An Implementation of Affix Grammars

Hans Meijer

Informatics Department, Nijmegen University, The Netherlands

Summary. Intermediate results and current problems in an ongoing implemention of minimally restricted, possibly ambiguous Affix Grammars are described. Affix Grammars are informally introduced. A Recursive Backup Parsing Algorithm, suitable for any context-free grammar which is not left-recursive is presented, together with a heuristic scheme which is particularly effective at the lexical level. The main intermediate result is a transcription for affixes which allows affixes to be referenced before they are defined. The implementation of context sensitivity, which is the main current problem, is discussed. Other remaining problems are listed.

This publication contains material which may be used in the author's forthcoming doctoral dissertation.

1. Introduction

Affix Grammars were developed by C. H. A. Koster [KOSTER 1971]. A modification was proposed by D. A. Watt [WATT]. Like other two-level grammars (Two Level Van Wijngaarden Grammars and Attribute Grammars), Affix Grammars are extensions of context-free grammars.

This implies that many of the theoretical results concerning context-free grammars are applicable. Moreover, Affix Grammars formalize the semantic aspects of languages, which makes them suitable for parsing. For the user, Affix Grammars are very much like Two Level Van Wijngaarden Grammars. For the implementer there are many similarities with Attribute Grammars.

1.1. An Example

First we give an example of a simple Affix Grammar for a first glance at the style we shall use throughout this paper.

```
five ary to unary (unary value>):
   number ("", unary value), ".".

number (>prefix, value>):
   digit (digit value), number (new prefix, value),
   times 5 plus (prefix, digit value, new prefix).
number (>prefix, prefix>): .

digit (zero>):  "0".
digit (one>):   "1".
digit (two>):   "2".
digit (three>): "3".
digit (four>):  "4".

times 5 plus (>n, >d, n4+n+d>):
   times 2 (n, n2), times 2 (n2, n4).
times 2 (>n, n+n>): .

zero::"". one::"i". two::"ii". three::"iii". four::"iiii".

five ary to unary.
```

This grammar accepts 5-ary numbers, like 23. and generates their values in unary, in this case iiiiiiiiiiiii. It is a context-free grammar in one-level Van Wijngaarden notation, extended with affixes. Left hand side and right hand side of rules are separated by colons. Members at the right hand side are separated by comma's. Rules are terminated by dots. For each alternative of a nonterminal a separate rule is written, we do not collect alternatives in one rule. The affixes are separated by comma's and enclosed in parentheses. Affix expressions are sequences of affix variables and constants separated by plusses.

The rule for *times 5 plus* inputs two affixes, named *n* and *d* (the flow symbol (">") is written here before the affixes). The first member *times 2* takes the affix *n*, doubles it and names the result *n2*. The second member *times 2* takes the value of *n2*, doubles it and names the result *n4*. Finally, the rule produces (the flow symbol is written after the affix expression) an affix which contains a*5+b i's, where a is the

number of i's in *n* and b is the number of i's in *d*.

The affix variables *zero*, *one*, etc. are defined using metarules. Metarules are also terminated by a dot, have an affix variable as their left hand side and an affix expression as their right hand side, separated by two colons.

Terminals and affix constants are written as string constants. The last line in the example is the specification of the initial symbol.

1.2. Project Goal

The set of Affix Grammars may be regarded as constituting a programming language which is especially suited for writing transducers or compilers. This idea has led in the past to the definition and implementation of the language CDL [CDL]. The purpose of our project is to write a compiler for Affix Grammars with as few restrictions as possible. The two most important aspects which we do not want to restrict are ambiguity and affix evaluation order (i.e., an affix need not already have a value when it is referenced).

We consider writing the (compiler-)compiler primarily as an engineering task. As a result, in the current stage of the project, no theoretical results are available for inclusion in this paper.

1.3. Outline Of The Paper

Chapter 2 gives an informal introduction to Affix Grammars. Readers who are familiar with Affix Grammars should skip this chapter.

Chapter 3 explains the algorithm which we use for parsing the context-free grammars underlying Affix Grammars.

Chapter 4 describes the main results of the project up to the current stage: the implementation of transducers, i.e., Affix Grammars which do not specify context conditions.

Chapter 5 very briefly indicates the problem which is currently being investigated: the implementation of context sensitivity.

Chapter 6 mentions some minor problems to be solved in the future.

In the text we assume some familiarity with (context-free) grammars and the languages Algol 60 and Algol 68.

1.4. Current Status

A transducer-compiler, written in Algol 68, generating transducers in Algol 60, runs under MVS on an IBM 370/158. Using this program, several transducers were generated:

- accepting context-free grammars with regular right parts,
- generating transducers with various optimizations,
- generating transducers in machine language,
- generating a compiler with a very general but expensive implementation of context sensitivity (for experimental purposes, see chapter 5).

Also a compiler for context-free grammars, utilizing the heuristics described in chapter 3, which generates parsers in assembler language, runs on the IBM 370/158. The generated parsers are callable as Algol 68 procedures. After each successful parse, the parse tree is available for inspection. This compiler is successfully being used in the department of Computer Linguistics of the Nijmegen University.

A compiler for transducer grammars was recently finished. It is written in CDL2 [CDL] and generates programs for an abstract, low level, stack-oriented machine. The compiler currently runs on a VAX, under VMS, but is easily portable to many other machines (CDL2 is a highly portable language). This compiler is the embryo of the final (compiler-)compiler mentioned above. For the time being it allows quick implementation of experimental transducers, written as Affix Grammars.

2. Affix Grammars

This chapter presents a brief outline of different versions of Affix Grammars. As indicated earlier, we follow a pragmatic approach and try to avoid formalism.

For most programming languages, one of the aspects which cannot be defined by a context-free grammar is the identification of properties of objects in the context where they are used. Part of the definition of a *serial clause* in Algol 68, like

*INT i = 3; ...; REAL y := a*i; ...; y*

using a context-free grammar might be

serial clause: declarations, expression.

declarations: declaration, semicolon symbol, declarations.
declarations:

declaration: declarer, defining identifier,
equals symbol, expression.

expression: applied identifier.

