Bottom-up Tree Acceptors

C. Hemerik J.P. Katoen
Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

November 1988
Revised August 1989

Abstract

This paper deals with the formal derivation of an efficient tabulation algorithm for table-driven bottom-up tree acceptors. Bottom-up tree acceptors are based on a notion of match sets. First we derive a naive acceptance algorithm using dynamic computation of match sets. Tabulation of match sets leads to an efficient acceptance algorithm, but tables may be so large that they can not be generated due to lack of space. Introduction of a convenient equivalence relation on match sets reduces this effect and improves the tabulation algorithm.

1 Introduction

Nowadays, many parts of a compiler can be generated automatically. For instance, the automatic generation of lexical and syntactic analyzers using notations based on regular expressions and context-free grammars is commonly used (see e.g. [1]). However, much research is still going on in the field of universal *code-generator generators*, which take a description of a machine as input and deliver a (good) code generator for that machine.

Code generation forms an important subject in compiler writing. Requirements traditionally imposed on a code generator are severe: the generated code must be correct and must utilize the resources of the machine (such as registers) efficiently. A fundamental issue in code generation is instruction selection. This forms the subject of the remainder of this section. The non-uniformity of the instruction set and addressing modes of the target machine determine the difficulty of instruction selection. As an example, we illustrate instruction selection for an expression on a register machine with a very simple instruction set. First, in example 1.1, the addressing modes are presented.

Example 1.1

addressing mode	format	meaning
immediate	#c	c
register	R_i	R_i
indexed	$c(R_i)$	$M(c + R_i)$
indirect	$*R_i$	$M(R_i)$

□

Here, $M(a)$ denotes the contents of address a, c is a constant, and R_i is a register. Suppose our target machine supports the following instructions.

Example 1.2

	instruction	definition
(1)	*MOV* #c, R_i	$R_i := c$
(2)	*MOV* $*R_j$, R_i	$R_i := M(R_j)$
(3)	*MOV* $c(R_j)$, R_i	$R_i := M(c + R_j)$
(4)	*ADD* R_i, R_j	$R_i := R_i + R_j$

□

Now consider the expression $R_1 := c_1 + M(c_2 + R_2)$. Using the instructions given above we may derive an instruction sequence for this expression as follows. At each step the subexpression that is matched is underlined.

Example 1.3

expression	instruction
$R_1 := c_1 + M(\underline{c_2} + R_2)$	*MOV* #c_2, R_3
$R_1 := c_1 + M(\underline{R_3 + R_2})$	*ADD* R_2, R_3
$R_1 := c_1 + \underline{M(R_2)}$	*MOV* $*R_2$, R_2
$R_1 := \underline{c_1} + R_2$	*MOV* #c_1, R_1
$R_1 := \underline{R_1 + R_2}$	*ADD* R_1, R_2

□

Observe that the derivation above looks like the parsing of a string. By replacing the definition of an instruction, which is of the form $R_i := \ldots$, by a production rule of the form $R_i \rightarrow \ldots$, definitions (1)-(4) of example 1.2 may be considered as a *code generation grammar*. This suggests the use of traditional parsing techniques for code generation. For

instance, Graham and Glanville use LR-parsing for instruction selection (see [9]). The main problem with this approach is the resolution of the large number of parsing conflicts caused by the fact that code generation grammars are inherently highly ambiguous. For example, an alternative instruction sequence may be derived for the expression $R_1 := c_1 + M(c_2 + R_2)$ as follows :

Example 1.4

expression	instruction
$R_1 := c_1 + \underline{M(c_2 + R_2)}$	*MOV* $c_2(R_2)$, R_2
$R_1 := \underline{c_1} + R_2$	*MOV* #c_1, R_1
$R_1 := \underline{R_1 + R_2}$	*ADD* R_1, R_2

□

A way to overcome this problem could be to use a more general parsing method like Earley's algorithm, as suggested in [5], but the resulting space and time complexity is unacceptable for practical use in code generators.

Neither of these methods takes into account a special property of code generation grammars, viz. that every operator symbol has a fixed rank. Using this property leads to the idea of considering the tree representation of an expression, rather than its string representation. In this way, the code generation (string) grammar becomes a so-called *tree grammar*. For the instructions of example 1.2 the production rules, represented by trees, become :

Example 1.5

instruction	tree representation
MOV #c, R_i	$R_i \rightarrow c$
MOV *R_j, R_i	$R_i \rightarrow M(R_j)$
MOV c(R_j), R_i	$R_i \rightarrow M(+(c, R_j))$
ADD R_i, R_j	$R_i \rightarrow +(R_i, R_j)$

□

Several code generation algorithms based on tree grammars have been described [1, 6, 12], but a theoretical framework is unfortunately missing. This is the more remarkable as a well-developed theory of tree grammars and tree automata has existed for some twenty years [3, 7, 11, 14]. A mathematical rigorous presentation in terms of universal algebra is presented in [8], which does not pay any attention to applications, however. A systematic treatment of some parts of the theory, aimed at code generation applications, is given in

[15]; a survey paper is in preparation [10].

In this paper we consider a particular class of tree acceptors, called *deterministic bottom-up tree acceptors*, which have a time complexity proportional to the size of the tree to be analyzed. Our main aim is to present algorithms for the efficient generation of compressed parse tables, and to show how the rather complex programs that construct these tables can be systematically derived. Of course, tree acceptors alone are not sufficient to solve the instruction selection problem, but they can easily be extended to bottom-up tree parsers and be combined with a dynamic programming technique. The resulting parsers yield all optimal derivations with respect to some externally specified cost per production rule. For details we refer to [15].

The organization of this paper is as follows. In sections 2 and 3 we present a simplified treatment of the theory of tree grammars and deterministic bottom-up tree acceptors. Section 4 shows how the transition functions of the acceptor can be tabulated, which leads to a linear time acceptance algorithm. In practical applications the size of the resulting tables may be prohibitive, however. Therefore in section 5, using ideas of Chase [4], an improved algorithm is described which generates compressed transition tables. Finally, section 6 contains some concluding remarks.

2 Tree grammars

In this section we define the basic concepts of the theory of tree grammars. Readers familiar with context-free (string-) grammars will notice that tree grammars are a generalization of right-linear (string-) grammars.

