TRANSLATION OF CONDITIONAL COMPILATION

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1997-13

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1997-13, NOVEMBER 1997

TRANSLATION OF CONDITIONAL COMPILATION

Maarit Harsu

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-4258-X
ISSN 0783-6910

Translation of conditional compilation

Maarit Harsu

Department of Computer Science
University of Tampere
P.O. Box 607, FIN-33101 Tampere, Finland
e-mail: `csnima@cs.uta.fi`

Abstract

This paper describes how to translate the compiler directives for conditional compilation in automated source-to-source translation between high-level programming languages. The directives for conditional compilation of the source language are translated into the corresponding directives of the target language, and the source program text of each branch of conditional compilation is translated into the corresponding target program text. Such translation raises a problem in conventional parsing because the source text is not a continuous stream of tokens of a legal program but rather a sequence of fragments that must be combined in certain ways to obtain one of several possible alternative programs. As a solution to this problem, a parsing technique is introduced which is able to cope with such input if certain reasonable conditions are satisfied.

Keywords: source-to-source translation, language conversion, conditional compilation.

1 Introduction

Some programming languages have compiler directives enabling conditional compilation. Compilers for these languages usually have a preprocessor which first evaluates the compilation condition and then searches for the branch to be executed. The preprocessor forwards only one branch to the compiler and neglects the other branches. Thus, the actual compiler parses and compiles only the branch fulfilling the compilation condition.

However, problems arise when implementing a source-to-source translator for this kind of language. In language conversion, the aim is naturally to translate the conditional compilation of the source language into the conditional compilation of the target language. Thus, the translator should parse and translate all the branches of the conditional compilation. However, normal continuous parsing is not always adequate. The successive branches of conditional compilation do not always form together a legal stream of tokens, and thus, the syntactical rules of the language may be broken. Consequently, the translator must somehow revoke its parsing process. When it has parsed one branch, and is starting to parse another one, it should restore the parsing situation in which it was when the head of the condition began. In other words, when a new branch begins, the converter should pretend not to have done any parsing during the earlier alternative branches handled so far. The described parsing method is called *multi-branch parsing*. Because the translator parses and translates all the branches, it has also the problem of how to store the nonterminal and terminal symbols of several branches.

We detected the problem of translating conditional compilation in source-to-source translation during the development of the PL/M-to-C converter translating PL/M [12] programs into C [11, 7]. The converter is implemented with TaLE (Tampere Language Editor) [8, 9], an object-oriented framework and a supporting editor for language implementation. In the earlier paper concerning the converter [11], we did not discuss the problem of translating conditional compilation at all. In [7], we tried to avoid the problem in usual cases by grammar transformations. In the present paper, we describe a more general solution. To implement this solution some modifications are required to the parsing code produced by TaLE.

This paper proceeds as follows. In the second section, we consider the problem in more detail. In sections 3 and 4, we present the solution, first informally and then as algorithms. These sections describe the solution in terms of general top-down parsing. Section 5 considers the solution especially from the point of view of TaLE. Finally, we give some concluding remarks. Throughout the paper, we assume a familiarity with the terms concerning parsing. For example, the term 'parse tree' is used as defined in [1].

2 Problem

The problem of translating conditional compilation is due to the difference between conventional compilation and source-to-source translation. When compiling, no problems arise because only one branch of the conditional compilation is parsed and translated. However, when translating, all the branches are parsed, and parsing them in pure consecutive order may be against the syntactic rules of the source language.

In source-to-source translation, the nonterminal and terminal symbols of all the branches should be stored somehow. During conventional parsing a parse tree of the program is logically constructed. When only one branch is parsed, the symbols of the branch are stored to the parse tree in a normal way. The branches of conditional compilation are actually alternatives of each other, and hence, the parse tree has place for the symbols of only one of the branches. Consequently, in source-to-source translation, the symbols of additional branches require some special storing convention.