This grammar allows the *applied identifier* to be any identifier, independent of the *defining identifiers*. Moreover, the identification of certain properties such as the mode, necessary for a correct interpretation of an applied identifier, is not defined. These context dependencies can not be expressed in a context free grammar: as there is an infinity of identifiers and modes, an infinite number of rules is required.

Affix Grammars, like other two level grammars, allow the (finite) specification of an infinite number of rules by parametrizing the rules of an underlying context-free grammar.

A nonterminal may be associated with parameters or affix positions. For each affix position a set of affix values is defined (the domain of the affix position). In the rules of the grammar, each affix position is occupied by an affix variable or affix value. When applying a rule in the derivation of a sentence (program) each affix variable must be replaced by an affix value in the domain of its position. This replacement must be done systematically, i.e., affix variables occurring more than once in the rule must be replaced by the same affix value.

In our example, we associate both the *defining identifier* and the *applied identifier* with one affix position, whose domain is the set of 'tag's. A tag may be considered as the "internal representation" of an identifier. We further associate the *declarer* with one affix position, whose domain is the set of 'mode's. For the moment, we extend our rule for *declaration* to

declaration (...):
declarer (mode), defining identifier (tag),
equals symbol,
expression (...).

In [KOSTER 1971] and [WATT] other conventions are used for delimiting nonterminals and affix variables or values. This is confusing, but using the original notation from [KOSTER 1971] would cause even greater confusion within this paper.

At this point we assume that for some particular tag *T* the external representation of *T* will be generated by *defining identifier (T)*. Likewise we require that all possible

representations for some particular mode *M* are derivable from *declarer (M)*.

We also introduce one affix position for *declaration*. Its domain is the set of all definitions, i.e., all tag/mode-pairs. It should be clear that we have to establish a relationship between the affix variables *definition, mode* and *tag:* the value of the definition is composed of that of the mode and the tag. For this purpose, Affix Grammars introduce predicates.

A predicate is written as a nonterminal with affix positions, but its "only" purpose is to operate on its affixes. Its contribution to the derivation of the sentence (program) is either the empty symbol or the "forbidden symbol" *omega*. Whether it produces the empty symbol or *omega* depends on the values at its affix positions. The predicates are defined by computable functions of the affix values, essentially:

IF the affix values satisfy a certain condition
THEN produce the empty symbol
ELSE produce omega
FI

No sentence (program) is allowed to contain the forbidden symbol. Thus, no sentence can be derived which does not satisfy the predicate's condition.

In our case, we introduce the predicate

define (definition, mode, tag)

with the trivial condition that the value of the affix variable *definition* must be composed of the values of the affix variables *mode* and *tag*. Now we can write the rule for *declaration* as

declaration (definition):
declarer (mode), defining identifier (tag),
define (definition, mode, tag),
equals symbol,
expression (...).

In order to identify the mode of an applied identifier, we must have all definitions available which are valid in the context of that applied identifier. For this, we introduce an affix variable *environ*, which stands for a set of definitions (its domain is the set of all sets of definitions). We also introduce a predicate

compose (environ, definition, old environ)

which checks whether the *environ* is composed of the *definition* and the *old environ*. It furthermore fails if the definition is already in the old environ. Using this we can write

declarations (environ):
declaration (definition), semicolon symbol,
declarations (old environ),
compose (environ, definition, old environ).
declarations (E): .

where *E* is the affix value representing an empty environ.

For the serial clause we can now write

serial clause (mode):
declaractions (environ), expression (environ, mode).

and for the expression

expression (environ, mode):
applied identifier (tag),
apply (environ, tag, mode).

with the predicate *apply* which checks whether the identifier occurs in the environ with the given mode.

The grammar, as it is now, not only defines which sequences of symbols contitute correct serial clauses, but also inhibits multiple definitions of identifiers, requires applied identifiers to be defined and states the mode of the value of serial clauses.

It remains to associate affixes with the member *expression* in the rule for *declaration*. Its mode must conform that of the declaration. In, for instance,

INT i = expr

the value of *expr* must be an integer. In Algol 68, expressions in this context may apply all identifiers defined in the serial clause. This requires that the complete environ must be available in all declarations.

```
serial clause (mode):
   declarations (environ, environ),
   expression (environ, mode).
declarations (environ, overall environ):
   declaration (definition, overall environ),
   semicolon symbol,
   declarations (old environ, overall environ),
   compose (environ, definition, old environ).
declarations (E, overall environ): .
declaration (definition, overall environ):
   declarer (mode), defining identifier (tag),
   define (definition, mode, tag),
   equals symbol,
   expression (overall environ, mode).
expression (environ, mode):
   applied identifier (tag),
   apply (environ, tag, mode).
```

2.1. Affix Flow

When using Affix Grammars, as they have been described up to now, for deriving sentences of the language, we should try all possible combinations of values for the affix variables, until one is found which satisfies all predicates. In particular, we should try all possible modes for the affix of *serial clause*.

On the other hand, if we think of parsing a given sentence using this grammar, it is obvious that for instance the tag associated with the defining identifier can have only one value, the internal representation of that identifier. Instead of trying all tags until the corresponding one is found, it can be immediately derived from the identifier. The same is true for the mode of the declarer. From these, the *definition* can be derived, etc.

To make Affix Grammars suitable for parsing in this sense, the affix positions of each nonterminal or predicate are specified as either derived or inherited. An affix position should be specified as derived, if its value contains information about the "contents" of the nonterminal and as inherited, if it contains information about the "context" of the nonterminal.