Definition 2.1 { ranked alphabet }

A ranked alphabet is a pair (V, r) such that V is a finite set and $r \in V \longrightarrow \mathcal{N}$

□

Note: throughout this paper $\mathcal{N}$ denotes the set of natural numbers.
Elements of V are called *symbols* and $r(a)$ is called the *rank* of symbol a. In the following the set V_n denotes the set of symbols with rank n, that is, $V_n = \{v \in V \mid r(v) = n\}$.

Definition 2.2 { $Tree(V, r)$, trees over a ranked alphabet }

The set $Tree(V, r)$ of trees over a ranked alphabet (V, r) is the smallest set X such that:

- $\mathrm{V}_0 \subseteq X$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in X : a(t_1, \ldots, t_n) \in X$

□

Definition 2.3 { tree grammar }

A tree grammar G is a 5-tuple (N, V, r, P, S) such that:

- $(N \cup V, r)$ is a ranked alphabet such that $\forall A \in N : r(A) = 0$
- $N \cap V = \emptyset$
- P is a finite subset of $N \times Tree(N \cup V, r)$
- $S \in N$

□

Elements of N, V, and P are called *nonterminals*, *terminals*, and *production rules*, respectively. S is called the *start symbol* of G.

Notational remark: upper-case letters are used to denote nonterminals, and lower-case letters stand for terminals. An element $(A, t) \in P$ is usually written as $A \to t$; A is sometimes called the *left-hand side*, and t the *right-hand side* of the production rule.

Definition 2.4 { $\Longrightarrow$, derivation-step (informal) }

Let (N, V, r, P, S) be a tree grammar. $\forall t_1, t_2 \in \text{Tree}(N \cup V, r)$:

$t_1 \Longrightarrow t_2$ if there exists $(A \to \alpha) \in P$, such that t_2 can be obtained from t_1 by substituting α for one occurrence of A in t_1

□

$\stackrel{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$. We say that t is *derivable* from A if $A \stackrel{*}{\Longrightarrow} t$. The set of trees, containing only terminals, derivable from the start symbol constitute the *language* that is generated by the corresponding tree grammar.

Definition 2.5 { $\mathcal{L}$, language generated by a tree grammar }

Let $G = (N, V, r, P, S)$ be a tree grammar.
$\forall A \in N$, function $\mathcal{L} \in N \longrightarrow \mathcal{P}(Tree(V, r))$ is defined by:

$$\mathcal{L}(A) = \{t \in Tree(V, r) \mid A \stackrel{*}{\Longrightarrow} t\}$$

$\mathcal{L}(S)$ is the language generated by G

□

The tree grammar defined in the following example shall be used as a running example throughout this paper.

Example 2.6

Let $G = (N, V, r, P, A)$ be a tree grammar, where

$N = \{A, B\}$;

$V = \{a, b, c, d\}$;

$r(a) = 2$; $r(b) = 1$; $r(c) = 0$; $r(d) = 0$;

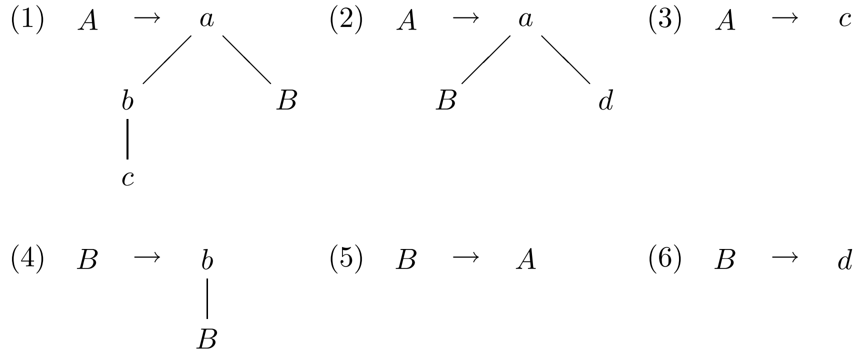

Figure 1: Production rules represented as trees

$$
\begin{aligned}
P = \{\ & (1)\ \ A \to a(b(c), B), \\
& (2)\ \ A \to a(B, d), \\
& (3)\ \ A \to c, \\
& (4)\ \ B \to b(B), \\
& (5)\ \ B \to A, \\
& (6)\ \ B \to d\ \}
\end{aligned}
$$

An alternative presentation of the elements of P is given in figure 1.
Some examples of derivations are :

$A \overset{(1)}{\Longrightarrow} a(b(c), B) \overset{(6)}{\Longrightarrow} a(b(c), d)$.

$A \overset{(2)}{\Longrightarrow} a(B, d) \overset{(4)}{\Longrightarrow} a(b(B), d) \overset{(5)}{\Longrightarrow} a(b(A), d) \overset{(3)}{\Longrightarrow} a(b(c), d)$.

Some elements of $\mathcal{L}(A)$ are : $c,\ a(b(c), d),\ a(a(b(c), b(d)), d)$.

□

3 Tree acceptors

A tree acceptor is a tree automaton which, given a tree grammar $G = (N, V, r, P, S)$ and a tree $t \in Tree(V, r)$, establishes whether $t \in \mathcal{L}(S)$. In this section we consider a particular kind of tree acceptors, viz. deterministic bottom-up tree acceptors, although we shall not stress the automata-theoretic concepts. The basic idea underlying this kind of acceptor is to extract from the grammar G a set PS of *patterns*, i.e. subtrees of right-hand sides of production rules, and to compute for a tree t the match set $MS_1(t)$ of patterns from which it may be derived. Tree t is accepted if and only if $S \in MS_1(t)$. As the match set of a compound tree can be simply computed from the match sets of its direct subtrees, an acceptor can be obtained which operates in time proportional to the size of the tree.

The following definitions are all relative to a given tree grammar $G = (N, V, r, P, S)$. We assume that G has no useless terminals and nonterminals, i.e. every (non)terminal, apart form S, occurs in some tree derivable from S and for all $A \in N : \mathcal{L}(A) \neq \emptyset$.

Definition 3.1 { $\underline{sub}$, subtree relation }

$\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(N \cup V, r) : t_j \ \underline{sub} \ a(t_1, \ldots, t_n)$

□

$\underline{sub}^*$ is the reflexive and transitive closure of $\underline{sub}$.

