Tree Parser

Periyasamy Kasilingam

A Major Technical Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

July 1987

ISBN 0-315-37107-2

ABSTRACT

Tree Parser

Periyasamy Kasilingam

A parser generator and a parser for tree grammars have been implemented. The parser has two passes; the top-down pass recognises the input and the bottom-up pass builds the output. The parser handles ambiguous inputs and writes out all possible parses. The parsing time is linear to the size of the input. The performance of the tree parser has been critically compared with similar parsers. The tree parser could be applied to code generation in compilers, syntactic pattern matching and tree transformation techniques.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my thesis supervisor Dr.Hendrik Boom who initiated this thesis. His valuable guidance during the course of this work and his patience in correcting the manuscript are greatly appreciated. His proven experience in the area of compilers enabled me to understand the subject in great detail.

I would like to thank the members of the Programming Language Study Group of the Department of Computer Science at Concordia University for their help and suggestions.

My special thanks is to my family members and friends for their moral support and encouragement.

Table of Contents

Chapter 1
Introduction

Parsing is the process of identifying the syntactic structure of the sentence in a language based on the rules of a grammar. A grammar is a way of specifying a potentially infinite language (set of strings composed from the symbols of the language) in a finite way [6]. The rules of the grammar are called productions which specify the valid patterns of the input to appear for a correct sentence. The parsing algorithm or technique very much depends on the underlying grammar. The program which implements the parsing algorithm is called a parser. The output of the parser is generally the syntactic structure of the sentence being parsed, usually a tree called a derivation tree. It shows how the input sentence is derived in steps from a symbol (called the start symbol of the grammar). The design of the parsing algorithm thus becomes the design of the rules of the underlying grammar and the implementation of the parser depends on the system on which it is implemented.

Context-free, Context-sensitive and Regular grammars are the ones normally used for the specification of programming languages. A tree grammar is a special kind of grammar in which the sentences of the associated language have a specific structure called a tree structure. Even though the notion of tree grammar was known much earlier (in the early seventies), its application became popular only recently. In the past few years, tree grammars have been used for tree pattern matching techniques applicable to syntactic pattern recognition, tree transformation and code generation in compilers. In these applications, the valid patterns are described by the productions of a tree grammar and the pattern matching is achieved through parsing the input.

In the code generation phase of compilers, the productions represent the patterns describing the instruction set of the target machine. The input is the expression tree from the parser which is then parsed by the tree parser to match the different patterns of various productions. Once a match is found, that pattern in the expression tree is replaced by the left side symbol of the production. At the same time, the production number is attached to another skeletal tree which preserves the structure of the expression tree. This process continues until the expression tree is reduced to a single symbol; Then the generation of code is simply a table look-up procedure which emits code for each production in the skeletal tree when the tree is traversed, usually in preorder. The code generation process could be deferred until the complete skeletal tree is built up so that ambiguities could be preserved or resolved before code generation.

In their paper, Aho and Ganapathi[1] have described an algorithm for tree pattern matching applicable to code generation. They have provided a tree automaton with attributes associated with each symbol. In fact, it was developed with the idea of incorporating the tree pattern matching along with dynamic programming to develop a language for code generation. The attributes are consulted at run time in case of ambiguous situations to select a unique match.

Turner[19] has described a very similar parsing technique with prefix grammars. A prefix grammar is essentially a tree grammar with the trees flattened into strings using polish prefix notation. The similarities between Turner's parsing technique and the tree parser presented here, are discussed in a later section.

Hatcher[9] has presented an algorithm for code generation using bottom-up tree pattern matching. He uses three passes over the input tree - the first bottom-up pass identifies the collection of patterns at each node of the expression tree, the second top-down pass selects the appropriate pattern at each node using the cost information passed from the parent node and the third pass emits the code by a table look-up. Hatcher claims that his technique generates optimal and efficient code.

The pattern matching techniques in trees have been well discussed by Hoffmann and O'Donnell[11]. They have described both bottom-up and top-down approaches and also have critically compared both the approaches. Yet their algorithms and constraints are based on strings; the trees are reduced to strings using polish notations.

Akoi[3] has described the tree grammar and top-down parsing algorithm and Akoi and Matsuura[4] have described the bottom-up parsing applicable to tree languages. Since both the publications are in Japanese, I am unable to explain the nature of their algorithms. However, they could still be compared with the tree parser presented here, if they were translated into English.

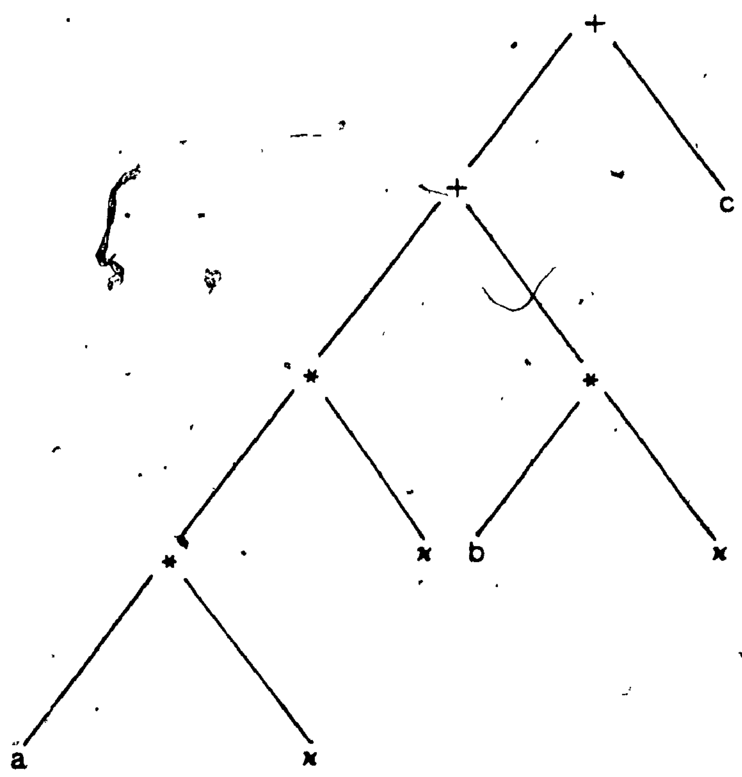
The tree parser presented here is a top-down parser based on tree grammars (which are context free tree grammars in our case). It reads an expression tree as input and constructs the skeletal tree with production numbers used in the derivation, as mentioned earlier. A special feature of interest about this parser is that it handles ambiguity; i.e. for the same input tree, the parser outputs all possible skeletal trees, if any. It preserves all the outputs so that a later phase can resolve the ambiguity based on any criteria. Also, precomputation of parsing tables makes the tree parser fast.