In the following examples, we use PL/M compiler directives for conditional compilation (`$IF`, `$ELSEIF`, `$ELSE` and `$ENDIF`). There are many situations in which continuous parsing does not cause any problem. If the conditional compilation has only one branch, normal parsing is always possible. Even if there are several branches, but the branches form a syntactically correct token stream, continuous parsing does not cause any problem. For example, the branches may consist of whole statements, as follows:

```
statement1;
$IF
statement2;
statement3;
$ELSE
statement4;
$ENDIF
statement5;
```

In this case, the parser can simply treat the compiler directives like comments and parse the program text in normal way. Because conditional compilation is often used in this way, the problem is easy to ignore in source-to-source translators. However, testing the PL/M-to-C converter with real industrial PL/M programs revealed many ways to use conditional compilation such that continuous parsing of the branches is not possible. In the following example a syntax error occurs when parsing the program text in consecutive order:

```
DECLARE
$IF
a BYTE;
$ELSE
a WORD;
$ENDIF
```

According to the syntactic rules of PL/M, variables are declared after the keyword `DECLARE`, different variables are separated from each other with commas, and the whole declaration ends with a semicolon. Thus, when parsing the above declaration continuously, a syntax error occurs when the variable `a` is detected for the second time. A more complicated (fictitious) example of nested conditional compilation in a declaration is shown below:

```
DECLARE
$IF
   a BYTE
   $IF
   DATA (1);    /* constant value for the variable a */
   $ELSE
   INITIAL (2); /* initial value for the variable a */
   $ENDIF
$ELSE
   a WORD
   $IF
   PUBLIC;      /* public attribute for the variable a */
   $ELSE
   EXTERNAL;    /* external attribute for the variable a */
   $ENDIF
$ENDIF
```

In PL/M, either constant or initial values can be given for a variable in a declaration, but not both. A variable can be defined either public or external, but not both. Thus, the above code cannot be parsed continuously.

Although there are several papers concerning source-to-source translation, we have found only one paper concerning the problem of translating conditional compilation, namely [2]. Other papers do not mention this problem at all. Instead, they often mention that difficult parts of the source language, for example those which have no correspondence in the target language, are left untranslated; for example, see [10]. Many source-to-source translators require the user assistance: the user must either edit the source program before-hand to a more suitable form for the translator or complete the translated target program afterwards by hand. This policy has likely been applied to the translation of conditional compilation, too.

In [2], translation of macros and other preprocessor structures is considered. The authors show how macro definitions and calls can be translated in source-to-source translation. They suggest a method which first expands the macros. After that parsing can be performed normally. When translating, they use a syntax tree and a fragment tree telling which fragment each token belongs to. According to the fragments, they can conclude whether the expanded macro should be translated into the body of the macro or into the call of the macro. The same practice can be applied to translating all textual substitutions including conditional compilation. According to the authors [3], the translation of a program is performed in several sessions, and each conditionally compiled program region is translated in different sessions. Thus, some parts of the program are translated several times.

3 Solution

In this section, we first introduce some assumptions which the solution is based on. Then we describe the solution step by step.

3.1 Assumptions

The solution is based on top-down parsing. We assume a parsing stack, which is used in parsing in the following way (the algorithm is generalized from [5], p. 121):

```
parsing_algorithm is
   push the start symbol onto an empty stack;
```

```
    while the stack is not empty do
       /* let X be the top stack symbol */
       /* let a be the current input token */
       if X is a nonterminal symbol then
          pop X from the stack;
          push the components of X onto the stack in reverse
             order;
       else if X is a terminal symbol and X = a then
          pop X from the stack;
          scan a;
       else
          /* syntax error */
       end if;
    end while;
 end parsing_algorithm;
```

We assume that the parsing stack is implemented as an array because we need the indexes of the terminal and nonterminal symbols. We also use the well-known convention of source-to-source translators that comment strings of the program are attached to the latest parsed nonterminal in the parse tree. We treat compiler directives like comments and apply the same convention to the compiler directives, too.

3.2 Storing parsing situations

An easy solution for the problem of translating conditional compilation would be to translate the program as many times as it has combinations of compilation conditions. At each translation time, we would parse and translate a different combination. Then we could merge all the different translations together. However, this would be very ineffective solution, we would parse and translate large parts of the program unnecessarily over and over again.

As mentioned, to solve the problem of translating conditional compilation properly we have to revoke parsing. When we detect the beginning of conditional compilation (if-control), we store the parsing situation. When we detect a new branch of conditional compilation (else-if-control or else-control), we restore the stored parsing situation. When we detect the end of conditional compilation (end-control), we can forget the stored parsing situation. If the end-control belongs to an inner conditional compilation of nested ones, we forget the current stored parsing situation and set the current situation to denote the previous situation. Thus, this multi-branch parsing method needs a stack structure for storing parsing situations.

