ACTA UNIVERSITATIS AGRICULTURAE ET SILVICULTURAE MENDELIANAE BRUNENSIS

Volume LIX 13 Number 7, 2011

DESIGN OF METHODOLOGY FOR INCREMENTAL COMPILER CONSTRUCTION

P. Haluza, J. Rybička

Received: August 31, 2011

Abstract

HALUZA, P., RYBIČKA, J.: *Design of methodology for incremental compiler construction.* Acta univ. agric. et silvic. Mendel. Brun., 2011, LIX, No. 7, pp. 137–146

The paper deals with possibilities of the incremental compiler construction. It represents the compiler construction possibilities for languages with a fixed set of lexical units and for languages with a variable set of lexical units, too. The methodology design for the incremental compiler construction is based on the known algorithms for standard compiler construction and derived for both groups of languages. Under the group of languages with a fixed set of lexical units there belong languages, where each lexical unit has its constant meaning, e.g., common programming languages. For this group of languages the paper tries to solve the problem of the incremental semantic analysis, which is based on incremental parsing. In the group of languages with a variable set of lexical units (e.g., professional typographic system TEX), it is possible to change arbitrarily the meaning of each character on the input file at any time during processing. The change takes effect immediately and its validity can be somehow limited or is given by the end of the input. For this group of languages this paper tries to solve the problem case when we use macros temporarily changing the category of arbitrary characters.

parser, incremental analysis, compiler, programming language, TEX

The analysis of text information belongs to the everyday routine of a modern computer scientist. Although today it is no longer a problem to work with a powerful hardware equipment, there is still room for improvement. A typical example of the operation with high demands on the speed of implementation needed is the source code compilation in any programming language. The only slight change in a high volume code can but need not mean a complete change of program activities. It depends on where it has been made. For example, a text change in the notes does not make any effect in the program activities and a new compilation following this change is basically useless. On the contrary, the variable name change requires the compilation of almost the entire code because it can cause a chain of error messages.

In most cases, adjustments are somewhere between these extremes, and therefore it is not always necessary to compile the entire source code, which is often composed of thousands of lines, again. The idea of the source code compilation in an incremental manner is based on the findings of interdependencies between different parts of the code and the subsequent reprocessing of only those parts that are directly affected by the performed change. It is obvious that the efficiency of the compilation incremental method rises with the increase in the code size.

The aim of this paper is to outline the possibilities of incremental compiler implementation both for common programming languages having constant set of lexical symbols and also for specific types of languages having variable set of lexical symbols. The typical representative of the second group of programming languages is for example the professional typographic system TEX.

Although the problem of the incremental compilation of common programming languages (or at least an incremental approach to some phases of the compilation) is resolved today, for the group of languages with a variable set of lexical units

this problem has not been solved yet. This paper therefore tries to discuss options and solutions in this so far insufficiently explored area.

Overview of literature and the present state

The implementation of an incremental compiler in the early phases is not different from the common one and is based on a sophisticated theory of formal languages and compilers dating back to the 1950s. For the study of formal languages one can use a number of now almost historic and predominantly foreign publications, which are in Czech supplemented by college textbooks of different quality levels.

To understand the principles of the theory of formal languages for the purpose of practical implementation of the compiler, publications from Hopcroft and Ullman (1978) and Molnár, Češka and Melichar (1987) are fully sufficient. From more comprehensive resources the first volume of the three-part set of posts (Rozenberg and Salomaa, 1997) can be recommended. In all the above mentioned sources can be found basic knowledge of the theory of formal languages and at least a theoretical procedure of the compilers construction of conventional programming languages. From the publications of Czech authors (though written in English) Meduna (2000) can be recommended.

Incremental compilation is widely spread in text editors with syntax highlighting and in environments of incremental compilers and interpreters. In fact, the term "incremental compilation" mostly means only an incremental approach to parsing, as one of the key stages of the compilation itself. Such incremental parser solves the problem of reconstruction of a parse tree, after the string is changed from xyz to $x\tilde{y}z$. If we have parse tree T for the string xyz generated by context-free grammar G and a string $\tilde{y}$, incremental parser is trying to create a new parse tree $\tilde{T}$ for the string $x\tilde{y}z$ using the minimum number of steps.

To use incremental compilation, we have to know the results from the previous compilation, including information about the state of the stack at each compilation step. Computational complexity of the incremental compiler to a large extent depends on the chosen implementation method of the compilation and on data structure for storing information necessary for the new run of the compiler. There have been suggested many methods for incremental analysis. Algorithms of incremental compilation of context-free language class LL (Yang, 1993; Li, 1995; Li, 1996; Shilling, 1992; Murching, Prasad and Srikant, 1990; Ferro and Dion, 1994) and LR (Yang, 1994; Agrawal and Detrich, 1983; Tomita, 1987; Horspool, 1990; Wagner and Graham, 1997; Wagner and Graham, 1998; Wagner, 1998) have been published. Melichar and Vagner (2008) published an elementary text about how to create an incremental parser of LL(1) languages as a part of educational materials in English, based on previous publications of Li (1995), but suitably accompanied by examples and illustrations. In the Czech language similar publications are still missing.

