UST Solutions Inc.

UltraGram

Visual parser generator

User Manual

Contents

1. Overview

UltraGram is a visual development environment for creating context free **LALR(1), LR(1), GLR** grammars, testing them, debugging against test files and generating a source code for selected programming language. **UltraGram** provides a simple and intuitive interface, extensive information about parsing process and parser tree, conflicts and errors, generates and displays **DFA** graph and **lookaheads**.

The process flow in general includes the following steps:

- Creation of a new project or opening an existing one
- Creation / opening / selection of a grammar file (***active grammar file***)
- Creation / opening / selection of a test file (***active test file***)
- Debugging session (making required changes in grammar file, compiling it and parsing test file with this compiled grammar)
- Optional generation of a source code for selected programming language

2. Development environment

The main screen of the application is shown in Fig. 1

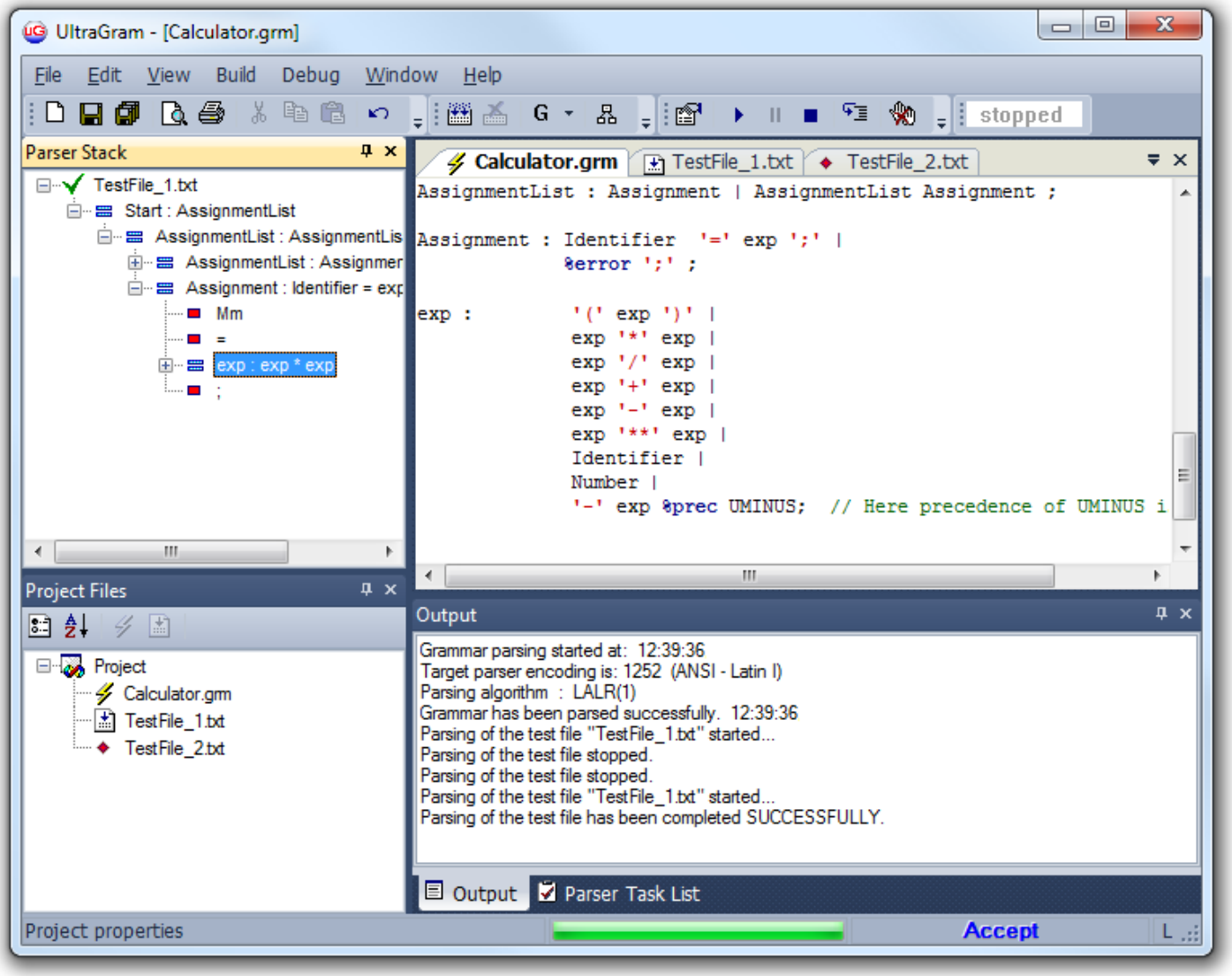

Fig. 1

Application toolbars

Application toolbars are presented by the following groups:

- Standard toolbar
- Build toolbar
- Debug toolbar

Standard toolbar. This toolbar provides standard functionality for file and editor operations

Fig 2.

All icons of this tool bar are standard and self-explanatory.

Build toolbar.

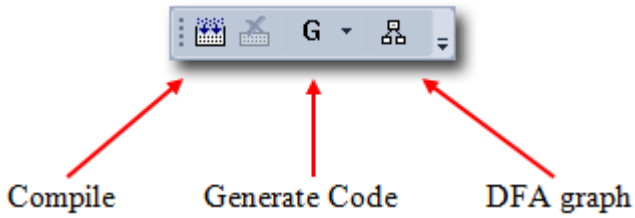

Fig 3.

Buttons on this toolbar initiate compiling of the active grammar file, creating **DFA** graph and initiating the process of generating source code for selected programming language.

Debug toolbar. This toolbar is equipped with buttons controlling parsing and debugging of the active test file with the grammar defined in the active grammar file.

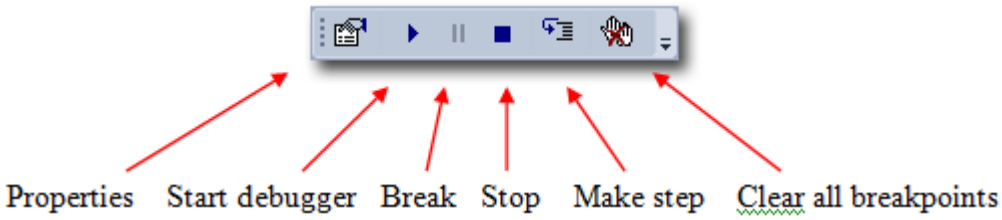

Fig. 4

All toolbar functions are also assessable from the application menus.

Application windows

Application presents the following windows:

Parser stack. This window displays the information from the parser stack in a from of a tree. The nodes of the tree represent the grammar rules and the tokens collected from the input.

Grammar and test files windows. These windows are used to enter grammar and test files. Note, that there can be many file windows in a project, but only one active grammar file and only one active test file (against which the grammar will be tested). Active grammar and active test files are marked with icons as on the Fig.5

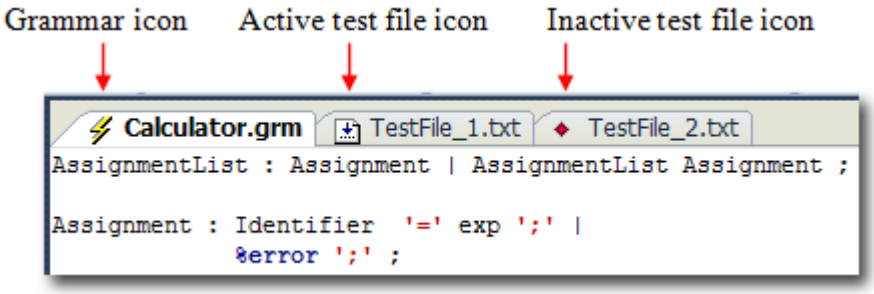

Fig. 5

Test file window is same type of an editor window as a grammar window. The only difference is that this window is equipped with a breakpoint stripe on the left side. To set or remove a breakpoint just click on the stripe in required place. Fig. 6

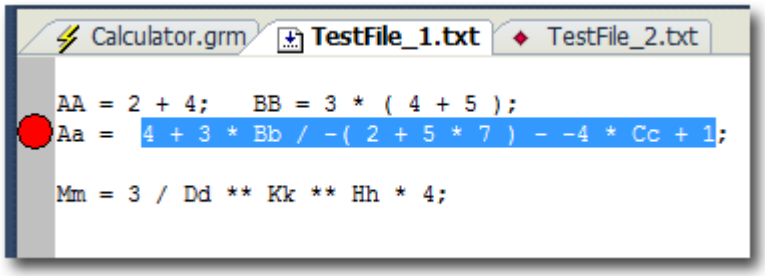

Fig. 6

Project files window. All files included in a project are listed in this window. Additionally the window provides functionality to select active grammar file and active test file. Fig. 7.

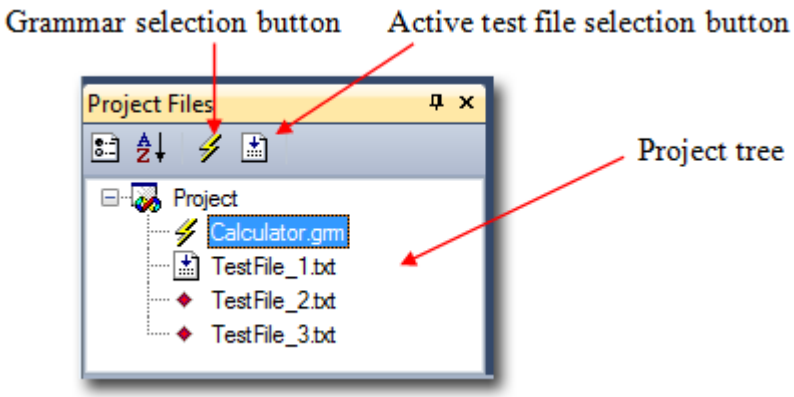

Fig. 7

Selection of the active grammar and active test file is really easy – just select appropriate file in the ***File tree*** and then click the required button.