The definitions are mentioned in the next section, followed by the discussion of the parsing algorithm in the subsequent section. After the parsing algotithm is the section which explains the precomputation procedure and tables. The section on implementation follows immediately and is followed by the Conclusion. Appendix A at the end gives an insight into some details of the programs. A list of references at the end cite the related works in the field.

Chapter 2

Definitions

The notations and the definitions described here are quite similar to those used by Rounds[16]. Let Σ denote the set of symbols treated as operators in the tree grammar. Basically these symbols represent the nodes or vertices in the tree. It is conventional to represent a tree upside down with the root at the top and the leaves at the bottom. As an example, a tree for the arithmetic expression a*x*x + b*x + c will look like



and Σ in this case will be {+,*,a,b,c}.

A ranked alphabet is a pair (Σ ,r) where Σ is a finite set of symbols and r is a mapping r : Σ --> N, associating a rank with each symbol in Σ ; N here is the set of natural numbers.

We write

$$\Sigma_n = r^{-1}\{n\}.$$

In trees, we refer the rank as the number of descendants of the symbol at a node. Thus for the above example, r(+) = r(*) = 2; r(a) = r(b) = r(c) = 0.

Let us first define context free grammars and later show the analogy between context free grammars and context free tree grammars.

A context free grammar[1] is a quadruple G = (N,T,S,P) where N is called the set of non-terminals, T is another set called the set of terminals, disjoint from N. S is an element of N called the start symbol of the grammar and P is the set of rewrite rules called productions. Each production is of the form X --> w, where X is a non-terminal and w is a string of symbols from N U T.

Let w be a string from N U T. By replacing one non-terminal occurrence of w by a string x, where the non-terminal and x constitute a production, another string could be derived from w. Such a replacement step is known as 'derivation'. We use the notation '-->' for derivation in one step, meaning that one of the non-terminal occurrences, say E, on the left side of --> is replaced by a string, say w, and E --> w is a production in the grammar. As an example, aEb --> awb is a one-step derivation if there is a production E --> w in the grammar. The set of all strings of terminal symbols that could be derived from the start symbol by successive replacement of the non-terminal occurrences by their associated productions is the language accepted by the grammar and these strings are the sentences of the language. If the string derived from the start symbol contains non-terminals, then it is called a sentential form in the grammar. Symbolically, it could be stated as

$$L(G) = \{ w \mid S \text{ -->* } w, \ w \text{ is a sentence}\}$$

(-->* refers to the transitive reflexive closure of --> and indicates derivation in zero or more steps).

Let us next define the tree and tree grammars. Let Σ be an arbitrary set of ranked symbols. The set Γ of trees over Σ is inductively defined as follows:

i) $\Sigma_0 \subseteq \Gamma$;

Σ_0 represents the set of symbols forming single node trees with no descendants. Typically Σ_0 forms the leaves of the trees.

ii) if $t_0,\ldots,t_{n-1} \in \Gamma$ and $\sigma \in \Sigma_n$

then $\sigma(t_0,\ldots,t_{n-1}) \in \Gamma$.

A tree grammar (being a context free tree grammar in our case) is a quadruple defined as $G_t = (V, \Sigma, S, P)$ where V is the set of non-terminals and Σ is the set of operators, S is in V, being the start symbol of the grammar, and P is the set of productions of the form N --> e where N is a non-terminal and e is a tree over V U Σ. A non-terminal in V has rank zero. We put the restriction that each non-terminal occurs at the left side of at least one production.

Σ contains the set of operators. An operator of rank n has n operands. Operators are analogous to the terminals in the context free grammar.We use the same notation --> for derivation in one step and -->* for the transitive reflexive closure of --> indicating derivations in zero or more steps. The sentences of the tree grammar are the trees derived in zero or more steps from the start symbol S. The language accepted by the tree grammar is the set of all sentences produced by the tree grammar. Mathematically,

$$L(G_t) = \{ t \mid S \longrightarrow^* t \text{ and } t \in \Gamma \}.$$

To save space in this document, Polish prefix notation is used to represent trees. (This is only for typographical convenience and is not the computational representation).

A smurf is a tree over V U Σ in the tree grammar. It is a single terminal, non-terminal or part or whole of the right side of a production. Turner[19] and Aho and

Ganapathi[1] refer to this as a 'pattern'. A smurf is said to be trivial iff it is a single non-terminal; consequently, a trivial production is the one which has a trivial smurf as the right side. The concept of trivial smurfs will be useful in summarizing the derivations due to trivial productions (chain productions [19]).

A smurf f is derived from another smurf e in one step(written e --> f) iff f is obtained from e by replacing one of the non-terminal occurrences in e (say N) by a subtree x (x being a subtree of f) and N --> x is in P.

Recognition is the process of identifying or verifying a sentence of the tree grammar. Parsing is the next phase which indicates the derivation steps of this sentence from the start symbol of the tree grammar. The output of recognition is the collection of smurfs which constitute the input tree whereas the output of parsing is another tree or trees over the production numbers of the tree grammar which indicate the order in which these smurfs are derived in zero or more steps from the start symbol.

We define Recognition and Parsing mathematically as follow:

For a sentence e in the tree grammar and a set of smurfs q, Recog(e,q) is defined as

$\text{Recog}(e,q) = \{ M \in q \mid M \rightarrow^* e\}$.

Notice that

$\text{Recog}(e,q) = \text{Union}\ (\text{Recog}(e,\{M\}) \mid M \in q)$ and

$\text{Recog}(e,\{M\})$ may be either $\{M\}$ itself or empty.

Since sentences are derived from the start symbol, we start the recognition process from Recog(e,S).

Let $PN = \{n \in N \mid P_n \in P\}$ is the set of all production numbers.