All mentioned publications deal with incremental parsing capabilities for current programming languages, i.e. languages with a fixed set of lexical units. In the case of a group of languages with a variable set of lexical units, the situation is in some respect worse. For example from the above mentioned representative of this group, the typographic system TEX, it is to some extent due to the fact that the process of compiling of the source code in TEX is understood and implemented in an entirely different manner than in conventional languages. The algorithms used by TEX are described by Olšák (2001).

MATERIALS AND METHODS

As already noted, the problem of an incremental compilation is to find a tree node from which there came to a change in comparison with the previous run of the compiler, and the substitution of a subtree by a new subtree based on the current situation on the input. But the reconstruction of the tree is only half of the final solution.

Imagine the following situation. Consider the source code in the following form:

```
var a: integer;
begin
  read(a);
  if a mod 2 = 0 then write(a,' is even number');
end.
```

Now we make a small change, and we get a slightly changed code in the following form:

```
var a: char;
begin
  read(a);
  if a mod 2 = 0 then write(a,' is even number');
end.
```

It is obvious that the reconstruction of the tree is no problem, because there was only an exchange of the contents of one node. The problem occurs when we try to compile, because the mistake will be its result. This simple example clearly shows that for a truly incremental compilation, including incremental semantic analysis, we must solve two problems:

1. to find a node in the tree and replace the modified subtree;
2. to find all places in the code that may be affected by this change.

In other words, both syntactic and semantic analysis must be incremental and both phases must be interconnected mutually.

In our case there was a change in the declaration of a variable caused by changing the data type of a variable from integer to character. Along with the verification of the fact that the new code is syntactically correct, there must follow the

verification that the new code is also semantically correct. Examining this example we can see that it is not correct, because the operation "modulo" with this character is not defined.

Let us now focus on the incremental parsing problems. Several times the languages with a fixed set of lexical units and languages with a variable set of lexical units have been mentioned. Now we will try to explain the used terminology.

Compilation of languages with a fixed set of lexical units

Under the group of languages with a fixed set of lexical units there belong languages, where each lexical unit has its constant meaning. For example, a sequence of characters "123" is always understood as a number.

It is obvious that this group of languages containts common programming and scripting languages that can be described by the context-free grammar. The standard approach to the compilers contruction of these languages includes three basic modules:

1. Lexical analyzer—reads characters from the input file and creates lexical units (tokens) from them. It keeps its type and content for each token, separators and comments are omitted. On each call the lexical analyzer returns one token.
2. Syntactic analyzer (parser)—gets the input sequence of tokens generated by the lexical analyzer and verifies the syntactic correctness of the input. The output is information about the syntactic structure (e.g. in the form of a parse tree), or an error message.
3. Semantic processor—creates the input for the output processing, performs controls of declarations, type controls, correct operands writing, etc., and generates an intermediate code.

The lexical analysis of common programming languages is purely a matter of routine. Incremental parsing uses the method described by Melichar and Vagner (2008) and it is known as interpretative recursive descent parsing (IRD). This method works with a modified FIRST and FOLLOW sets, which defines as follows:

$$FIRST'(\alpha) = \{X \in (N \cup \Sigma) \mid \alpha \Rightarrow^* X\beta\} \cup \{\epsilon \mid \alpha \Rightarrow * \epsilon\},$$

$$FOLLOW'(A) = \{X \in (N \cup \Sigma) \mid S \rightarrow^+ \gamma AX\beta\} \cup \{\epsilon \mid S \Rightarrow^* \gamma A\}.$$

The algorithm for the construction of parsing table M' containing nonterminals is similar to the algorithm for construction of the standard parsing table M.

For implementation purposes it is necessary to modify the original $LL(1)$ grammar G on an augmented $LL(1)$ grammar G' as follows:$G' = (N \cup \{S'\}, \Sigma \cup \{Ⓐ, \#\}, P', S')$, where $P' = \{S' \rightarrow SⒶ\} \cup \{B \rightarrow \beta\# \mid B \rightarrow \beta \in P\}$, $S' \notin N$ is the new starting symbol, Ⓐ, $\# \notin \Sigma$.

The symbol Ⓐ represents a symbol of acceptance, the symbol # is used as a symbol marking the end of the right side of grammar rule.

We assume at the same time, the right side of the grammar G' rule is stored in a data structure that has the following features:

- the right side of the rule is stored as a string;
- is it possible to point at any element of the structure.

IRD compiler uses a parsing table M' defined as a map $M' : N \times (\Sigma \cup \{\epsilon\}) \rightarrow \{X \mid X$ is a pointer at the beginning of the right side of the rule$\} \cup \{$error$\}$. The parsing table construction is the same as in the standard case. IRD compiler uses the actual pointer (AP), which indicates the specific symbol of the right side of the grammar rule. The symbol, which the pointer points to, is the actual pointer. For clarity, the sign with dot notation is used because it can clearly express the value of the actual pointer. If the actual pointer points to the symbol X in the rule $A \rightarrow \alpha X\beta$, where $X \in N \cup \Sigma \cup \{Ⓐ, \#\}$, its value is expressed as $A \rightarrow \alpha.X\beta$.