Parser task list window. This window (Fig. 8) displays different information related to the

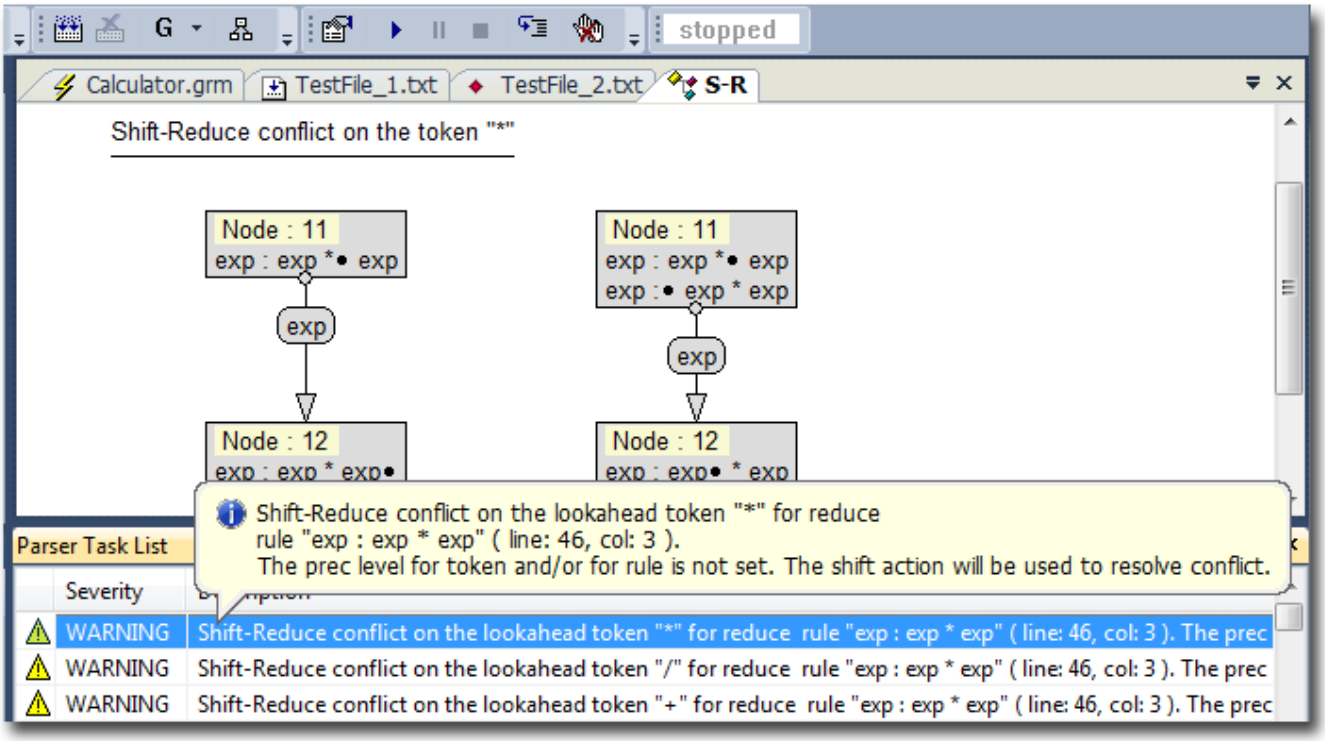

Fig. 8

active grammar. Each line can have different severity (INFO/WARNING/ERROR) that requires user to act in certain way. In case of WARNINGs related to shift-reduce and reduce-reduce-reduce conflicts, the double-click on the line will bring up a new conflict graph window with visualized conflict process flow.

Conflict graph window. On this window (see Fig. 8) the sequences of actions that lead to creation of shift-reduce or reduce-reduce conflicts are visualized. In the window there are two graphs. Both of them start from a same point of a single DFA node and end in this or some other single DFA node. The start point production may be resolved in some ways, some transitions possibly could occur. In the sample on the Fig. 8 the graph on the left side shows the result of "**exp**" transition from start point production to the end DFA node. The graph on the right shows that the start point production is first resolved in the start DFA node by "**exp : . exp ∗ exp"** and only then transition by "**exp"** from the start DFA note to the end node occurs. This illustrates the typical shift-reduce conflict on particular lookahead token ("∗" in this sample) when decision (whether to shift or to reduce) should be made. It is possible to explicitly control the behavior (decisions) of the parser in cases like that by using ***precedence section*** in grammar file and ***%prec*** keyword (explained later in the manual).

DFA window. This window (Fig.9) is displayed after the corresponding button on the **Build** toolbar is clicked.

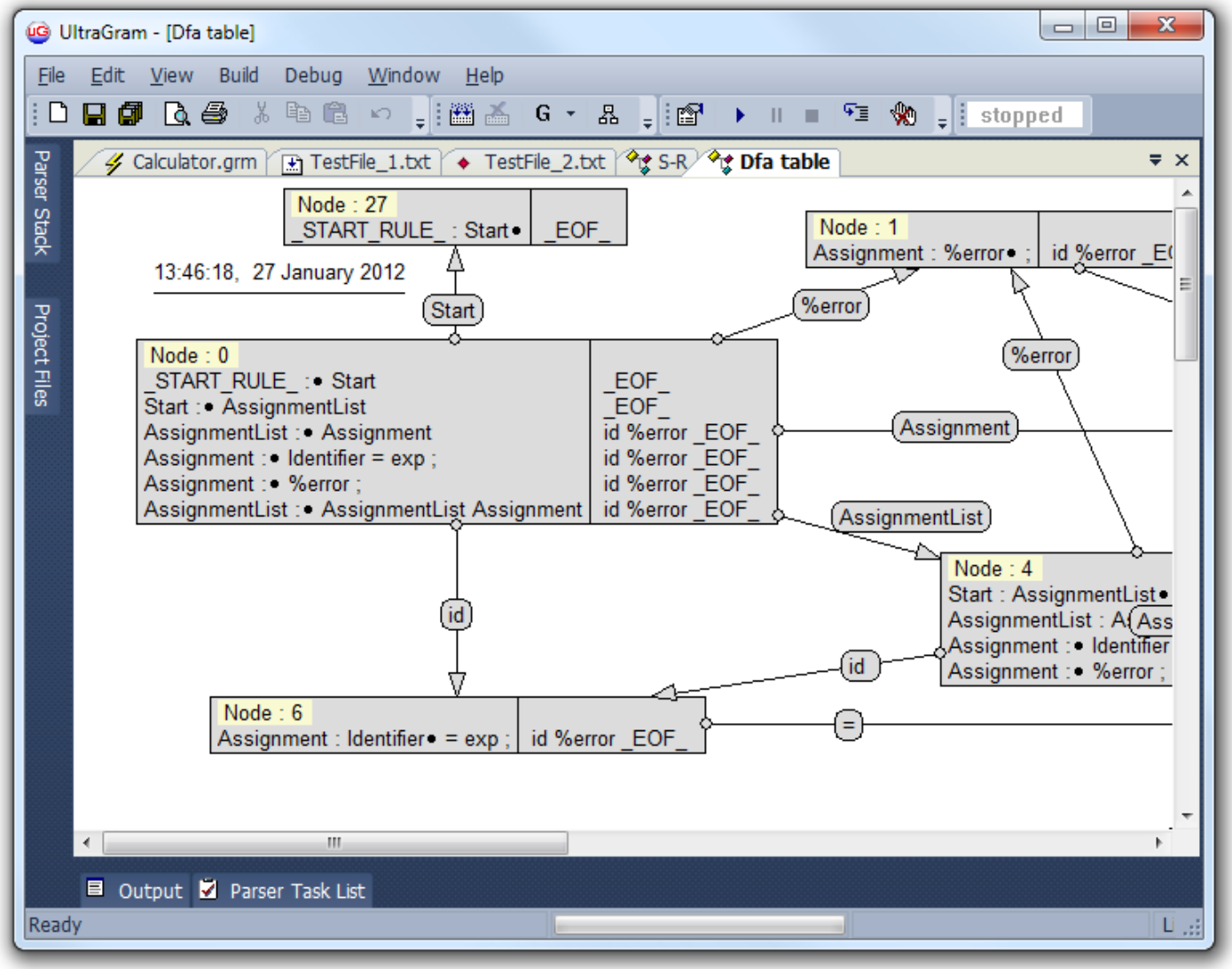

Fig. 9

Grammar must be successfully compiled before that. All nodes in on the window can be moved and resized. By default the size of each node is limited just to displaying of all productions. If there is a need to observe corresponding production lookaheads then the width of the node rectangle should be increased. For purpose of optimal layout, there is a possibility insert breaks in the transition lines from one node to another. To do so, just click on the required line and then press right mouse button. Choose required option from the pop-up menu. Fig. 10

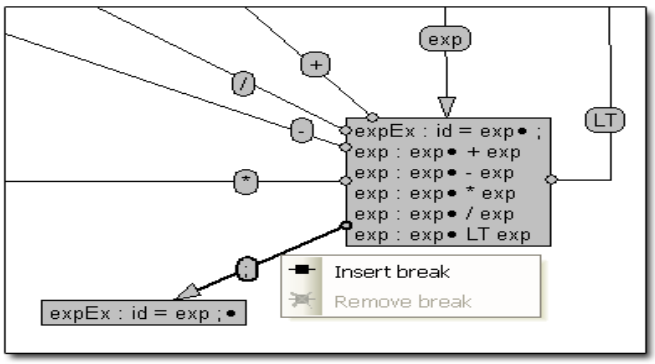

Fig. 10

Popup windows. There are several popup windows in the application. The main of them are:

- New project window (accessible via menu “New project”).