Definition 3.2 { PS, pattern set }

$PS = \{t \mid \exists A, t' : (A \rightarrow t') \in P : t \ \underline{sub}^* \ t'\}$

□

Notice that $N \backslash \{S\} \subseteq PS$ holds, since every element in $N \backslash \{S\}$ occurs in some tree derivable from S. The closure of a pattern s is a set of patterns. This set contains s and the nonterminals from which s is derivable. Similarly, the closure of a set of patterns is defined as follows.

Definition 3.3 { closure }

$\forall s \in \mathcal{P}(PS)$ function $closure \in \mathcal{P}(PS) \longrightarrow \mathcal{P}(PS)$ is defined by:

$$closure(s) = s \cup \{A \in N \mid \exists p : p \in s : A \stackrel{*}{\Longrightarrow} p\}$$

□

Obviously, for all $s \in \mathcal{P}(PS) : closure(s) \subseteq PS$. There are various ways to handle the acceptance problem. One possible way, commonly known as the *bottom-up* method (or bottom-up pattern matching), is to derive the start symbol S starting with a given tree t. The bottom-up method relies on the notion of *match sets*, sets of subpatterns that match at a particular tree node. These sets are defined recursively as:

Definition 3.4 { MS_1, match set }

The function $MS_1 \in Tree(V, r) \longrightarrow \mathcal{P}(PS)$ is defined by:

- $\forall a \in V_0 : MS_1(a) = closure(\{a\})$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(V, r)$:

$$MS_1(a(t_1, \ldots, t_n)) =$$
$$closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall j : 1 \leq j \leq n : p_j \in MS_1(t_j)\})$$

□

The relevance of match sets is expressed by the following lemma.

Lemma 3.5 $\forall t \in Tree(V, r) : MS_1(t) = \{t' \in PS \mid t' \stackrel{*}{\Longrightarrow} t\}$

Proof : by structural induction over $Tree(V, r)$.

1. base step: let $a \in V_0$. We derive:

$$
\begin{array}{ll}
 & MS_1(a) \\
= & \quad \{\text{definition 3.4, } a \in V_0\} \\
 & closure(\{a\}) \\
= & \quad \{\text{definition 3.3}\} \\
 & \{a\} \cup \{A \in N \mid A \stackrel{*}{\Longrightarrow} a\} \\
= & \quad \{N \subseteq PS,\ a \in PS \wedge a \stackrel{*}{\Longrightarrow} a\} \\
 & \{t' \in PS \mid t' \stackrel{*}{\Longrightarrow} a\}
\end{array}
$$

2. induction step: for $1 \leq n$, let $a \in V_n$. Then we have:

$$
\begin{array}{ll}
 & MS_1(a(t_1, \ldots, t_n)) \\
= & \quad \{\text{definition 3.4, } a \in V_n, 1 \leq n\} \\
 & closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall j : 1 \leq j \leq n : p_j \in MS_1(t_j)\}) \\
= & \quad \{\text{induction hypothesis, set calculus}\} \\
 & closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall j : 1 \leq j \leq n : p_j \stackrel{*}{\Longrightarrow} t_j\}) \\
= & \quad \{\text{definition 2.4}\} \\
 & closure(\{a(p_1, \ldots, p_n) \in PS \mid a(p_1, \ldots, p_n) \stackrel{*}{\Longrightarrow} a(t_1, \ldots, t_n)\}) \\
= & \quad \{\text{definition 3.3, } \forall s \in \mathcal{P}(PS) : closure(s) \subseteq PS\} \\
 & \{t' \in PS \mid t' \stackrel{*}{\Longrightarrow} a(t_1, \ldots, t_n)\}
\end{array}
$$

□

Lemma 3.6 $\forall t \in Tree(V, r) : S \in MS_1(t) \ \equiv \ t \ \in \mathcal{L}(S)$

Proof : straightforward by using definition 2.5 and lemma 3.5. □

So, by computing $MS_1(t)$ for a given tree t, it is easy to decide whether t belongs to the language generated by G or not.

Example 3.7

Consider the grammar of example 2.6. Its pattern set is :

$$PS = \{a(b(c), B),\ b(c),\ c,\ B,\ a(B, d),\ d,\ b(B),\ A\}$$

For $t = a(b(c), b(a(d, d)))$ bottom-up computation of $MS_1(t)$ is depicted in figure 2, where each node of t is annotated with the match set of the tree rooted at that node. In this figure we see that $A \in MS_1(t)$, hence (see lemma 3.6), $t \in \mathcal{L}(A)$.

□

Notational remark: elements of a match set added by a closure operation are separated from other elements by a semicolon.

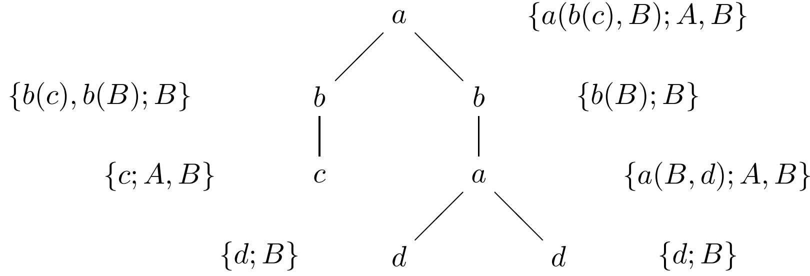

Figure 2: An example of dynamic computation of match sets

4 Tabulation of match sets

A program computing MS_1 can easily be implemented, but is very inefficient. At each determination of the acceptance of a tree, match sets (and closures) must be recalculated. Fortunately, since $\mathcal{P}(PS)$ is a finite set (due to the fact that PS is finite) the number of match sets is finite. This allows *tabulation*, a general applied technique for a more efficient implementation of recursively defined functions (see [2]), of match sets. Observe that $MS_1(a(t_1, \ldots, t_n))$ is of the form $f_a(MS_1(t_1), \ldots, MS_1(t_n))$, where f_a is defined as follows.

