive call or circular recursion in %s', [fun.name],
fun.srcDec);
  end;
end;
if HayError then exit;
//Actualiza el programa principal
TreeElems.main.UpdateCalledAll; //No debería dar error de recursividad,
porque ya se verificaron las funciones
if TreeElems.main.maxNesting>128 then begin
  {Stack is 256 bytes size, and it could contain 128 max. JSR calls, without
  considering stack instructions.}
  GenError('Not enough stack.');
```

This function is called in TCompiler_PIC16.DoOptimize() because this update needs to be done for some optimization tasks and before code generation.

UpdateFunLstCalled() fulfills two additional tasks:

- Detect cases of recursion or circular recursion.
- Detect possible cases of stack overflow due to excessive nesting.

10.6.8 named elements

Although not specified directly, there is only one set of elements that can be called. These are:

1. Functions
2. Constant Declaration
3. Declaration of variables
4. Type Declaration

Units can also be referenced, but that functionality has not been implemented yet.

A particular case of references, for purposes other than the simple determination of use, are the references that are created from certain assembly instructions to the assembly labels of the same block.

10.7 program frames

Program frames are special syntax elements that are also represented as nodes within the syntax tree:

Within the compiler, program frames are modeled as syntax elements that inherit from the `TEleProgFrame` class.

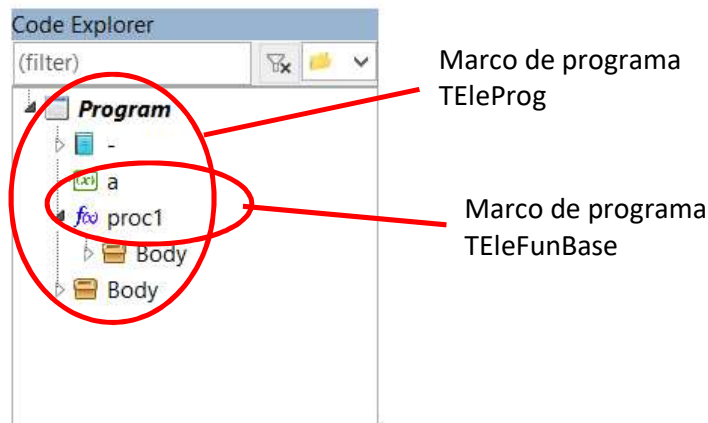
```
TEleProgFrame = class ( TxpElement )
public
  function BodyNode : TEleBody ;
  public // Handling calls to exit()
firstObligExit : TEleSentence ; {Reference to the first exit(), in mandatory
code}
  procedure RegisterExitCall ( exitSent : TEleSentence );
  public // Initialization
  procedure Clear ; override ;
  Create constructor ; override ;
  destroyer Destroy ; override ;
end ;
```

The main characteristic of a program framework is that it has a body or `TEleBody` element, which is what actually contains the statements. Additionally, a program framework may include various sections of declarations of variables, constants, types, or functions/procedures.

In the current version of the compiler, the following elements are defined as program frames:

- `TEleProg` -> Main program.
- `TEleFunBase` -> Function or procedure.
- `TeleUnit` -> Unit.

The code explorer of the IDE can help us to better visualize the program frames:



Units are also considered as program frames, because they include declarations and also have a body, which would be the `INITIALIZATION` section. The `FINALIZATION` section can be thought of as another body, but this would be a special case.

The body of a program frame, the `TEleBody` class element, is the one that ultimately contains the program statements.

10.8 AST end state

The AST serves as a workspace for various processes. Most of them are produced in the process of optimization and synthesis; and are described in the “Documentation – Optimization and Synthesis” document, but in the Analysis process (described in this documentation) the syntax tree ends up fulfilling the following rules:

1. The AST contains all the information necessary to perform the optimization and synthesis phases. That is, once filled in Analysis, no other source code information is needed to generate code.
2. The order of the AST elements reflects the order in which the source code is scanned.
3. All the AST nodes have their names already resolved. That is, all referenced names have already been searched for and found in the source code (and exist in the AST).
4. All the nodes of the AST are already differentiated as elements. I mean, you already know who `TEleExpress` or `TEleVarDec` or `TEleUnit` or...
5. All the nodes of the AST, of type expression (`TEleExpress`), already have their data type defined. This is possible because of Pascal's characteristic as a "typed" language.
6. AST nodes, of type expression (`TEleExpress`), might not have storage defined. Unlike the data type, the storage cannot be defined in advance, for optimization reasons, and must be defined in later phases.
7. Some `TEleConstDec` nodes might not be evaluated to their value. This happens when the constant is declared as an expression that needs to be evaluated (See section 4.1).
8. Some `TEleExpress` nodes of type `otConst` might not be evaluated in their value. This happens when they refer to an unevaluated `TEleConstDec`.

11 SENTENCES

Every Pascal program is mainly composed of statements. Statements are the instructions that make up a program.

The following code, for example, has 3 statements:

```
begin
procl ( x, y );
b := 1 ;
  if a = 0 then a = 1 ; end ;
end ;
```

Conditionals like this, like REPEAT or WHILE loops, are considered as a single statement that can contain internal blocks of code.

The statements are grouped into blocks of code, represented by the classes:

- TEleBody -> Blocks of code within functions, units or the main program.
- TEleBlock -> Blocks of code within conditional structures, loops, or BEGIN ... END blocks.

Do not confuse code blocks with program frames, described in section 10.7 **Error! Reference source not found.** Code blocks are containers that include declarations and a main code block. Code blocks contain only statements.

11.1 Sentence Types

All statements are modeled within the compiler as an element of the TEleSentence class. However, this statement can be of various types, which are represented by the following enumeration:

```
TxpSentence = (
sntNull, // Default value
sntAssign // Assignment
sntProcCall, // Procedure call
sntAsmBlock, // ASM block
  sntBeginEnd, //BEGIN-END block
  sntIF,       //Conditional IF
  sntREPEAT,   //REPEAT Loop
  sntWHILE,    //WHILE Loop
  sntFOR,      //FOR loop
  sntCASE,     //Conditional CASE
  sntExit      //Exit instruction
);
```

This classification is consistent with the typical definitions of the Pascal language, (Review the syntax diagrams that define the Pascal language).

It remains this way, despite the fact that the compiler has a different approach (See section 7.2), as an initial step in the compiler implementation, because previous implementations (which unified the `sntProcCall` and `sntAssign` types) generated identification problems. and implementation.

For this reason, we are trying to maintain the simple definition of Pascal as always, hoping that it will also facilitate implementation and keeping in mind possible future extensions.

All `TEleSentence` objects have the type defined in their "sntType" field, which is basically their only specialized field:

```
TEleSentence = class(TxpElement)
public
  sntType: TxpSentence; //Sentence type
  function sntTypeAsStr: string;
  constructor Create; override;
end;
```

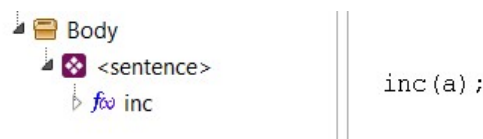
The statement element is intended to be a container for expressions whose treatment will depend on the type of statement.

11.2 Procedure Call Statements

Statements of type expression are those that are identified as being of type `sntProcCal`.

These types of statements include a single main node that represents a function or method that could also be an operator in its method version (See Section 7.4 for more details).

The following figure shows the diagram, in the syntax tree, that generates a statement that only calls a procedure:



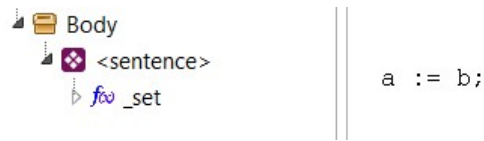
This expression falls into the `sntExpres` category, because it is a procedure call.

11.3 Assignment Statements

Assignment statements are identified as being of type `sntAssign`.

This type of statement is similar to the "Procedure Call" type, but mainly represents a call to the `_set()` assignment method.

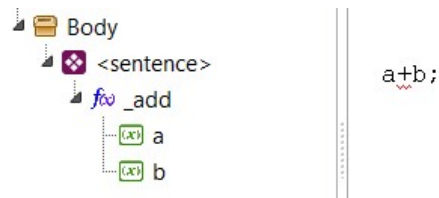
The following figure shows the statement that generates an assignment expression:



You can see that the assignment operator has been replaced by its corresponding `_set()` method, and therefore it could be seen as a particular case of "Procedure Call", but assignments are special methods that also deserve to be treated separately. special.

It can be considered that the `_set()` method is defined in such a way that the first parameter is always of type VAR because it is expected to obtain a physical address to assign to.

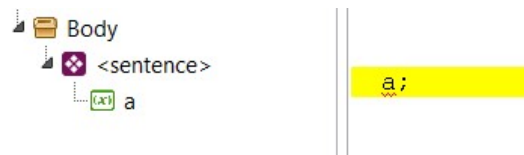
Expressions of type "a+b", would also be encoded as methods, but the compiler only allows expressions with operands for assignments, so these types of expressions will generate an error:



The rules that the compiler applies to verify that an expression is valid can be seen in the `T Analyzer.AnalyzeSentence` method and in summary are:

1. That is a call to a function/procedure/method (without using an operator), or
2. Let it be an assignment expression, with a single operand on the left hand side.

Expressions that only include one operand are also rejected as valid statements because they do not produce a valid method:



Such an expression will give a compile-time error. A variable type node can only appear at a more nested level of the expression.

As a conclusion it can be said that:

Todas las sentencias válidas que representan a una expresión (tipo `sntExpres`), incluyen un único nodo hijo (clase `TEleExpress`) de tipo `otFuncnt`.

11.3.1 expressions

In section 7, we explained how expressions are handled in the compiler, but we didn't say how they are implemented internally.

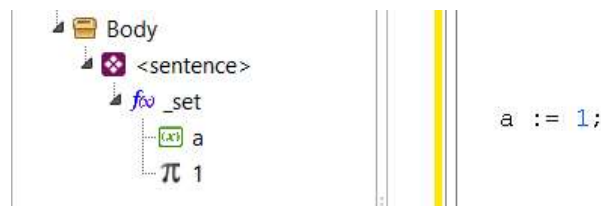
Here we will detail, a little more deeply, the implementation of the expressions within the compiler.

As we saw in the previous section, expressions are handled as objects of the TElExpress class, inside Statement nodes.

Also each of the operands and methods that make up the expression are also objects of the TElExpress class. That is, all nodes within an expression statement are of the same TElExpress class.

We also saw that expression statements start with a single child node of type otFunc.

The following figure shows the statement that generates an assignment expression:



11.4 Assembler Blocks

Assembler blocks are blocks that contain assembler instructions. They are statements of type "sntAsmBlock".

As the analysis of these blocks is relatively complex, a special chapter has been created in section **Error! Reference source not found..**

11.5 conditionals

Conditional expressions are a special type of statement that models IF statements, in their various forms:

```
IF ... THEN ... END ;
IF ... THEN ... ELSE ... END ;
```

```

IF ... THEN ... ELSIF ... END ;

IF ... THEN ... ELSIF ... ELSE ... END ;

```

Unlike standard Pascal, a Modula-2-like syntax is supported in this compiler.

For the filling of the AST, for uniformity, the last ELSE section is read as if it were a section: ELSIF true THEN ... because the behavior is similar. In this way all conditionals are reduced to a more general case:

```

IF < condition > THEN < block >

ELSIF < condition > THEN < block >

ELSIF < condition > THEN < block >

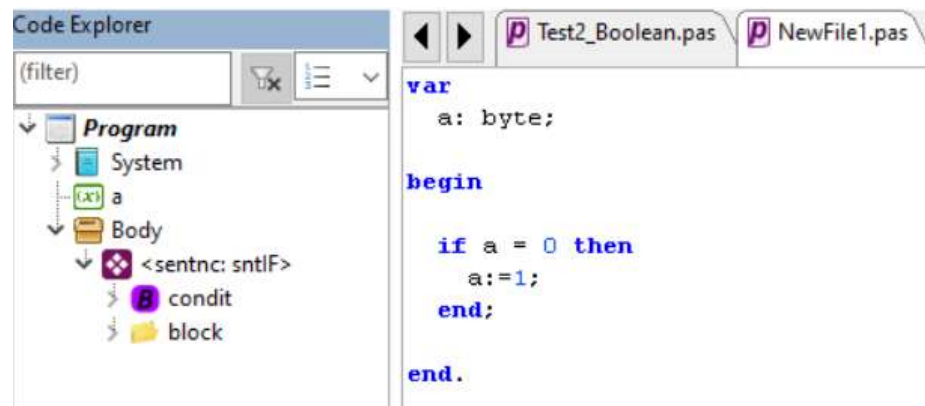
...