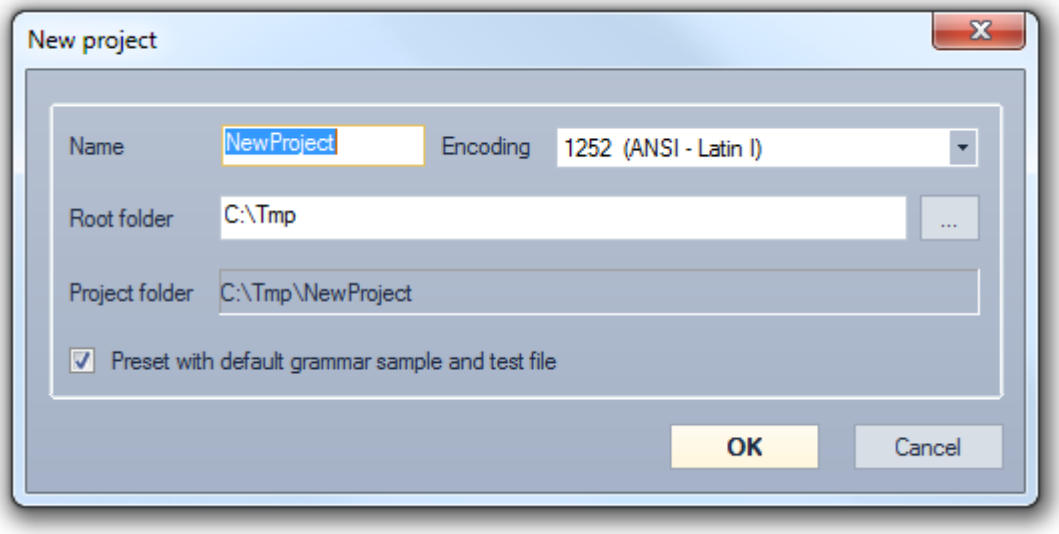

Fig. 11

The fields of this window are self-explanatory. It is important to note that selected **encoding** is related to the target parser that supposed to be generated as result of this project. In other words the target parser data tables will be build based on this selected encoding. The other necessary comment is that the default grammar sample that is used to preset the new project (just for highlighting general guidelines) is located in the file “***Grammar.gtm***” in the installation folder. This file can be freely modified to reflect specific user preferences.

- Properties window (accessible via toolbar button Properties). This window has several pages that can be selected by selecting nodes of the tree on the left side of the window.

- The General page (Fig. 12) contains several elements. The first one is the **Default projects root folder**. **UltraGram** will try to create all new projects in this folder (this value can be changed for individual project in the New Project dialog) .

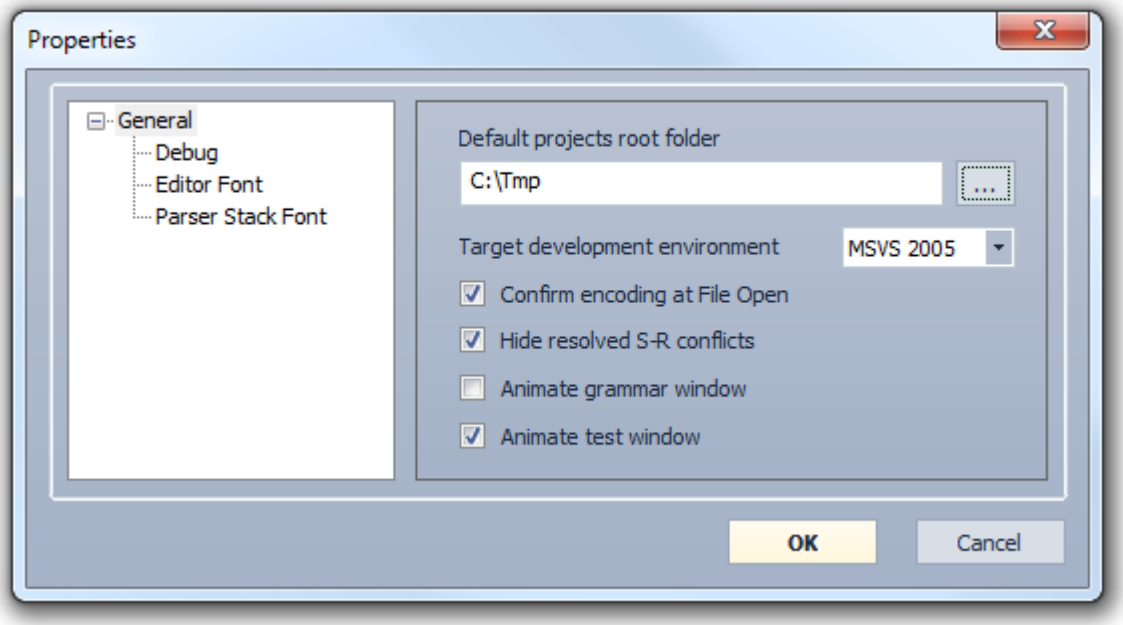

Fig. 12

“Target development environment” controls the format of output solution and project files for Microsoft Visual Studio.
The check box “**Confirm encoding on File Open**” controls application behavior in case menu “**File - Open**” is selected. If this check box is selected, then each time a new file is opened and added to the project, a special dialog window with a list of installed code pages will appear. Here it is possible to choose correct encoding for the specified file.
The check box “**Hide resolved S-R conflicts**” controls whether information about resolved Shift-Reduce conflicts is displayed in the “Parser Task window” (the technique of resolving this type of conflicts is described later in this manual).
Two checkboxes “**Animate grammar window”** and “**Animate test window”** enable the highlighting of different grammar elements during parsing or when selection on the parser tree changes. This feature helps to understand in details what is going on during parsing, however when the grammar of test files are constantly modified and parsing process is restarted it might be better to disable animation to avoid manual repositioning of grammar and test file windows.

- “**Debug**” window Fig. 13

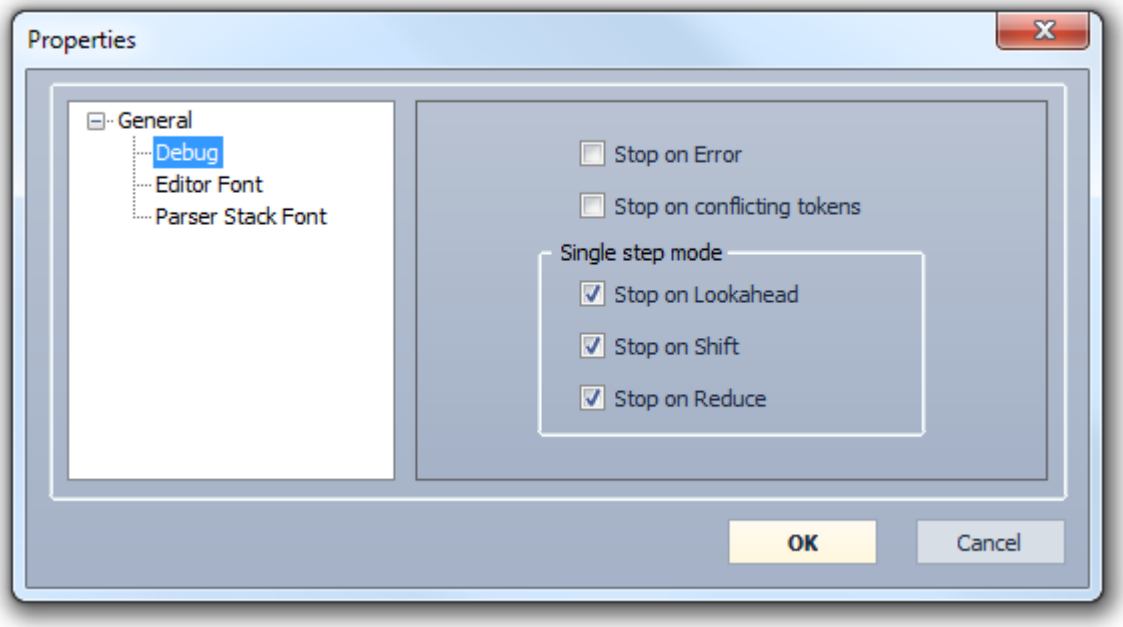

Fig. 13

This window has a set of check boxes. The first one “Stop on error” controls the behavior of the parser in case it encounters an error in the text of the active test file. If this type of error is handled by the grammar (***%error*** token is explained below) – parser will stop (pause execution) or continue based on the state of the check box.
(Obviously if an error is not handled – parser will stop in all cases).
The second one “Stop on conflicting tokens” controls behavior in case when more then one matching token is found in the input text. This situation is described in details in the ***pragmas*** section.
The group of check boxes “Single step mode” affects parsing process only if user decides to parse test file by using single steps (“Make step” button on the “Debug” toolbar). In this case parser will stop with resolution defined by this set of check boxes. Note that the actual stop state will be shown in the status line of the application. Fig. 14. (In this picture parser stopped on Reduce.)