Then IRD compiler configuration is defined as the triad (x, γ, AP) where $x \in \Sigma^*$ is the still unread part of the input string, γ is the contents of stack and AP is the value of the actual pointer.

But there is another method of the syntactic analysis, based on the storage of stack content into the path from the actual symbol to root symbol in the parsing tree. This modified method of incremental parsing using recursive descent describes the following algorithm, the input of which is a parsing table M', augmented grammar G' and the input string w. (Melichar and Vagner, 2008)

1. Set the actual pointer to point to the symbol S at the right-hand side of the rule $S' \rightarrow SⒶ$.
2. Set $s := FIRST(w)$.
3. Repeat steps 4, 5, 6, 7 until accept or error appears.
4. Comparison: If the actual symbol is a terminal symbol and if it is the same symbol as the symbol s, advance the actual pointer to the next symbol, advance the input to the next symbol and set $s := FIRST(remainder\_of\_input)$. If the actual symbol is not the same symbol as s, then the result of the parsing is an error.
5. Expansion: If the actual symbol is a nonterminal symbol, set $r := M(actual\_symbol, s)$. Push the actual pointer into the pushdown store. If $r = error$ then the result of the parsing is an error, otherwise set $AP := r$.
6. End of rule: If the actual pointer is the symbol # (the end symbol of a rule), pop the actual pointer from the pushdown store and advance it to the next symbol.
7. Accept: If the actual symbol is Ⓐ (accept) terminate the parsing and the result of the parsing is yes.

Compilation of languages with a variable set of lexical units

Conventional programming languages can be included into the group of languages with a fixed set of lexical units. Each input symbol is during the lexical analysis correctly recognized and the output from the lexical analyzer is information about the lexical unit type. Significant there is the fact that each character is clearly assigned to a given lexical unit type, and this assignment is constant.

In the group of languages with a variable set of lexical units, it is possible to change arbitrarily the meaning of each character on the input file at any time during processing. The change takes effect immediately and its validity can be somehow limited or is given by the end of the input. An example might be the instruction causing that the sequence of characters "{abc}" and "1abc2" will have the same meaning for a given time, as we mark "1" meaning assigned to the left brace and the character "2" has assigned the meaning of the right brace. This system can be achieved easily using the TEX macro \catcode.

So under the term *a variable set of lexical units* we can imagine a set of input symbols assigned to the type of lexical units, but the assignment can be volatile during the compilation.

One of the representatives of a group of languages with a variable set of lexical units is the professional typographic system TEX, which greatly differs from conventional programming languages in its methods for analysis of the input text. The TEX activity is divided into individual processors, which have their own functions—the input processor, the token processor, the expand the processor and the main processor.

The input processor reads the file lines sequentially from the input lines of the text, adjusts them and gives output lines ready for the token processor. An output line of the text from the input processor is an internal data structure of TEX and this is the same in all implementations of TEX.

The token processor processes the input lines prepared by the input processor and its output is a sequence of lexical units—tokens. A token is either an ordered pair (ASCII code, category), or a control sequence. In all algorithms following after the token processor there are the input text characters not processed any more, but only tokens. A detailed description of each state of the token processor is described by Olšák (2001).

All implementations of TEX divide the input characters into categories. At any given moment, each character can be included in only one category and only once (Matoušek, 2001; Olšák, 2001). Categories are identified by an integer from the interval ⟨0, 15⟩. There is a total of 16 categories. Each category has its own meaning and default assigned characters. The categories of characters are shown in Tab. I. If the category description includes a star character, it means that this category has meaning only in the algorithms of the token processor and never appears in its output. If the category includes the word (plain), then this category is not assigned to the character set by default, but is set up in plain format (Olšák, 2001).

The follow-up part of the compiler is the expand processor. It provides macros expansion, which means their use. It interchanges the control sequence from the input, by which a macro is identified, with a sequence of tokens in the output. This sequence of tokens is stored in the memory in the macros learning phase, when the main processor has saved the body of the macro definition. Once all the tokens in the output queue processor are

I: *List of characters categories in system TEX (Olšák, 2001)*

category	value	default assignment
0*	escape sequence at the beginning	\
1	opening the group	{ (plain)
2	closing the group	} (plain)
3	math mode switch	$ (plain)
4	separator in the tables	& (plain)
5*	end of the line	^^M (ASCII 13)
6	macro parameters lable	# (plain)
7	power constructor	^ (plain)
8	index constructor	_ (plain)
9*	ignored character	^^@ (ASCII 0, plain)
10	space	
11	letter	A to Z, a to z
12	other characters	characters remaining
13	active characters	~, ^^L (plain)
14*	bracketed comments at the beginning	%
15*	illegal character	^^? (ASCII 127)

unexpandable, the expand processor passes the output token sequence to the main processor.

The main processor controls the whole activity of TEX. After the start it requests the first token from the expand processor and interprets it as the command of the main processor. This activity is repeated until the input end. By means of the main procesor command various activities are realized leading to the establishment of the final print material and its output to the desired file format.

The lexical analyzer in the system TEX is a function that fills the input processor and the token processor. The concrete form of the own procedure depends on the selected programming language and environment, so there can be created different variations of treatments that work in the same manner in their final form.

