Univ. u Novom Sadu
Zb. Rad. Prirod.–Mat. Fak.
Ser. Mat. 22, 1 (1992), 179-193

Review of Research
Faculty of Science
Mathematics Series

ON TRANSFORMATION OF CONTEXT FREE GRAMMARS INTO A FORM SUITABLE FOR USE IN COMPILER GENERATORS

Mirjana Ivanović,[1] Saša Živkov, Zoran Budimac
Institute of Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia

Abstract

Algorithms for transformation of arbitrary context free grammar into reduced non-circular grammar without empty rules are implemented. Implementation is organized as a preprocessor that takes arbitrary grammar as an input and produces its equivalent suitable for usage in compiler generators. Preprocessor can be used as a first step in many compiler generators, as well as in the various applications in formal language theory. Some transformation algorithms are restated and some important implementation issues are described.

AMS Mathematics Subject Classification (1991): 68N20, 68Q50
Key words and phrases: reduced grammars, non-circular grammars, empty rules, equivalent grammars, compiler generators.

1. Introduction

Compilers are specialized programs that translate high-level programming languages to machine languages of specific computers. First step in

[1]Work is partially supported by Science Fund of Serbia

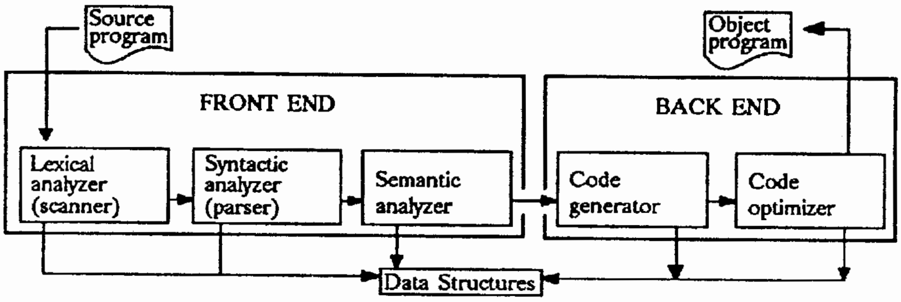

Figure 1: The structure of compiler

every compiler construction process is the description of the source language (the language to be translated). Description, as a rule, takes the form of context free grammar (later on: CF-grammar) over some alphabet.

The common structure of any compiler is displayed in Figure 1. It consists of two parts:

- front-end which performs the analysis of the source program and is independent of the target machine language, and

- back-end which performs the synthesis of the target (object) program, and is fully dependent on the target computer and its language.

Both parts share common data structures.

Over the past decade automated compiler generators (or "compiler-compilers") emerged as a standard tool in compiler construction. They automatically produce almost the whole front-end of a compiler i.e., complete scanner and parser and some parts of semantic analyzer. Resulting front-end (the output of compiler generator) is either represented as a source program of some high-level programming language, or as an abstract syntax tree extended with all necessary information for compiler generation. The latter approach has been especially popular in the last several years [2].

Automatic generation is made based on the specification of a source language that usually takes the form of CF or attribute grammars. However, not every CF-grammar is a suitable input for compiler generator. Only non-

circular reduced CF-grammars **without empty rules** are considered to be a suitable device for the specification of programming languages and implementation of compiler generators (see [5]; [1]; [6]).

There are a lot of implemented compiler generators to date. However, most of them either assume that the input grammar is already non-circular, reduced and without empty rules, or just check whether the grammar is of the desired form or not. Very few of them actually involve conversion of arbitrary input CF-grammar into a desired form. The existence of such tool would facilitate the preparation of the source language specification and, consequently, the usage of the whole compiler generator.

This paper describes the implementation of such preprocessor. Preprocessor implements some of the known algorithms for transformation of arbitrary CF-grammar into its equivalent, suitable for compiler generation. Some of the needed algorithms are restated (with respect to the versions given in [6]) in order to reflect implementation issues and stress the more efficient usage of memory space. Only restated algorithms will be described in this paper. In the rest of the paper necessary definitions will be given, followed by all restated algorithms for grammar transformation. The final part of the paper will describe important implementation issues of the preprocessor.