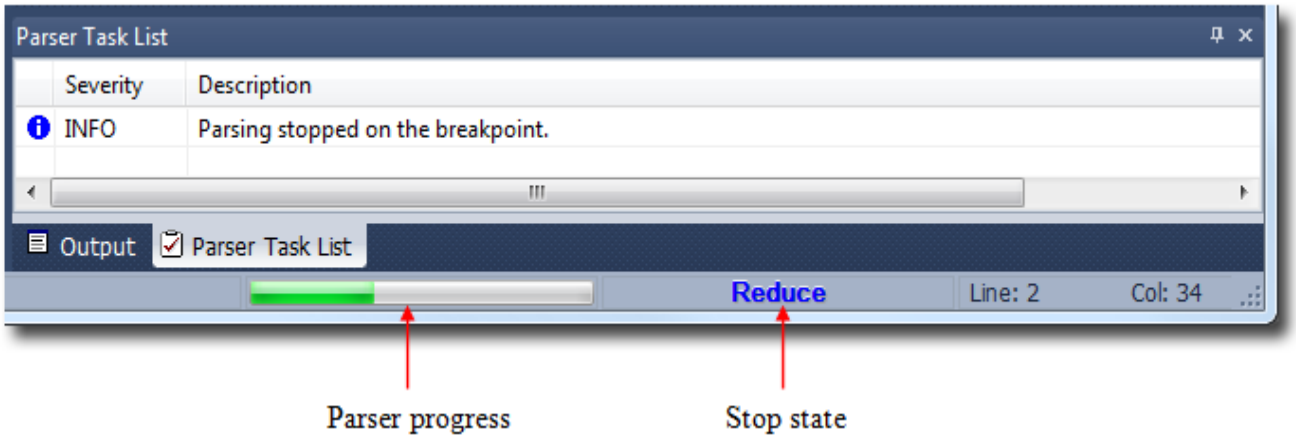

Fig. 14

Note also that if the parsing process completed successfully the displayed stop state will be set to **Accept**.

- Generate source code file window (Fig. 15). This window is activated by pressing corresponding button on the "Build" toolbar. Here it is possible to select the name of the main parser class and specify folders for the parser source code and for the test project. Note that this window is accessible only if grammar file was compiled with no errors, otherwise the toolbar button will be disabled.

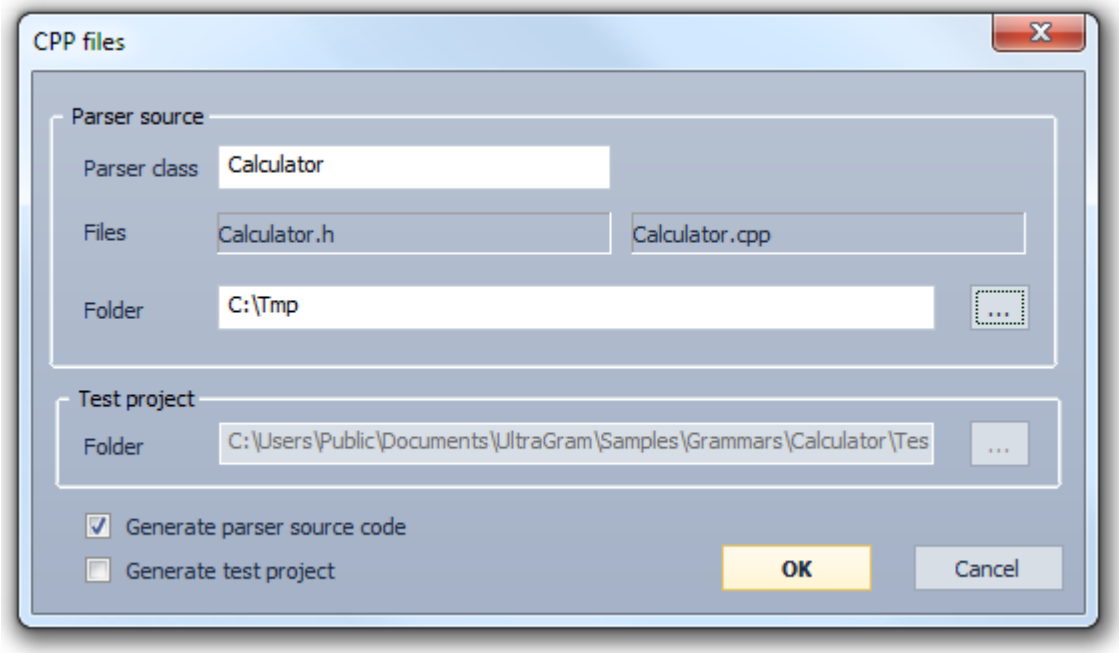

Fig. 15

3. Creating grammars

Grammar file must have the following sections:

- Optional pragmas section
- Optional macro section
- Tokens section
- Production section

3.1. Pragma section

The first optional pragma section starts with the keyword `'%pragma'` and defines directives to the parser / lexer to use specific parsing algorithm and produce specific behavior in some situations. The following directives are defined:

```
alg   glr   rtc   dkey   intok
```

The first directive `'alg( lalr | lr1 )'` dictates the algorithm that will be used to construct parsing tables. At the moment two algorithms are available **LALR(1)** and **LR(1).** Here is the example how to make this selection :

```
alg( lalr ) ;
```

Default algorithm is **LALR(1).** It will be used if this directive is not specified. Note, that using **LR(1)** algorithm generally will result in more reliable parsing and might produce less conflicts of different kind (like **Reduce-Reduce** conflict ..), however the size of generated parsing tables will be bigger and in some cases might be not practical.

The directive `'glr( on | off )'` generally affects parser behavior at run time in the points where **Reduce-Reduce** and **Shift-Reduce** conflicts are encountered (note, that this type of conflict can happen on both **LALR(1)** and **LR(1)** table construction algorithms). For this case parser at the point of conflict will try to detect a successful (that does not result in a parsing error) action that should be taken. This might result is some performance hit, however this might be an only way to successfully use ambiguous and complex grammars. Another form of this directive allows to target specific type of conflict: directive `glr( on, RR )` enables GLR mode only for **Reduce-Reduce** conflict and `glr( on, SR )` enables GLR for only for **Shift-Reduce** conflicts.

These `rtc` and `dkey` directives apply to the order of how tokens are processed and how decisions in case of conflicting tokens are made.

When parser enters a DFA state, lexer starts to analyze input text searching for a match in the set of applicable lookahead tokens. In ideal case there is only one token that matches the sequence of input characters. Lexer recognizes this token and returns it to parser for further processing. However in some cases on the same sequence of input characters lexer can find more then one lookahead token. In this case what token should be returned to the parser? For example: there are following declarations in the tokens section:

```
'ABC'                                   Keyword,        'ABC';
'([a-z]|[A-Z])([a-z]|[A-Z]|[0-9])*'     Identifier,     'id';
```

If both of them are listed as lookaheads in some particular DFA state and the input text contains characters 'ABC' lexer will detect both tokens! In this case the default rule is: ***the token that is defined later in the list of tokens has higher priority then the token that is defined first***. According to this rule lexer will return

to the parser the second token **'Identifier'** for further processing. Now if there is more then one parsing path (an expected sequence of tokens starting from this one) and parser entered the correct one – everything is fine. But if some number of steps later parser will fail – it means that lexer returned a wrong token – the correct one should be the **'Keyword'**. The most natural way of solving this problem is to change the order of the token declarations. In many cases this will be enough. However in some cases this approach along with fixing the problem in one place will create similar problem in another place.

UltraGram provides two ways of handling this issue.

The first one is initiated by the directive **'dkey(on | off)'.** The parameter **'on|off'** turns on or turns off this pragma (example: **'dkey(on);'**). When this pragma is turned on it will assigns the high priority to the tokens that have fixed length and all characters defined in upper or lower case (in other words tokens that look like keywords in a programming language will receive high priority). Under this category falls the token **'Keyword'** in the sample above. So when lexer detects both tokens on some character sequence, it will pass the keyword token to the parser regardless of the token position in the declarations section.

The second solution for choosing the correct token can be enabled by the directive **'rtc(on|off, <level>)'.** The value **<level>** can be in a range from **1** to **10** .
Example: **'rtc(on, 4);'**. When this pragma is turned on and conflicting tokens are encountered a special analyzing process starts that allows to choose a correct token. The reliability of the process can be tuned by the **<level>)** value. This value to some extend represents how far in the text parser will look to decide whether the token is correct or not. This is a very powerful solution, however it can slightly slowdown the overall parsing process. The value of the **<level>** parameter is not recommended to be always set to the maximum of 10. The best thing to do is to use the lowest possible value that resolves all errors. This lowest value for reliability purposes may be increased by one or two points.

The algorithm of using pragmas for best performance and reliability is as follows:

1. First try to enable the **dkey** pragma.
2. If parser fails – disable **dkey** and enable **rtc** pragma. Tune the value of the **level** parameter until parser succeeds
3. If success – try to enable **dkey** pragmas and check changes in performance.

The next pragma directive **'intok(on|off, on|off[,<valid char set>])'** controls the way tokens (terminal symbols) are declared and how error correction is made. The first parameter allows / disallows declarations in the production section, the second parameter controls how declared symbol will be compared with the input text. If parameter is **on** then lexer will look for exact mach, if **off** then case insensitive mach. The third parameter is related to typo error correction mechanism. In details it is described in the "Type error correction" section below.