Let PT be the set of trees over PN, where the rank of a symbol n in PN is the number of non-terminal occurrences on the right side of the production P_n.

The parse tree t_p is defined as follows:

Each vertex n of t_p, having a rank k, is associated with k-tuple subtrees $(t_1,\ldots,t_k)$ with roots $n1,\ldots,nk$ ($n1,\ldots,nk$ all in PN) such that the non-terminals on the left side of productions $P_1,\ldots,P_{nk}$ exactly match the non-terminal occurrences on the right side of the production P_n.

The set of all parse trees PT_p is a subset of PT. Each element of PT_p represents the derivation of a sentence in the tree grammar, in terms of production numbers.

If t_p is a parse tree with root n having a rank k and associated subtrees $t_1,\ldots,t_k$, then 'expansion' $Ex(t_p)$ of t_p is the result of replacing n by the right side of production

P_n and then replacing the non-terminal occurrences in the right side of production P_n in order, by the expansions $Ex(t_1),\ldots,Ex(t_k)$ of the subtrees $t_1,\ldots, t_k$.

If t_e is a parse tree for a sentence e, then expansion $Ex(t_e)$ is the same as e.

M -->* e iff e = Ex(t) for some parse tree t with M as its top symbol.

Thus $Ex(t_e)$ is said to derive the input e from the parse tree t_e. Mathematically,

Parse (e,q) = { sequences $t_1,\ldots,t_k$ | t_i is in PT_p,
t_i is a parse tree starting with non-terminal N_i,
N_i is the i-th non-terminal of X,
X is a smurf in q,
replacement of each N_i in X by $Ex(t_i)$ results in the derivation of e.}

Notice that for a trivial smurf X,

Parse (e,{X}) = { t | e = Ex(t), n is the root of e and X is the left side of production P_n } and

Recog (e,q) = { X ϵ q | Parse(e,X) is non empty}.

Chapter 3

Parsing Algorithm

The tree parser presented here is functionally similar to a conventional LR parser. It has a parser generator routine which generates the parsing tables from the specifications of the grammar. These tables are stored in a compact way and the 'parser' routine (the LR parser calls this as driver routine) later uses them to parse the input. The performance of the tree parser is comparable with a conventional LR parser. In addition, it also handles ambiguities while parsing but the LR parser does not.

The 'recogniser' of the tree parser is a top down algorithm, identifying the smurfs from the start symbol of the grammar. As it gets down to the leaves in a top-down left-right order, it identifies all possible sentential forms at each node or vertex of the input tree. After recognising the input tree, it outputs the set of smurfs in a bottom-up order. For the effective computation of both the 'recogniser' and the 'parser', the valid subtree patterns at each vertex are all precomputed and stored in four tables. The 'parser' routine after recognition constructs the parse tree from the productions used in the derivation.

The following terms are used in the parsing algorithm

described below:
We consistently use the symbol q to denote a state (set of smurfs) and Q to denote a set of states (i.e., set of set of smurfs).

'Expand' is a relation between smurfs. Expand(u,v) holds iff v is non-trivial and either u = v or u -->* w --> v and w is trivial. Thus v is the first non-trivial smurf in any derivation u -->* v. If u itself is non-trivial, then expand (u,u) holds with no derivation steps. For all the smurfs in the tree grammar, expand is precomputed and tabulated.

The Closure of a set of smurfs q is defined as
Closure(q) = q U { y | expand(x,y) holds and $x \in q$}.

The set q is said to be closed iff closure(q) = q.

The input tree has operators at its nodes. The leaves have 'terminal symbols' which are operators with no operands. We write $f(x_1,\ldots,x_n)$ for a subtree whose root operator is f and whose descendants are $x_1,\ldots,x_n$.

The following is the algorithm for the 'recogniser' which computes $Recog(f(e_1,\ldots,e_n),q)$ when q is closed. The functions Scan, Down, Across and Up are defined in the subsequent paragraphs.

```
S1 : U := Scan (f,q);
S2 : for i := 1 to n do
         RM := Down (i,U);
         U := Across (i,U,Recog($e_i$,RM));
     endfor;
S3 : return (Up (U,q));
```

U and RM are sets of smurfs and they keep track of the current possibilities much like the states (sets of configurations) of an LR parser. U and RM are eventually coded as small integers in the same way as LR states.

We classify smurf sets into states, substates and retracts based on their role in the above algorithm. A single smurf set may be a state, a substate and a retract at different times depending on its usage in the algorithm. We define the states, substates and retracts as follows:

a state is Closure({S}) or a set of smurfs produced by Down.

a substate is a set of smurfs produced by Scan or Across.

a retract is a set of smurfs produced by Up.

The four parsing functions are defined below:

Scan : For a set of smurfs q, Scan(f,q) is the set of smurfs in q with f as root.

Scan (f,q) = $\{ f(x_1,\ldots,x_n) \in q \}$.

Down : For a set of smurfs q, Down is defined for each operand i of each operator f in the grammar. It is the collection of operands and their expands when each operator is analysed in a top down order. Thus,

Down (i,q) = Closure ($\{x_i \mid x_i$ is the i-th son of some $f(x_1,\ldots,x_n) \in q$ and some operator f$\}$).

Across : Across is defined for each operand i of each operator f in the grammar over a set of smurfs q and a set of retracts r. It is the subset of smurfs from q for which the root of subtrees is operator f and the set of all possible i-th operands have been found in r. Precisely,

Across (i,q,r) = $\{ f(x_1, \ldots,x_n) \in q$ for some $f \mid xi \in r\}$

Up : This is the set of all smurfs (a subset of q) returned after recognising the input subtree for an operator f. Typically,

Up (U,q) = $\{ M \in q \mid$ expand (M, X) for some $X \in U\}$.

This set is analogous to the set of 'handles' in an LR parser, defined by Aho and Ullman[2].

3.1. Precomputation

The precomputation process starts with the start symbol of the grammar. All the sentences of the language originate from the set of smurfs created from the start symbol. This is the initial state of precomputation. Each step in the precomputation performs one of the following:

a) creates a new state to be processed

b) creates a new substate (returned by Scan or Across) for the state under processing

c) creates a retract (returned by Up) upon recognition of a part of input.