The described solution seems very obvious and easy. However, the problem is how to store the parsing situations, based on the standard top-down parsing algorithm. We propose the following method. We collect the terminal and nonterminal symbols parsed during the first branch of conditional compilation. Then we parse the same symbols during each subsequent branch. When we have applied this method to all branches, normal parsing can continue, and no syntactic errors occur.

As can be seen from the top-down parsing algorithm (in subsection 3.1), when the parsing of a nonterminal is started, the nonterminal is first popped from the parsing stack. Thus, if a nonterminal has been popped, its parsing operation has begun. Consequently, to know the parsed nonterminal and terminal symbols we collect all the popped symbols during the first branch of conditional compilation. Then we call the parsing functions of these nonterminals during the following branches.

However, we need not and must not collect all the popped symbols. Consider the nonterminal `A` having the following structure

```
A -> B C D.
```

If `A` has been parsed during the first branch of conditional compilation, it has been popped from the parsing stack. In addition, its components have been popped. Thus, our pop list contains the nonterminal `A`, the components of `A`, the components of the components of `A`, and so on, as far as they have been popped. However, if parsing of `A` has been completed during the first branch, it is sufficient to call only the parsing function of `A` in the following branches. When calling the parsing function of a nonterminal, the parsing sequence proceeds implicitly to the components of the nonterminal and to the components of the components as far as they exist. Thus, we should remove the component symbols from the pop list such that the pop list contains as large nonterminals as possible.

3.3 Reducing pop lists

To remove the undesired component symbols from the pop list, we should be able to conclude the component relations between the symbols. We assumed the parsing stack to be implemented as an array. Hence, we can store the array index of each symbol popped, and attach the index to the same symbol in the pop list. From these indexes we can conclude the component relations between the symbols.

Consider again the nonterminal `A` shown above. If `A` is popped from the index 1, the component nonterminals are popped from the indexes 3, 2, and 1, respectively. Consequently, we obtain an algorithm for reducing a pop list. If we find a sequence of symbols having the indexes of the form 1, 3, 2, 1 from the pop list, we can remove the whole sequence except the first symbol. We know that the other symbols of the sequence are the component symbols of the first nonterminal. In general, we can search for these kinds of removable sequences as follows. We start from the end of the pop list. We mark each item of the pop list in turn (first we mark the last item). We move backwards item by item as long as i = j + 1 holds for the index i of the current item and the index j of the previous item. When no such indexes are found, we compare the index of the current item with the index of the marked item. If they are equal, we have found a removable sequence, and we can remove it. Searching and removing removable sequences continues until all the items have been marked and no more removable sequences can be found. The resulting pop list is the reduced pop list.

By using the described method, we can find out the parsed nonterminal and terminal symbols. These symbols have been popped from the parsing stack during the first branch. To restore the parsing situation, we put (not push) these symbols again to the parsing stack to the same indexes as they were earlier, and call their parsing functions, which effect popping them from the parsing stack and parsing them. Note that when putting symbols to the parsing stack again, we must be sure that we do not put them on the indexes which already have symbols. To consider this further, we define:

Definition 1 *A pop list is well-structured if all the indexes of its nonterminal and terminal symbols are greater than the index of the top item of the parsing stack and if each index between the lowest and the highest index of* *the pop list appears exactly once.*

3.4 Obtaining well-structured pop lists

If the pop list is well-structured, there is no problem in putting the additional nonterminal and terminal symbols back to the parsing stack. To find out in which situations the pop list is well-structured, we first examine the situation in which the pop list is not well-structured. Consider the following grammar:

```
Statement     -> Statement1 | Statement2 | ...
StatementList -> Statement+
Statement1    -> 'IF' Condition 'THEN' Statement
Statement2    -> 'DO' StatementList 'END'
```

In the above grammar, nonterminal names begin with uppercase letters, terminal names are enclosed with quotes, and plus (`+`) denotes a list having one or more elements. Suppose we have the following input text:

```
/* Input 1 */
$IF
IF cond1 THEN DO
   Statement3
$ELSE
IF cond2 THEN DO
   Statement4
$ENDIF
   Statement5
END
```

Table 1 shows the terminal and nonterminal symbols pushed onto the parsing stack and popped from it.