Definition 4.1 { f_a, transition function for symbol a }

- $\forall a \in V_0 : f_a \in \mathcal{P}(PS)$, where $f_a = closure(\{a\})$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall s_1, \ldots, s_n \in \mathcal{P}(PS)$:
  function $f_a \in \mathcal{P}(PS)^n \longrightarrow \mathcal{P}(PS)$ is defined by:
  $$f_a(s_1, \ldots, s_n) = closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall j : 1 \leq j \leq n : p_j \in s_j\})$$

□

Obviously all f_a's have a finite domain (since the number of match sets is finite), so we may tabulate f_a. This means that we will have an n-dimensional table for a symbol of rank n. We do not have to tabulate f_a for the entire powerset $\mathcal{P}(PS)$, but only for its reachable part, i.e. the smallest set $Z \subseteq \mathcal{P}(PS)$ closed under all f_a's. To compute the reachable part Z and the tabulation of f_a, $\tilde{T}_a \in Z^n \longrightarrow Z$, where $n = r(a)$, the standard reachability algorithm (see e.g. [13]) is used. This leads to the following algorithm, which is called A_1.

```
|[ var x, y : P(PS)
  | Z, W, G := ∅, P(PS), ∅
  ; for all a ∈ V_0
   do y := f_a; W, G := W \ {y}, G ∪ {y}; T̃_a := y od
  ; do G ≠ ∅ ⟶ x :∈ G
               ; for all a ∈ V \ V_0
                do |[ var n : N
                     | n := r(a)
                     ; for all (s_1, ..., s_n) ∈ (Z ∪ {x})^n \ Z^n
                      do y := f_a(s_1, ..., s_n)
                         ; if y ∈ W ⟶ W, G := W \ {y}, G ∪ {y}
                           ▯ y ∉ W ⟶ skip
                           fi
                         ; T̃_a(s_1, ..., s_n) := y
                      od
                     ]|
                od
               ; G, Z := G \ {x}, Z ∪ {x}
   od
]|
```

To allow transition tables to be indexed with numbers rather than match sets, we introduce an enumeration $E \in \mathcal{N} \longrightarrow_p Z$ of match sets ($\longrightarrow_p$ denotes a partial function). E is an injection. We define transition tables in terms of the enumeration as follows.

Definition 4.2 { T_a, transition table for symbol a }

- $\forall a \in V_0 : T_a \in dom(E)$, where $E(T_a) = f_a$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall i_1, \ldots, i_n \in dom(E)$:
  function $T_a \in dom(E)^n \longrightarrow dom(E)$ is defined by:

$$E(T_a(i_1, \ldots, i_n)) = f_a(E(i_1), \ldots, E(i_n))$$

□

The corresponding definition of match sets is now.

Definition 4.3 { MS_2, match set }

The function $MS_2 \in Tree(V, r) \longrightarrow dom(E)$ is defined by:

- $\forall a \in V_0 : MS_2(a) = T_a$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(V, r)$:

$$MS_2(a(t_1, \ldots, t_n)) = T_a(MS_2(t_1), \ldots, MS_2(t_n))$$

□

The correspondence between MS_1 and MS_2 is stated in the following lemma.

Lemma 4.4 $\forall t \in Tree(V,r) : MS_1(t) \;\equiv\; E(MS_2(t))$

Proof : by structural induction over $Tree(V,r)$, using definitions 3.4 and 4.1-4.3. □

Match sets containing the start symbol S have a special meaning (see lemma 3.6) and are called *accepting states*.

Definition 4.5 { F, set of accepting states }

$F = \{n \in dom(E) \mid S \in E(n)\}$

□

Lemma 4.6 $\forall t \in Tree(V,r) : t \in \mathcal{L}(S) \;\equiv\; MS_2(t) \in F$

Proof : use definition 4.5 and lemmata 4.4 and 3.6. □

As can be observed from definition 3.3, taking the closure of a set of patterns consists of deriving left-hand sides of production rules. This calculation can be simplified by tabulating the closure of nonterminals in a table Nclosure. Formally :

Definition 4.7 { Nclosure }

The function $Nclosure \in N \longrightarrow \mathcal{P}(N)$ is defined by :

$\forall A \in N : Nclosure(A) = \{B \in N \mid B \stackrel{*}{\Longrightarrow} A\}$

□

Computing the Nclosure of a nonterminal is equivalent to determining the reflexive and transitive closure of $P \cap (N \times N)$. We use Warshall's algorithm to calculate the transitive closure. The relationship between closure and Nclosure is stated in lemma 4.8.

Lemma 4.8

$\forall s \in \mathcal{P}(PS) : closure(s) = s \cup (\bigcup A, \alpha : (A \rightarrow \alpha) \in P \wedge \alpha \in s : Nclosure(A))$

Proof : use definitions 3.3, 2.4, and 4.7.

□

The following algorithm A_2 is a more efficient version of algorithm A_1 in which the match sets are enumerated and Nclosure is used. In algorithm A_2, the sets Z, G, and W are characterized by $\{E(i) \mid 0 \leq i < p\}, \{E(i) \mid p \leq i < q\}$, and $\mathcal{P}(PS) \setminus (Z \cup G)$, respectively.

|[**con** $G = (N, V, r, P, S)$: *tree grammar*
 ; **var** $E : \mathcal{N} \longrightarrow_p \mathcal{P}(PS)$
 ; $F : \mathcal{P}(\mathcal{N})$
 ; $p, q : \mathcal{N}$
 ; $T_a : \mathcal{N}$ *for all* $a \in V_0$
 ; $T_a : \mathcal{N}^n \longrightarrow \mathcal{N}$ *for all* $n : 1 \leq n : a \in V_n$
 ; $Nclosure : N \longrightarrow \mathcal{P}(N)$

```
; proc compute_Nclosure =
 |[ | (* first, transitive closure by means of Warshall's algorithm *)
    for all A ∈ N do Nclosure(A) := {B ∈ N | (B → A) ∈ P} od
   ; for all B ∈ N
    do  for all A ∈ N
        do  if B ∈ Nclosure(A) ⟶ Nclosure(A) :=
                                  Nclosure(A) ∪ Nclosure(B)
             ▯ B ∉ Nclosure(A) ⟶ skip
            fi
        od
    od (* second, reflexive closure *)
   ; for all A ∈ N do Nclosure(A) := Nclosure(A) ∪ {A} od
  ]|

; func closure =
 (↓ s : 𝒫(PS) | 𝒫(PS))
 |[ var r : 𝒫(PS)
  | r := s
  ; for all (A → α) ∈ P
   do if α ∈ s ⟶ r := r ∪ Nclosure(A)
       ▯ α ∉ s ⟶ skip
      fi
   od
  | r
 ]|
 )

(* main program *)
| compute_Nclosure()
; q := 0 ; p := 0
; for all a ∈ V_0
 do E(q) := closure({a})  (* new match set *)
    ; T_a, q := q, q + 1
 od
; do  p ≠ q ⟶ for all a ∈ V \ V_0
               do  |[ var n : 𝒩
                    | n := r(a)
                    ; for all (p_1, ..., p_n) ∈ {0, ..., p}^n \ {0, ..., p − 1}^n
                     do  E(q) := closure({a(t_1, ..., t_n) ∈ PS | t_i ∈ E(p_i)})
                         ; |[ var k : 𝒩
                            | k := 0
                            ; do E(k) ≠ E(q) ⟶ k := k + 1 od
                            ; if k = q ⟶ q := q + 1  (* new match set *)
```

```
                              [] k ≠ q ⟶ skip
                             fi
                            ; T_a(p_1, ..., p_n) := k
                           ]|
                       od
                      ]|
                  od
                  ; p := p + 1
   od
  ; p := 0 ; F := ∅  (* determine accepting states *)
  ; do p ≠ q ⟶ if S ∈ E(p) ⟶ F := F ∪ {p}
                   [] S ∉ E(p) ⟶ skip
                  fi
                  ; p := p + 1
   od
]|
```