RESULTS

Incremental parsing: augmenting of existing methodology for languages with a variable set of lexical units

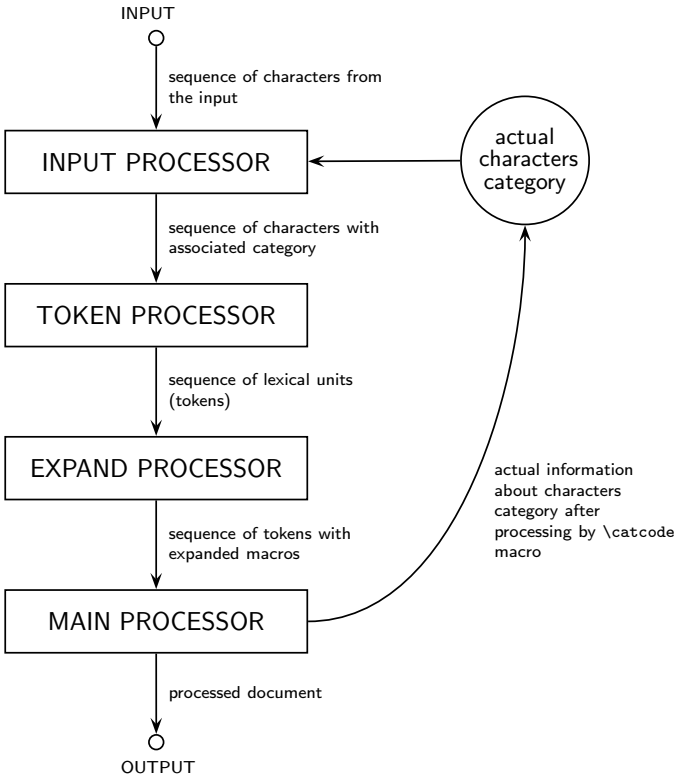
The basic problem of the group of languages with a variable set of lexical units is the fact that the input symbols can have different (variable) significance during the compilation.

During the lexical analysis of the input text in TEX it is necessary to assign a particular category to each character. The problem occurs when the category of the character has changed. This is in the system TEX the role of macro \catcode, which allows to set a different category to each character. The change takes effect immediately and is valid until the end of the group or to the end of the file if the validity is not defined with a group. If we want to compile a TEX source text incrementally, we must keep this fact in mind.

At first glance it might seem that a change in the character category should necessarily affect parsing of the input to such an extent that it will be necessary to re-compile the input text in the next step. But this is not necessary. In fact, it is necessary to re-analyze only the concerned group or if appropriate the rest of the input file if the group is not bounded.

The parsing table remains even after such an intervention in the input text—representing a change in the character category—unchanged. This is undoubtedly due to the fact that the form of the parsing table is not dependent on the input text, but on the allowed lexical units of the given language. It is obvious that the set of permitted lexical units is constant at any moment.

A fundamental change in the behavior of the compiler in case of the incremental compilation of TEX must therefore occur at the phase of lexical analysis that aims to retrieve gradually characters from the input and create lexical units from them—i.e. at the phase of input processing. At this time there must be a data structure containing information about the current categories of characters. Once the macro \catcode is processed it is needed to store the new information about the category change of



1: *Incremental parsing in TEX—solution to the problem with \catcode macro*

the given character. Afterwards the lexical analyzer during character reading and category assignment has to go from the current information, which can include the temporarily changed category. The next compilation phases may remain unchanged. The entire process is illustrated in the Fig. 1.

Incremental semantics analysis

The methodology of the incremental parsing analyzer using the recursive descent method described by Vagner and Melichar (2008) is based on the fact that the output of incremental parsing is a parsing tree representing the last modification of the source code serving as the input to the semantic analysis called thereafter.

If we want to link the semantic with syntactic analysis and do semantic actions already at the stage when we are building a new parsing tree, we must necessarily have also information about the interdependencies between parts of the source code.

Semantic tree construction

Information on the interactions of semantic links are linked with the grammar of the given language, thus we can bind them with the rules and include them to the implementation using the method of recursive descent.

A good way to express semantic relations is the semantic dependency tree. Tree nodes represent the semantic actions performed during a call in the appropriate place of the processed rule, then the edges represent a possible modification during the change of the source text. The implementation, however, is also a semantic operation and so it can be implemented by inserting a link into a good place of a corresponding grammar rule.

Let p_1, $p_2 \in P$ are two rules from context-free grammar $G = (N, \Sigma, P, S)$. Furthemore, let X is a set of semantic actions, $X_1 \in X$ a $X_2 \in X$ are two semantic actions located in grammar rules p_1, p_2 so that p_1 is the shape $N_1 \rightarrow \alpha X_1 \beta$, p_2 is the shape $N_2 \rightarrow \gamma X_2 \delta$, N_1, $N_2 \in N$, $\alpha, \beta, \gamma, \delta \in (N \times \Sigma \times X)^*$.

Then bind $X_2 \uparrow X_1$ we can represent by a new form of the semantic action $B(X_1)$ located in grammar rule p_2, which then will have the form $N_2 \rightarrow \gamma X_2 B(X_1) \delta$.

During the incremental compilation the presence of the semantic action $B(X_1)$ in the corresponding rule will cause the need to re-compile the construction described by the rule, which includes the semantic action X_1 on the right side.