2. Preliminaries

2.1 Basic Definitions

Definition 1. *CF-grammar* is ordered quadruple $G = (V_n,\ V_t,\ S, \Phi)$, where:

- V_n is a finite non-empty set of non-terminal symbols,

- V_t is a finite non-empty set of terminal symbols,

- $S \in V_n$ is an initial symbol of the grammar,

- Φ is a finite set of grammar rules of the form $\alpha \rightarrow \beta$, where $\alpha \in V_n$ and $\beta \in (V_n \cup V_t)^*$.

In usual notation, ε denotes the empty word, V^* denotes the set of all possible words over the alphabet (finite set) V, V^+ denotes the set of all possible words over V except the empty word (i.e. $V^+ = V^* \backslash \varepsilon$) and $\alpha \Rightarrow^* \beta$ denotes that non-terminal α produces word β by applying some grammar rules from Φ, one or more times.

Definition 2. *Non-circular CF-grammar* is a CF-grammar in which there is no $\alpha \in V_n$, such that $\alpha \Rightarrow^* \alpha$.

Definition 3. *Reduced CF-grammar* is a CF-grammar that satisfies the following constraints:

- For every $\alpha \in V_n$ there is $\beta \in V_t^*$ such that $\alpha \Rightarrow^* \beta$
- For every $\alpha \in V_n$ there is at least one $S \Rightarrow^* \tau_1 \alpha \tau_2$, $, \tau_1, \tau_2 \in (V_n \cup V_t)^*$

Definition 4. *Empty rule* is a grammar rule of the form $\alpha \to \varepsilon$, where $\alpha \in V_n$.

Definition 5. *CF-grammar without empty rules* is a CF-grammar that doesn't contain empty rules (as defined in previous definition).

Consequently, non-circular, reduced grammar without empty rules is every CF-grammar which satisfied the constraints defined in definitions 2, 3 and 5. Every CF-grammar has its equivalent that is non-circular, reduced and without empty rules. This assertion is proved by quoting all necessary steps leading from arbitrary CF-grammar toward the desired one.

In the following two sections of the paper some of the necessary steps will be shortly described. All the presented algorithms are restated in order to reflect implementation issues and stress the more efficient usage of memory space.

2.2 Toward Reduced CF-Grammar

Intuitively, CF-grammar is reduced if it contains only useful symbols i.e., symbols that produce language constructions. Symbols of the grammar are useless if they are inactive or unreachable. By removing all such symbols from CF-grammar, its reduced equivalent is produced. Clearly, reduced grammar will produce the same language as the original one, because only useless symbols and appropriate rules are removed.

Active symbols

Definition 6. $\alpha \in V_n$ is *active* if there is at least one $\beta \in V_t^+$, such that $\alpha \Rightarrow^* \beta$. Otherwise, the symbol α is *inactive*.

All inactive symbols and appropriate rules have to be removed from original grammar. The following algorithm will remove inactive symbols and their rules from an arbitrary CF-grammar $G = (V_n, V_t, S, \Phi)$ giving equivalent CF-grammar $G' = (V_n', V_t, S, \Phi')$ with only useful symbols and corresponding rules.

Algorithm:

```
$W_{new} \leftarrow \alpha \mid \alpha \rightarrow \beta \ \in \ \Phi, \ \ \beta \in V_t^*;$          initial set of active symbols
$m \leftarrow \mid V_n \mid;$          the number of non-terminal symbols
REPEAT
    $W_{old} \leftarrow W_{new}$
    $W_{new} \leftarrow W_{old} \cup \{\alpha \mid \alpha \rightarrow \beta \in \Phi, \beta \in (V_t \cup W_{old})^*\}$
    $m \leftarrow m - 1$
UNTIL $(W_{new} = W_{old}) \lor (m = 0)$
$V_n' \leftarrow W_{new}$
$\Phi' \leftarrow \{\alpha \rightarrow \beta \in \Phi \mid \alpha \in V_n', \ \beta \in (V_t \cup V_n')^*\}$
```

Reachable symbols

Definition 6. The symbol is *reachable* if for every $\alpha \in V_n \cup V_t$ there is at least one $S \Rightarrow^* \tau_1 \alpha \tau_2$, such that $\tau_1, \tau_2 \in (V_n \cup V_t)^*$.