Given the transition tables, the acceptance problem is easily solved by simple table look-ups. The table-driven acceptor is described by the following algorithm. The time complexity of this algorithm is proportional to the size of the input tree.

```
|[ con F : P(N)
 ;      T_a : N                                   for all a ∈ V_0
 ;      T_a : N^n ⟶ N                   for all n : 1 ≤ n : a ∈ V_n
 ;      t : Tree(V, r)

 ; var accepted : bool

 ;func ms_2 =
  (↓ t : Tree(V, r) | N
  | if t :: a ⟶ T_a
   [] t :: a(t_1, ..., t_n) ⟶ T_a(ms_2(t_1), ..., ms_2(t_n))
   fi
  )

 | accepted := (ms_2(t) ∈ F)
]|
```

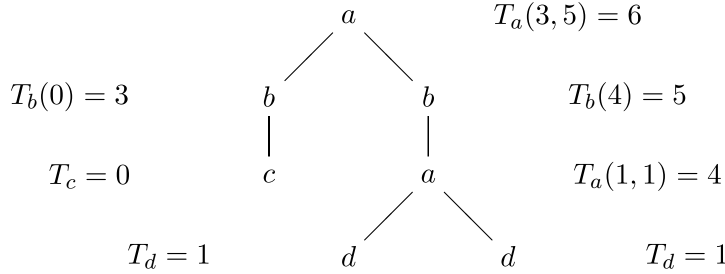

Figure 3: An example of table-driven acceptance

Example 4.9

Consider the grammar of example 2.6. The tables generated by algorithm A_2 are :

E	match set
0	$\{c; A, B\}$
1	$\{d; B\}$
2	$\emptyset$
3	$\{b(c), b(B); B\}$
4	$\{a(B,d); A, B\}$
5	$\{b(B); B\}$
6	$\{a(b(c), B); A, B\}$
7	$\{a(B,d), a(b(c), B); A, B\}$

T_a	0	1	2	3	4	5	6	7
0	2	4	2	2	2	2	2	2
1	2	4	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2
3	6	7	2	6	6	6	6	6
4	2	4	2	2	2	2	2	2
5	2	4	2	2	2	2	2	2
6	2	4	2	2	2	2	2	2
7	2	4	2	2	2	2	2	2

T_b	
0	3
1	5
2	2
3	5
4	5
5	5
6	5
7	5

$T_c = 0,\ T_d = 1;\ F = \{0, 4, 6, 7\}$

Given the input tree, for instance $t = a(b(c), b(a(d,d)))$, table-driven bottom-up acceptance proceeds as demonstrated in figure 3. In this figure each node of t is annotated with a number corresponding to the match set with which it was annotated in figure 2. Since $6 \in F,\ t \in \mathcal{L}(A)$.

□

5 Optimized tabulation

In practice, code generation tree grammars (see introduction) are rather extensive. This means that transition tables may be very large. These tables could be compressed after they are computed, but uncompressed tables may be so large that they cannot be generated, even if the compressed tables are of manageable size. However, an optimization is possible that enables compressed transition tables to be directly generated. This optimization is based on an equivalence relation on match sets. The equivalence relation is based on the observation that some patterns only occur as the j-th subtree of a tree labelled with a symbol of rank n ($n \geq j$). The main advantage is that with generation of match sets one can iterate over the equivalence classes instead of the match sets. This is quite lucrative, provided that the mapping of match sets onto equivalence classes is not (nearly) a bijection (in which case no improvement is made). David Chase first considered this optimization, but he only gave an informal treatment. Here, we derive an improved algorithm for the generation of match sets based on the ideas presented in [4].

The j-th childset of a symbol, say a, $1 \leq j \leq r(a)$, is the set of patterns that appear as j-th subtree of a tree in PS labelled with a. Formally :

Definition 5.1 { $CS_{a,j}$, j-th childset of symbol a }

$$\forall n, j : 1 \leq j \leq n : \forall a \in V_n : CS_{a,j} = \{t \mid \exists t_1, \ldots, t_n : a(t_1, \ldots, t_n) \in PS : t_j = t\}$$

□

Example 5.2

The childsets of the symbols of the grammar of example 2.6 are:

$CS_{a,1} = \{b(c),\ B\}$, $CS_{a,2} = \{B,\ d\}$, and $CS_{b,1} = \{c,\ B\}$.

□

Lemma 5.3 $\forall n, j : 1 \leq j \leq n : \forall a \in V_n : CS_{a,j} \subseteq PS$

Proof : PS is closed under taking subtrees. □

Using childsets we may refine the definition of match set as follows.

Definition 5.4 { MS_1, match set }

The function $MS_1 \in Tree(V, r) \longrightarrow \mathcal{P}(PS)$ is defined by:

- $\forall a \in V_0 : MS_1(a) = closure(\{a\})$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(V, r)$:

$$MS_1(a(t_1, \ldots, t_n)) =$$
$$closure(\{a(p_1, \ldots, p_n) \in PS \mid \forall j : 1 \leq j \leq n : p_j \in MS_1(t_j) \cap CS_{a,j}\})$$

□

The only difference between definition 5.4 and definition 3.4 is the intersection with $CS_{a,j}$. This does not affect the value of $MS_1(a(t_1, \ldots, t_n))$, because the only patterns missing from $MS_1(t_j) \cap CS_{a,j}$ are those patterns that do not appear as the j-th subtree of a tree in PS labelled with a.