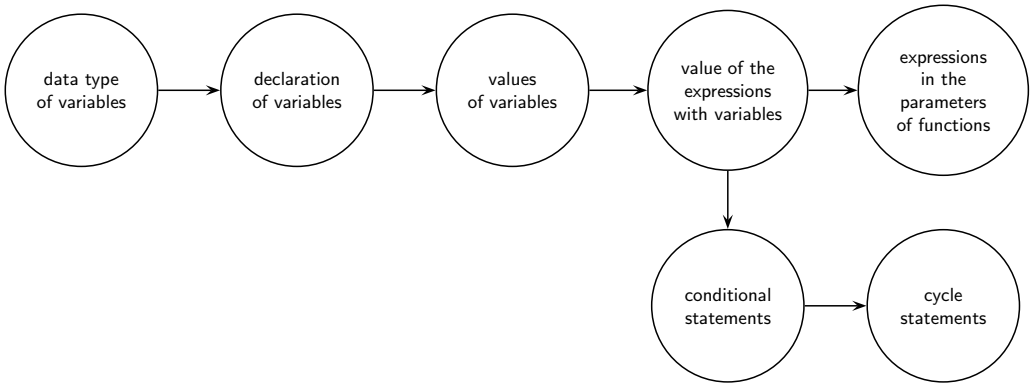
Example of tree construction

Let us come back to the example with the change of the variable data type. In general such a change can have entirely fatal consequences:

1. By changing the variable type there may be affected the whole declaration part of the program.
2. By changing the declaration part of the program there may be affected the value of all declared variables.
3. By changing the value of all declared variables there can be affected the value of all expressions which the variables occur in.
4. By changing the values of expressions with variables there can be influenced the result of calling the functions with the expression in the parameter.
5. By changing the values of expressions with variables there can be affected also all conditional statements, because the condition can be evaluated differently.
6. A change in the evaluation of the condition there may be affected all the cycle statements that are controlled by this condition. One slight change in a variable declaration can result in the need of reprocessing of the most of the code. Individual dependencies are clearly illustrated in the Fig. 2.

Let us show you now the construction of a dependence tree in a concrete example. Consider the following sequence of statements:

```
var a,b,c:integer;
begin
  a:=10;
  b:=5;
  c:=a+b*3;
  write (a, b, c);
  b:=a+c;
  write (b);
end.
```



2: *Dependencies in the source code*

We build the dependence tree in the following steps:

1. Declaration of variables. Consider rewriting rules of context-free grammar in the usual form (capitals represent the semantic action):

   declaration → INITLIST decitem ; decitem'
   decitem → var : id SAVETYPE

   We create a bind between the variable data type and variable declaration by inserting the binding semantic action BIND(INITLIST):

   decitem → var : id SAVETYPE BIND(INITLIST)

   Now let us show you how to capture this link in a semantic tree. For each variable we create a node and its predecessor for the data type of the variable (see Fig. 3).
2. Assigning values to variables. Consider rewriting rules of context-free grammar in the usual form (capitals represent the semantic action):

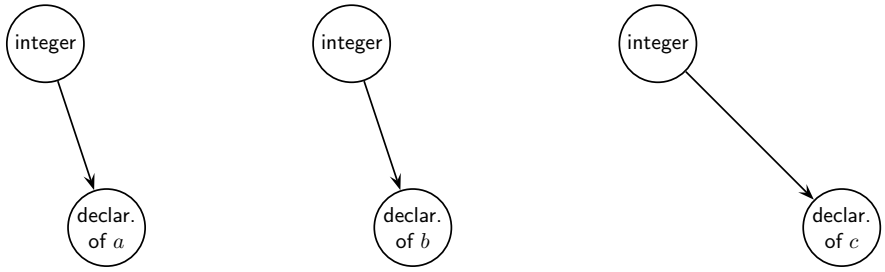
   assignment → id LVALUE := expression
   factor → (expression) | id RVALUE | num

   We create a bind by inserting the binding semantic action BIND(LVALUE):

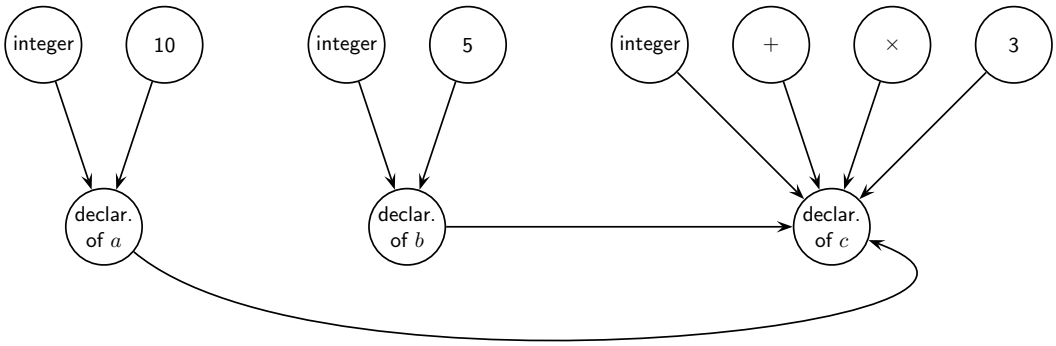
   factor → (expression)
   factor → id RVALUE BIND(LVALUE)
   factor → num BIND(LVALUE)