All unreachable symbols have to be removed from the grammar together with the rules containing unreachable non-terminals. The following algorithm will remove the unreachable symbols and corresponding rules from CF-grammar with only active symbols $G' = (V_n', V_t, S, \Phi')$, giving the equivalent CF-grammar $G'' = (V_n'', V_t', S, \Phi'')$ containing only reachable symbols.

Algorithm:

```
$W \leftarrow S \ ;$          initial set of reachable symbols
$m \leftarrow \mid V_n' \mid;$          the number of non-terminal symbols
REPEAT
```

```
    $W_{temp} \leftarrow \{\alpha \in (V_n' \cup V_t) \mid \beta \to \tau_1\alpha\tau_2 \in \Phi', \beta \in W_{temp}\}$
    $W \leftarrow W \cup W_{temp}$
    $m \leftarrow m - 1$
UNTIL $(W = W_{temp}) \vee (m = 0)$
$V_n'' \leftarrow W \cap V_n'$
$V_t' \leftarrow W \cap V_t$
$\Phi'' \leftarrow \{\alpha \to \delta \in \Phi' \mid \alpha, \delta \in W_{temp}^+\}$
```

By applying the above two algorithms (in quoted order) any CF-grammar will be transformed into the equivalent reduced one. The order of application of algorithms is important because some new unreachable symbols can be introduced into the grammar, after all inactive ones are removed. On the contrary, the removal of unreachable symbols does not influence the appearance of inactive ones.

2.3 Toward Non-Circular CF-Grammar

Circular derivations (derivations of the form $\alpha \Rightarrow^* \alpha$, where $\alpha \in V_n$) can introduce ambiguity into compiler generator and force it into endless loops. The algorithm that transforms arbitrary CF-grammar into the equivalent non-circular grammar is based on backtracking algorithm *RemoveCirc(α)* which will remove all rules of the form $\alpha \Rightarrow^* \alpha$, $\alpha \in V_n$. The resulting set of rules will be designated as $\Phi(\alpha)$. Initially every rule in the set Φ is unmarked.

```
Algorithm RemoveCirc($\alpha \in V_n$)
$w \leftarrow \alpha$;        initial word produced by applying first $\alpha$
{Find $r$, such that $r \leftarrow \alpha \to \beta_1\beta_2\ldots\beta_n \in \Phi$,
$\beta_i \in V_n \cup \{\varepsilon\}$,  $i = 1,\ldots,n$, and $r$ is unmarked}
IF such $r$ can be found THEN
    mark the rule $r$
    $w = \ldots\alpha\ldots \leftarrow \ldots\beta_1\beta_2\ldots\beta_n\ldots$ ;
{replace the occurrence of $\alpha$ in $w$ with the right-hand side of the rule }
    IF $w = \alpha$ THEN
        $\Phi(\alpha) = \Phi \backslash \{\delta_1 \to \delta_2 \mid \delta_1 \to \delta_2 \in \Phi$ and is marked $\}$
```

```
            STOP
    ELSE
            FOR i=1 TO n
                    $RemoveCirc(\beta_i)$
            END
    END
END
```

Based on *RemoveCirc(α)*, the following algorithm for transformation of arbitrary CF- grammar $G = (V_n, V_t, S, \Phi)$ into the equivalent non-circular grammar $G' = (V_n, V_t, S, \Phi')$ is produced. It simply applies *RemoveCirc(α)* to every $\alpha \in V_n$.

Algorithm

```
$V \leftarrow V_n;$              temporary set of non-terminal symbols
$\Phi' \leftarrow \Phi;$              resulting set of rules
$\alpha \leftarrow \beta \in V_n;$           $\alpha$ is any non-terminal from $V_n$
REPEAT
    $\Phi(\alpha) \leftarrow RemoveCirc(\alpha)$
    $\Phi' = \Phi' \setminus \Phi(\alpha)$
    $V = V \setminus \alpha$
UNTIL $V = \emptyset$
```

The order in which the above three algorithms will be applied does not influence the resulting grammar. However, the most efficient sequence can be established depending on statistical figures over some common cases.