In our notation, we indicate the flow of affix positions in the left hand side of rules. An affix position is specified as inherited/derived by writing a ">" preceding/following the affix position. Our grammar fragment becomes

serial clause (mode>):
declarations (environ, environ),
expression (environ, mode).
declarations (environ>, >overall environ):
declaration (definition, overall environ),
semicolon symbol,
declarations (old environ, overall environ),
compose (environ, definition, old environ).
declarations (E>, >overall environ): .
declaration (definition>, >overall environ):
declarer (mode), defining identifier (tag),
define (definition, mode, tag),
equals symbol,
expression (overall environ, mode).
expression (>environ, mode>):
applied identifier (tag),
apply (environ, tag, mode).

with the assumption that

defining identifier (tag>): ...
applied identifier (tag>): ...
declarer (mode>): ...

and that the predicates are handled as if they have the left hand sides

define (definition>, >mode, >tag)
apply (>environ, >tag, mode>)
compose (environ>, >definition, >old environ)

We now can follow the "flow" of the affixes. In the rule for *declaration* for instance, if for each *declarer* there exists exactly one *mode* and for each *identifier* exactly one *tag*, the predicate *define* can map these on to exactly one *definition*, which is in turn produced by *declaration*.

We observe that in this rule the *identifier* delivers a tag and that *define* uses it. Likewise, *define* delivers a definition, which is used by *declaration*, the left hand side, for delivery in rules where it occurs in the right hand side. In other words, the first affix position of *define* assigns a value to the affix variable *definition*, which is assigned to the first affix position of *declaration*. Therefore, derived positions in the right hand side of rules and inherited positions in the left hand side are called defining positions, while derived positions in the left hand side and inherited positions in the right hand side are called applied positions.

2.2. Well Formedness

Basically, the choice of flow for affix positions is free. In order to make Affix Grammars suitable for parsing, however, defining positions should not produce an infinite number of

affixes. For reasons of complexity, they should even produce only very few affix values.

To this end, several well-formedness conditions were introduced in [KOSTER 1971]. One of them states, that predicates, if they do not fail, must uniquely map their inherited affixes into their derived affixes. In our example we have assumed that the predicates *define, apply* and *compose* satisfy this condition. The condition could be weakened by allowing predicates to produce only a finite (preferably small) number of values, thus introducing ambiguity at the affix level.

Another condition is that each variable occurring at an applied position should have exactly one defining occurrence. If it did not have a defining occurrence, all values in the domain of the variable would be applicable at the applied occurrence. Again, this might be allowed for finite domains.

On the other hand, if a variable is defined more than once, the systematic-replacement rule requires that both occurrences should define the same value.

There is yet another condition which states that variables must be defined before they are applied: if the (unique) defining occurrence appears at an affix position of the n'th member at the right hand side, no applied occurrence should appear in the first n members of the right hand side.

This condition ensures that the values of affixes, during a left to right parse, are available when they are used. It implies that affix variables may only depend on the left context of the members which use them. This well-formedness condition allows predicates to be evaluated during the parsing of the sentence. Thus, certain context conditions are checked during parsing and prohibit useless parsing (affix-directed parsing).

In our example we have violated some of the well-formedness conditions. In the rule for *declaration* the value of *mode* is defined in both *declarer* and *expression*. This fact expresses the context-condition that the mode of the expression must conform to that of the declaration. Furthermore, in the rule for *serial clause* the variable *environ* is both defined and applied in the same member *declarations*. This is a direct consequence of the fact that in Algol 68 identifiers, etc. may be applied before they are defined.

The main purpose of this project is to investigate implementations of Affix Grammars with as weak well-formedness restrictions as possible. Basically, the (parser/ transducer/ compiler)-generator should accept any Affix Grammar, but in order for the generated program to be terminating, the grammar should obey certain restrictions (defining affixes finite, finite domains for only-applied affix variables, no circular

affix definitions, etc.), the analogue being a compiler which also accepts non-terminating programs.

Of course, the generator must issue warnings or error-messages for each dangerous situation it can statically detect.

2.3. Watt's Extension

In Affix Grammars, the predicates are defined to be any computable function. In practice, these functions must be actually expressed in some language like lambda calculus, Algol 68, machine language, etc. Thus Affix Grammars are, for their interpretation, dependent on the semantics of that language.

The predicates allow trivialization of the grammatical aspects of language definition. Any language could be defined by an Affix Grammar like the following:

```
program (object>): read (sentence),
                   compile (sentence, object).
read (sentence>): symbol (token), read (remainder),
                  concatenate (token, remainder, sentence).
read (E>): .
```

Here, *concatenate* is a very simple predicate and *compile* a very complicated one. In [WATT] a modification called Extended Affix Grammars is described which restricts Affix Grammars to have only two predicates, *synthesize* and *analyze*, which are predefined for each type of object in affix domains (integer, string, set, tuple, etc.). Subsequently the grammars are extended by allowing affix expressions at affix positions, which embody the *synthesize* and *analyze* predicates. Affix expressions at applied positions are synthesizing, affix expressions at defining positions are analyzing.

In the (Extended) Affix Grammars used here, affix expressions are written as sequences of affix variables and constants (affix terminals), separated by the symbol +, which can be interpreted as the operation concatenation (, addition, set union, ...) for affix values of type string (, integer, set, ...). If not stated otherwise we shall assume that the affix values are all of type string. We shall try to get by with as few "pre-defined" predicates as possible.

In our example, the rule for *declarations* is now written

```
declarations (definition + environ>, >overall environ):
    declaration (definition, overall environ),
    semicolon symbol,
    declarations (environ, overall environ).
```

which indeed is a simplification. On the other hand, the predicate *apply* must now be written as a set of rules

```
apply (>tag + mode + environ, >tag, mode>): .
apply (>tag1 + mode1 + environ, >tag, mode>):
    not equal (tag, tag1),
    apply (environ, tag, mode).
apply (>tag + mode, tag>, mode>): .
```

where *not equal* is assumed to be defined elsewhere. In practice, a few rules like *not equal* will be predefined.

For domain specification of affix variables metarules are used, like

```
definition:: tag + mode.
environ:: definition + environ.
environ:: .
overall environ:: environ.
```

These are again context free grammars, defining languages which are the domains of the affix variables. Like Two Level Van Wijngaarden Grammars, Affix Grammars are equivalent to Chomsky Type 0 grammars.