The precomputation proceeds until none of the above is possible.

For each new state q created, Scan computes a substate for each operator f in the grammar. This set of substates are entered in the Scan table for the respective indices f and q. Also, this substate is added to the list of substates if it does not exist already.

For each of the substates, Down computes a state which is a collection of operands of operator f in the substate. As before, Down table is updated for the respective indices and the state is added to the set of states if it is new.

Once the retracts have been found for the collection of operands computed by Down (empty for the first time), the Across operation computes a set of substates for each operator f. Consequently the list of substates is expanded accordingly if new substates are computed by Across. The Across table is also updated for the corresponding indices.

After all the substates of a state q have been computed and processed, the retracts are finally computed and added to the set of retracts for that state q. These retracts are also entered in the Up table for the respective indexes.

These four steps interact and are repeated until no further state, substate or retract can be created. For instance, if Scan produces a substate then Down and Across follow computing further states and substates respectively. After the retracts have been computed, Scan proceeds again for the next state produced by Down in the previous step. This may create another new substate and the computation proceeds. Down will use the new retracts to continue the iteration.

3.2 Precomputation Algorithm

We simultaneously construct

- a set Q of 'states'
- sets Q'(q,f,i) of 'substates' for each state q in Q,

operator f and integer i (with i <= rank of f)

- sets R(q) of 'retracts' for each q in Q

by generating these elements as follows:

S0 : Initially Q is empty.

S1 : Insert closure ({S}) into Q; S being the start symbol.

S2 : For each state q in Q and each n-ary operator f in the grammar, the following steps are iterated until all the states are processed with all operators.

S21 : Insert Scan(f,q) into Q'(q,f,0) as a substate.

S22 : For each U in Q'(q,f,i-1), 1 <= i <= n, insert Down(i,U) into Q as a state.

S23 : For each U in Q'(q,f,i-1), 1 <= i <= n, and r in R(Down(i,U)), insert Across(i,U,r) into Q'(q,f,i) as a substate.

S24 : For each U in Q'(q,f,n) insert Up(U,q) into R(q) as a retract.

The following example is an illustration of the precomputation process.

The nullary operators are {a,b}

The non-terminals are {E,F,G}

There are two binary operators {+,*}

The Start symbol is E

The following are the productions: (Note that the right side of each production is a tree, here written in polish prefix form.)

E --> +EE

E --> *EE

E --> F

E --> G

F --> a

G --> b

The smurf set computed from this grammar is {a,b,E,F,G,+EE,*EE}.

The initial state is the closure of the start symbol; i.e. closure({E}) = {E,+EE,*EE,a,b} = q_1.
These are the smurfs that could be expanded directly from the start symbol. Note that there is no production E --> a or E --> b but still the smurfs a and b appear in q1because of the trivial productions E --> F and E --> G respectively. As 'expand' between the smurfs is precomputed and tabulated, these trivial expansions are computed only once thus saving a lot of time in the later processing.

Consider the operator +, for example. Scanning q_1 for + yields the substate Scan(q_1,+) = Q'(q_1,+,0) = {+EE}. The Down operation for the first operand of + yields Down(1,+,q_1) = Closure({E}) = {E,+EE,*EE,a,b} = q_1. The second operand also creates the same. Since no retracts have been computed yet, Across does not yield anything at this moment. So the operation on + is suspended until a new retract is computed. Scanning for the operator a yields a substate Scan(q_1,a) = Q'(q_1,a,0) = {a}. Since a is a nullary operator, there is no further processing by Down and Across on a. The Up operation on Q'(q_1,a,0) will compute the retract {a,E}. Once this retract is computed, Across on + continues to compute the substate {+EE} for the first and second operands. In the next step of iteration, Scan does not operate since no new state was computed by Down; neither does Down. The Up operation on + now computes the retract Up(+,Q'(q_1,+,2)) = {E,+EE}.

Proceeding in this way for all operators, we get the following table:

Initial state : q_1= {E,+EE,*EE,a,b}.

Scan(q_1,a) = {a};

Scan(q_1,b) = {b};

Scan(q_1,+) = {+EE};

Scan(q_1,*) = {*EE};

Down(1,+,q_1) = Down(2,+,q_1) = Down(1,*,q_1) = Down(2,*,q_1) = q_1

There is no entry in the Down table for the operators a and b, since they are operators with no operands.

Up(a,q_1) = {a,E} = r1;

Up(b,q_1) = {b,E} = r2;

Up(+,q_1) = {+EE,E} = r3;

Up(*,q_1) = {*EE,E} = r4.

$\mathrm{Across}(1,+,q_1,r1) = \mathrm{Across}(2,+,q_1,r1) = \{+EE\}$;

$\mathrm{Across}(1,+,q_1,r2) = \mathrm{Across}(2,+,q_1,r2) = \{+EE\}$;

$\mathrm{Across}(1,+,q_1,r3) = \mathrm{Across}(2,+,q_1,r3) = \{+EE\}$;

$\mathrm{Across}(1,+,q_1,r4) = \mathrm{Across}(2,+,q_1,r4) = \{+EE\}$;

$\mathrm{Across}(1,*,q_1,r1) = \mathrm{Across}(2,*,q_1,r1) = \{*EE\}$;

$\mathrm{Across}(1,*,q_1,r2) = \mathrm{Across}(2,*,q_1,r2) = \{*EE\}$;

$\mathrm{Across}(1,*,q_1,r3) = \mathrm{Across}(2,*,q_1,r3) = \{*EE\}$;

$\mathrm{Across}(1,*,q_1,r4) = \mathrm{Across}(2,*,q_1,r4) = \{*EE\}$.

After creating all the four parsing tables, the parser is now ready to parse the inputs. The parsing process is divided into two phases - recognition and parse tree construction. Recognition identifies the input tree belonging to the grammar and the latter constructs the parse tree.

3.3 Recognition

In the recognition phase, the input is analysed top-down by scanning over each vertex and identifying the subtree below it recursively. It is a top-down algorithm performing left-to-right processing. The scanning and downward analysis are all done in the precomputation process itself and so recognition simply uses these tables and identifies the input very quickly. An error in recognition (which indicates that the input sentence is invalid) is flashed out immediately when there is no entry in any of the parse tables for a set of appropriate parameters. Recognition in that case is stopped immediately and the next phase is abandoned.