3. Implementation in general and data structure

All the algorithms given here are implemented in programming language Modula-2 and form a preprocessor that takes as its input an arbitrary CF-grammar and produces an equivalent non-circular reduced CF- grammar without empty rules. The resulting grammar can be directly given as an

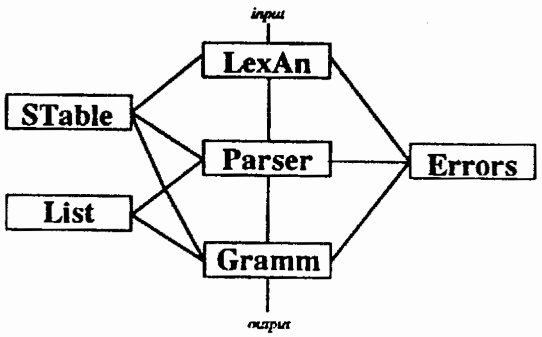

Figure 2: A general structure of a preprocessor

input to most compiler generators. The implemented preprocessor is itself a simple compiler (see Figures 1 and 2). It has to translate the source language into the target one, applying the above three algorithms. Target language can be produced immediately after semantic analysis (module **Gramm**) and no explicit code generation is needed.

The preprocessor takes as its input an arbitrary CF-grammar. Such arbitrary CF-grammar must be submitted to the preprocessor in precise form, that can be described by the following Backus- Naur notation:

```
<grammar> ::= <rule> {<rule>}
<rule> ::= <non-terminal> '=' {<symbol>}
<non-terminal> ::= "<" <words> ">"
<words> ::= <word> {("  "|"-") <word>}
<word> ::= <char> {<char>}
<char> ::= 'A'| ... |'Z'|'a'| ... |'z'|'0'| ... |'9'
<symbol> ::= <terminal> | <non-terminal>
<terminal> ::= "'" <words> "'"
```

Module **LexAn** implements the lexical analyzer of input grammar. It recognizes the next token in input and its type (`'<'`, `'='`, `'>'`, identifiers, etc.). **LexAn** is called whenever a new token is needed by the rest of the modules.

Module **Parser** implements the syntactic analyzer of input grammar. **Parser** is essentially driven by the above description of input grammar form. It calls **LexAn** whenever it needs a new token, builds the internal structure of one rule and reports any errors that might occur in the specification of input grammar.

Module **Gramm** takes the rules from **Parser** and builds an internal representation of the whole grammar. Module also contains all necessary operations over the grammars: for insertion of a rule into the grammar, for writing the grammar to the external media, for removing the rule(s) from grammar, ... as well as the implementation of the three mentioned algorithms.

Module **List** implements a general list, which is a basic data type for representation of all other structures - grammar rules, grammar itself and various sets. Every node of a list contains addresses of elements and elements themselves and can hold elements of different types. Module **List** is a central point of the whole preprocessor, because all other modules depend on its implementation.

Module **STable** implements a dictionary of all symbols occurring in input grammar and module **Error** reports all errors that occur during the work of the preprocessor.

A (general) list is defined in the following way:

```
List    = POINTER TO Header;
Header  = RECORD
            InfoSize: CARDINAL;
            First: Elem
          END;

Elem    = POINTER TO Element;

Element = RECORD
            Info: ADDRESS;
            Link: Elem
END;
```

General list defined in this way, can accept only the list elements of the same size. However, the generality of lists is not much reduced because in this particular implementation elements of the lists will be always of the equal size. In that way a lot of memory space is saved because the information about the size of the elements is saved only in the header node of the list and not in every list element.

Grammar rule, the whole grammar and the sets are all implemented as general lists. Elements of grammar rule and sets are pointers to symbol table, and elements of grammar are pointers to rules i.e., another lists. However, this difference is not "known" to the most of procedures dealing with lists - they treat the list in a uniform manner, no matter whether it represents the rule, the set or the grammar.