Table 1: A sample content of the parsing stack.

4		='IF'		
3		=Condition	='DO'	=Statement
2		='THEN'	=StatementList	-[Statement]
1	=Statement1	=Statement	-'END'	

In Table 1, the numbers of the leftmost column denote the indexes of the terminal and nonterminal symbols. The symbols marked with (`=`) are both pushed onto the parsing stack and popped from it, the symbols marked with (`-`) are only pushed onto the parsing stack. To get the above situation of the parsing stack, we perform the following actions. First, we push the nonterminal `Statement1` onto the parsing stack. Then we pop it from the parsing stack and substitute it with its components (`'IF'`, `Condition`, `'THEN'`, and `Statement`). We pop the first three of them (`'IF'`, `Condition`, and `'THEN'`). Actually, when the nonterminal `Condition` is popped, it is substituted with its components, which are again popped and substituted with their components; we omit the details here. (The resulting situation is still the same.) We pop the nonterminal `Statement` and because the next coming statement is `Statement2`, we push its components. We pop the component `'DO'`.

The nonterminal `StatementList` is a list structure having no separator. We assume the following parsing actions for such structures. The list nonterminal (here `StatementList`) is popped from the parsing stack, the nonterminal of the list body (here `Statement`) is pushed onto the parsing stack as optional (optionality is presented with enclosing brackets). In addition, the body nonterminal is pushed as such, and the parsing function of the body nonterminal is called. While the structure of the body occurs in the program text, the body nonterminal is pushed, and its parsing function is called. When the whole list structure is parsed in this way, the optional body is popped from the parsing stack. In the case of Table 1, we have popped the body nonterminal `Statement`. Because the parsing of the whole list is not finished, we have not yet popped the optional body.

According to Table 1, we obtain the following pop list (the index and the name of the symbol):

```
1  Statement1
4  'IF'
3  Condition
2  'THEN'
1  Statement
3  'DO'
2  StatementList
3  Statement
```

We can reduce the pop list as follows:

```
1  Statement1
3  'DO'
2  StatementList
3  Statement
```

We can present the situation of the parsing stack (Table 1) in a more simplified form:

```
2  [Statement]
1  'END'
```

This is the situation in which the pop list is not well-structured: we cannot put the items of the pop list to the parsing stack properly. However, suppose we have, instead of the above input text, the following input text:

```
/* Input 2 */
$IF
IF cond1 THEN DO
   Statement3
   Statement5
END
$ELSE
IF cond2 THEN DO
   Statement4
   Statement5
END
$ENDIF
```

In this situation, the items of the parsing stack (optional statement and the terminal symbol `'END'`) are also popped, and thus, added to the pop list. Our pop list is then as follows:

```
1  Statement1
3  'DO'
2  StatementList
3  Statement
2  [Statement]
1  'END'
```

It can be reduced two times, first as follows:

```
1  Statement1
3  'DO'
2  StatementList
1  'END'
```

and second as follows:

```
1  Statement1
```

Thus, if we had this situation, our pop list would be well-structured and we could put the nonterminal `Statement1` to the parsing stack, because the index 1 would be free.

As shown above, nonterminal and terminal symbols move from the parsing stack to the pop list because popped items are taken away from the parsing stack while the pop list contains the popped items. When we move symbols from the parsing stack to the pop list, the item count of the parsing stack decreases and that of the pop list increases. The smaller the item count of the parsing stack, the more likely the indexes of the pop list are greater than the top index of the parsing stack. Thus, the pop list is more likely well-structured. The larger the item count of the pop list, the more likely we can reduce it and make it well-structured.

3.5 Parsing conditional compilation

If the pop list is well-structured, the proposed multi-branch parsing is possible, because we can restore the parsing situation by putting the nonterminal and terminal symbols of the pop list to the parsing stack. We are now trying to characterize the cases where the pop list is well-structured. In the following theorem, we denote S the set of nonterminals whose parsing has started during the first branch of conditional compilation.