The following is the recognition algorithm for recognising the input e in a closed set of smurfs q.

Algorithm Recog(e,q)

Let q be the initial state (the set of smurfs computed as closure({start symbol})). Let e be the input tree. We use the notation $e = f(x_1,\ldots,x_n)$ where f is the n-ary operator at the root of the input tree e and $e_1,\ldots, e_n$ are its n operands.

```
S1 : U := Scan(f,q); (* Scan for the subset of smurfs with f
                        as root.  Error, if Scan(f,q) is
                        not available *)
S2 : for i := 1 to n do
     RM   :=  Down(i,U);  (* get the collection of i-th
                        operands.  Error, if Down(i,U) is
                        not available *)
     U  :=  Across(i,U,Recog(e_i,RM));  (* the recursive call
                        recognises the i-th operand e_i in
                        RM.   Across  filters  out  those
                        elements of U that are compatible
```

```
                                    with the smurfs returned by the
                                    recursive call.  Error, if there is
                                    no entry in the Across table for
                                    these arguments *)
    endfor
S3 : return(Up(U,q)); (* returns the retracts corresponding
                                    to U and q.  Error if Up table
                                    entry for U and q is not found *)
```

3.4 Parse Tree Construction

The output of recognition is a collection of retracts returned at each level of recursion. Since the analysis is top-down, recognition stores the retracts in bottom-up order in a linear array Q. Hence the second phase constructs the parse tree in bottom-up fashion starting from the leaves. The retract is a set of smurfs constituting the subtree at that level of recursion where it is returned. If there are multiple choices (i.e., different sets of smurfs constituting the same subtree) then recognition outputs all of them. It is then the task of the second phase to construct the ambiguous tree for the multiple choices. This is the most significant feature of the tree parser.

The parse tree written out by the second phase is a tree of production numbers. These are the productions used in the derivation of the input tree from the start symbol of the tree grammar. Obviously, in order to construct the parse tree from the output of recognition, there must exist a mapping between the set of smurfs and the productions. We define the mapping as follows:

Prodsmurf(u,v) is a sequence of production numbers where the sequence is cycle-free. A sequence is cycle-free if there is no repetition of production numbers.

The sequence for prodsmurf(u,v) represents the productions used in the derivation of v from u (i.e., u -->* v). If the derivation involves only one step, then the sequence will contain only one production number. If the derivation involves no step, then the sequence is empty and so prodsmurf(u,v) is an empty sequence. Thus prodsmurf(u,u) is always an empty sequence. If there is a cycle in any of the sequences, then a status information is stored along with that sequence indicating the presence of the cycle. The retracts as well as the set of production chains for all combinations of smurfs in each retract are all precomputed. For the ease of construction of the parse tree, the recogniser writes out a pointer to the input tree along with each retract. This pointer points to the root of the subtree which has been recognised by that retract.

We use a linear array called 'Table' to store the nodes of the parse tree temporarily while they are being constructed. Initially Table is empty.

Let Q be the set of retracts written out by the recogniser.

$Q = (q_1, \ldots, q_n)$, q_1 written out first in bottom-up order.

The following algorithm describes the construction of the parse tree.

For each q in Q, do the following:

i) For each pair of smurfs s_1, s_2 in q, check whether Prodsmurf(s_1, s_2) is non-empty; if it is, then enter the first node in the sequence Prodsmurf(s_1, s_2) into Table.

ii) For each node t in Table that are stored in step (i), construct the subtree with node t as the root. If the node t is the first node of a sequence, then the subtree is constructed with the last node in that sequence as the root. This implies that the sequence itself is to be constructed with each node having only one son. The construction procedure involves the following steps:

For each non-terminal occurrence N on the right side of the production at the node t, select the node r from Table (r is not t) satisfying the following conditions:

a) the left side non-terminal of the production at node r be N.

b) Expansion Ex(t) matches the subtree pointed to by the pointer written out with the retract q.

Assign node r as the i-th son of node t (N being the i-th non-terminal occurrence).

If there are multiple choices for the son of a node t, then all the possible subtrees are assigned as the sons of a dummy node which is then assigned as the son of t.

The example below illustrates this feature.
The following are the productions:

P_1: E --> + G G

P_2 : E --> + F G

P_3 : G --> * F G

P_4 : F --> * a b

P_5 : F --> a

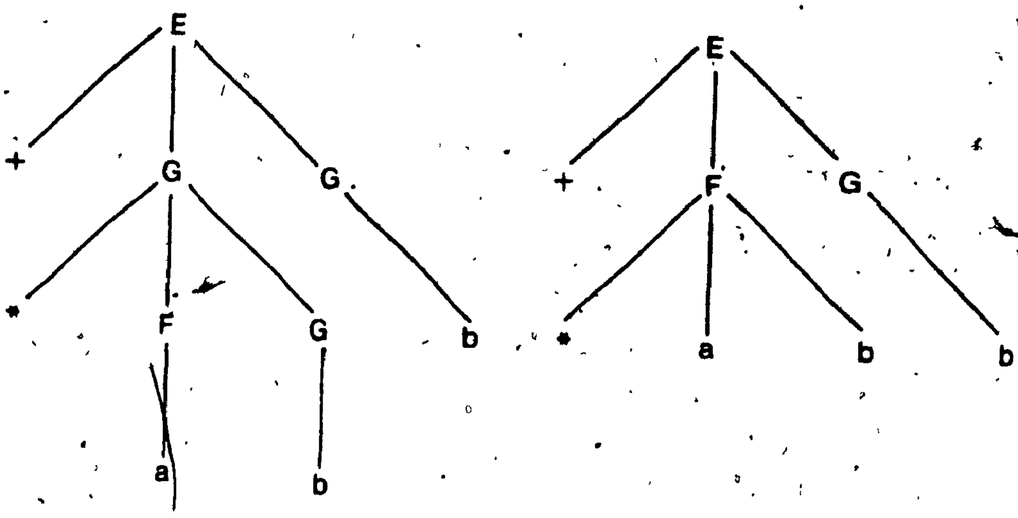
P_6 : G --> b

The smurfs are { a,b,E,F,G,+GG,+FG,*FG,*ab }.