For the same reasons as mentioned in section 4 a program computing MS_1 can be easily implemented, but is rather inefficient. Again, tabulation is possible. Observe that $MS_1(a(t_1, \ldots, t_n))$ is now of the form $f_a(g_{a,1}(MS_1(t_1)), \ldots, g_{a,n}(MS_1(t_n))\ )$ where f_a is defined as in definition 4.1 and $g_{a,j}$ is defined as follows.

Definition 5.5 { $g_{a,j}$, map function for j-th child of symbol a }

$\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall s \in \mathcal{P}(PS) :$

function $g_{a,j} \in \mathcal{P}(PS) \longrightarrow \mathcal{P}(PS)$ is defined by: $g_{a,j}(s) = s \cap CS_{a,j}$

□

Again for practical reasons we use an enumeration $E \in \mathcal{N} \longrightarrow_p \mathcal{P}(PS)$ of match sets. We define the transition tables in terms of the map function and the enumeration E as follows.

Definition 5.6 { T_a, transition table for symbol a }

- $\forall a \in V_0 : T_a \in dom(E)$, where $E(T_a) = f_a$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall i_1, \ldots, i_n \in dom(E) :$
  function $T_a \in dom(E)^n \longrightarrow dom(E)$ is defined by:

$$E(T_a(i_1, \ldots, i_n)) = f_a(g_{a,1}(E(i_1)), \ldots, g_{a,n}(E(i_n))\ )$$

□

These transition tables are similar to those defined in definition 4.2.
An important observation is that the intersection of a match set with some childset $CS_{a,j}$, for some symbol $a \in V_n$, where $1 \leq j \leq n$, induces an equivalence relation over match sets.

Definition 5.7 { equivalence relation $\approx_{a,j}$ }

$\forall s, s' \in \mathcal{P}(PS) : \forall n, j : 1 \leq j \leq n : \forall a \in V_n : s \approx_{a,j} s' \equiv (s \cap CS_{a,j} = s' \cap CS_{a,j})$

□

Definition 5.8 { equivalence class $\varepsilon_{\approx a,j}$ }

$\forall s \in \mathcal{P}(PS) : \forall n, j : 1 \leq j \leq n : \forall a \in V_n : \varepsilon_{\approx a,j}(s) = \{s' \in \mathcal{P}(PS) \mid s \approx_{a,j} s'\}$

□

In other words: the equivalence class $\varepsilon_{\approx a,j}(s)$ is the set of all match sets that are equivalent (under $\approx_{a,j}$) to s. An equivalence class $\varepsilon_{\approx a,j}(s)$ is represented by $s \cap CS_{a,j}$, which is called the *representer-set* of $\varepsilon_{\approx a,j}(s)$. Since the number of equivalence classes is finite we can tabulate the representer-sets. We introduce an enumeration of representer-sets

$R_{a,j} \in \mathcal{N} \longrightarrow_p \mathcal{P}(PS)$ for all $n, j : 1 \leq j \leq n$, and $a \in V_n$. The mapping of (the enumeration of) match sets onto (the enumeration of) representer-sets is performed by an index map table $\mu_{a,j}$.

Definition 5.9 { $\mu_{a,j}$, index map table for j-th child of symbol a }

$\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall i \in dom(E) :$

function $\mu_{a,j} \in dom(E) \longrightarrow dom(R_{a,j})$ is defined by:

$$R_{a,j}(\mu_{a,j}(i)) = E(i) \cap CS_{a,j}$$

□

Notice that $\mu_{a,j}$ is, in fact, the tabulation of $g_{a,j}$, which was defined in definition 5.5. This means that transitions only need to be tabulated for representer-sets (instead of match sets). This is reflected in the following definition.

Definition 5.10 { T'_a, transition table for symbol a }

- $\forall a \in V_0 : T'_a \in dom(E)$, where $E(T'_a) = f_a$
- $\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall i_1, \ldots, i_n \in dom(E) :$
  - $\mu_{a,j} \in dom(E) \longrightarrow dom(R_{a,j})$
  - $T'_a \in (dom(R_{a,1}) \times \ldots \times dom(R_{a,n})) \longrightarrow dom(E)$
  - $E(T'_a(\mu_{a,1}(i_1), \ldots, \mu_{a,n}(i_n))) = f_a(g_{a,1}(E(i_1)), \ldots, g_{a,n}(E(i_n)))$

□

The corresponding definition of match set becomes :

Definition 5.11 { MS_2, match set }

The function $MS_2 \in Tree(V, r) \longrightarrow dom(E)$ is defined by :

- $\forall a \in V_0 : MS_2(a) = T'_a$
- $\forall n : 1 \leq n : \forall a \in V_n : \forall t_1, \ldots, t_n \in Tree(V, r) :$

$$MS_2(a(t_1, \ldots, t_n)) = T'_a(\mu_{a,1}(MS_2(t_1)), \ldots, \mu_{a,n}(MS_2(t_n)))$$

□

Notice that MS_2 is still related to MS_1 by lemma 4.4. Using the definitions above and the invariants given below we may derive the following tabulation algorithm (named A_3). First, we give the invariants of the program.

- $0 \leq p \leq q$
- $(\forall n : 1 \leq n : \forall a \in V_n : new_a = (\exists j : 1 \leq j \leq n : p_{a,j} < q_{a,j}))$
- $(\forall n, j : 1 \leq j \leq n : \forall a \in V_n : 0 \leq p_{a,j} \leq q_{a,j})$
- $(\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall i : 0 \leq i < q :$
  $0 \leq \mu_{a,j}(i) < q_{a,j} \;\wedge\; R_{a,j}(\mu_{a,j}(i)) = E(i) \cap CS_{a,j})$

- $(\forall n, j : 1 \leq j \leq n : \forall a \in V_n : \forall i_1, \ldots, i_n : 0 \leq i_j < p :$
  $0 \leq \mu_{a,j}(i_j) < p_{a,j}$
  $\wedge\ 0 \leq T'_a(\mu_{a,1}(i_1), \ldots, \mu_{a,n}(i_n)) < q$
  $\wedge\ E(T'_a(\mu_{a,1}(i_1), \ldots, \mu_{a,n}(i_n))\ ) = f_a(g_{a,1}(E(i_1)), \ldots, g_{a,n}(E(i_n))\ )\ )$