3. Recursive Backup Parsing of Context Free Grammars

In this chapter we shall use a small, ambiguous, context-free grammar as an example:

```
(1) s: x.          (3) a: y, x.        (6) x: "0".
(2) s: x, a, s.    (4) a: s, y, a.     (7) y: "1".
    s.             (5) a: s, s.
```

The grammar is found in [HOPCROFT].

3.1. Characterization

The Recursive Backup Parsing Algorithm [KOSTER 1974] is a top-down method, immediately based on leftmost rewriting. It is suitable for any context-free grammar which is not left-recursive. In particular, it handles ambiguity in a reasonably efficient way, its backup administration is elegant and transparent. Furthermore, if the grammar is LL(k), its behaviour is linear.

3.2. The Algorithm

In leftmost rewriting, we maintain a string of terminals and nonterminals, the rewrite string, which we manipulate by replacing the leftmost nonterminal by some alternative for that nonterminal. In a depth-first version, we replace the leftmost nonterminal by one of its alternatives and explore recursively all possible derivations of the updated string. It is then

replaced by a second alternative, and so on. The exploration continues as long as the terminals to the left of the leftmost nonterminal match the (leftmost part of the) input string. If the grammar is not left-recursive, this process always terminates.

After having replaced the leftmost nonterminal by an alternative, we explore the updated string *U*. During exploration the string may be changed at will, but we require that after exploration, whether matches were found or not, the string is restored to *U* and the alternative just explored is replaced, reversely, by its left hand side.

This balancing is essential for the algorithm. Roughly speaking, each routine undoes its own global effects.

For backup we maintain a backup stack. When the leftmost nonterminal is replaced by an alternative, an identification of this alternative is saved on that stack. Returning from the exploration of this alternative, we take the identification from the stack, replace the nonterminal by the next alternative, etc.

For the example grammar the algorithm will, parsing the string 001100, eventually arrive at the rewrite string

00asyas

with a backup stack B. We have named the rules 1, 2, ... The algorithm will run through the following configurations:

```
00      asyas  B
00     yxsyas  B3
001     xsyas  B37
0010     syas  B376       {mismatch}
001     xsyas  B37 (6)    {last alternative for x}
00     yxsyas  B3 (7)     {last alternative for y}
00      asyas  B (3)
00    syasyas  B4
00    xyasyas  B41
000    yasyas  B416       {mismatch}
00    xyasyas  B41 (6)    {last alternative for x}
00    syasyas  B4 (1)
00 xasyasyas  B42
000 asyasyas  B426       {mismatch}
00 xasyasyas  B42 (6)    {last alternative for x}
00    syasyas  B4 (2)
00      asyas  B (4)
00     sssyas  B5
    ...
00     sssyas  B5
00      asyas  B (5)
```

We write the top element of the backup stack in parentheses when it is actually taken from it, but must be retained until

the next alternative, if any, has been determined.

This general approach can be simplified. We need not actually maintain the terminal prefix of the rewrite string. Instead, we may have a pointer into the input string. If the leftmost nonterminal rewrites to a terminal string, we compare it with the substring to the right of the pointer. If they do not match, we backup immediately, i.e., we continue with the next alternative, if any. If they do match, we increase the input pointer by the length of the terminal string and continue rewriting. After having explored this configuration, we decrease the input pointer by the same amount. The input string together with its pointer behave like a stack.

The remaining right part of the rewrite string is also a stack. Returning from an alternative we remove this alternative from the (rewrite) stack, but only put the left hand side back, if there are no other alternatives: if we restore it and go on with the next alternative, we must immediately remove it again.

This leads to the following transcription of the rules for *a:*

```
a: push yx on the rewrite stack;
   continue;
   pop yx from the rewrite stack;

   push sya on the rewrite stack;
   continue;
   pop sya from the rewrite stack;

   push ss on the rewrite stack;
   continue;
   pop ss from the rewrite stack;

   push a on the rewrite stack;
   backup
```

where *continue* is:

```
pop 1 element from the rewrite stack;
call it as a subroutine
     pushing the return point on the backup stack
```

and *backup* is:

```
pop 1 element from the backup stack;
jump to it
```

thus, a simple return-from-subroutine.

The transcription of the rule for x is:

```
x: IF the string with length 1 at the input pointer is "0"
   THEN add 1 to the input pointer;
        continue;
        subtract 1 from the input pointer
   FI;
   push x to the rewrite stack;
   backup
```

It remains to indicate if and when the input sentence has been completely recognized. This is the case when the input pointer points just beyond the input string and the rewrite stack is empty. We shall leave the first part of this condition to the grammar writer. He must choose a terminator character, write it at the end of the input string and prescribe its function in the grammar.

Detection of emptiness of the rewrite stack is very simple. Let *m* be a name different from all nonterminals of the grammar. The algorithm includes

```
m: write a message indicating that the input string
       was matched;
   push m to the rewrite stack;
   backup
```

Now, the algorithm is fired by

```
initialize the backup- and rewrite stacks as empty stacks;
make the input pointer point to the first character of the
   input string;
push m to the rewrite stack;
continue
```

It may be concluded that the algorithm is an enumeration of all sentences of the language generated by the given grammar. It terminates since the enumeration in the depth stops on a mismatch and the grammar is not left-recursive. The backup administration is completely covered by balancing the rewrite-, input- and backupstacks.

3.3. Using An Algorithmic Stack

As we have seen, the backup stack behaves exactly like a stack of return addresses for subroutine calls. The *continue/backup* subalgorithms are essentially subroutine call/return over the backup stack, respectively, where *continue* POPs the subroutine to be called from the rewrite stack.

Thus the rewrite stack is a stack of subroutines. It is only a small step to implement the rewrite stack as a subroutine (representing the top element) which eventually calls the subroutine which represents the remainder of the stack. Then POPping the stack is just calling the subroutine representing it and PUSHing the stack is creating a subroutine

which represents the PUSHed element and eventually calls the remainder.

In a language like Algol 68 we can dynamically create procedures from other procedures if we pass them as parameters. Thus we do not maintain a global rewrite stack, but pass it as a parameter to the procedures which are the transcriptions of the rules of the grammar.