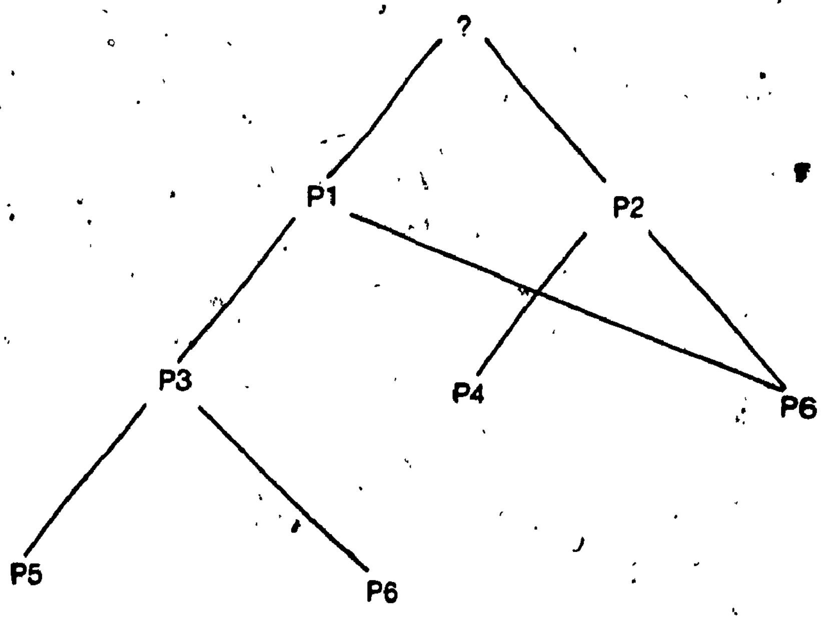
For the input + * a b b , the following retracts are returned by the recogniser in the bottom-up order.

{ a,F }	pointer to a
{ b,G }	pointer to b
{ G,*FG,F,*ab }	pointer to * a b
{ b,G }	pointer to b ... the second b in this case
{ E,+GG,+FG }	pointer to + * a b b

There are two parse trees corresponding to this input, as below:



The output of the tree parser for this input is a parse tree with production numbers which is given below:



(Note that the node with '?' indicates that there are multiple choices at that node and the number of choices corresponds to the number of sons of the node with '?').

Chapter 4
Implementation

The tree parser has been written in Pascal and tested on MicroVAX-II running under the VMS operating system. Few of the routines use the special features in the VAX Pascal Compiler (printing the pointer values, getting the memory statistics etc). The precomputation and parsing have been separated so that precomputation routine runs only once for a grammar and generates the parsing tables. The table entries are coded as small integers and stored in a compact manner. The parsing routine later uses these tables. The tree parser outputs all possible subtrees at each node of the parse tree. At present, the same data structures have been used for precomputation and parsing to simplify the coding and debugging efforts. However, making them different from each other might improve efficiency. Different kinds of data structures have been tried out for the sake of run-time efficiency and storage and their effects will be discussed.

4.1. Representation of the Various Elements

As stated earlier, the basic entity in a tree grammar is a tree as opposed to a string in conventional context free grammars. Trees in tree parser are written using Pascal record structure containing necessary informations and pointers. Typically, each node in the tree will contain the symbol (name or information), its type, number of sons and so on. Since smurfs and the right sides of productions are all trees, they are all represented using the same record structure. The input grammar is read as strings with trees written in prefix notation and the smurfs and productions are later constructed from these strings by a separate procedure. Both of them are entered into separate linear arrays so that any particular smurf or production can be easily indexed for future reference.

Once the smurfs are constructed, the next step is to create the Expand relation between the various smurfs. This is a square table indexed by the smurf indices on both sides. Each entry in the table is a 1 or 0 indicating whether the relation is true or false. The Expand table is thus coded as bit vectors of Pascal enumerated type [0..1] to save space.

The states, substates and retracts are all sets of smurfs. They are implemented as linked lists, each element being a smurf index. Coding the lists with smurf indices rather than smurfs themselves, makes the construction and manipulation of the lists much easier; at the same time, it avoids the duplication of the smurf(tree) structures everywhere. As the precomputation process computes the various sets of smurfs, they are entered into a linear array (as in the case of smurfs) along with their types as state, substate or retract. This will be helpful later while creating the parsing tables. It also eliminates the duplicate entries when the same set belongs to more than one type (e.g. a smurf set may be a state in one case and a retract in another). Some attention has been paid to reduce the storage. The entries of Prodsmurf are set of lists of production numbers. These sets themselves have been implemented as lists. The same list structure used for the smurf set is also employed for these relations except that the list elements now represent the production numbers.

The choice of data structure for the parsing tables affects the speed of the parser. Because these tables are created by the precomputation routine and used by the parsing routine, they are expected to be compact and easily accessible. Also they must be compatible with the data structures of other elements. The sizes of the tables depend on the input grammar.

At first, all the tables were designed as multi-dimensional arrays, the number of dimensions depending on the parameters of each table. The parser was found to be fast but the tables occupied a lot of memory (approximately 400K bytes). Also in this case, each table was found to be at least 75% sparse. Hence it became necessary to switch over to an alternate structure.

The next choice was a structure of multi level pointers, the number of levels depending on the number of parameters of the parsing functions (Scan, Down, Across and Up). Each level had either an array of pointers or array of values. The first level was initialised to an array of pointers, since all tables had two levels at the minimum. As an example, Scan table had an array of pointers at the first level corresponding to the state indices; each pointer pointed to an array of values corresponding to the operators. Similarly, Down table had two levels of array of pointers, the first level for the substates and the second level for the operators; the last level was an array of values corresponding to the number of operands of each operator. The advantage of this scheme over the previous one is that a level of pointers will be created only if at least one entry in the level is non-empty. The dynamic data allocation feature using function 'new' in Pascal is much useful in this case. Even though this technique was found to occupy less storage compared to the previous one, it is slightly slower because of the pointers. Also when there are very few entries in an array, the whole array must still be created, obviously underutilized. Since the parse tables are known to be highly sparse for a reasonable number of parameters, a lot of memory is still wasted in pointers.