3.2 Macro section

The macro section start from the keyword **%macro**. In this section it is possible to define a number of expressions that can be later used in the token declarations. Single macro definition has the following format

```
identifier      'expression'  ;
```

Here: **identifier** is unique name of the macro. By this name macro can be referenced in the token expression or in another macro.
expression is a sequence of characters enclosed in single quotes. Note, that this sequence is not checked in any way and is taken literally. Later on when a reference to the macro is met, this

reference is simply substituted by the macro expression. Only after this in case of token declarations expression is processed.

Example:

a simple macro to define a number :

```
num   '0-9' ;
```

This is a macro to define alpha character followed by a number:

```
anum  'a-z{num}' ;
```

In this case number is presented by a reference to the first macro. As result this macro will be

equivalent to the following

```
anum  'a-z0-9'  ;
```

3.3 Tokens section

Terminal symbols (also known as a "token type") in UltraGram can be defined in a special "**tokens**" section or directly in the "**production**" section. In the second case ("**production**" section) each terminal symbol is defined **inline** as a sequence of characters enclosed in single quotes. Example:

```
statement  :   'begin'   itemList   'end' ;
```

While parsing the input test, lexer will look for sequences of characters that mach sequences in single quotes (in the example above these are '**begin**' and '**end**'). This behavior can be modified by using **pragma intok** that turns **on / off** character case checking (exact mach or case insensitive match). However there are cases when more complicated rules must be applied. These rules must be defined in the section starting with keyword **%tokens**.

3.3.1 Token declaration

In the "**%tokens**" section each **terminal symbol** represents a class of syntactically equivalent tokens. You use this symbol in grammar rules to mean that a token in that class is allowed. Each symbol has the following format:

```
'expression' [identifier [,'alias']][,%ignore] ;
'expression' [identifier [,'alias']][,%chars valid_char_set] ;
```

Here:

expression defined the rule by which the token in the input text will be recognized
identifier is a unique name of the symbol
alias is an alias of this symbol.
%ignore directive instructs to skip the token
%chars directive instructs to enable type error correction on this token using **valid_char_set**
valid_char_set is a name of a token defining range of characters that should be used in type error recovery. For details see "**Typo error correction**" below.

Each token declaration must have the following structure:

- Format string enclosed in single quotes
- Token name and optionally an alias separated by a comma;

Token alias must be enclosed in single quotes. Each token declaration must end with a semicolon. Examples:

```
'[1-9][0-9]*'                              Number ;
'([a-z]|[A-Z])([a-z]|[A-Z]|[0-9])*'        Identifier,       'id';
'\('                                       OParen,           '(' ;
```

Normally in the text that should be parsed there are special sequences of characters that should be recognized and ignored by parser (these are for example white spaces that separate tokens, or comments embedded in the text). To handle cases like that these sequences must be described as regular tokens but marked with a keyword `%ignore` at the very end. Example:

```
'(//)[^\n]*'            CppStyleComment,  '//',       %ignore;
'[ \n\t\r]+'            wspace,                       %ignore;
```

For these type of tokens it is possible to completely omit name and optionally an alias and live only `%ignore` keyword. Example:

```
'[ \n\t\r]+'            %ignore;
```

Each token declaration must start with a format string. For building a token format string a set of characters and formatting characters should be used. Formatting characters are listed below:

`( )` define a group
example: `'(abc)+'` - matches `'abc'` or `'abcabc'` or `'abcabcabc'` ...

`[ ]` define a character choice group. Group can consist of characters and/or ranges. Only one character will be considered.
example: `'[abc]'` - any char that matches `'a'` or `'b'` or `'c'`
example: `'[a-z0-9]'` - any char in the range from `'a'` to `'z'` and from `'0'` to `'9'`

^ logical NOT. Can be used in the choice group `[ ]`. Result will be anything that matches the characters in the choice group before `'^'` and is not in the choice group after it.
exampe: `'[.^abc]'` - any char that is not `'a'` and not `'b'` and not `'c'`
example: `'[.^a-c]'` - any char except `'a'`, `'b'`, `'c'`
example: `'[a-n^be]'` - any char in the range `'a-n'` except `'b'` and `'e'`
Note, that if the first character is a dot `'.'` it can be omitted so `'[.^a-c]'` will be equal to the following : `'[^a-c]'`

Functionality of the choice group in case of logical NOT can be extended by supplying an expression enclosed in parentheses instead of a char or char range.
example: `'[.^(\*/)]'`. Lexer will thread this as follows: any character that is not a first character of a sequence that matches expression in parentheses: `'*/'`
This is very useful feature to define for example XML style comments:

```
'(<!\-\-)[.^(\-\->)]*(\-\->)' XmlComment, %ignore;
```

In this example lexer will first search for a sequence of characters `'<!--'` that match the first group enclosed in parentheses. After this lexer will analyze and skip each following character

if this character is not the first one in the sequence `'-->'` Note that individual characters from this sequence will be skipped if they *do not form exact sequence* (for instance are separated by a space or some other character). This what is happening in the group enclosed in the brackets: `[.^(\-\->)]*.` But if the exact sequence is detected – lexer will advance to the final group enclosed in parentheses, process the input and complete the token.

`|` logical OR. Matches character or group on the left or right side.
example: `'a | b'` matches `'a'` or `'b'`
example: `'(abc)|(qwer)'` matches `'abc'` or `'qwer'`

`.` dot. Matches *any* character.
example: `'a.b'` matches `'adb'` and `'a7b'` and `'amb'` and `'aab'`…

`*` zero or more repetitions of character or group:
example: `'a*'` can be nothing, or `'a'` or `'aa'` or `'aaa'` or...

`+` one or more repetitions of character or group
example: `'(abc)+'` - matches `'abc'` or `'abcabc'` or `'abcabcabc'` ...

`?` zero or one repetition of character or group
example: `'a?'` matches nothing or `'a'`

`{`*num1,num2*`}` repetition range. Defines repetitions from *'num1'* to *'num2'*
example: `'a{2,3}'` matches `'aa'` or `'aaa'`

`{`*ref*`}` macro reference. Note, that the opening and closing curly `'{' '}'` are included.
example: `'{num}'` will be translated to `'0-9'` if there is a following macro definition: `num '0-9' ;`

`-` defines character range. Accepts only characters on both sides.
example: `'0-9'` matches any char in the range from `'0'` to `'9'` Note: `'0'` and `'9'` are in the range as well.
example: `'a-z'` matches any char in the range from `'a'` to `'z'`

`'` single quote. Special case.

`\` back slash. Special case.

Formatting characters are used to define format string. If there is a need to use any of them as characters then the back slash should be prepended.
example: `'ab\(rrr\)'` - matches the string `'ab(rrr)'`
example: `'c+\\.'` matches `'cc\a'` or `'cccc\7'` or...
example: `'abc\'drf'` matches `'abc'def'`
example: `'[0-9]\.[0-9]+'` matches `'1.2'` or `'3.1415'` or `'0.567'` or …

It is also legal to use the following control characters: **\n \t \r \b \v\a\f** and space character.
example: `'[ \t\r\n\f\b]+'` defines a string of one or many "white space" characters .

Sometimes there is a need to represent characters by their hexadecimal values. This can be done by using `'\x'` prefix. Example: the white space can be presented as `' '` and as `'\x20'`. One hexadecimal value is limited to maximum of four hexadecimal characters. In other words `'\xFFFF'` is the limit. Note, that the expression like this `'\xFFFFA'` will be treated as a sequence of two characters the second of which is a character `'A'`.

In some cases token declaration can have quite significant length (see XML grammar as example) and do not feed in the width of the screen. In situations like this it is perfectly valid to split the format string into two or more lines. In this case the following rule applies: all white space characters before and after line breaks will be ignored. Example: line `'abcde f'` will be equal to the following two lines

```
'abc
de f'
```

However the line `'abc de f'` spited in the above way will loose the first space but keep the second.

3.3.2 Precedence section

In optional <precedence section> declarations for operator precedence allow you to specify when to shift and when to reduce. Use the `%left', `%right' or `%nonassoc' declaration to declare a token and specify its precedence and associativity, all at once. The syntax of a precedence declaration is as follows:

```
%left symbols... ;
```
or
```
%right symbols... ;
```

. **UltraGram** supports **%left**, **%right** and **%nonassoc**, precedence declarations that can only be used once for a given token. The associativity of an operator **OP** determines how repeated uses of the operator nest: whether `X **OP** Y **OP** Z' is parsed by grouping X with Y first or by grouping Y with Z first. **`%left'** specifies left-associativity (grouping X with Y first) and **`%right'** specifies right-associativity (grouping Y with Z first). **`%nonassoc'** specifies no associativity, which means that `X **OP** Y **OP** Z' is considered a syntax error.

The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.
Example:

```
...
%left       '+' '-';
%left       '*' '/';
%right      '**';
%left       UMINUS;
```

In many cases the precedence of a token depends on the context. For example, a minus sign normally has a high precedence as a unary operator, and a lower precedence as a binary operator. **UltraGram** supports **%left**, **%right** and **%nonassoc**, precedence declarations that can only be used once for a given token. For context-dependent precedence, there is an additional mechanism: **%prec** modifier.

Note, that precedence of the tokens at the bottom of the list is higher then precedence of the tokens at the top. In this sample the highest precedence will have the token **UMINUS** and the lowest the tokens **'+'** and **'-'**.