For example, the following procedure will assign n-th element of list L to (untyped) variable info. It can be applied to take n-th element of grammar rule, n-th element of the set, as well as to take n-th rule of the grammar.

```
PROCEDURE Nth(L: List; n: CARDINAL; VAR info: ARRAY OF BYTE );
VAR
   ptr: Elem;
   i: CARDINAL;
   tmp: POINTER TO BYTE;
BEGIN
   ptr := L^.First;
   IF n > 0 THEN
      DEC(n);
      WHILE (n > 0) AND (ptr <> NIL) DO
         ptr := ptr^.Link;
         DEC(n)
      END;
      IF ptr <> NIL THEN
         tmp := ptr^.Info;
         FOR i := 0 TO a^.InfoSize - 1 DO
            info[i] := tmp^;
            INC(tmp)
         END
      END
   END
END Nth;
```

4. Implementation of algorithms

As an illustrative example of the correspondence between the restated algorithms and their actual implementation, we refer to the following implementation of the algorithm for deletion of all inactive symbols from the grammar. Operations over grammars involved in this procedure are the following:

Procedure **InitList(VAR L: LIST; size: CARDINAL): List** creates an empty list, where all elements will be of the size size.

Function procedure **TakeNonTerminals(g: List): List** returns the set (i.e. the list) of all non-terminals of grammar g.

Function procedure **TakeTerminals(g: List): List** returns the set of all terminals of grammar g.

Function procedure **TakeSymbols(g: List; Set: List): List** returns the following set of symbols $\{\alpha \mid \alpha \rightarrow \beta \in \Phi, \beta \in Set^*\}$.

Function procedure **CardNumber(S: List): CARDINAL** returns cardinal number of the set S.

Function procedure **Union(S1, S2: List): List** returns the set which is the union of sets $S1$ and $S2$.

Function procedure **Same(S1, S2: List): BOOLEAN** returns logical truth value TRUE if sets $S1$ and $S2$ are equal, otherwise returns FALSE.

Function procedure **TakeRules(g: List, Set: List): List** returns the following set of rules $\{\alpha \rightarrow \beta \in \Phi \mid \alpha \in Set\}$.

The following function procedure **Active(g1: List): List** returns the grammar where all inactive symbols and corresponding rules are removed.

```
PROCEDURE Active(g1: List; VAR g2: List);

VAR
  m: CARDINAL;
  Vn, Vt, Wnew, Wold: List;
BEGIN
  InitList(Wnew, TSIZE(Symbol));
  Vn := TakeNonTerminals(g1);
  Vt := TakeTerminals(g1);
  Wnew := TakeSymbols(g1, Vt); $W_{new} \leftarrow \{\alpha \mid \alpha \rightarrow \beta \in |Psi, \beta \in V_t^*\}$
  m := CardNumber(Vn);        $m \leftarrow| V_n |$
  REPEAT
    Wold := Wnew;        $W_{old} \leftarrow W_{new}$
    Wnew := Union(Wold, TakeSymbols(g1, Union(Wold, Vt)));
      $W_{new} \leftarrow W_{old} \cup \{\alpha \mid \alpha \rightarrow \beta \in \Phi, \beta \in (V_t \cup W_{old})^*\}$
    DEC(m)        $m \leftarrow m - 1$
  UNTIL (Same(Wold, Wnew)) OR (m=0);        $(W_{new} = W_{old}) \vee (m = 0)$
  g2 := TakeRules(g1, Wnew);
    $\Phi' \leftarrow \{\alpha \rightarrow \beta \in \Phi \mid \alpha \in W_{new},\ \beta \in (V_t \cup W_{new})^*\}$
END Active;        $V_n' \leftarrow W_{new}$
```

5. Example

Data structure for grammar representation is displayed in Figure 3. Additional pointers placed in symbol table entries are pointing to the first and last rule of the same non-terminal. These pointers served as a device for faster manipulation with rules within a grammar. The data structure displayed here represents internally the following grammar:

```
<S> = <A> <B>.
<A> = 'a' <A>.
<A> = ε.
<A> = <C>.

<B> = 'b' <B>.
<B> =  ε.
<C> = 'c' <C>.
<D> = 'd'.
```

As a result of transformation of input grammar, the following grammar is produced.

```
<NEW> =  ε.
<NEW> = <S>.
<S> = <B>.
<S> = <A>.
```

