Ben-Gurion University of the Negev
Department of Mathematics & Computer Science

The Compilation of Functional Unification Based Language

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree of Ben-Gurion University of the Negev

by

Bar Orion Barak

(barakb@cs.bgu.ac.il)

The research work for this thesis has been carried out at
Ben-Gurion University of the Negev, Israel
under the direction of Dr. Michael Elhadad

September 26, 2000

Subject: **The Compilation of Functional Unification Based Language**

This thesis is submitted as part of the requirements for the M.S.c degree

Written By:	Barak Bar Orion
Advisor:	Dr. Michael Elhadad
Department:	Mathematics & Computer Science
Faculty:	Natural Science

Ben-Gurion University of the Negev, Israel.

Author Signature: ____________________ Date: _______________

Advisor Signature: ____________________ Date: _______________

Dept. Committee Chair Signature: ____________ Date: _______________

Abstract

This work is focused on the definition of and search for efficient techniques for compilation of FUF, a unification-based language utilized in the field of natural language generation.

FUF is a programming language designed for the purpose of natural language realization by use of a unification mechanism. The inputs for FUFare a structure which represents the semantics and a structure which represents the grammar of a natural language. Its output is a structure that, when linearized, generates a sentence written in the language's grammar, and having the required semantics.

In earlier works, the internal data structure was represented either in the form of a list or in the form of a directed graph, where each method had its strengths and its weaknesses. A description of the unification process, in terms of primitive operations was never presented in earlier works. In this paper, such a description is given. Moreover, I have described in this paper how to translate FUF into a set of primitive machine commands.

Specifically, in this paper I defined a set of primitive machine commands yielding a result code, and suggested a number of optimizations that generate a faster code. I also took a closer look at some of the problematic points involved in the compilation process. Among these are the treatment of paths, grammar alternatives, and efficient backtracking.

At the conclusion of the paper, I have conducted a number of evaluation tests, which showed that my approach resulted in better run-times, as compared with previous works.

To Daniella—for her loving support;
and to Negev—for long walks taken whenever it saw fit.

Contents

Part I

Chapter 1

Introduction

My research focuses on the definition and optimization of new techniques for the compilation of FUF , a unification-based language used in the field of Natural Language Generation.

The strategy I chose in order to find new optimizations was based on the following two approaches:

- The definition of a virtual machine for the execution of FUF and,
- porting and evaluation of existing optimization strategies from other languages and compilers to this new virtual machine architecture. Specifically, I deployed in the resultant compiler optimization strategies used in logic programming (Prolog compilation).

Chapter 2

Background

The Functional Unification Formalism FUF [5] is a programming language designed for the needs of linguistic realization in a unification-based framework. FUF uses graph unification to combine an input structure that corresponds to a sentence specification, with the grammar of a natural language. The result is a structure that, when linearized, produces a sentence compatible with both the semantic sentence specification and the rules of the grammar. In particular, SURGE (see [10]) the Systemic Unification Realization Grammar of English, is a grammar written in FUF which has been used by a number of research and development groups around the world.

2.1 Review of FUF

2.2 The FD

The input for FUF is a semantic description of the text to be generated and a unification grammar. The structure that describes the semantics of the text to be generated is called a Functional Description (FD). An FD is a list of attribute-value pairs, called features. The attribute of a feature must be a symbol, while the value of a feature can be a leaf (symbol string ...) or, recursively an FD. For example see Figure 2.2.1 on page 18

A given attribute in an FD must have at most one value. Therefore, the FD **((cat v)(cat np))** is illegal. If the attribute **cat** does not appear in the FD, it follows that its value is not constrained and it can be anything. Thus, the FD **()** (empty list of pairs) represents all the objects in the world.

```
((cat s)
(voice  passive)
(prot ((n ((lex John)))))
(verb ((v ((lex sell)))))
(goal ((n ((lex car)
           (number plural))))))
```

Figure 2.2.1: A simple FD

2.2.1 Paths

Any position in an FD can be unambiguously referred to by the path leading from the top of the FD to the value considered. For example, in Figure 2.2.1 the path to **John** is **{prot n lex}**. Paths are represented as lists of atoms between curly brackets. A path can be absolute or relative. An absolute path gives a way from the "top" of the FD down to the value. A relative path starting with "^" refers to the FD *embedding* the current feature. For the FD of Figure 2.2.1 is shown next:

```
((cat s)
 (voice passive)
 (verb ((v ((lex sell)))
        ({^ prot n lex} John)))
 (goal ((n ((lex car)
         (number plural))))))
```

Figure 2.2.2: The same simple FD

One can have several "^"s in a row to go several levels up or "**^number**" to go "number" levels up, For example: (({^ ^ ^} value)) has the same meaning as (({^3} value)))

In FUF a path can appear in the left as in the right side of feature as in Figure 2.2.3.