The transcription of the rules for s looks like:

```
RULE s = (STACK q) VOID:
     BEGIN STACK q1 = VOID: x (q);    CO PUSH x CO
           q1;                        CO continue CO
           STACK q2 = VOID: s (q);    CO PUSH s CO
           STACK q3 = VOID: a (q2);   CO PUSH a CO
           STACK q4 = VOID: x (q3);   CO PUSH x CO
           q4
     END
```

The rewrite stack is wholly embedded in the call/return stack of the Algol 68 implementation. In fact, the call/return stack is both the rewrite and backup stack. No explicit balancing is necessary since we use the parameter q in all alternatives directly. It is a disadvantage that we no longer have "pure" stack operations: any procedure may call any other procedure in lower regions of the stack.

We observe that the last PUSH of each alternative is superfluous: it is immediately POPped by a call. Also, we can avoid the temporary identifiers by writing the routine texts directly as actual parameters.

Then, the transcription of the rules for a is

```
RULE a = (STACK q) VOID:
     BEGIN y (VOID: x (q));
           s (VOID: y (VOID: a (q)));
           s (VOID: s (q))
     END
```

This is still rather ugly because of the many VOID's. See the next section for an elegant version in Algol 60.

The other rules are transcribed straightforwardly:

```
RULE x = (STACK q) VOID: match ("0", q);
RULE y = (STACK q) VOID: match ("1", q)
```

The match routine is

```
PROC match = (STRING t, STACK q) VOID:
     IF INT index = scanner + UPB t - 1;
        index <= UPB input CO will fit CO
     THEN IF input [scanner:index] = t
          THEN scanner +:= UPB t;
               q;
               scanner -:= UPB t
          FI
     FI
```

The calling program is

```
STRING input; read (input); INT scanner := 1;
STACK m = VOID: print (("match", new line));

s (m)
```

Although the parser and the transducer which will be explained in the next chapter are actually implemented in machine language using the version of the algorithm presented in the former section, the implementations are illustrated here by means of this Algol 68 version.

3.4. Algol 60 Version

Since Algol 60 knows the coercion "proceduring" by virtue of its call-by-name mechanism, the Recursive Backup Parsing Algorithm may be elegantly expressed in Algol 60. The body of the procedure which implements the rules for *a* is, for instance, written as

```
y (x (q));
s (y (a (q)));
s (s (q))
```

Because of its simplicity, one need not even write the grammar first. One can immediately write the parser with (almost) the same effort of writing the grammar.

But in order to be able to use call-by-name, the actual parameters, like *y (a (q))* must be expressions. Thus, the rule transcriptions must be type-procedures, not just procedures. The type itself has no relevance, we may choose real, integer or Boolean. From the Algol 60 Revised Report it is not clear whether type-procedures may only be called in expressions nor whether type-procedures always must return values (it only states that it must if the procedure is called in an expression). Most implementations allow type procedures to be called in procedure statements. Under this assumption, the parser for our sample grammar, in Algol 60, in an even more condensed version, reads:

```
procedure parse (sentence, message);
   integer array sentence; procedure message;
   begin  Boolean procedure s (q); Boolean q;
             begin match (0, q);
                   match (0, a (s (q)))
             end;
          Boolean procedure a (q); Boolean q;
             begin match (1, match (0, q));
                   s (match (1, a (q)));
                   s (s (q))
             end;
          Boolean procedure match (s, q);
             integer s; Boolean q;
             if s = sentence [pointer]
             then begin pointer := pointer + 1;
                        match := q;
                        pointer := pointer - 1
                  end;
          Boolean procedure write; message;

          integer pointer;
          pointer := 1;
          s (write)
   end
```

3.5. Heuristics

In practice it is desireable to speed-up the Recursive Backup Parsing Algorithm. We shall describe a heuristic which showed good results in some experiments.

For some or all nonterminals and for some or all positions in the input string all different lengths which were found for these nonterminals at these positions are remembered. In general this requires a 3-dimensional table, where the first dimension indexes the nonterminals, the second one indexes the input positions and the third one all possible lengths.

Before discussing the implementation of this table, let us consider the implementation of its access. When entering the procedure for nonterminal *a* we must first establish whether we have already explored this nonterminal at this postition. For the time being we shall assume that we have an array of truth values named *already*. If we have already explored the nonterminal here, we can just iterate through the different lengths:

```
RULE a = (STACK q) VOID:
     IF already [index for a, scanner]
     THEN FOR length FROM 0 TO max length
          DO IF occurred [index for a, scanner, length]
             THEN scanner +:= length;
                  q;
                  scanner -:= length
             FI
          OD
     ELSE explore
     FI
```

If we must explore, we must basically apply the usual algorithm. But, for each match found for the nonterminal, we must indicate the length of that match. We can take the same approach that we took earlier for testing the emptiness of the rewrite stack. Instead of passing just the STACK *q* to the alternatives, we pass a reduce procedure which on its turn calls *q*. This reduce procedure is called whenever a match for the nonterminal starting at the current position is found. If we also reserve a local variable to remember the current position, we can easily establish the length of all matches (we use a refinement notation):

```
explore:
   INT start position = scanner;
   STACK reduce = VOID:
         BEGIN INT length = scanner - start position;
               occurred [index for a,
                         start position,
                         length] := TRUE;
               q
         END;
   y (VOID: x (reduce));
   s (VOID: y (VOID: a (reduce)));
   s (VOID: s (reduce)).
```

Experiments indicate that the average time consumption for "normal" grammars is acceptable.

There are several ways to reduce the storage complexity by increasing the time complexity. The bit table *occurred* is sparse: only a relatively small number of different lengths occurs and far from all nonterminals occur at all positions. This indicates that a linked list implementation may be profitable. Also worth investigating is hashing on the pair [nonterminal index, position] and storing all different lengths found for that pair in a linear linked list.

Also, it seems unnecessary to apply the heuristic to all nonterminals. Some nonterminals may have optimized transcriptions, others may not. In a syntax like that of Algol 68 it is probably sufficient to optimize only the rules for 'tag's and 'denotation's.