Theorem 1 *If parsing of all the nonterminals in set S has both started and finished during the first branch of conditional compilation, the pop list is well-structured.*

Proof We first proof the following reverse statement. If the pop list is not well-structured, the set S contains nonterminals whose parsing has started but not finished. The proof of this reverse statement is as follows. If the pop list is not well-structured, it has some nonterminal symbols whose components are lacking. Because these components have not been popped, their parsing functions have not been called. Consequently, parsing of the nonterminals which have the unparsed nonterminals as their components has not been finished. These nonterminals are those items of set S whose parsing has started but not finished. Hence, the reverse statement holds, and thus, the original statement holds, too. □

Note that when conditional compilation occurs, there are some nonterminals whose parsing has started but not finished. At least this holds for the start symbol of the language. However, these nonterminals are not meaningful from the point of view of the well-structured pop lists. We require only that if parsing of a nonterminal has started during a branch it must also be finished during the branch to get a well-structured pop list.

Multi-branch parsing is always possible if the pop list is well-structured. However, parsing can be possible even if the pop list is not well-structured. In the previous subsection, we illustrated the relation between pop lists, one of which being well-structured and the other not. It seemed that well-structured pop lists can be obtained from ones which are not well-structured by just moving symbols from the parsing stack to the pop list. In practice it is not such simple, because the calling sequence in which these symbols will be popped has started. Thus, we cannot explicitly pop the symbols from the parsing stack, or they would be popped more than once, which causes errors.

When we have a pop list which is not well-structured, parsing succeeds in some situations in the following way. Consider the first input text (Input 1) of the previous example (in subsection 3.4). The branches of the conditional compilation contain the beginning of the conditional statement (`Statement1`). The common end of the statement is after the whole conditional compilation. When we note that the pop list is not well-structured, we search for the end of the conditional compilation (end-control). We read input text from this new location until the pop list is well-structured, i.e., until the necessary symbols are popped from the parsing stack. Then we return to the earlier location.

However, it is possible to find situations in which parsing does not succeed if the pop list is not well-structured. Consider the following situation:

```
$IF assignment
a
$ELSE
RETURN
$ENDIF
$IF assignment
= 1;
$ELSE
2;
$ENDIF
```

It is easy to conclude from the above example that parsing does not succeed with our method. However, the example is fictitious and not very sensible. Note that because the statement of the first branch is unfinished, the same kind of statement is expected to appear in the second branch, too. Suppose the whole conditional compilation is in more sensible form as follows:

```
$IF
a = 1;
$ELSE
RETURN 2;
$ENDIF
```

Now parsing succeeds (the pop list would be well-structured). Because the statements of the branches are finished, they are allowed to differ each other.

After all, if the pop list is well-structured, multi-branch parsing succeeds. In addition, even if the pop list is not well-structured, parsing succeeds in some cases, but not always. Thus, a well-structured pop list is a sufficient but not a necessary condition for the success of multi-branch parsing.

3.6 Storing the symbols of the additional branches

We still have to solve the problem of storing the nonterminal and terminal symbols of the additional branches of conditional compilation. We cannot store them to the parse tree to their own places, because it has only room for the symbols of one branch.

We assumed that compiler directives (and comment strings) are attached to the latest parsed nonterminal. Thus, each nonterminal has a comment list containing the comment strings and the compiler directives attached to it. To store also the additional nonterminals, we expand the structure of the comment list item such that each item has a list of symbols attached to it. Thus, the symbols appearing in a program text after, for example, the compiler directive if-control are attached to this directive, and the if-control itself is attached to the preceding nonterminal.

In source-to-source conversion, when a nonterminal has been translated, the comments and the compiler directives of the nonterminal are written. If a compiler directive has other symbols attached to it, translation functions of these symbols are called. This convention works very well in usual cases. However, translation functions are called one after another and they are separated from their original context. If more global information is needed in conversion, the result of the translation may not be valid. How well this convention works depends on how the user has divided the grammar of her language to different nonterminals and how she has divided the semantic actions to the translation functions of different nonterminals.

4 Implementation of the solution

To translate conditional compilation, we have to make some additions to the parsing code. We need some additional functions mainly concerning parsing stack and scanning. In this section, we show the algorithms in essence considered in the previous section. In addition, we show how multi-branch parsing can be implemented without an explicit parsing stack.

4.1 Parsing stack

We need a new data structure, a stack for pop lists. The top of the stack has the situation of the current conditional compilation (the current pop list).