The third and most recent scheme uses hash tables, which are well-known for compact storage and fast access[18]. It is possible to have tables of reasonable size so that the hashing technique will be fairly fast and there will be enough room for all the computed values. The entries of the hash tables are records which contain the appropriate parameters and the values computed. For example, Scan table entries are records of three parameters, viz. an operator index, a state index and the value. Since all entries in the records are now indices of arrays, the tables become more compact and easily accessible. Also they could easily be passed between routines.

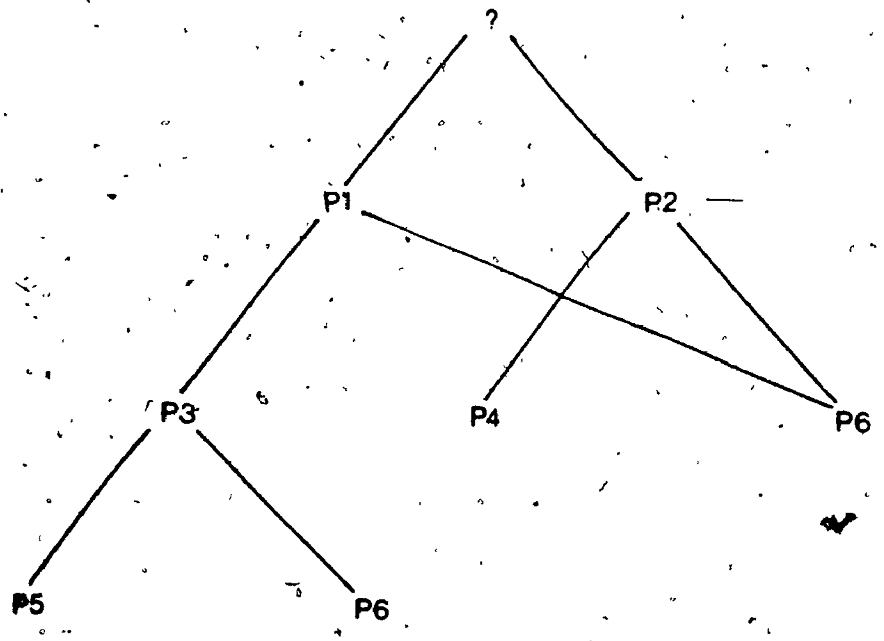
The output of the tree parser is also a tree but with production numbers. This is also implemented using Pascal record structure with pointers. At nodes where multiple choices are possible(i.e., the nodes with '?'), zeroes are stored instead of production numbers in order to distinguish them from other nodes of the tree.

4.2. Printing the Output

Printing a tree structure on paper is quite cumbersome, especially when the size of the tree (depth and breath) is not predictable. The tree output by the tree parser is a good example of this kind. Generally a more complicated printing routine is written in such cases which receives the depth of the tree and the width of the paper as input, computes the printing positions of each node and prints the tree. The tree parser, instead, makes use of a special feature in VAX/VMS Pascal compiler to print the tree in a tabular format. The VAX/VMS Pascal compiler has a built-in function called 'Hex' which converts a pointer value into its character string representation. For example, if P is a pointer holding an address FFFA4 (address in Hexadecimal) then the statement

```
        writeln (Hex(P))
```

will output the string 'FFFA4'. The parser makes use of this novel feature to output the tree in a user-readable format. Each node is output on one line with its address, followed by the information or contents of the node and then the addresses of the descendants. Thus it is possible to output a tree with each node having an arbitrary number of descendants. A typical output is shown below with its internal representation:



(a) internal structure.

Node	Information	Descendants	
35620	?	355D8	35608
355D8	P1	35578	355C0
35578	P3	35548	35560
35548	P5		
35560	P6		
355C0	P6		
35608	P2	355A8	355C0
355A8	P4		

(b) tree parser output

Note that the node P6 (with address 355C0) is shared by two parent nodes (355D8 and 35608) because of the multiple choices at the upper level. Sharing in this way eliminates storage of redundant information; this is especially important when the trees are fairly big and the '?' nodes arise at the upper levels (close to the root).

Chapter 5
Conclusion

A top-down parser based on tree grammar has been presented. The interesting feature of this parser is that it outputs all possible parse trees in ambiguous situations. The parser traverses the input tree twice and so the parsing time will be linearly proportional to the size of the input tree.

Turner[19] has independently constructed a very similar algorithm for parsing, using prefix grammars. A prefix grammar is essentially a tree grammar in which the trees are flattened into strings using polish prefix notation. The 'items' of Turner's algorithm are the same as the 'smurfs' in tree parser. Both the algorithms precompute the states from the grammar specification and code them as small integers which represent the indexes of the state array. The terminals are treated as nullary operators in both cases.

Even though the purpose is the same in both the algorithms, there are quite a few differences between them. Turner's algorithm has two passes like the tree parser but the passes are reversed. Turner's first pass is bottom-up and collects the patterns and numbers the states at each vertex and his second pass is top-down identifying the input tree. The tree parser, on the other hand, collects the patterns in the first top-down traversal and then builds the parse trees while proceeding up in the second pass. This change in traversing directions leads to one significant advantage in tree parser. That is, it preserves all possible parse trees in ambiguous situations until one of them is selected at a later stage. Keeping all the parses is not so hazardous and is affordable at the current lower cost of memories. Turner's algorithm collects all the possible patterns at each vertex but it doesn't build all the parse trees. The time taken for parsing an input is found to be linear to the size of the input and so the tree parser is quite comparable in performance with Turner's and Aho and Ganapathi's algorithms.

Cattell's[8] method of code generation also applies tree pattern matching algorithm. The input tree is first converted into TCOL (Tree-base COmmon Language) representation before pattern matching. His pattern matcher is a top-down single pass algorithm; it emits code as soon as a match is found during the top-down traversal. Also the searching technique is exhaustive; i.e., at each node of the input TCOL tree, a number of patterns may match and the appropriate one is selected which gives the least expensive code sequence. Selecting a locally cheapest one may cause blocking later and force backtracking. This situation will never arise in the tree parser as it outputs all parse trees and the appropriate one is selected later. Computing the patterns of the machine description is quite similar to the precmputation process of the tree parser.