   Again, let us show you how to capture this link in a semantic tree. On the right side of an assignment statement there is an expression in general. Therefore, for each variable declaration node we create an ancestor with operators and constants. If the expression includes a variable, then the ancestor is the variable declaration node. This step is not necessary with compiling compilers which do not monitor the actual value of the variable (see Fig. 4).
3. List of variables values. Consider rewriting rules of context-free grammar in the usual form (capitals represent the semantic action):

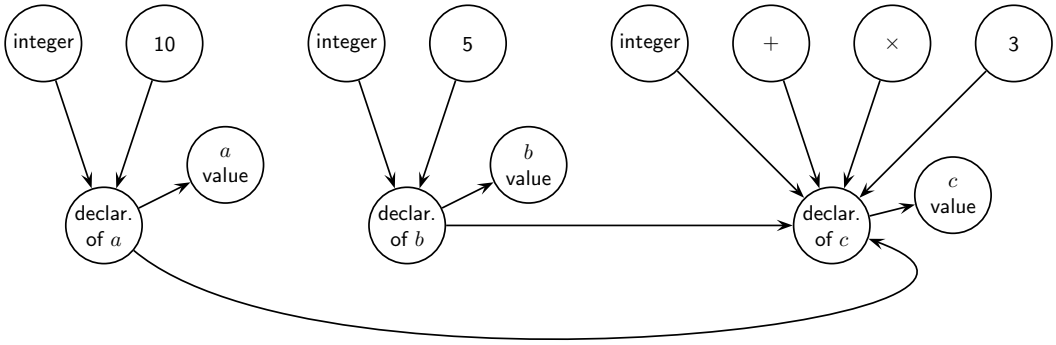
   expression → INITEXP term expression'
   factor → (expression) | id RVALUE | num



3: *Example of tree construction, step 1*



4: *Example of tree construction, step 2*



5: *Example of tree construction, step 3*

We create a bind between the variable data type and variable declaration by inserting the binding semantic action BIND(INITEXP):

factor → (expression)
factor → id RVALUE BIND(INITEXP)
factor → num BIND(INITEXP)

Again, let us show you how to capture this link in a semantic tree. We create successors of the variables declaration nodes with the current value of the variable (see Fig. 5).

4. Assigning values to variable. Analogous to the step 2 only with the difference that it is not the first assignment of value, therefore we must create a new successor of the declaration node. Again, this is not necessary for compiling compilers (see Fig. 6).
5. Listing the value of variable. Analogous to the step 3 (see Fig. 7).

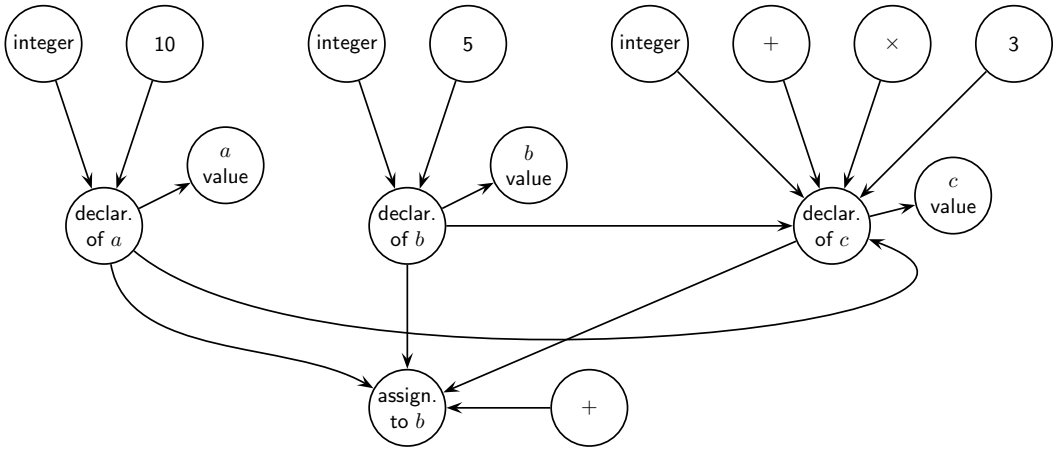
Now let us show how the chart can identify which parts of the program will be affected by the change (see Fig. 8). Suppose code modification in the form:

```
var a, b, c : integer;
begin
  a:=4;
  b:=5;
  c:=a+b*3;
  write (a, b, c);
  b:=a+c;
  write(b);
end.
```