The parsing stack originally has the operations push and pop (as shown in the top-down parsing algorithm), which pushes a nonterminal or a terminal symbol onto the parsing stack and pops a symbol from it, respectively. We need new push and pop operations for the pop list stack. These new operations are called push_list and pop_list, and the code of them is as follows:

```
push_list is
   create a new pop list;
   add the new pop list onto the top of the pop list stack;
   increase 'pop_saving';
end push_list;

pop_list is
   take the pop list from the top of the pop list stack;
   set the pop list just under the taken pop list to be the
      new top of the pop list stack;
   delete the pop list taken away;
   decrease 'pop_saving';
end pop_list;
```

We need the global variable pop_saving to tell whether the nonterminal or terminal symbol popped by the function pop should be added to the current pop list (on the top of the pop list stack). Note that the variable pop_saving has to be an integer variable (not a boolean variable), because conditional compilation may be nested. We show the additions to the original function pop later. We need also a new operation restoring the parsing stack situation:

```
restore is
   reduce the current pop list;
   if the current pop list is well-structured then
      set the variable 'skipped' the value false;
      put each symbol of the pop list to the parsing stack
         to the same index from which it has been popped;
      set 'last_symbol' to denote the latest parsed symbol;
      for each put symbol
         parse the symbol;
         add the symbol to the nonterminal list of
            'last_symbol';
```

```
      end for;
   else
      set the variable 'skipped' the value true;
      mark the current scanning position;
      set the new scanning position to be just after the
         next end-condition;
   end if;
end restore;
```

We need the global variable `skipped` to tell whether we have skipped over the rest of the conditional compilation. If the pop list is not well-structured, we mark the current scanning position and skip scanning to the next occurence of an end-control. The original `pop` operation restores the scanning position. All the necessary additions to the `pop` function are shown below:

```
pop is
   if the parsing stack index is greater than zero then
      // added code begins
      if 'pop_saving' is greater than zero then
         add the top symbol of the parsing stack to the
            pop list on the top of the pop list stack;
      end if;
      if 'skipped' and the pop list is well-structured
      then
         set the scanning position to the marked one;
         set the variable 'skipped' the value false;
      end if;
      // added code ends
      decrease the parsing stack index;
   end if;
end pop;
```

4.2 Scanning

We need two new scanning operations for conditional compilation. We need a boolean function `is_conditional`, which returns true if and only if a compiler directive for conditional compilation is coming next in the input. In addition, we need the operation `process_conditional`, which acts according to the type of the directive, as follows:

```
process_conditional is
   if if-control is coming next then
      push_list();
   else if else-if-control or else-control is coming next
   then
      restore();
   else if end-if-control is coming next then
      pop_list();
   end if;
end process_conditional;
```

The operation `process_conditional` just calls the added operations of the parsing stack, which actually has a new operation for each different control of conditional compilation.

4.3 Parsing conditional compilation without a stack

In the latest two chapters (3 and 4), we have described the multi-branch parsing method and its implementation for conditional compilation. We have assumed the parsing stack. However, multi-branch parsing could also be implemented in recursive descent parsers having no explicit parsing stack. These parsers have a parsing function for each nonterminal symbol of the language. This function calls the parsing functions of the nonterminal components, and scans the terminal components. To our nonterminal `A`, the parsing function (having the same name `A`) would be as follows:

```
A is
   B;
   C;
   D;
end A;
```

More examples of these parsing functions can be seen, for example, in [5], pp. 33-38. Instead of the explicit stack, recursive descent parsers have an implicit run-time stack containing the called procedures in the form of activation records. To allow parsing of conditional compilation, the parsing function of the nonterminal `A` should be modified as follows:

```
A(i: integer) is
   B(i);
   C(i+1);
   D(i+2);
   if pop_saving is greater than zero then
      add the current nonterminal (A) to the pop list;
   end;
end A;
```

We use the global variable pop_saving for the same purpose as in the stack implementation (to tell whether nonterminal and terminal symbols should be saved to the pop list). We obtain the indexes of the pop list items by adding an integer parameter for each function corresponding a nonterminal of the language. If the nonterminal A receives the parameter i, it passes the same value i to the first component B, the value i+1 to second component C, and the value i+2 to the third component D. We can reduce the pop list normally (as described in subsection 3.3). To parse the additional symbols, we just call the parsing functions of the corresponding nonterminal symbol or scan the corresponding terminal symbol.