Again another, more elegant, dynamic approach is to allow only a fixed number of [nonterminal index, position] pairs to remember their lengths, with priority for the most frequently occurring pairs. If a pair is thrown out, it must again follow the more elaborate algorithm. This scheme probably stabilizes at the most frequently used rules.

If we restrict the heuristic to the lexical level, all attempts to derive or require a separate pass for the lexical level are superfluous. However long-winded the rules for 'identifier', 'number', etc. are, the heuristic will usually "terminalize" these nonterminals completely. As to which nonterminals are to be considered lexical may be determinable by a simple calculus.

4. Transducers

In this chapter we use an example borrowed from [KNUTH]. It is a grammar which describes the transcription of rational numbers in binary notation, like

1001.011

into a sum of powers of two, in this case

2↑(0+1+1+1)+0+0+2↑(0)+0+2↑(-(1+1+1)+1)+2↑(-(1+1+1))

This result may seem of little interest but the grammar has a few illuminating aspects.

The idea is to handle both the integer and fractional part as binary integers. A 0-bit is transcribed into a term 0. A 1-bit is transcribed into a power of two. The exponent is written as a sum of two terms, the second being the position value of the 1-bit. The first term, in case of the fractional part, is the amount by which the binary point is shifted, i.e., the number of bits in the fractional part with a minus sign for shifting left. In case of the integer part the first term is 0.

The initial symbol of the grammar is *number*, with

number (value>): rational (value), terminator.

The first alternative of *rational* is simple:

rational (value>): integer (value, "0", length).

Here, the *length* is ignored, the shift term, to be passed to each digit, is fixed to *"0"*. The second alternative reads

```
rational (whole value + plus + fraction value>):
   integer (whole value, "0", whole length),
   point symbol,
   integer (fraction value,
            minus + "(" + fraction length + ")",
            fraction length).
```

where the first member is like the first alternative. In the third member, the shift term is built from the *fraction length*. Here, the second affix can not be evaluated before the whole number is parsed, since the third affix depends on the last digit.

The alternative for *integer* which does the main job is

```
integer (digit value + plus + integer value>,
         >shift,
         "1" + plus + integer length>):
   digit (digit value, shift + plus + integer length),
   integer (integer value,
            shift,
            integer length).
```

The value of the integer is the sum of the values of its components. The length is increased by 1. The exponent is composed of the shift term and the length of the remainder. This, too, is the use of an affix which is not yet available.

The other alternative for *integer* initializes values:

```
integer (digit value>, >shift, "1">):
   digit (digit value, shift).
```

The integer length is 1 here. The exponent is just the shift term, the position value is 0.

The rules for *digit* are straightforward:

```
digit ("0">,                                    >exponent): "0".
digit ("2" + power + "(" + exponent + ")">, >exponent): "1".
```

Other trivial rules are

```
point symbol: ".".
terminator:   " ".
plus ::       "+".
minus ::      "-".
power ::      "↑".
```

4.1. Transducer Restriction

Recall that the example grammar represents a transducer: no defining position contains an affix expression. For instance, the first alternative for *integer* defines the affixes *shift* (2nd position of left hand side), *digit value* (1st position of member *digit*), *integer value* (1st position of member *integer*) and *integer length* (3rd position of member *integer*). From these affixes, together with the meta-affixes and the constant terms the other (applied) affixed are composed.

4.2. Affix Mappings

If we consider the first alternative for *integer*, we observe that after having matched the member *digit* its defining affixes (in this case *digit value* only) are determined up to the values of its applied affixes. If the grammar were left-to-right well-formed, the applied affixes would be determined at this point and therefore also the defining affixes. In our case, as in our example, the applied affix depends on *shift* and *integer length*. The latter also depends on *shift*, among others. This means that we cannot even evaluate the derived affixes on the left hand side, since they indirectly depend on *shift*, whose value may not be known yet after having matched the nonterminal at the left hand side (which in our case is true; the value of *shift* is not known before the whole input string is matched).

Thus, the derived affixes cannot be evaluated before the whole input string is matched. They are determined up to the values of the inherited affixes of the same (incarnation of their) nonterminal.

This means that the derived affixes should not be evaluated to elementary objects (strings), but to mappings yielding these objects when called with the proper arguments (i.e., the values of the inherited affixes).

In terms of Algol 68, let AFFIX be the mode of affix values. Let a nonterminal have h inherited and d derived affix positions. Then we express the inherited affixes as *AFFIX* objects and the derived affixes as

PROC (AFFIX, ..., AFFIX) AFFIX

mappings, with h parameters. We call the mode of these derived affixes *DERh* for h parameters. Thus

MODE DER2 = PROC (AFFIX, AFFIX) AFFIX

Let us now consider how these procedures are actually constructed. The first derived affix of the left hand side of our example rule is *digit value + plus + integer value*. It must be written as a procedure with one parameter. The actual value of this parameter will eventually be the value of the inherited

affix *shift*. Therefore, we can interpret the inherited affixes at the left hand side as formal parameters:

```
(AFFIX shift) AFFIX: evaluate derived affix
```

The first term, *digit value* is a DER1 object, which is to be called with the expression *shift + plus + integer length* as actual parameter. Here, *integer length* is a DER1 object, to be called with *shift*. It is easy to see, that ultimately the original expression can be expressed in terms of constants, meta-affixes and inherited/left affixes (the formal parameters):

```
(AFFIX shift) AFFIX:
   digit value (shift + plus + integer length (shift)) +
   plus +
   integer value (shift)
```

The other derived/left affix becomes

```
(AFFIX shift) AFFIX:
   "1" + plus + integer length (shift)
```

We assume that string denotations are also denotations for mode *AFFIX*. We can always introduce a conversion operator.

Having established how the derived affixes can be expressed as functions of the inherited affixes we still must guarantee the availability of the global terms in these procedures. The meta-affixes cause no problems. We shall assume that these are available throughout the program as objects of the mode *AFFIX*. Since meta-affixes are expressed in terms of constants and other meta-affixes only, this is easily accomplished.