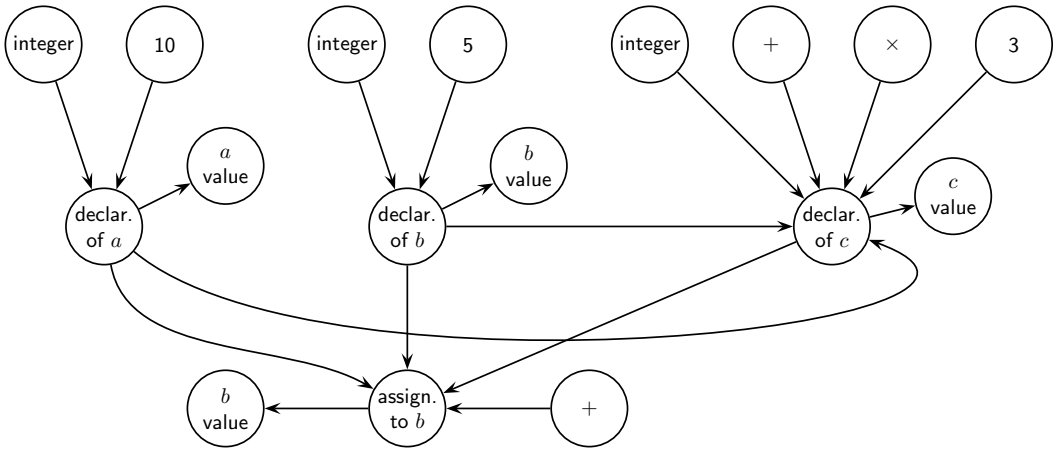
There is a trivial modification, when into the variable a was instead of the value 10 stored the value 4 at the beginning. All touched sections of the code are marked by a hatched line in the chart.

As we can see, the new value in the variable *a* due to the command line 3 will affect firstly the record of *a* variable in the symbol table, where the new value is stored. This change will affect the command to extract the value of the variable on the line 6, but also assign an expression to the variable *c* on the line 5. This assignment to the variable *c* affects the values of the variable *c* on the line 6 and also the assignment to the variable *b* on the line 7, which is also influenced by changing the value of *a* from the line 3. Finally, the new value in the variable *b* will cause a change during the processing of the line 8, where the value of *b* is listed.

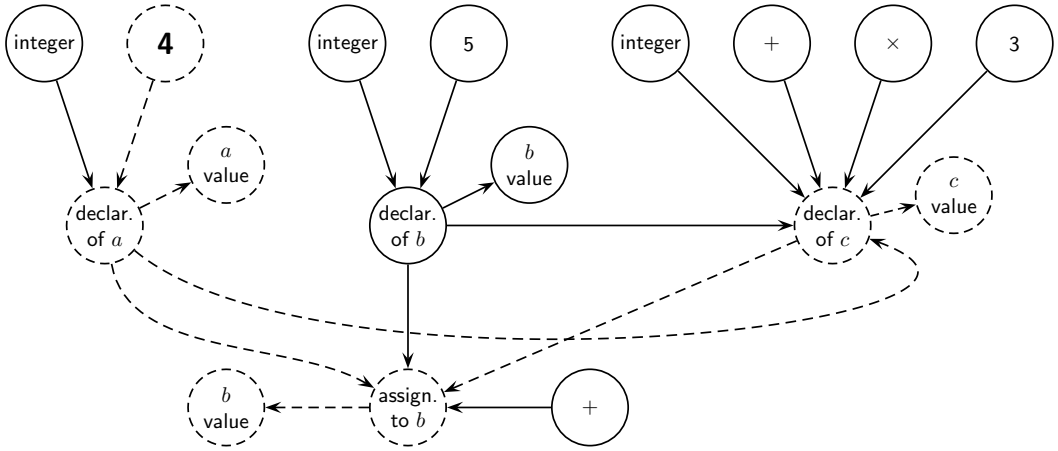
In a similar way we can solve even more complex programming structures as loops or conditional statements, because in both cases the essential



6: *Example of tree construction, step 4*



7: *Example of tree construction, step 5*



8: *Visualisation of touched sections of the source code*

element is the expression in the condition, respectively the variables that occur in it.

To make the system truly functional it is needed to mark particular nodes by a unique identifier to make it clear to which instruction in the source code they apply. Next it will connect information from the dependency graph (tree) with the information obtained from the parsing tree. At the moment we have information about the place in the program in which a change has occured (from the parsing tree) and we know which other parts of the program are affected by this change (from the graph of dependencies), nothing prevents the successful implementation of the incremental compiler.

DISCUSSION AND CONCLUSION

The paper deals with problems in the implementation of incremental compilers for various types of programming languages. As for the group of languages with a variable set of lexical units there is proposed an extention of the incremental design methodology for parsing.

Nevertheless, the change of the function of lexical analyzer is substantial. During the compilation of languages with a fixed set of lexical units its only function is that it creates the lexical units from the input symbols, and by this verifies if there are any invalid characters on the input. During the compilation of languages with a variable set of lexical units is the lexical analyzer or a machine close to it (as the input processor for TEX) basically the most important part of the compiler, since its decision on the current lexical unit type depends the success and effectiveness of the repeated compilation of the text in case of a small input change.

Following the incremental parsing there is then outlined the construction of the incremental semantic analyzer and thus the incremental compiler for any group of programming languages.

The proposed procedure can be used for both compilers and interpreters. The only difference will be the access to the change of a variable value. Whereas in case of the interpreter changing the value of the variable, it is necessary to find immediately a new value in order to continue the compilation, in the case of the compiler this requirement is waived, since the actual value of the variable is determined at the last stage, when you run the compiled code. To some extent it can be stated that the construction of the dependency graph for the semantic analysis will be easier for compilers than for interpreters.