FDs as Graphs

Sometimes, it is best to represent an FD as a graph. This representation provides a clear interpretation of the path mechanism and makes the reading and understanding of relative paths much easier. The structured format of FDs can be viewed as equivalent to a direct graph with labeled arcs as pointed out in [14]. The correspondence is established as follows: an

```
((a b)
 (a c))

;; is the same as:
(({a} {b})
 (b c))

;; or even:
(({b} {a})
 (b c))
```

Figure 2.2.3: Paths on both sides of a feature

FD is a node, each pair **(attribute value)** is a labeled arc leaving this node. The **attribute** of the pair is the label of the arc and the value is the adjacent node. Internal nodes in the graph have therefore no label whereas leaves are atomic values. The equivalence is illustrated in Figure 2.2.4 on page 19.

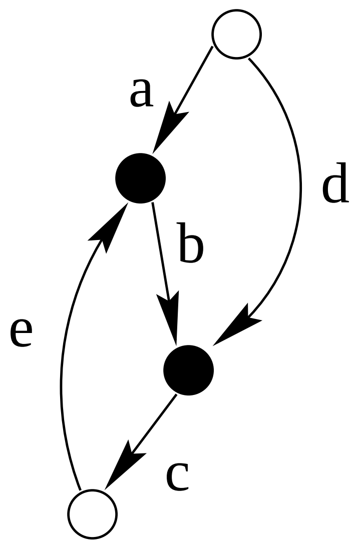


Figure 2.2.4: Graphical representation of FD.

The graph of Figure 2.2.4 can be expressed via list of pairs as

```
((({a b c e} {a}))
 ({a b} {d}))

;; using absolute paths, or

((a ((b {^2 d})
     (b ((c ((e {^3}))))))))

;; using relative paths.
```

Figure 2.2.5:

Functional Descriptions vs. First-order Terms

It is important to understand the difference between functional unification (FU) and the better known structural unification (SU) used in PROLOG for example. The FD and first-order term in Figure 2.2.6 can be used to represent the fact that the book **The Lord Of The Rings** moved from **room 256** at **building Gimel** to **room 30** at **building Aleph**.

```
((process move)
 (object ((concept book)
          (title "The Lord Of The Rings")
          (pages 965)))
 (from   ((concept position)
          (building Gimel)
          (room 256)))
 (to     ((concept position)
          (building Aleph)
          (room 30))))

move(book("The Lord Of The Rings",256),
      position(Gimel,256),
      position(Aleph,30))
```

Figure 2.2.6: FU vs. SU

This example illustrates the following differences:

- *Symbolic labels for substructures:* the arguments are labeled in the feature structure notation and referred to by position in the first-order term.

- *Fixed arity:* features can be added at will to the FD. FDs are used to represent *partial information.* This is not the case for first-order terms.

- *Functions and arguments:* first-order terms have names which play a central role in the unification process. In FDs all the information plays the same role.

- *Variable and coreference:* in standard unification a variable is used to denote that the value is unknown (there are no constraints on it) or the value is the same as another value, *e.g.,* **like(X,X)**. In FU no constraints means that the variable will not appear in the FD, while coreference is handled by the path mechanism, *e.g.*, **((process like)(agent {object}))**.

The Unification Grammar

The second input to FUF is a unification grammar. A grammar which has the syntax of FD but it has other constructions that make it resemble a program rather than a simple data structure. The most significant of these constructs is the *alt* construction. *alt* stands for "alternation". Figure 2.2.7 demonstrates the syntax of *alt.* The semantics of alt explained at Chapter 9.

```
((attribute1 value1)
 (attribute2 value2)
   ...
 (alt (annotation*) (fd1 fd2 ... fdn))
   ...
 (attributen valuen))
```

Figure 2.2.7: alt

Another grammar construction is the *pattern* which imposes a constraint on the ordering of the subconstituents of the unified FD. For example, in a sentence containing the constituents *prot*, *goal* and *verb*, the following *pattern* can be used:
(pattern (prot verb goal)).

How unification is done

The unifier works top-own recursively: it unifies first the top level FD against a grammar and then recursively, it unifies each of its constituents with the same grammar. The constituents can be recognized by one of the following characteristics:

- The FD contains a feature with attribute **cat**.
- Every sub FD mentioned in *pattern* is a constituent.
- Every FD explicitly specifies with the *cset* (Constituent SET) mechanism is a constituent.

The cset

As mentioned above the *cset* is a mechanism enabling the user to specify explicitly which features of an FD correspond to constituents and therefore need to be recursively unified. For example, in Figure 2.2.8 the constituents are FD1 and FD2.

```
((attribute1 FD1)
 (attribute2 FD2)
   ...
 (cset (attribute2 attribute1))
   ...
 (attributen valuen))
```

Figure 2.2.8: cset

Types

In a pure FUnctional Grammar, there is no structure over the set of values, it is a flat collection of symbols with no relation between each other. All constraints among symbols must be expressed in the grammar. In linguistics, however, grammars assume a rich structure between properties: some groups of features are mutually exclusive while others are only defined only in the context of other features.