The availability of the derived affixes at the right hand side, in this case *digit value, integer value* and *integer length,* is a little more complicated. The member *digit* must produce the DER1 object *digit value,* the member *integer* must produce the DER1 objects *integer value* and *integer length*. In other words, we basically want *digit* to be a parameterless, one-valued function of mode *PROC DER1* and *integer* a parameterless, even two-valued function. We could simulate many-valued functions in Algol 68 by means of structures or try to find another language, but procedure-valued-procedures lead to scope problems, which would also appear in other languages.

These problems may be solvable, but we prefer an elegant solution, based on the idea of continuations. It very nicely combines with the Recursive Backup Parsing Algorithm. The concept of continuations is already used for many years and plays an important role in Denotational Semantics. The application in Algol 68 looks complicated but its realization in a low level language is simple and efficient.

The parsing procedure for our example rule is

```
RULE integer = (STACK q) VOID:
     BEGIN digit (VOID:
                  integer (q));
           ...
     END
```

The fact that an integer must be matched after a digit, is expressed by specifying *VOID: integer (q)* as the continuation of *digit*, while the continuation of the rule as a whole, *q*, is specified as the continuation of member *integer*, the last member of this alternative. Note that the actual sequentialization is done in the procedure which implements the terminals.

Thus everything to be done after the completion of *digit* is expressed in its continuation. As a result, anything which *digit* yields as a result, must be passed to its continuation.

Since *digit* yields one DER1 object, its continuation must have the mode

```
PROC (DER1) VOID
```

Likewise, *integer* must have a continuation with mode

```
PROC (DER1, DER1) VOID
```

We write *CONTnXm* for *PROC (DERm, ..., DERm) VOID* with n parameters.

The transcription of the rule for *integer* then becomes

```
PROC integer = (CONT2X1 q) VOID:
     BEGIN digit ( (DER1 digit value) VOID:
                   integer ( (DER1 integer value,
                                   integer length) VOID:
                             q (composed value,
                                composed length) ) );
           digit ( (DER1 digit value) VOID:
                   q (single value, single length) )
     END
```

with the refinements

```
composed value:
   (AFFIX shift) AFFIX: digit value (shift +
                                     plus +
                                     integer length (shift)) +
                        plus +
                        integer value (shift).
```

```
composed length:
   (AFFIX shift) AFFIX: "1" + plus + integer length (shift).

single value:
   (AFFIX shift) AFFIX: digit value (shift).

single length:
   (AFFIX shift) AFFIX: "1".
```

There is one last complication. The transcription of the last derived affix of the third member of the second alternative of *rational* gives rise to an infinite expansion:

```
AFFIX: whole value ("0") +
       plus +
       fraction value
          (minus +
           "(" + fraction length
                    (minus +
                     "(" + fraction length (...
```

Therefore, we must also procedure the inherited/right affixes:

```
AFFIX:
   BEGIN
      INH zero = AFFIX: "0",
          left shift =
             AFFIX: minus +
                    "(" + fraction length (left shift) + ")";
      whole value (zero) +
      plus +
      fraction value (left shift)
   END
```

This recursion is very near circular affix definition. Some results concerning transducers may be found in [KUEHLING].

5. Context Sensitivity

In this chapter we merely state the consequences of affix expressions at defining positions and indicate one possible solution.

5.1. Problem Statement

Recall that defining affix positions assign an affix value to the affix variable(s) occurring at that position. It may be that even a multiplicity of affix values is assigned, but then only the degree of ambiguity is increased: via the backup mechanism each of these values will be treated individually.

If a single affix variable occurs at the defining position, as is always the case with transducers, the value is assigned to that variable, which is thereby defined. If an affix expression occurs at a defining position, all terms of that expression are simultaneously defined. If *A* is the affix value (a string) and *a+b+c* is the affix expression, we have to solve the equation *a+b+c = A* If domain specifications are given for *a*, *b* and *c*, each combination of values in these domains such that the equation is satisfied constitute definitions for *a*, *b* and *c*. Again, via backup, each possible combination must be taken individually.

If one of the terms is a constant, it will be treated as a variable with a domain of one value. If for one of the terms no metarules are given, its domain is the set of all strings. It is clear that at least the multiplicity of the defining value must be finite. If we do not allow any of the terms on an applied position and at most one of the terms on a defining position to be infinite, this condition seems satisfied (it is sometimes useful to name part of a defining value without having to elaborately specify its structure).

If an affix variable occurs more than once as a term at a defining position, one of them defines its value, which is subsequently taken as single-valued domain for the other occurrences, according to the systematic-replacement rule.

In the above example, the domains of *a*, *b* and *c* are specified as context-free grammars (the metarules). Thus *x::a+b+c* is a grammar. Parsing *A* according to *x* associates parts of *A* with *a*, *b* and *c*. The part which is associated with *b* for instance, will be used as affix value in any applied occurrence of *b* and as grammar in any (other) defining occurrence.

Affix values, in our implementation, are not strings but functions which eventually produce the strings they represent. As a consequence, the grammars for *a*, *b* and *c* must parse the string represented by the function *A* and produce functions which represent the strings which are found as values for *a*, *b* and *c*.

5.2. An Experimental Solution

We have constructed a solution in Algol 68, which is very complex, but is useful as a vehicle for developing new ideas. We shall not give the details here.

The principle idea is that the function representing the affix value and the grammar which parses it, call each other, passing themselves as continuations. Each time both have reached a terminal symbol and these are found equal, they call their continuations like a ping-pong game. Furthermore the grammars gradually build the function which represents the

string they match, which will be used in further applied and defining positions.

Using a transducer grammar, we generated an Algol 68 program for a grammar describing

$$a^n\ b^n\ c^n$$

which proved that the transcription basically works.