To illustrate better the individual operations in the compiling process and also for teaching the theory of formal languages a new web application has been created at the Faculty of Economics at Mendel University. The application can automate the process of building a compiler using the recursive descent method. Currently it allows you to perform grammar transformations into the desired shape, the calculation of FIRST and FOLLOW sets, construction of the parsing table and even testing if the specifed sentence belongs to the language described with the language grammar. All of it rendering the parse tree of the tested sentence. Of course, there is the output in the form of the compiler source code in the programming language C. In the coming months the implementation of an incremental compiler with all the necessary components will be completed making this application an invaluable tool of exceptional quality.

SUMMARY

The paper deals with the possibilities of the incremental compiler construction. It represents possibilities of compilers constructions for the common programming languages and for languages with variable set of lexical units, too. Based on the known algorithms for standard compiler construction the metodology design for the construction of incremental compilers is derived. In the case of the incremental compilation of common programming languages this paper is based on the published methodology of incremental parser constructing, which extends as for the construction of the incremental semantic analyzer and thus an incremental access to the whole compiling process. The main change in the function of the incremental compiler for TEX source parsing must occur at the level of the lexical analysis, which is represented by the input processor and token processor. This lexical analysis has to read input characters and create the lexical units. At this time the data structure containing necessary information about the current characters categories must exist. While the parsing of \catcode macro is completed, it is necessary to save the new information about changing the character category. At the end of the paper is presented a web application for visualizing and automating the transformation of context-free grammars.

REFERENCES

AGRAWAL, R., DETRO, K. D., 1983: An Efficient Incremental LR Parser for Grammars with Epsilon Productions. Acta Inf., 19, pp. 369–376.

ALONSO, M. A., CABRERO, D., VILARES, M., 1997: A New Approach to the Construction of Generalized LR Parsing Algorithms.

FERRO, M. V., DION, B. A., 1994: Efficient Incremental Parsing for Context-Free Languages. Proceedings of the 5th IEEE International Conference on Computer Languages, pp. 241–252.

HOPCROFT, J. E., ULLMAN, J. D., 1978: Formalne jazyky a automaty. Bratislava: Alfa, 343 pp.

HORSPOOL, R. N., 1990: Incremental Generation of LR Parsers. Computer Languages, Vol. 15, issue 4, pp. 205–223. ISSN 0096-0551.

LI, W. X., 1995: A Simple and Efficient Incremental LL(1) Parsing. Lecture Notes in Computer Science, Vol. 1012. ISBN 987-3-540-60609-3.

LI, W. X., 1996: Building Efficient LL Incremental Parsers by Augmenting LL Tables and Threading Parse Trees. Computer Languages, Vol. 22, No. 4, pp. 225–235.

MATOUŠEK, M., 2001: Transformace strukturně značkovaných dokumentů. Diplomová práce. Brno: MZLU v Brně. 76 pp.

MEDUNA, A., 2000: Automata and Languages: Theory and Applications. London, Springer Verlag, 921 pp.

MELICHAR, B., VAGNER, L., 2008. Compiler Construction. 132 pp.

MOLNÁR, Ľ., ČEŠKA, M., MELICHAR, B., 1987: Gramatiky a jazyky. Bratislava, Alfa, 192 pp.

MURCHING, A. M., PRASAD, Y. V., SRIKANT, Y. N., 1990: Incremental Recursive Descent Parsing. Computer Languages, 15 (4), pp. 193–204.

OLŠÁK, P., 2001: TEXbook naruby. Brno: Konvoj, 468 pp. ISBN 80-7302-007-6.

ROZENBERG, G., SALOMAA, A. (eds.), 1997: Handbook of Formal Languages. Vol. 1: Word, Language, Grammar. Berlin: Springer-Verlag, 873 pp. ISBN 3-540-60620-0.

SHILLING, J. J., 1992: Incremental LL(1) Parsing in Language-Based Editors. IEEE Trans. Softw. Eng., 19 (9), pp. 935–940.

TOMITA, M., 1987: An Efficient Augmented-Context-Free Parsing Algorithm. Computational Linguistics, Vol. 13, pp. 31–46.

WAGNER, T. A., 1998: Practical Algorithm for Incremental Software Development Environments. Dissertation Thesis. 148 pp.

WAGNER, T. A., GRAHAM, S. L., 1997: Incremental Analysis of Real Programming Languages, 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 31–43.

WAGNER, T. A., GRAHAM, S. L., 1998: Efficient and Flexible Incremental Parsing. ACM Transactions on Programming Languages and Systems. Vol. 20, No. 2.

YANG, W., 1993: An Incremental LL(1) Parsing Algorithm. Information Processing Letters, Vol. 48, Issue 2, pp. 67–72.

YANG, W., 1994: Incremental LR Parsing. 1994 International Computer Symposium Conference Proceedings, vol. 1, pp. 577–583.

Address

Ing. Pavel Haluza, doc. Ing. Jiří Rybička, Dr., Ústav informatiky, Mendelova univerzita v Brně, Zemědělská 1, 613 00 Brno, Česká republika, e-mail: haluza@mendelu.cz, rybicka@mendelu.cz