Not all tokens listed in the precedence section must have formal declarations. For example the token **UMINUS** in this section may not de declared, and as result will refer only to precedence level. The precedence of **UMINUS** can be used in specific rules:

```
exp:    ...
      | exp '+' exp
      ...
      | '-' exp %prec UMINUS ;
```

3.4 Production section

Production section starts with keyword `%production` followed by the start rule name. After that rule declarations follow. Each rule has the hollowing format:

```
name   :    rule_components   ;
```

where:

name is the nonterminal symbol that this rule describes and each
rule_component is an optional list of terminal or nonterminal symbol related to this rule and separated by a whitespace.

Example : `exp : exp '-' exp ;`

This rule defines that two groupings of type “exp”, with a ’-‘ token in between, can be combined into a larger grouping of type “exp”.

Multiple rules for the same **name** can be written separately or can be joined with the vertical-bar character `|' as follows:

```
name:    rule_components1
               |    rule_components2
              … ;
```

Example : `exp : exp '-' exp | exp '+' exp ;`

If rule components in a rule is empty, it means that **name** can match the empty string. Example : here is how to define a comma-separated sequence of zero or more `exp' groupings:

```
exp_opt   :    // empty rule
          |  expseq     ;

    expseq   :  exp    | expseq   ','   exp    ;
```

The entry point (main rule) is defined by a name that follows the **%production** keyword. There must always be a rule with the name that corresponds to it.

3.4.1 %text token

The **%text** token it a powerful feature that allows creation of simplified versions of complex grammars or creation of grammars for files with partially unknown format. This token allows to skip the part of the input text until the *first valid* sequence of tokens that follow the **%text** token will be detected. Example:

```
expEx :      %text id '=' exp ';'   ;
```

Sequence of tokens that follow the `%text` token is considered valid if the rule where this token resides can be successfully reduced. So, in the example above some text from the input will be skipped (and assigned to the **%text** token) until the sequence that matches `id '=' exp ';'` can be found and entire rule reduced.

As result of this the **%text** token will have the minimal possible length, however this length will not be less then 1 character. So, if for example there is a rule:

```
expEx :     %text %text id '=' exp ';'  ;
```

then the first **%text** token will always have length of 1 character.

Note, that the usage of the **%text** token due to additional data processing can in some cases (that depend on particular grammar) slowdown the parser performance.

3.4.2 %error token

One of the tokens that can be used in productions is an **%error** token. This token notifies parser the there can be a possible error in the input text. The usage and behavior of this token is similar to the **%text** token. However there are some differences. First of all **%error** token has the lowest priority and its processing will start only after all other options will fail. The second and in fact the major difference is that **%error** token does not consider **%ignore** tokens. This means that in case of **%error** token parser expects the worst scenario (like damaged file) and will not make any difference between normal formatted text or some comments residing in it. Parser will just try to skip one char after another (and assign them to the **%error** token) until parsing of the input can be successfully resumed. This behavior allows to handle errors in grammars that already utilize **%text** tokens. Here is the fragment of the above grammar with generic error handling:

Example:

```
expEx :     id '=' exp ';'
            |   %error ';' ;
```

3.5 Typo Error correction

Along with error correction mechanism based on the **%error** token, there is another way of handling errors. This mechanism is targets a specific class of typographical errors – errors that in many cases result from manual text typing and include duplication, omission, transposition or substitution of characters. This type of error correction can be enabled only for token declarations that have fully defined pattern – do not contain wildcards and have fixed length (in other words look like keyword tokens). Here are examples of such declarations:

```
'Doctor'                              doc;
'[Ee][Nn][Gg][Ii][Nn][Ee][Ee][Rr]'    eng ;
'(begin)|(BEGIN)'                     begin;
```

When UltraGram parses input text it can run into the case when none of the defined tokens can be found due to a typo error in the input. For example if expected tokens are **doc** and **eng** (from the sample above) and input text from the current location contains: "*Doktor…*" (note the spelling error) parser will fail (unless recovery with **%error** token is utilized). To enable handling of typo errors an additional "valid char set" token should be declared and attached to appropriate token declaration. "Valid char set" token declaration is a regular token that describes characters that might accidentally typed into the input text. Example:

```
'[a-zA-Z]'          MyValidCharSet;
```

This declared a token that will match any char from a to z in lower or upper case. The length of this token must not exceed one character. After this "valid char set" token can be attached to some other token using keyword **%chars** in the following way:

```
'Doctor'  doc, %chars MyValidCharSet ;
```

This tells to the parser that if exact match vas not found, then it is allowed to try variants when one of expected characters is substituted with a character that corresponds to the **MyValidCharSet** token. If this will fail, attempt is made to try variants with a single inserted character (example: "*Docvtor…"*). If this will fail a check for variants with missing characters will be made (example: "*Dotor…"*). If this will fail a check for variants with swapped characters will be (example: "*Dotcor…"*). If one of the variants listed above will succeed – token will be consumed and parsing will continue in normal way.

There is a mechanism that allows to enable this type of error recovery on tokens that do not have explicit declaration and are declared inline. To attach "valid char set" token to all inline tokens the following form of the **intok** pragma should be used:

```
intok(on, on|off, valid_char_set)
```

Here:

valid_char_set is a name of token declared must be in the **%tokens** section and that will be attached to all inline tokens for typo error correction.

It is possible to declare multiple "valid char set" tokens and attach them to explicit token declarations as necessary.

3.6 UNICODE support

When a new project is created it is possible to select required encoding including **UNICODE UCS-2**. Files in the **UNICODE** format sometimes have "Byte Order Mark" (BOM) that is placed in the first character of a file or character stream to indicate the endianness (byte order) of all the 16-bit code units of the file. When this type of a file is opened in the **UltraGram IDE** the BOM is handled automatically. However if a grammar that is under development does not have explicit handling of BOM then the generated parser (source code) will not have BOM support also. This may result in the situation when some test files can be perfectly parsed in the IDE but will fail or produce unexpected results when parsed with the generated parser (complied from generated source code). To avoid this situation it is recommended to include BOM handling in the grammar file. To do this the grammar file should be extended with the following fragments:

```
%tokens
'\xFEFF'  BOM; // "Byte Order Mark" for UNICODE UCS-2 LE


... some other tokens follow here...

%production Start
Start : BOM_opt some_rule ...
BOM_opt : | BOM;

... some other rules follow here  ...
```

Now grammar will correctly process BOM (if any) and the rest of the file.

3.7 How to resolve Shift-Reduce and Reduce-Reduce conflicts

Shift-Reduce and **Reduce-Reduce** conflicts happen when during construction of the parsing tables algorithm runs in the situation when more then one action could be taken. These situations occur as result the way grammar is defined or a paring algorithm that is used (generally **LALR(1)** parsing algorithm tends to produce these conflicts comparing to **LR(1)** algorithm). By default UltraGram always will behave in certain way (described below) when conflict happens. However default behavior might not be a desired one. In general the best thing to do is to eliminate conflict by modifying the grammar file. If this can not be done then switching from **LALR(1)** to **LR(1)** algorithm might help (note, that size of the parsing tables will significantly increase). If conflicts still exist then the following options might be taken:

- in case of SR conflicts parser should be explicitly instructed what to do (see below)
- in case of RR conflicts there are two options:
    a) parser should be explicitly instructed what to do (similar to SR case, see below)
    b) parser should be allowed to detect a successful behavior at runtime. This can be done by enabling GLR parsing algorithm (see Pragma section for details)

If a **SR** or **RR** conflict are detected during parsing of the grammar, **UltraGram** prints all information about them in the **Parser Task List** window. After clicking on the single line in this window a new tab window is opened where two sequences of transitions that create this conflict are displayed. These sequences always start from one point and go trough identical transitions however they end up in different states and this is what makes parser confused since it does not know which path to choose. By analyzing these sequences it is possible to understand how conflict is created. In some cases SR or RR conflicts can be eliminated in natural way just by modifying the grammar. In some cases conflicts can be eliminated by selecting LR(1) table construction algorithm and / or using the **`glr`** parsing (see paragraph 3.6. note, that glr does not eliminate conflict – it just chooses a successful parsing path between all conflicting). However in other cases the best thing to do is to resolve them using technique with setting precedence of tokens and rules. Here is how it can be done. Lets say this transitions and the text of the SR message look in the following way (Fig.16):

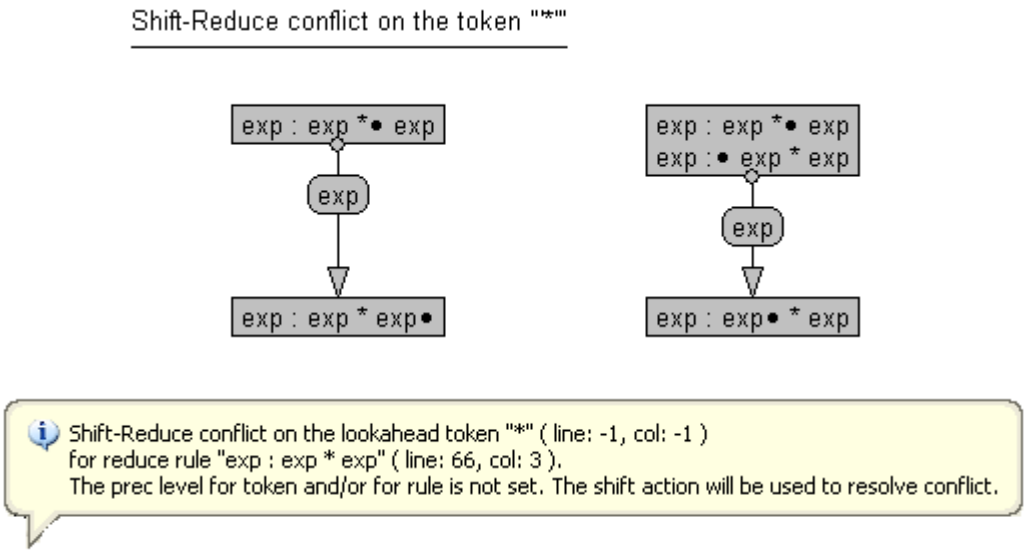

Fig. 16

As the text states conflict happens on the lookahead token **`'*'`** for the reduce rule **`exp : exp * exp`**