$|[$ **con** $G = (N, V, r, P, S) : tree\ grammar$

; **var** $E : \mathcal{N} \longrightarrow_p \mathcal{P}(PS)$
; $F : \mathcal{P}(\mathcal{N})$
; $T'_a : \mathcal{N},$ — $for\ all\ a \in V_0$
; $T'_a : \mathcal{N}^n \longrightarrow \mathcal{N}$ — $for\ all\ n : 1 \leq n : a \in V_n$
; $\mu_{a,j} : \mathcal{N} \longrightarrow \mathcal{N}$ — $for\ all\ n, j : 1 \leq j \leq n : a \in V_n$
; $R_{a,j} : \mathcal{N} \longrightarrow_p \mathcal{P}(PS)$ — $for\ all\ n, j : 1 \leq j \leq n : a \in V_n$
; $p_{a,j}, q_{a,j} : \mathcal{N}$ — $for\ all\ n, j : 1 \leq j \leq n : a \in V_n$
; $CS_{a,j} : \mathcal{P}(PS)$ — $for\ all\ n, j : 1 \leq j \leq n : a \in V_n$
; $new_a : bool$ — $for\ all\ n : 1 \leq n : a \in V_n$
; $p, q : \mathcal{N}$
; $Nclosure : N \longrightarrow \mathcal{P}(N)$

; **proc** *compute_Nclosure* (* *see algorithm* A_2 *)

; **func** *closure* (* *see algorithm* A_2 *)

; **proc** *compute_childsets* =
 $|[$ **var** $j, n : \mathcal{N}$
  | **for all** $a \in V \setminus V_0$
   do $j, n := 1, r(a)$
     ; **do** $j \neq n + 1 \longrightarrow CS_{a,j} := \emptyset\ ; j := j + 1$ **od**
   od
  ; **for all** $a(t_1, \ldots, t_n) \in PS$
   do $j := 1$
     ; **do** $j \neq n + 1 \longrightarrow CS_{a,j} := CS_{a,j} \cup \{t_j\}$ **od**
   od
 $]|$

; **proc** *compute_reprsets* =
 $(\downarrow p : \mathcal{N}$
 $||[$ (* *compute equivalence classes of matchset* $E(p)$ *for all* $a \in V \setminus V_0$ *)
   for all $a \in V \setminus V_0$
   do $|[$ **var** $j, n : \mathcal{N}$
     | $j, n := 1, r(a)$
     ; **do** $j \neq n + 1 \longrightarrow R_{a,j}(q_{a,j}) := E(p) \cap CS_{a,j}$
       ; $|[$ **var** $k : \mathcal{N}$
        | $k := 0$

```
                                        ; do R_{a,j}(k) ≠ R_{a,j}(q_{a,j}) ⟶ k := k + 1 od
                                        ; if k ≠ q_{a,j} ⟶ skip
                                          ▯ k = q_{a,j} ⟶ (* new representer-set *)
                                                       q_{a,j}, new_a := q_{a,j} + 1, true
                                          fi
                                        ; μ_{a,j}(p) := k
                                      ]|
                                     ; j := j + 1
            od
           ]|
      od
 ]|
)

(* main program *)
| compute_Nclosure()
; compute_childsets()
; for all a ∈ V \ V_0
 do  |[ var j, n : 𝒩
        | j, n := 1, r(a)
        ; do j ≠ n + 1 ⟶ q_{a,j}, p_{a,j}, j := 0, 0, j + 1 od
       ]|
      ; new_a := false
 od
; q := 0 ; p := 0
; for all a ∈ V_0
 do E(q) := closure({a})
     ; T'_a, q := q, q + 1
 od
; do p ≠ q ⟶ compute_reprsets(p) ; p := p + 1 od
; do (∃a : a ∈ V \ V_0 : new_a)
   ⟶for all (a : a ∈ V \ V_0 : new_a)
       do  |[ var j, n : 𝒩
              | n := r(a)
              ; for all (p_1, ..., p_n) ∈ {0 ... q_{a,1} − 1} × ... × {0 ... q_{a,n} − 1}\
                                            {0 ... p_{a,1} − 1} × ... × {0 ... p_{a,n} − 1}
               do E(q) := closure({a(t_1, ..., t_n) ∈ PS |: t_i ∈ R_{a,i}(p_i)})
                   |[ var k : 𝒩
                    | k := 0
                    ; do E(k) ≠ E(q) ⟶ k := k + 1 od
                    ; if k = q ⟶ q := q + 1   (* new match set *)
                      ▯ k ≠ q ⟶ skip
                     fi
```

```
                ; T'_a(p_1, ..., p_n) := k
               ]|
             od
            ; j := 1 ; do j ≠ n + 1 ⟶ p_{a,j} := q_{a,j} ; j := j + 1 od
           ]|
          ; new_a := false
       od
      ; do p ≠ q ⟶ compute_reprsets(p) ; p := p + 1 od
  od
 ; p := 0 ; F := ∅  (* determine accepting states *)
 ; do p ≠ q ⟶ if S ∈ E(p) ⟶ F := F ∪ {p}
                 [] S ∉ E(p) ⟶ skip
                fi
               ; p := p + 1
  od
]|
```

The table-driven acceptor is presented below.

```
|[ con F : P(N)
 ;      T'_a : N                                              for all  a ∈ V_0
 ;      T'_a : N^n ⟶ N                                for all n : 1 ≤ n : a ∈ V_n
 ;      μ_{a,j} : N ⟶ N                          for all n, j : 1 ≤ j ≤ n : a ∈ V_n
 ;      t : Tree(V, r)

 ; var accepted : bool

 ;func ms_2 =
  (↓ t : Tree(V, r) | N
  | if t :: a ⟶ T'_a
    [] t :: a(t_1, ..., t_n) ⟶ T'_a(μ_{a,1}(ms_2(t_1)), ..., μ_{a,n}(ms_2(t_n)))
    fi
  )

 | accepted := (ms_2(t) ∈ F)
]|
```

Compare the results given in the following example with those of example 4.9.