END ;

```

This modeling will be reflected in the syntax tree and will help simplify subsequent code generation.

The following image shows how a simple conditional is modeled in the AST.

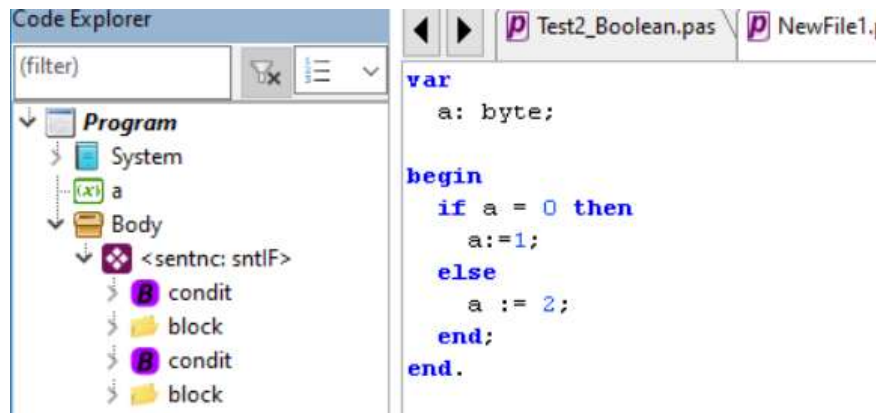


The entire conditional itself is considered to be a statement of type sntIF.

The "condit" element is a container that represents the boolean conditional expression that controls the IF. Internally it contains the condition expression (if there is only one) or the set of expressions that needs to be evaluated to get the result of the condition.

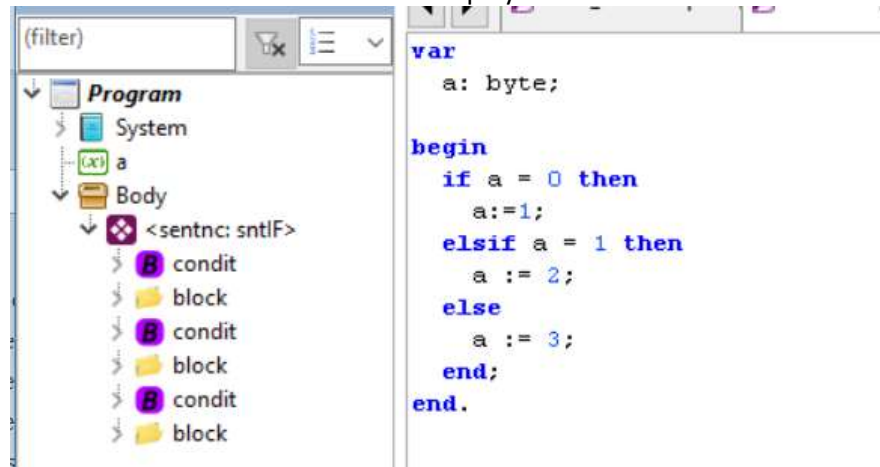
The "block" element is from the TElBlock class and is similar to a TElBody container, because it can contain one or more statements or instructions.

The following images show the case of a conditional in the AST, which includes an ELSE:



In this case, the second condition only contains a constant, with the fixed value TRUE, which is how the ELSE is modeled.

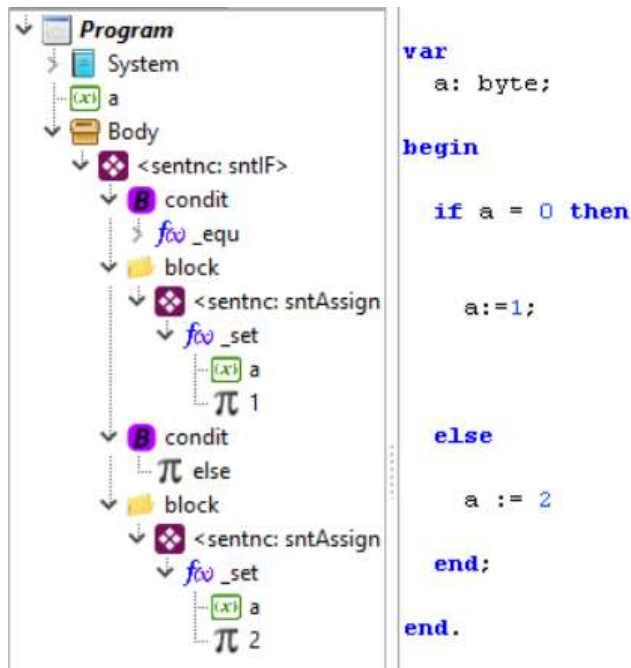
A conditional with an ELSIF will be displayed in the AST as shown in the figure:



The creation of the conditionals in the AST is done in TAnalyzer.AnalyzeIF().

11.5.1 Structure of a conditional

To study the structure of a conditional in the AST, let's consider a simple conditional, like the one shown in the following figure:



Here you can see the internal structure of the conditions and code blocks. Simple conditions contain only one expression, as shown in the figure, but in complex expression conditions, the optimization routines can split the expression into multiple expressions that are compiled as if they were separate statements (See Optimization section of the “Documentation – Optimization and Synthesis.docm”). When this happens, the expression that controls the condition will always be the last one.

blocks behave similarly to the body of a subroutine, that is, they work as statement containers. These sentences, in addition, can be of different types.

It can also be noted that the condition that represents the ELSE contains only one expression that is of constant type, and that must have the value TRUE.

The structure of conditional statements may seem a bit complex, but it has been designed this way to make optimization and code generation easier.

11.6 Exit() Statements

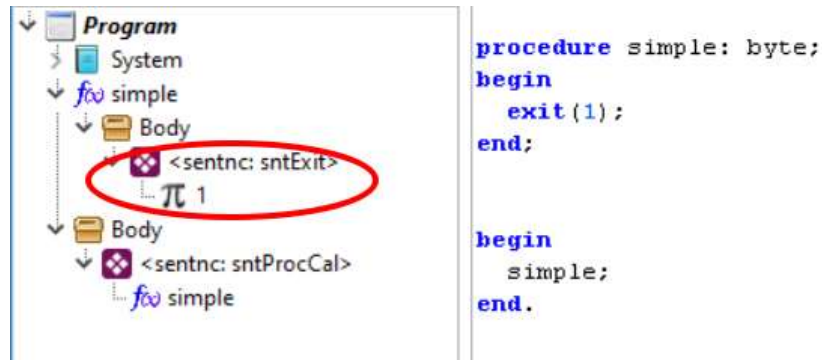
exit() statements, which are the common statements that allow you to exit a procedure or terminate the main program, are identified as statements of type sntAssign.

The exit() instruction is handled as a special type of statement within the language, instead of being defined as a system function ¹⁰, due to its particular behavior regarding the parameter it can receive.

¹⁰ This is how it worked in previous versions of the compiler.

An `exit()` statement can receive one or no parameters, and the parameter can be of any type.

Within the AST, the `exit()` statement is represented as a statement that may or may not receive an expression as a child node:



The subroutine that parses this statement is `TAnalyzer.AnalyzeEXIT()`:

```

procedure TAnalyzer.AnalyzeEXIT ( exitSent : TEleSentence );
...
begin
  ProcComments ;
  parentNod := TreeElems.curCodCont.Parent ;
  if parentNod.idClass = eleProg then begin
    prog := TEleProg(parentNod);
    //It's the main body
    if GetExitExpression(oper) then begin
      GenError('Main program cannot return a value.');
```

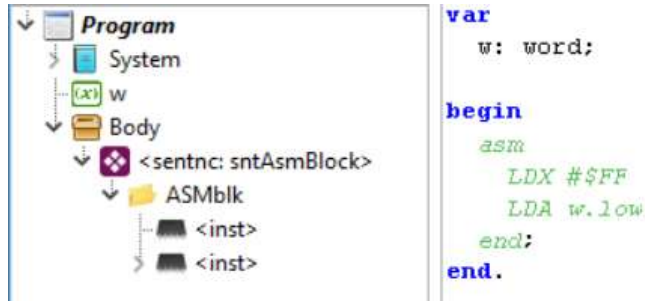
```
end;  
//Lleva el registro de las llamadas a exit()  
func.RegisterExitCall ( exitSent );  
end ;  
end ;
```

Most of the code shown is verification of the rules that define the need to include the `exit()` parameter. As defined, it is not allowed to specify a parameter in the `exit()` if it is used inside the main program.

An important detail, and one that will be used for later optimization, is the call made to `RegisterExitCall()`, to register the call of the `exit()` statement.

12 ASSEMBLE BLOCKS

Assembly blocks, or ASM blocks, are a special type of statement that represents a common assembly block, defined by the ASM ... END delimiters:



The assembler blocks are processed by the “ParserASM_6502” unit. Here are the parsing routines. The code generation routines are located in the GenCodBas_PIC16 unit.

This documentation describes the analysis process. The code generation is described in the document “Documentation – Optimization and Synthesis”.

The assembler blocks generate, in the AST, an element called "ASMBlk" that internally contains various elements representing the instructions.

The “ASMBlk” element is the assembly block, which is modeled with the TElAsmBlock class and behaves as a container for ASM instructions. Each instruction within a TElAsmBlock represents an object of the TElAsmInstr class.

12.1 lexical analyzer

Even though ASM blocks are part of the programming language syntax. The "lexer" that parses the Pascal source code (TContext object), does not directly recognize the assembler syntax.

Therefore, the facility of the TContext lexer is used to change its token identification routine and be able to “teach” it another syntax. This change can be seen in the TParserAsm_6502.ProcessASMBlock() routine of the ParserASM_6502 unit:

```

procedure TParserAsm_6502.ProcessASMBlock ( cpx0 : TCompilerBase );
...
begin
  cpx := cpx0; //Reference to compiler.
  cpx.Next;    //Get ASM
  cpx.curCtx.OnDecodeNext := @DecodeNext; //Set a new lexer
  StartASM;
  asmBlock := TElAsmBlock.Create;
  asmBlock.srcDec := cpx.GetSrcPos;

```

```

asmBlock.name := 'ASMblk';
cpx.TreeElems.AddElementAndOpen(asmBlock);
...
//Current token is delimiter END.
cpx.curCtx.OnDecodeNext := nil; //Restore lexer here, in order to take the
"END" with the new lexer and avoid problems of syntax.
cpx.Next; //Take END with default lexer.
cpx.TreeElems.CloseElement ; // Close ASM block
end ;

```

The way to change the “scanner” of the lexical analyzer is through the “OnDecodeNext” event. In this way we can define our custom “scanner”, which recognizes the assembler syntax.

In our case, we set the TParserAsm_6502.DecodeNext() routine for recognition of new tokens. This routine must follow the rules defined for the customization of the lexical analyzer.

Even though we define new rules for token identification, the types of tokens are the same as defined in the LexPas unit.

12.2 ASM Instruction Types

The TElAsmInstr class, which models the 6502 assembly instructions, has the following definition:

```

TElAsmInstr = class ( TxpElement )
addr : integer ; // StartingAddress. Used only in code generation.
iType : TiType ; // ASM instruction type
// Fields to generate instructions.
opcode : word;
addMode: byte;
operVal: integer;
operRef: TxpElement;
operNam: string;
constructor Create; override;
end;

```

With these fields it should be possible to represent all ASM instructions, except instructions with element operands (See section 12.3.2).

The "addr" attribute indicates the physical address of the ASM instruction. In the analysis it is only initialized to -1. It will be used in code generation.

The "iType" field defines the type of the assembler instruction. It is defined by the TiType enumerator:

```

TiType = (
itOpcode, // Common instruction with an Opcode and Operand.
itLabel, // An ASM label.
itOrgDir, // Instruction ORG
itDefByte // Instruction DB

```


);

Depending on the value of "iType" the following fields of TElAsmInstr will take on a special meaning.

The type "itOPcode" is the common type of assembly instructions.

The type "itLabel" identifies the labels, which are treated here as one more instruction.

The type "itOrgDir" represents an "ORG <address>" directive.

The type "itDEfByte" represents a DB directive that is used to define a single byte.

Not all of these types are implemented in the current version of the compiler, as can be seen in TGenCodBas.GenCodeSentences(), for the "sntAsmBlock" case.

The various types of implemented instructions are described below.

12.3 Common Instructions

They are identified when "iType" is equal to "itOPcode". This is the case for common instructions that include an Opcode, an Address, and optionally an Operand, as in the following examples:

```
LDA $A0, X
LDX value
BEQ jump
CLC
```

Common instructions can, in turn, have different types of operands depending on the value that is put in TElAsmInstr.operVal.

value of "operVal"	Operand Type
>=0	Operating direct. It reads from "OperVal".
= -1	operating element.
= -2	Operating "\$"

12.3.1 direct operands

An operand is said to be direct when it is a literal and the value of the operand can be read directly in the same parsing phase. Some examples are:

```
LDA $1234
ADC#15
```

In this type of instructions, the “opcode”, “addMode” and “operVal” fields of TElAsmInstr are used:

```
TElAsmInstr = class ( TxpElement )
  addr   : integer; //Starting Address. Used only in code generation.
  iType  : TiType;  //ASM instruction type
  //Fields to generate instructions.
  opcode : word;
  addMode: byte;
  operVal: integer;
  ...
end ;
```

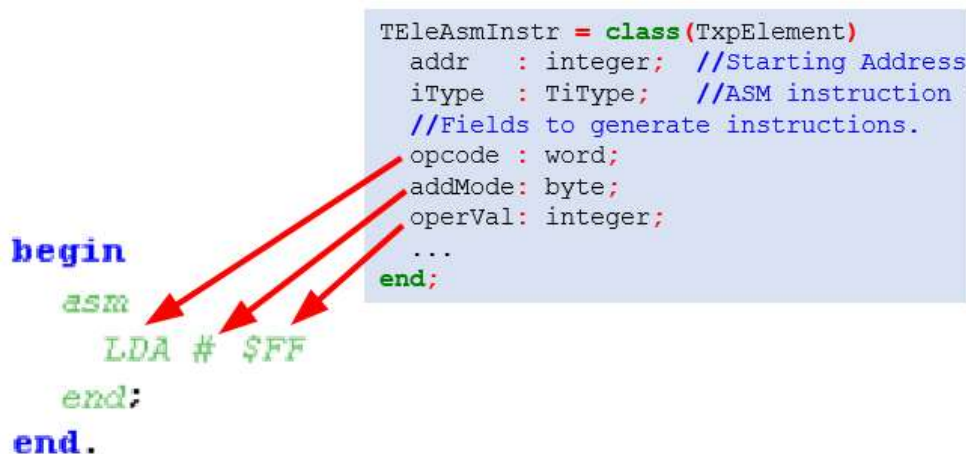
Formally, the “opcode” field should be of the TP6502Inst type, but it has been defined as a number, because we do not want to depend on the P6502Utils unit, on the XpresElemP65 unit, where this class is defined. This unit should be as independent as possible from the target CPU hardware.

Under normal conditions, “opcode” is a positive value that is converted to a TP6502Inst, through a simple “casting”, so that it identifies a CPU OpCode.

The “addMode” field represents the address mode of the instruction. It is also defined as a number, for the same reason that justifies defining “opcode” as a number. It is converted to the routing identifier, by simple casting with TP6502AddMode().

The “operVal” field would be the value of the operand, of the ASM instruction defined by “opcode” and “addMode”.

In simple instructions like: “LDA #ff” or “JSR \$1234” or “BNE \$FE”, it stores the numeric value of the operand, as a positive number.



In these cases, the value of “operVal” can be resolved in the parse phase, but when the operand of the assembly instruction is a reference to a language element (such as a variable), we

say that it is an “operand element”. ”, and cannot be resolved in the analysis phase, but the reference is saved, to be resolved later.

The element operands are described in section 12.3.2.

12.3.2 Element Operands

An ASM instruction with operand element is one that includes references to other elements of the Pascal program (which contains the ASM block) or to labels of the same ASM block.

Examples of these types of instructions are:

```
LDA #const_pascal
STA var_pascal
JSR func_pascal
NBS label1
```

Items that can be referenced in these instructions are:

- Pascal Variables (Element “eleVarDec”)
- Pascal constants (Element “eleConsDec”)
- Pascal Functions (Element “eleFunc”)
- Labels of the same ASM block (Element “eleAsmInstr”).

For this type of expression, with operands that cannot be completely defined in the “operVal” field, the value of “operVal” is set to -1 and “operRef” is used to point to the referenced element.

```
var
  varA: byte;
begin
  asm
    LDA varA
  end;
end.
```

```
TEleAsmInstr = class(TxpElement)
  addr   : integer; //Starting Address
  iType  : TiType;  //ASM instruction
  //Fields to generate instructions.
  opcode : word;
  addMode: byte;
  operVal: integer;
  operRef: TxpElement;
  ...
end;
```

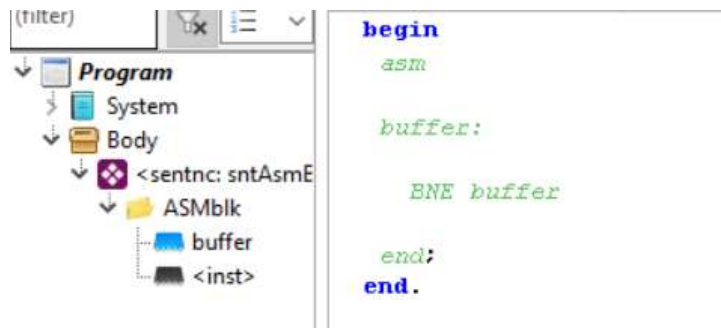
12.3.3 label operands

There is a case where the operand of the instruction does not reference an element of Pascal code, but instead references a tag in the ASM block itself.

This would be the case for instructions like:

```
bne jump
JSR subroutine
```

These instructions are encoded by also creating an Operand in the instruction.



This case is handled like any other element operand case, except that the referenced element is an element of the same ASM block.

Note that the “buffer” tag is also represented as a TElAsmInstr instruction (See section 12.4).

This type of operands are recognized in TParserAsm_6502.CaptureOperand() and can be of two types:

- Operands already defined
- Operands to be defined

OPERANDS ALREADY DEFINED

The already defined operands are labels that are declared before they are referenced, as in the following case:

```
jump:
bne jump
```

This case is easy to process because the tag is created before it is used. This tag is assigned directly in “operRef”, as seen in TParserAsm_6502.CaptureOperand():

```
function TParserAsm_6502.CaptureOperand: boolean;
...
begin
...
end else if cpx.tokType = tkIdentifier then begin
    if IsLabelDeclared(cpx.token, lblEle) then begin
        //Es un identificador de etiqueta
        curInst.operVal := - 1 ;           // Indicates to use "operRef"
```