With the above modifications, multi-branch parsing could be implemented in recursive descent parsers, too. However, the modification to these parsers is harder than to the parsers having a parsing stack, because the same modification is required in all the parsing functions.

5 Modifications to TaLE

In this section, the necessary parsing modifications are considered from the point of view of TaLE.

5.1 Backgrounds

TaLE (Tampere Language Editor) is a tool supporting the development of language implementation software in an object-oriented programming environment. It emphasizes software engineering qualities rather than contributions in formal languages. TaLE does not expect the user to write a language specification, but to edit the classes representing language structures under the control of a specialized editor. In order to use TaLE, the user need not learn any textual metalanguage. TaLE is meant to be simple and easy to use, to attract also users who intend only to implement a small task concerning textual representations of data, specifications, algorithms, etc.

TaLE is written in C++ [13]. There are actually two versions of TaLE, one generates C++ code and the other Java [4] (both are written in C++). There are several papers about TaLE: an overview can be found in [8], an example of using TaLE in [9], and the introduction of the Java version in [6]. The PL/M-to-C converter is implemented with the C++ version.

A main principle in designing TaLE has been reuse. Reuse is due to the class division of TaLE which is different from that of more traditional systems. In TaLE, each language structure (nonterminal symbol in the grammar) is implemented as an independent software unit, i.e., a class. Each nonterminal knows only its component nonterminals. Thus, the syntactic information about a language is dispersed in several classes.

In TaLE, parsing is controlled by metaobjects. Each nonterminal class has a metaclass, which knows the components of the nonterminal. The instance of the metaclass, metaobject takes care of the instantiation of its actual nonterminals when needed. A metaclass has two basic operations, `look_ahead` and `make`. The operation `look_ahead` is a boolean function returning true if and only if the corresponding nonterminal appears next in the input. The function `make` is the instantiation operation returning a new instance of the actual nonterminal class.

TaLE automatically generates the parsing code for the given grammar. However, TaLE provides no semantic support for a language, instead, the user is expected to write the semantic functions for her language. The semantic processing begins by calling the function `process` of the start symbol of the language. TaLE provides the feature to generate the default `process` functions for each class. For example, the default `process` function of a structural class contains the calls of the `process` functions of the components. The `process` function of each nonterminal of the PL/M-to-C converter implements the translation of the nonterminal.

5.2 Modifications

Some concepts concerning conditional compilation in TaLE are different from the concepts used in the present paper. TaLE has a parsing stack called *work list*. The items of the work list are not nonterminal or terminal symbols, instead the work list contains the corresponding metaobjects. Similarly, the items of the pop list are metaobjects, too. In TaLE, calling the parsing function of a nonterminal means calling the `make` function of the corresponding metaobject. This effects creating and parsing the nonterminal.

Consider again the nonterminal `A`. The `make` function of the metaclass `MetaA` is in essence as follows:

```
Notion* MetaA::make(void)
{
   worklist->pop();        // metaobject of A is popped
   worklist->push(meta_D);
   worklist->push(meta_C);
   worklist->push(meta_B);
   return new A;
}
```

In the above code, `Notion` is a superclass of all the nonterminal classes. `MetaA` is a metaclass for a nonterminal `A`. The parameters `meta_D`, `meta_C` and `meta_B` are metaobjects, instances of the metaclasses `MetaD`, `MetaC` and `MetaB`, respectively.

Each nonterminal of the language has a metaclass and the corresponding `make` operation. In addition, TaLE has different `make` functions for different predefined language structures: for a keyword (terminal symbol), for a set of keywords, for a list with a separator, for a list without a separator, and for an optional structure. (The `make` function of the list without a separator is such as assumed in subsection 3.4.) Together all these `make` functions act like the top-down parsing algorithm.

TaLE stores comments and compiler directives to the latest created nonterminal (like in our assumptions). If a compiler directive has nonterminals attached to it, we call the `process` function of the nonterminals to get the corresponding translation code.