Example 5.12

Consider again our running example and consider the childsets of our grammar as given in example 5.2. The tables generated by algorithm A_3 are :

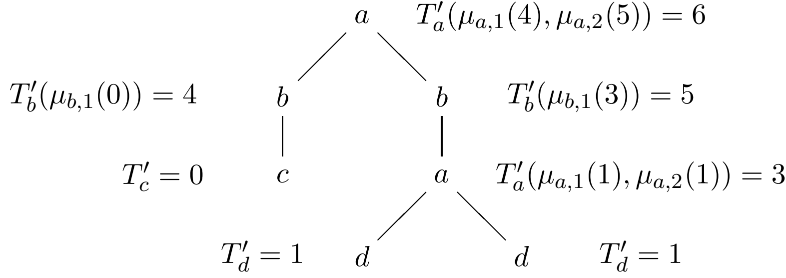

Figure 4: Example of matching using compressed tables

E	match set	$\mu_{a,1}$	$\mu_{a,2}$	$\mu_{b,1}$
0	$\{c; A, B\}$	0	0	0
1	$\{d; B\}$	0	1	1
2	$\emptyset$	1	2	2
3	$\{a(B, d); A, B\}$	0	0	1
4	$\{b(c), b(B); B\}$	2	0	1
5	$\{b(B); B\}$	0	0	1
6	$\{a(b(c), B); A, B\}$	0	0	1
7	$\{a(B, d), a(b(c), B); A, B\}$	0	0	1

T'_a	0	1	2
0	2	3	2
1	2	2	2
2	6	7	2

T'_b	
0	4
1	5
2	2

$R_{a,1}$	
0	$\{B\}$
1	$\emptyset$
2	$\{b(c), B\}$

$R_{a,2}$	
0	$\{B\}$
1	$\{d, B\}$
2	$\emptyset$

$R_{b,1}$	
0	$\{c, B\}$
1	$\{B\}$
2	$\emptyset$

$T'_c = 0,\ T'_d = 1;\ F = \{0, 3, 6, 7\}$

For example, take $E(4)$, which equals $\{b(c), b(B), B\}$.

$E(4) \cap CS_{a,1} = E(4) \cap \{b(c), B\} = \{b(c), B\} = R_{a,1}(2)$, so $\mu_{a,1}(4) = 2$

$E(4) \cap CS_{a,2} = E(4) \cap \{B, d\} = \{B\} = R_{a,2}(0)$, so $\mu_{a,2}(4) = 0$

$E(4) \cap CS_{b,1} = E(4) \cap \{c, B\} = \{B\} = R_{b,1}(1)$, so $\mu_{b,1}(4) = 1$

The bottom-up acceptance of $t = a(b(c), b(a(d, d)))$ now proceeds as depicted in figure 4. Since $6 \in F$, $t \in \mathcal{L}(A)$.

□

6 Concluding remarks

We have derived a rather efficient tabulation algorithm for table-driven bottom-up tree acceptors. The basic idea for the optimized tabulation is quite simple, nevertheless it leads to a complex algorithm. The presented algorithms are derived by step-wise refinement : starting with the well-known reachability algorithm we elaborate this towards an algorithm for the efficient generation of compressed parse tables. Our opinion is that such a systematical derivation gives us more insight in a complex algorithm.
Experiments with an implementation (in Pascal) of the algorithm have demonstrated a considerable improvement in table generation. For example, a code generation grammar with 33 production rules (representing a part of the Intel 8085 instruction set) reduced the number of table entries from 9208 to only 635, of which 468 were index map table entries. Index map tables take up most of the space, but traditional compression techniques can be used to reduce that space since index maps are inherently sparse.
Although the tabulation algorithm has an exponential time complexity we believe that for code generation grammars the mapping of match sets onto equivalence classes is such that a significant improvement is made.

Bottom-up tree acceptors can easily be extended to bottom-up tree parsers. To that purpose the matchset $MS(t)$ of a certain subtree t does not just contain the set of patterns from which t can be derived, but also an encoding of the corresponding derivations.
In instruction selection applications there will be a great (even infinite) number of derivations. By associating a cost with each production rule and by recording for each pattern in a match set the encoding of a minimal cost derivation starting from that pattern, minimal cost parses can be obtained. Thus instruction sequences can be selected that are optimal with respect to e.g. time or memory utilization. For a detailed account of these optimizing parsers we refer to [15].

Acknowledgements
Thanks are due to Huub ten Eikelder, Berry Schoenmakers, and Pieter Struik (all of the Eindhoven University of Technology) for reading and commenting on draft versions of this paper. We would also like to thank an anonymous referee for his helpful comments.

References

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

[2] R.S. Bird, Tabulation techniques for recursive programs, ACM Computing Surveys **12**(4) (1980), 403-417.

[3] W.S. Brainerd, Tree generating regular systems, Information and Control **14** (1969), 217-231.

[4] D.R. Chase, An improvement to bottom-up tree pattern matching, Proc. of the 14th ACM Conf. on Principles of Programming Languages (1987) 168-177.

[5] T.W. Christopher, P.J. Hatcher and R.C. Kukuk, Using dynamic programming to generate optimized code in a Graham-Glanville style code generator, ACM SIGPLAN Notices **19**(6) (1984) 25-36.

[6] T.W. Christopher and P.J. Hatcher, High-quality code generation via bottom-up tree pattern matching, Proc. of the 13th ACM Conf. on Principles of Programming Languages (1986) 119-130.

[7] J. Doner, Tree acceptors and some of their applications, Journal of Computer and System Sciences **4** (1970) 406-451.

[8] F. Gécseq and M. Steinby, *Tree Automata* (Akadémiai Kiadó, Budapest, 1984).

[9] S.L. Graham and R.S. Glanville. A new method for compiler code generation, Proc. of the 5th ACM Conf. on Principles of Programming Languages (1978) 231-240.

[10] C. Hemerik and Y.M. van Dinther, Acceptors and parsers for regular tree grammars, in preparation, Eindhoven University of Technology.

[11] C.A. Hoffmann and M.J. O'Donnell, Pattern matching in trees, Journal of the ACM **29**(1) (1982) 68-95.

[12] P.K. Turner, Up-down parsing with prefix grammars, ACM SIGPLAN Notices **21**(12) (1986) 167-174.

[13] M. Rem, Small programming exercises 5, Science of Computer Programming **4**(3) (1984) 323-333.

[14] W.C. Rounds, Mappings and grammars on trees, Math. Syst. Theory **4** (1970) 257-287.

[15] Y.M. van Dinther, The systematic derivation of acceptors and parsers for tree grammars, Master's Thesis, Eindhoven University of Technology (1987). (in Dutch)