As stated earlier, the tree parser may write out multiple parse trees for a single input. In such case, the routines which use the output of the tree parser, have to select the appropriate parse tree. For example, the code generator in a compiler may use the tree parser; the multiple choices in that case will indicate the possibility of several set of codes for the same input. The code generator may then select the one which involves less memory access or fewer instructions. Usually this is done by incorporating attributes with each symbol in the tree grammar and evaluating the attributes. The tree parser could be made to process and write out attributed trees. The attributes could be computed during either or both the traversals or carried out at the end as a separate process. It is also possible to compute certain attributes of the input tree during precomputation which will make the attribute evaluation easier. The tree parsing will then resemble the method described by Aho and Ganapathi [1]. The tree parser is well suited for applications like syntactic pattern recognition[14] and tree transducers[20].

References

1. Aho,A.V. and Ganapathi,M.
"Efficient Tree Pattern Matching - an Aid to Code Generation"
Twelfth ACM symposium on Principles of Programming Languages, Jan 1985, pp 334-340.

2. Aho,A.V. and Ullman,J.D.
"Principles of Compiler Design"
Addison-Wesley Publishing Company, 1979.

3. Aoki,K.
"A Context-free Tree Grammar and its Top Down Parsing Algorithm"
Trans.Inst.Electron. and Commn. Engg. Japan Part D (Japan), Vol.J66D, No.3, March 1983, pp 294-301 (in Japanese).

4. Aoki,k. and Matsuura,K.
"A Bottom-Up Parsing Method for Complete Context-free Tree Languages"
Trans.Inst.Electron. and Commn. Engg. Japan Part D (Japan), Vol.J68D, No.5, May 1985, pp 1079-1086 (in Japanese).

5. Barrett,W.A. and Couch,J.D.
"Compiler Construction - Theory and Practice"
Science Research Associates 1979.

6. Bauer,F.L. and Eickel,J.
"Compiler Construction - an Advanced Course"
Lecture Notes in Computer Science, No.21,

Springer-Verlag 1974.

7. Boom, H.J.
"Tree Grammars and Code Generation"
Technical Report PLSG-2, Dept. of Computer Science
Concordia University, Montreal, 1982.

8. Cattell, R.G.G.
"Formalization and Automatic Derivation of Code Generators"
UMI REsearch Press 1982.

9. Hatcher, P.J.
"High Quality Code Generation via Bottom-Up Tree Pattern Matching"
Thirteenth Annual Symposium on Principles of Programming Languages, Jan 1986, pp 119-129.

10. Hoffmann, C.M. and O'Donnell, M.J.
"An Interpreter Generator using Tree Pattern Matching"
Proceedings of the Sixth Annual Symposium on Principles of Programming Languages, 1979, pp 169-179.

11. Hoffmann, C.M. and O'Donnell, M.J.
"Pattern Matching in Trees"
Journal of ACM, Vol.29, No.1, Jan 1982, pp 68-95.

12. Johnson, S.C.
"YACC - Yet Another Compiler Compiler"
Bell Labs, Murray Hill 1977.

13. Keller, S.E. et al.
"Tree Transformation Techniques and Experiences"
SIGPLAN Notices, Vol.19, No.6, June 1984, pp 190-201.

14. King-Sun Fu and Bhargava,B.K.
"Tree Systems for Syntactic Pattern Recognition
IEEE Transactions on Computers, Vol C-22, No.12, Dec 1973, pp 1087-1099.

15. Rounds,W.C.
"Context-free Grammars on Trees"
IEEE Conference Record on Symposium on Switching and Automata Theory, 1969, pp 143-148.

16. Rounds,W.C.
"Mappings and Grammars on Trees"
Mathematical System Theory, Vol.4, No.3, 1970, pp 257-287.

17. Tai,J.W.
"Attributed Parallel Tree Grammars and Automata for Syntactic Pattern Recognition"
Proceedings of the Fifth International Conference on Pattern Recognition, Miami, U.S.A., Dec 1980 (NY, IEEE 1980), pp 1001-1003.

18. Tai,K.C.
"On the Implementation of Parsing Tables"
SIGPLAN Notices, Vol.14, No.1, Jan 1979, pp 46-54.

19. Turner,P.K.
"Up-Down Parsing with Prefix Grammars"
SIGPLAN Notices, Vol.21, No.12, Dec 1986, pp

20. Waite,W.M. and Goos,G.
"Compiler Construction"
Springer Verlag 1984.

21. Wei-Chung Lin and King-Sun Fu
"Conversion and Parsing of Tree Transducers for Syntactic Pattern Analysis"
Int.Jour. Computers and Information Sciences, Vol.11, No.6, Dec 1982, pp 417-458.

Appendix A

The tree parser has been implemented on MicroVAX-II running under the VMS operating system. It has been tested with several grammars including the one with 70 productions. This grammar is derived from an existing LISP compiler. The precomputation created 68 smurfs and a total of 232 smurfsets (states, substates and retracts). The Scan, Up, Down and Across tables have 187, 221, 44 and 697 entries respectively. During the test runs, it was observed that the number of entries in the Scan table is linearly proportional to the number of operators and smurfs and those in the Down table depends on the number of operands of each operator. This prediction could be helpful in estimating the size of the hash tables before the precomputation. The current version of the parse tables using hashing scheme occupies approximately 40K bytes of memory whereas the previous versions required 1000K bytes and more. The precomputation process took approximately 2.5 minutes on MicroVAX-II and required a virtual storage of 40K bytes for the computation. This virtual storage is claculated at run-time by the VAX/VMS run-time routine 'Lib$stat_VM' callable from Pascal.

As stated earlier, the current implementation has two separate routines - precomputation and parsing.

Parsing includes procedures for the two passes - top-down recognition and bottom-up building of the parse tree. Most of the functions and procedures are available in both the routines. For the ease of debugging and understanding, the routines have been divided into a number of small subprograms. Typically, the subprograms for the construction, comparison and writing of the smurfs, accessing and manipulating the hash tables etc. are the operations common to the both the routines. At present, all these subprograms have been duplicated in both the routines because they use identical data structures for the various elements. However both the precomputation and parsing routines can be modified independently to improve their efficiency. Since the parsing tables are coded as small integers, which are passed between these routines, modifications in any of the routines will not affect the other.