```

curInst.operRef := lblEle ; // Label reference.
  cpx.Next ;
  exit ( true );
end ;
...
end ;

```

OPERATING TO DEFINE

The operands to be defined are labels that appear after they are referenced, as in the following case:

```

jmp jump
jump:

```

This situation poses the problem that you cannot define which tag element is referenced, because this tag will (should) appear later.

The section of code, which identifies undefined label operands, is shown:

```

function TParserAsm_6502.CaptureOperand : boolean ;
...
begin
...
  end else if cpx.tokType = tkIdentifier then begin
...
ele := cpx.TreeElems.FindFirst ( cpx.token ); // identify element
  if ele = nil then begin
    // Undefined identifier ( like a tag ) . can be defined later.
    curInst.operVal := -1 ; _ // Indicates to use "operRef"
    curInst.operRef := nil; //Will be later linked.
    curInst.operNam := UpCase(cpx.token); //Keep name to find reference.
    //Los saltos indefinidos, se guardan en la lista "undJumps"
    curBlock.undefInstrucs.Add(curInst);
    cpx.Next;
    exit(true);
  end else begin
    // Identifies a language element
...
  end ;
...
end ;

```

For this case, the value of “operRef” is set to NIL and the “operNam” field is used to store the name of the label that will be searched for later, when the reference is completed in the TParserAsm_6502.EndASM() routine.

To know that this reference must be completed later, it is added to the list “curBlock.undefInstrucs”. This list will be used in TParserAsm_6502.EndASM().

But, in addition, "curBlock.undefInstructs" will be used in the code generation process. That is why it has been declared as part of the ASM block "curBlock" (from the TElAsmBlock class), because that way the list will reach the code generator.

12.3.4 Operating "\$"

The operand "\$" is the one used to refer to the current position.

This would be the case for instructions like:

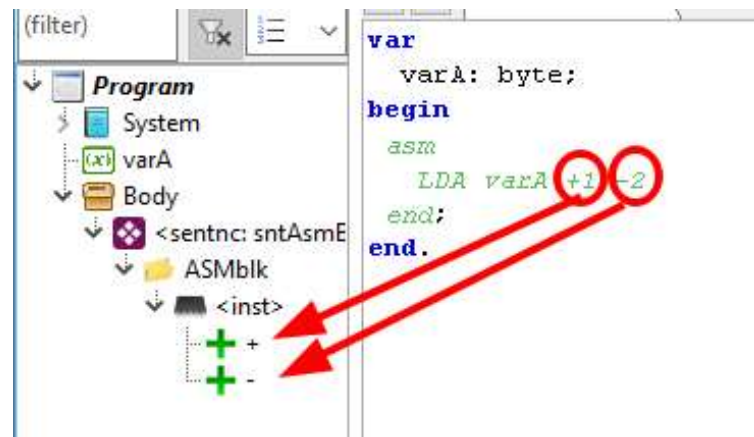
```
LDA $
```

This case is identified by setting "operVal" to -2.

12.3.5 Operations

Operations are understood as the arithmetic or logical transformations to be applied to an operand.

Operations are created within the AST as child nodes:



The operations are modeled with the TElAsmOperat class:

```
TElAsmOperat = class ( TxpElement )
operation : TAsmInstOperation ; // operations
value : word ; // Value
Create constructor ; override ;
end ;
```

The implemented operations, values of the TAsmInstOperation enumeration, are small but sufficient for acceptable assembly instruction support:

```
TAsmInstOperation = (
aopSelByte, // Select a byte (.low, .high, >, <)
```

```

aopAddValue, // Add a value
aopSubValue // Subtract a value
);

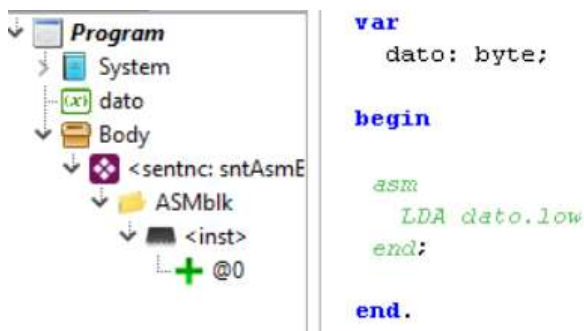
```

Each operation is complemented with a number that is stored in `TEleAsmOperat.value`.

The expressions that can be achieved with this format are simple, but cover most cases very well.

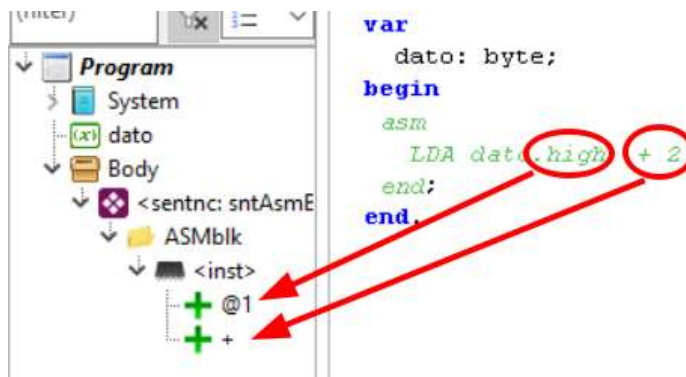
The operations apply to the operand defined in “`TEleAsmInstr.operVal`” or in “`TEleAsmInstr.operRef`” and can be zero or many operations:

For example, the instruction can be modeled well: `LDA $+16-3` or `LDA data.low`



In this case, the operation “`data.low`” (equivalent to “`<data>`”) is modeled as a child node with the tag “`@0`”.

More complex parameters will imply that more operations are generated, as in the case of `LDA data.high + 2`:



It is convenient to have the instructions that make references to variables, functions or even declared constants represented in this way, because as we are in the analysis phase, the addresses of the variables or functions are not yet resolved.

Note that the expressions, which will be processed in this way, do not obey precedence criteria, but simply the order in which they are written in the instruction (from left to right).

12.4 Instructions Label

These types of instructions are identified when the “TEleAsmInstr.iType” field takes the value “itLabel”.

ASM instruction labels are also modeled as if they were independent instructions, as an object of the TEleAsmInstr class.

For example, the following tag declaration:

```
label1:
```

It will generate an “itLabel” statement. The tag name will be stored in “TEleAsmInstr.name”, which is an inherited attribute.

Labels, which are included on the same line as a statement, are also considered to be separate statements. Let's consider the case:

```
label1: LDA #00
```

This assembler line will generate two instructions, one for the label and one for the instruction, as shown in the following figure:



Unlike common instructions, label-instructions are created with the TParserAsm_6502.AddInstructionLabel() function:

```

procedure TParserAsm_6502.AddInstructionLabel ( lblName : string ; srcDec :
TSrcPos );
begin
...
curInst := TEleAsmInstr.Create ;
curInst.name := lblName ;
curInst.srcDec := srcDec ;
cpx.TreeElems.AddElementAndOpen ( curInst );
  curInst.inst := -1 ; _ // Mark as a jump instruction.
curInst.addr := -1 ; _ // Indicates that the physical address has not yet been
set.
labels.add ( curInst ); // Add to list of tags
end ;

```


One detail about label instructions is that they are created with the “addr” field set to -1, to indicate that the label is not yet mapped in RAM. This is important because the code generation routines need to know when a tag is already defined or not.

All the labels are registered in the "labels" list to then perform an exploration in search of references to be completed. This scan is done in TParserAsm_6502.EndASM():

```

procedure TParserAsm_6502.EndASM ; // Terminate ASM code processing
...
var
jmpInst : TElAsmInstr ;
begin
  //Completa los saltos indefinidos
  for jmpInst in curBlock.uncInstrucs do begin
    if not CompleteUndefJump(jmpInst) then begin
      //No se encuentra "jmpInst" en "labels".
      cpx.GenError(ER_UNDEF_LABEL_, [jmpInst.name], jmpInst.srcDec);
    end;
  end;
end ;

```

12.5 ORG Directive

This type of directive is identified when the “TeleAsmInstr.iType” field takes the value “itOrgDir”.

The ORG directives are also modeled as instructions, that is, as objects of the TElAsmInstr class.

An example of an ORG directive is:

```
ORG $
```

The ORG directive allows you to define the memory location from where the assembly of the following instructions will start.

Inside the AST, they appear as normal instructions with the name ORG:



There can be many ORG directives and they can appear in any position, but it is clear that by their nature, they can break the order of the normal sequence of execution and could cause runtime errors if not properly controlled.

The ORG directive also accepts operations on the operand:

```
ORG $-2
```

12.6 DB Directive

This type of directive is identified when the “TEleAsmInstr.iType” field takes the value “itDefByte”.

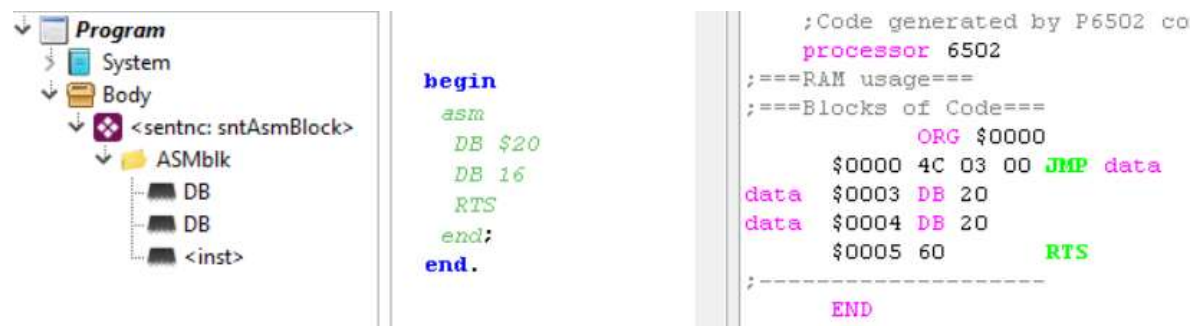
DB directives are also modeled as instructions, that is, as objects of the TEleAsmInstr class.

An example of a DB directive is:

```
DB$12
```

The DB directive allows you to define values to be placed directly at the current position in the assembly.

Inside the AST, they appear as normal statements with the name DB:



The DB directive can be used to define variables within the code section. However for this, it is recommended to insert a jump before until after DB

```

JMP code
fact:
DB 0 ;data
code:
LDA data

```