For a flavour of this transcription we give some modes and the Algol 68 procedure which does the ping-pong:

```
MODE ELEMENT = CHAR,
     EVAL = PROC(ACTION)VOID,
     ACTION = PROC(ELEMENT,EVAL)VOID,
     AFFIX = PROC(ACTION,AFFIX)VOID;

PROC parse = (ELEMENT elem, EVAL object, cont)VOID:
     object ((ELEMENT element, EVAL remainder)VOID:
               IF elem = element
               THEN cont ((ELEMENT elem, EVAL cont)VOID:
                            parse (elem, remainder, cont))
               FI)
```

6. Further Developments

This chapter lists a few minor points concerning the implementation of, and further research on, Affix Grammars.

6.1. Optimizations

Parsing time may be significantly reduced by traditional methods like look ahead. Even if the grammar is ambiguous, the degree of local ambiguity may be reduced by look ahead. Again, the parsing algorithm allows easy implementation of look ahead.

Several well known transformations of the grammar into an equivalent grammar should be applied: removal of empty derivations, rule substitution, right recursion removal. The consequences of these transformations for the affix level have to be investigated. Rule substitution at the lexical level may reduce parsing time by factors and also seems to cause few problems at the affix level (the affix handling associated with identifiers and numbers is generally simple).

During parsing the operations on the rewrite- and reduce stacks are "pure" stack operations: no references are made to lower regions of these stacks. Therefore, these lower regions may be saved on background storage to reduce foreground storage requirements. Even the stack which contains the incarnations

of affix procedures might be, possibly using a paging strategy if the machine does not provide one.

The "very lexical" level, i.e., where nonterminals generate only single characters, like letters and digits, might be implemented in a special way using character-indexed class tables. Again, this requires only a trivial change in the parsing algorithm.

A more adventurous optimization is related to affix-directed parsing. Each affix procedure must have its parameters evaluable before it can be evaluated itself. It is worthwhile searching for an algorithm where these affixes are evaluated as soon as the parameters are available. This is especially important for defining affixes since they may cut off lengthy subparses when certain context conditions are locally not satisfied. On the other hand affix directed parsing is not always efficient. What is cheaper: avoiding parsing by early detection of unsatisfied context conditions or avoiding affix operations by earlier failing parses?

6.2. Extensions

There are several ways to extend the syntax and semantics of Affix Grammars to make them more comfortable as a programming language.

Like others [EAGLE] we should like to have a syntax which more resembles Two Level Van Wijngaarden Grammars, although the reasons for this are probably quite irrational.

As we have mentioned earlier, certain predicates should be predefined for Affix Grammars, like *unequal*. Also a small set of predicates to handle table-structured affixes (for symbol tables and the like) which may be assumed to be implemented efficiently would be of great practical value. Also the existence of certain lexical-level rules for letters, digits, numbers, identifiers, etc. would simplify the programming of grammars.

The (nearly always empty) rules which are needed for the "code generation" aspects could be avoided by introducing parameters at the metalevel. This introduces more syntactical tokens and structure, but our experience in writing several non-trivial grammars shows that about half of the grammar consists of artificial-looking rules which generate empty strings and always do so since no context conditions are involved.

Apart from strings, integers should be allowed as types for affix variables, perhaps even sets. This immediately opens the discussion concerning strong typing, dynamic typing, etc.

Allowing regular right parts seems very attractive. Many repetitional structures like sequences and lists are defined much more naturally using the closure operation than with recursion. It also makes the restriction on left-recursion more acceptable. Again, there are non-trivial consequences for the affix-level.

It must be possible to define multi-pass grammars explicitly. Basically this requires only the transfer of one or more affix values from one initial symbol to another.

It might be helpful to allow more freedom in the specification of the flow of affixes: specification at the right hand side of rules, specification at more than one occurrence of an affix position or no specification at all for some positions (those which are "obvious" for the grammar writer). [FRANZEN] describes an algorithm for automatic determination of affix flow.

6.3. More Elaborate Questions

Is it possible to associate the way affixes are implemented here with a bottom-up parsing algorithm?

How should the generated transducer/compiler handle errors in its input string? Is there a way to create an error-handling mechanism without any explicit specification by the grammar writer? The wording of the error messages is possibly derivable from the wording of the nonterminals and affix variables. We are very much aware of the fact that if ever the generator must be of practical use, it must at least provide a good quality error handling which requires minimal effort from the user. This is probably the hardest problem in the project.

What is the complexity of the generated transducers/ compilers? Are they correct?

References

[CDL] Koster, C.H.A.
Using the CDL Compiler Compiler.
In: Compiler Construction, An Advanced Course
(F.L. Bauer, J. Eickel, eds.),
Lecture Notes in Computer Science, 21,
Springer Verlag, Berlin-Heidelberg-New York, 1974.

[EAGLE] Franzen, H., Hoffman, B., Petersen, I.R.
Ein Parser-Generator fuer Erweiterte Affix-Grammatiken.
Diplomarbeit, Technische Universitaet Berlin,
Fachbereich Informatik, Bericht Nr. 76-24, Oktober 1976.

[FRANZEN] Franzen, H. and Hoffmann, B.
Automatic Determination of Data Flow in
Extended Affix Grammars.
Technische Universitaet Berlin, Fachbereich Informatik,
Bericht Nr. 79-20, September 1979.

[HOPCROFT] Hopcroft, J.E., Ullman, J.D.
Formal Languages and Their Relation to Automata.
Addison-Wesley, 1969.

[KNUTH] Knuth, D.E.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2, 127-145 (1968).

[KOSTER 1971] Koster, C.H.A.
Affix Grammars.
In: ALGOL 68 Implementation (J.E. Peck, ed.),
North-Holland Publishing Company, Amsterdam, 1971.

[KOSTER 1974] Koster, C.H.A.
A Technique for Parsing Ambiguous Grammars.
In: Lecture Notes in Computer Science, 26,
Springer Verlag, Berlin-Heidelberg-New York, 1974.

[KUEHLING] Kuehling, P.
Affix-Grammatiken zur Beschreibung von
Programmiersprachen.
Dissertation, Technische Universitaet Berlin,
Fachbereich Informatik, Februar 1978.

[WATT] Watt, D.A.
Analysis-oriented Two-level Grammars.
Ph.D. thesis, University of Glasgow, January 1974.

Nijmegen, January 8, 1980.