In general case to resolve this conflict precedence should be defined for both : for the lookahead token and for the reduce rule. The precedence of the reduce rule (if not set explicitly using **%prec** modifier) will be calculated based on the precedence of the last token in that rule. ***Note, that by default precedence of the tokens is not defined. It is assigned only after token is listed explicitly in the precedence section.*** As result of all this the following will happen:

1) If precedence of the lookahead token ***or*** precedence of the reduce rule is undefined then conflict is not resolved and default action **SHIFT** will take place
2) If precedence of the token ***and*** precedence of reduce rule are defined then:
   - If lookahead token has higher precedence **SHIFT** will take place.
   - If reduce rule has higher precedence then **REDUCE**.
   - It lookahead token and reduce rule precedence are equal then: if token is right associative (**%right**) then **SHIFT** otherwise **REDUCE**.

In the sample above (**Fig. 16**)the last token in the reduce rule is the same token as a lookahead. So, the following line that sets precedence and associativity will resolve it to **REDUCE**:

```
%left       '*';
```

(See “Calculator” grammar as a sample)

Sometimes the reduce rule precedence can not be calculated and should be set explicitly. To do this use fictitious tokens (as **UMINUS** in the sample in **3.3.2 Precedence section**)

Reduce-Reduce conflicts can be resolved using same technique with fictitious tokens.

4 Code generation

After grammar is successfully compiled (and tested) it is possible to generate a source code for selected programming language. Generally **UltraGram** creates a file with a main parser class and several support files with different tables.

4.1 C++ code generation

This process generates 5 files. All file names depend on the name of the parser class (see Fig. 14). For example: if the parser class name is set to `'CalkScript'` then generated files will be:

`CalkScript.h`	- header file with the main parser class
`CalkScript.cpp`	- source file with main parser class implementation
`CalkScriptConst.h`	- file with some constants definitions
`CalkScriptDataTable.h`	- header file with parser tables
`CalkScriptDataTable.cpp`	- source file with parser tables

The main parser class is derived from the `ParserBase` class that is located in the corresponding file in the “*Include*” folder of the installation folder (for the case of C++ source code). Methods of the main parser class override virtual methods of the base class. As result of this it is possible to modify the main parser class directly in order to achieve required functioning. The following methods of the parser class should be considered.

Public methods:

```
virtual void SetText( LPCTSTR pText );
```

Method sets the text that should be parsed

```
LPCTSTR GetText();
```

Returns the pointer to the text that was set by the **SetText** method

---

```
virtual bool Parse();
```

Parses the text.

Return value. If parsing is successful (does not have any unrecoverable errors) this method will return **true**, otherwise it will return **false**.

Protected methods:

```
virtual bool UserAbort();
```

While parsing is in progress, this method is checked frequently from the main parser loop to determine whether parsing should be aborted.
Return value. By default this method returns **false**. Modify this method and make it return **true** if there is a need to abort at any given time.

```
virtual void OnAcceptToken( int nTokenIndex, const TextInfo & ti );
```

This method is called by the parser each time a new token is accepted from the lexer.

Parameters:

int nTokenIndex index of the token. Note that using this index it possible to obtain the name and alias of the token in the way they are defined in grammar file. To do that use this index to retrieve this information from the array retuned by the function:

```
TOKNAME * Get...TokName();
```

Here '...' will be the name of the parser class. For the class name **'CalkScript'** in the example above, the function will be:

```
TOKNAME * GetCalkScriptTokName();
```

To get the name and alias of the token something like this can be used:

```
...
TOKNAME * pArray = GetCalkScriptTokName();
TOKNAME & tn = pArray[ nTokenIndex ];
...
```

TextInfo & ti describes location of the token in the parsed text.

```
virtual bool OnErrorStop( const TErrorInfo & ei, bool bCanRecover );
```

This method is called when parses encounters an error in the parsed text.

Parameters:

TErrorInfo ei contains information about location of the error in the parsed text, start position of the error token and error token length (if parser can recover from the error and assign a sequence of characters to the **%error** token) and a pointer to array of lookahead tokens (token indexes) that parser was looking for at the error point.

bool bCanRecover indicates whether parser can recover from this error or not.

Return value.
If this function return **true** then parsing will be stopped. If **false** parsing will continue. Note, that if the value of **bCanRecover** parameter is **false** – parsing will stop regardless of the function return value.

```
virtual StackData * OnReduce( int nProductionIndex );
```

This method is called each time parser is ready to proceed with reduce action. At this moment all other operation (like getting a lookahead or shifting token to the parser stack) are completed and parser is ready to remove items from the stack and push a new item (reduce operation).

Parameters:

int nProductionIndex is an index of the production that will be reduced.

Return value.
The return value of the function supposed to carry specific user data. This value will be associated with the reduced rule that will be pushed on the parser stack.

```
virtual StackData * GetProductionItemData( int nProductionItemIndex );
```

Method returns pointer to the **StackData** associated with required production item.

Parameters:

int nProductionItemIndex index of required production item. Note, that this index starts from 0 and represents items in the current production. Example: in the fragment of the code below from some **'OnReduce'** function the commented line after the **case** operator represents the rule in a way it was defined in the grammar file.

```
...
case Assignment_85BE:
// Assignment : Identifier = exp ;
...
```

For production items of this rule:

'Identifier' will have index 0
'=' will have index 1
'exp' will have index 2
';' at the end will have index 3.

Note, that the rule name (`'Assignment'` in this sample) and the following colon `':'` must be ignored in the process of index calculation.

```
virtual bool GetProductionItemInfo( int nProductionItemIndex, TextInfo & ti );
```

This method can be called from the overridden **`OnReduce`** method in order to obtain information about location and length of particular item on the parser stack.

Parameters:
`int nProductionItemIndex` index of an item on the parser stack. Index is calculated using same convention as in the function **`GetProductionItemData`**.

`TextInfo & ti` after function returns this parameter will contain information about the start position in the text and the length of the production item specified by **`nProductionItemIndex`**

Return value. **`true`** if success **`false`** otherwise.

Some of these methods take as parameters variables of the following types:

```
struct TextPos
{
      int nOffs;
      int nLine;
      int nCol;

      bool Equals( const TextPos & right );
      void SetToZero();
};


struct TextInfo
{
      TextPos     startPos;
      int         nLength;
      void Reset();
};


struct TErrorInfo
{
      enum Type
      {
            MISSING_TOKEN,
            CONFLICTING_TOKENS,
            INCOMPLETE_TOKEN
      };
      Type        errorType;
      TextPos     errorPos;
      TextInfo    errorTokenInfo;
      int         nLookAheadsCount;
      int *       pLookAheadTokenIndices;
};
```

Structures **`TextPos`** and **`TextInfo`** are self explanatory and describe position in the text and token position and length in the text accordingly.

Structure **TErrorInfo** provides information when some error occurred. Fields of it have the following meaning:

errorType – specifies the nature of an error. If the value is set to **MISSING_TOKEN** then it means that the parser failed to find required token at current position. If the value is set to **CONFLICTING_TOKENS** then parser detected more then one valid token of the same length at the current position. . If the value is set to **INCOMPLETE_TOKEN** then parser can not complete **%text** token (valid sequence to tokens that follow **%text** token can not be found)

errorPos – specifies position in text where an error occurred.

errorTokenInfo - specify the parameters of the error token constructed by the parser. This fields of this structure are valid if parser encountered a recoverable error and the error token with this parameters may be accepted. Note, that the start offset of the error token does not necessarily point to the offset specified in the **errorPos** field (see above). Often (but of course depending on the grammar) the error token will have smaller offset then offset where the error happened and greater length then expected but missing token.

pLookAheadTokenIndices – points to array of token indices. Tokens with these indices are valid / expected at the current text position. These are lookaheads that parser is looking for. *NOTE: this pointer is valid only while execution is inside of the* **OnErrorStop***(…) function. If there is a need to save this information for future use – a independent storage should be allocated and the contents of array should be copied to it.*

nLookAheadsCount – specifies size of array pointed by **pLookAheadTokenIndices**.

4.2 .NET code generation (VB, C#)

This process generates 2 files. File names depend on the name of the parser class (see Fig. 14). For example: if the parser class name is set to **'CalkScript'** then generated for C# files will be:

CalkScript.cs	- source file with main parser class implementation
CalkScriptDataTable.cs	- source file with helper class and parser tables

The main parser class is located in the first file. The second file contains a helper class with some data required for parser initialization. The name of the helper class also corresponds to the name of the file.

The main parser class is derived from the class **ParserBase** that is located in the **Ust.Parser.dll** assembly in the namespace **Ust.Parser**. Methods of the main parser class override virtual methods of the base class. As result of this it is possible to modify the main parser class directly in order to achieve required functioning. The following methods of the base parser class should be considered.