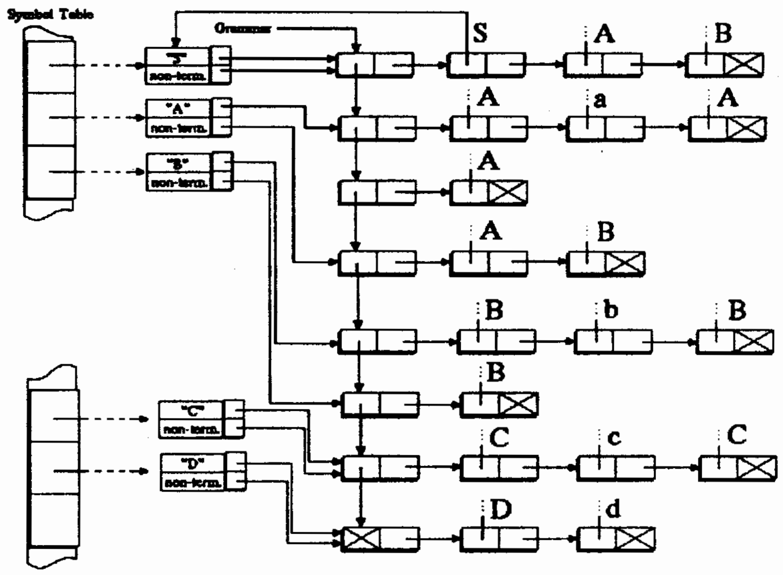

Figure 3: Internal representation of the grammar

```
<S> = <A> <B>.

<A> = 'a'.
<A> = 'a' <B>.
<B> = 'b'.
<B> = 'b' <B>.
```

Resulting grammar is suitable for most compiler generators, because it is reduced, non-circular and without empty rules.

6. Conclusion

The preprocessor for transformation of an arbitrary CF-grammar into its reduced non-circular empty-rules- free equivalent is implemented. The implementation is, above all, memory efficient, and can serve as an additional first step to most of the existing compiler generators.

The implementation is done based on restated algorithms for grammar transformations. Restated algorithms reflect the implementation issues better and lead to better memory usage.

The main advantages of the described implementation are:

- Dynamic nature. Because of that memory requirements of the whole preprocessor are proportional to the size of input grammar.

- Modular structure. Preprocessor is easy to change and maintain. The form in which grammars are specified is easily adjusted to suit almost any compiler generator. It is accomplished by changing only one procedure (for writing the grammar to external media).

- Uniform data structures. Most of the procedures are common for grammar rules, grammars and sets which reduce the size of preprocessor to one half of its possible size. That way, there is more memory space available for grammar processing.

References

[1] Aho, A., Sethi, R., Ullman, J.: , Compiler Principles, Techniques, and Tools,(1985) Addison-Wesley Publishing Company.

[2] Crelier, R.: OP2: A Portable Oberon-2 Compiler. In Proceedings of 2. International Conference "Modula-2 and beyond", (Loughborough, England), (1991), 58-67.

[3] Dobler, H., Pirklbauer, K.: Coco-2 - A new Compiler-Compiler, SIGPLAN Notices 25 (1991) 5.

[4] Ivanović, M.: An Implementation of Parser based on Simple Precedence Grammars, Master thesis, University of Novi Sad, (1988) Novi Sad (in Serbian).

[5] McGettrick, A.D.: The Definition of Programming Languages, (1980), Cambridge University Press, Cambridge.

[6] Tremblay, J.P., Sorenson, P.G.: The Theory and Practice of Compiler Writing, (1985), McGraw Hill, New York.

[7] Rechenberg, P., Mössenböck, H.: A Compiler Generator for Microcomputers, (1989), Prentice Hall, London, UK.

REZIME

O TRANSFORMACIJI KONTEKSTNO SLOBODNIH GRAMATIKA U OBLIK POGODAN ZA KORIŠĆENJE U GENERATORIMA KOMPAJLERA

Implementirani su algoritmi za transformaciju kontekstno slobodne gramatike u redukovanu, necirkularnu gramatiku bez praznih pravila. Implemantacija je oblika predprocesora koji uzima gramatiku kao svoj ulaz i proizvodi njen ekvivalent pogodan za korišćenje u generatorima prevodilaca programskih jezika. Predprocesor se može koristiti kao prvi korak u mnogim generatorima prevodilaca, kao i u primenama u teoriji formalnih jezika. Neki algoritmi korišćeni za transformaciju gramatika su preformulisani, a prikazani su i najvažniji detalji implementacije.

Received by the editors October 16, 1991