We have implemented the modifications described in this paper to the TaLE code. The modifications mainly concern two classes of TaLE: work list and scanner. However, the compiler directives for conditional compilation are in some cases difficult to lead to their own places in the target code. Thus, we have made some additional minor changes. For example, we have implemented an optimation which checks whether the nonterminals of the branch of conditional compilation are items of a list. In affirmative cases we treat the branches like there were no conditional compilation. These cases do not cause any syntactic errors, but the output of the compiler directives is easier to lead to the proper place, if we ignore the conditional compilation.

We have compared the efficiency of multi-branch parsing with the normal parsing of TaLE. Table 2 shows the results of the comparison.

Table 2. Efficiency comparison of multi-branch parsing.

	program 1	program 2
PL/M	12.40	12.43
PL/M'		10.19

In Table 2, program 1 is a PL/M program having a lot of directives for conditional compilation. Program 2 is the same program, but the directives for conditional compilation are enclosed with comment marks, and some tiny modifications have been done to prevent the syntax errors due to the commented directives. The sizes of program 1 and program 2 are 270 kb and 279 kb, respectively. In Table 2, PL/M means normal PL/M language allowing the directives for conditional compilation. PL/M' has been modified from PL/M such that the directives for conditional compilation are not allowed. The numbers are translation times in seconds. According to Table 2, the additional parsing stack operations do not slow down parsing (compare the numbers 12.40 and 12.43). Instead, finding the directives takes time (compare the numbers 12.43 and 10.19): the directives for conditional compilation must be distinguished from the other directives (in PL/M beginning with the same letter '$'). We performed the comparison under TaLE. TaLE uses top-down parsing, thus, the time ratio should be approximately the same in any environment using top-down parsing.

6 Conclusions

Very little attention is paid to the conversion of conditional compilation in source-to-source translation. However, in practice it is important that also the programs containing compiler directives of conditional compilation can be automatically translated into the target language. If a syntax error occurs due to conditional compilation the whole program is left without translation. In this paper, we have proposed a parsing method which enables parsing of conditional compilation.

We have implemented the necessary modifications for conditional compilation to TaLE. TaLE originally had a parsing stack (work list) to which we have applied the main modifications. By using the parsing stack the parsing situations are easy to store and restore. However, we have shown that multi-branch parsing could also be implemented in recursive descent parsers having no explicit parsing stack.

Multi-branch parsing is not quite perfect. There are situations of conditional compilation in which parsing does not succeed with our method. However, these situations are very rare, and we have defined the conditions in which multi-branch parsing succeeds.

Acknowledgements

The author gratefully acknowledges Kai Koskimies and Erkki Mäkinen for their review of this paper.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] K. Andrews, P. del Vigna and M. Molloy, Macro and file structure preservation in source-to-source translation, *Software - Practice and Experience* **26** (3), 281 - 292 (1996).

[3] K. Andrews, Private communication (via e-mail), in October, 1997.

[4] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[5] C. N. Fischer and R. J. LeBlanc, *Crafting a Compiler with C*, The Benjamin/Cummings Publishing Company, 1991.

[6] M. Harsu, J. Hautamäki and K. Koskimies, A language implementation framework in Java, In Bosch, J., Hedin, G., Koskimies, K. (eds.), Proceedings of LSDF'97 Workshop on Language Support for Design Patterns and Object-Oriented Frameworks, Department of Computer Science, University of Karlskrona/Ronneby, Research Report 6/97.

[7] M. Harsu, Automated conversion of PL/M to C, Report A-1997-7, Department of Computer Science, University of Tampere.

[8] J. Hautamäki, Language Implementation with TaLE, Department of Computer Science, University of Tampere, Master of Science Thesis, 1996.

[9] E. Järnvall, K. Koskimies and M. Niittymäki, Object-oriented language engineering with TaLE, *Object Oriented Systems*, **2**, 77 - 98 (1995).

[10] V. D. Moynihan and P. J. L. Wallis, The design and implementation of a high-level language converter, *Software - Practice and Experience*, **21** (4), 391 - 400 (1991).

[11] M. Niittymäki, Implementing a PL/M-to-C converter with TaLE, Proceedings of the Fourth Symposium on Programming Languages and Software Tools, Visegrad, Hungary, June 1995, 9 - 20.

[12] PL/M Programmer's Guide, Intel Corporation, USA, 1987.

[13] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1987.