Public methods and properties:

```
virtual string Text { set; get; }
```

Property to set / get the text that should be parsed

```
virtual bool Parse()
```

Parses the text.

Return value. If parsing is successful (does not have any unrecoverable errors) this method will return **`true`**, otherwise it will return **`false`**.

Protected methods:

`virtual bool UserAbort()`

While parsing is in progress, this method is checked frequently from the main parser loop to determine whether parsing should be aborted.
Return value. By default this method returns **`false`**. Override this method and make it return **`true`** when there is a need to abort parsing.

`virtual void OnAcceptToken(int` ***`nTokenIndex`*****`, Ust.Parser.TextInfo`** ***`ti`*****`)`**

This method is called by the parser each time a new token is reseived from the lexer.

Parameters:
`int nTokenIndex` index of the token. Note that using this index it possible to obtain the name and alias of the token in the way they are defined in grammar file. To do that use this index to retrieve this information from the array retuned by the function:

```
public static TOKNAME [] TokName()
```

This function is located in the helper class **`'...'DataTable`** Here **`'...'`** will be the name of the parser class. For the class name **`'CalkScript'`** in the example above, the class name will be will be:

```
public class CalculatorDataTable
```

To get the name and alias of the token something like this can be used:

```
TOKNAME [] arr = TokName();
TOKNAME tn = arr[ nTokenIndex ];
```

`TextInfo ti` describes location and length of the token in the parsed text.

`virtual bool OnErrorStop(Ust.Parser.TErrorInfo` ***`ei`*****`, bool`** ***`bCanRecover`*****`)`**

This method is called when parses encounters an error in the parsed text.

Parameters:
`TErrorInfo ei` contains information about location of the error in the parsed text, start position of the error token and error token length (if parser can recover from the error and assign a sequence of characters to the **`%error`** token) and a reference to array of lookahead tokens (token indices) that parser was looking for at the error point.

`bool bCanRecover` indicates whether parser can recover from this error or not.

Return value.
If this function returns **true** then parsing will be stopped. If **false** parsing will continue. Note, that if the value of **bCanRecover** parameter is **false** – parsing will stop regardless of the function return value.

---

virtual Ust.Parser.StackData OnReduce(int *nProductionIndex*)

This method is called each time parser is ready to proceed with reduce action. At this moment all other operation (like getting a lookahead or shifting token to the parser stack) are completed and parser is ready to remove items from the stack and push a new item (reduce operation).

Parameters:
int nProductionIndex is an index of the production that will be reduced.

Return value.
The return value of the function supposed to carry specific user data. This value will be associated with the reduced rule that will be pushed on the parser stack.

---

virtual Ust.Parser.StackData GetProductionItemData(int *nProductionItemIndex*)

Method returns reference r to the **StackData** associated with required production item.

Parameters:
int nProductionItemIndex index of required production item. Note, that this index starts from 0 and represents items in the current production. Example: in the fragment of the code below from some **'OnReduce'** function the commented line after the **case** operator represents the rule in a way it was defined in the grammar file.

```
...
case Assignment_85BE:
// Assignment : Identifier = exp ;
...
```

For production items of this rule (that follow the colon):

'Identifier' will have index 0
'=' will have index 1
'exp' will have index 2
';' at the end will have index 3.

Note, that the rule name (**'Assignment'** in this sample) and the following colon **':'** must be ignored in the process of index calculation.

---

virtual bool GetProductionItemInfo(int *nProductionItemIndex*,
Ust.Parser.TextInfo ti)

This method can be called from the overridden **OnReduce** method in order to obtain information about location and length of particular item on the parser stack.

Parameters:
int nProductionItemIndex index of an item on the parser stack. Index is calculated using same convention as in the function **GetProductionItemData.**

TextInfo ti after function returns this parameter will contain information about the start position in the text and the length of the production item specified by **nProductionItemIndex**
Note, that **ti** cannot be **null** when this method is called.

Return value. **true** if success **false** otherwise.

---

virtual void Reset()

This method resets internal variables of the parser including lexer and parser stack. After a call of this method parsing of the text will be started from the very beginning.

Some of these methods take as parameters variables of the following types:

```
struct TextPos
{
      int nOffs;
      int nLine;
      int nCol;

      bool Equals( const TextPos & right );
      void SetToZero();
};


struct TextInfo
{
      TextPos     startPos;
      int         nLength;
      void Reset();
};



struct TErrorInfo
{
      enum Type
      {
            MISSING_TOKEN,
            CONFLICTING_TOKENS,
            INCOMPLETE_TOKEN
      };
      Type        errorType;
      TextPos     errorPos;
      TextInfo    errorTokenInfo;
      int []      pLookAheadTokenIndices;
};
```

Structures **TextPos** and **TextInfo** are self explanatory and describe position in the text and token position and length in the text accordingly.

Structure **TErrorInfo** provides information when some error occurred. Fields of it have the following meaning:

errorType – specifies the nature of an error. If the value is set to **MISSING_TOKEN** then it means that the parser failed to find required token at current position. If the value is set to **CONFLICTING_TOKENS** then parser detected more then one valid token of the same length at the current position. . If the value

is set to **INCOMPLETE_TOKEN** then parser can not complete **%text** token (valid sequence to tokens that follow **%text** token can not be found)

errorPos – specifies position in text where an error occurred.

errorTokenInfo - specify the parameters of the error token constructed by the parser. This fields of this structure are valid if parser encountered a recoverable error and the error token with this parameters may be accepted. Note, that the start offset of the error token does not necessarily point to the offset specified in the **errPos** field (see above). Often (but of course depending on the grammar) the error token will have smaller offset then the offset where the error happened and greater length then expected but missing token.

pLookAheadTokenIndices – reference to array of token indices. Tokens with these indices are valid / expected at the current text position. These are lookaheads that parser is looking for.

4.3 How to use generated source code

Libraries

To create a software application using generated code a parsing kernel is required. Depending on the **UltraGram** packaging the parsing kernel can be available as:

- a source code that can be added to the target application and compiled all together
- a library file that must be linked or references from the target application. The following libraries containing parser kernel are supplied :

  ustp.dll, ustpu.dll – ANSI and UNICODE versions of the kernel. By default these files are installed in the Windows **System32** folder.
  ustp.lib, ustpu.lib – ANSI and UNICODE lib files. Located in the Lib subfolder of the installation folder.

  ustp.jar – Java archive file located in the installation folder.

  Ust.Parser.dll - **.NET 2.0** assembly located in the **Assemblies** subfolder of the installation folder. During installation this assembly is also registered in **GAC**.

Note, that the C++ libraries are compiled using **MS Visual Studio 8** with a compiler option **"Treat wchar_t as Build-in Type"** set to **Yes**.

Generated code

Generally to make parser to perform some useful work two types of steps should be taken:

1. Collection of values of required tokens during parsing and storing them (or storing some related secondary objects).
2. In particular points of the parsing process (in fact in points where particular productions are to be reduced) retrieving objects stored in step 1 and using them as parameters for some activity (like a function calls or for creation of some other dependent objects, that in their turn can also be stored for further use)

Implementation of these steps will be explained base on the sample.

As it was mentioned above **UltraGram** generated 5 files for C++ language. From example above these file are:

CalkScript.h	- header file with the main parser class
CalkScript.cpp	- source file with main parser class implementation
CalkScriptConst.h	- file with some constants definitions
CalkScriptDataTable.h	- header file with parser tables
CalkScriptDataTable.cpp	- source file with parser tables

First two files contain definition and implementation of the main parser class. These files should be modified directly to achieve specific parsing goals.

Starting the parsing process is very simple – after the instance of the parser class is created, method **SetText** should be called to specify the text to parse and then parsing should be initiated by calling **Parse** method. While parsing is in progress, protected method **OnReduce** will be called multiple times. Inside of this method there is a **switch-case** statement. Each time parser is ready to reduce some production – this method will be called and execution will enter particular **case** statement. Values of particular items from corresponding production can be retrieved using **GetProductionItemInfo** method. Value in this case mean a section of the input text (text that is parsed) that correspond to the single production item. In most cases these values should be stored for further use. This can be done by using the **UserStackData** class that is derived from class **StackData**. The **UserStackData** is class is declared in the file containing main parser class. This class should be equipped with one or more fields that will carry required values or references or pointers to some object that represent these values. In most cases the instance of the class **UserStackData** should be created each time execution enters **OnReduce** method and should be returned by this method. Parser will take the returned pointer (reference in case of .NET) and attach it to the rule that will be pushed to the parser stack after production is reduced. When execution will enter the **OnReduce** method next time, instances of the **UserStackData** class (that were previously stored) can be extracted from the items of production that will be reduced by using **GetProductionItemData** method. After some processing (if necessary) new instance of **UserStackData** can be created and returned by **OnReduce** method and so on and so on…Note, that the class **StackData** (and **UserStackData** as well) is equipped with the method **FinalRelease**. Parser will call this method for each item of a production when this production is reduced. This allows to perform a cleanup and automatically delete instances of **UserStackData** from the heap and may be perform some additional cleanup activities.

The helper file **CalkScriptDataTable.*** declares data tables that describe the source grammar and are used in during parser initialization. Additionally to this there is a useful function that returns an array of structures of a **TOKNAME** type. This structure carries token name and alias in the way they were defined in the grammar file. This information can be very useful in case parser encounters an error in the parsed text and there is a need to communicate to the caller application / user what was the set of expected token.

Samples of the source code can be found in the subfolder **\Samples**.