The schema in Figure 2.2.9 indicates that a noun can be either a pronoun, a proper or a common noun. Note that these three features are

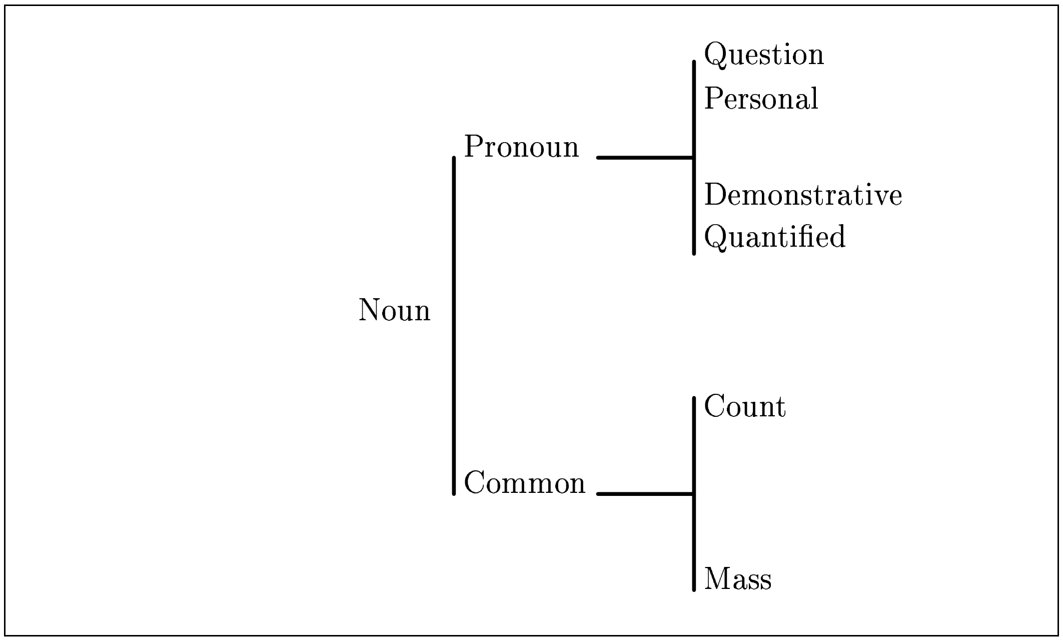


Figure 2.2.9: Types

mutually exclusive. In order to enforce the the correct constraints, it is necessary to use the meta-FD NONE

This approach has some limitations:

- Both the grammar and the FD are redundant and longer than needed.
- The branches of the alternations in the grammar are interdependent.

To overcome this limitation of pure Functional Unification, FUF uses types as suggested in [4].

In addition to those mentioned so far, FUF has other control structures. For complete specification of FUF and its grammar, see [5].

An example of FUF input and output

The grammar gr0 is shown on Figure 2.2.11 on page 25, while its input and the reasult of the unification are shown in Figure 2.2.13 page 26 and Figure 2.2.14 on page 27 respectivly.

```
((alt (((noun pronoun)
        (common none)
        (pronoun
         ((alt (question personal demonstrative quantified)))))
       ((noun proper) (pronoun none) (common none)))
       ((noun common)
        (pronoun none)
        (common ((alt (count mass))))))))

;; The input FD describing a personal pronoun is then:
((cat noun)
 (noun pronoun)
 (pronoun personal))
```

Figure 2.2.10:

```
((alt top (
   ;; a grammar always has the same form: an alternative
   ;; with one branch for each constituent category.

   ;; First branch of the alternative
   ;; Describe the category S.
   ((cat s)
    (prot ((cat np)))
    (goal ((cat np)))
    (verb ((cat vp)))
    (alt (
      ((voice active)
       (subject {^ prot})
       (object  {^ goal}))
      ((voice passive)
       (by-obj ((cat pp)
                (prep ((lex "by") (cat prep)))
                (np   {prot})
                (pattern (prep np))))
       (subject {^ goal})
       (object  {^ by-obj}))))
    (verb ((number {^ ^ subject number})
           (voice  {^ ^ voice})))
    (pattern (subject verb object)))
   ;; Second branch: NP
   ((cat np)
    (n ((cat noun)
        (number {^ ^ number})))
    (alt (
      ;; Proper names don't need an article
      ((proper yes)
       (pattern (n)))
      ;; Common names do
      ((proper no)
       (pattern (det n))
       (det ((cat article)
             (lex "the")))))))
```

Figure 2.2.11: The grammar gr0 (part one)

```
      ;; Third branch: VP
   ((cat vp)
    (alt (
      ((voice active)
       (pattern (v dots))
       (v ((cat verb) (number {^ ^ number}))))
      ((voice passive)
       (pattern (v1 v dots))
       (v1 ((cat verb) (lex "be") (number {^ ^ number})))
       (v ((cat verb) (ending past-participle)))))))
   ;; Fourth branch: PP
   ;; does nothing.
   ((cat pp))
   ;; Fifth branch: Article
   ((cat article))
   ((cat prep))
   ((cat noun))
   ((cat verb))))))
```

Figure 2.2.12: The grammar gr0 (continued)

```
;; An input fd.
((cat s)
  (prot ((n ((lex  man) (proper no)))))
  (verb ((v ((lex eat)))))
  (goal ((n ((lex  meal) (proper no))))))
```

Figure 2.2.13: An input to the grammar in Figure 2.2.11

```
;; The output of the unifier.
((cat s)
 (prot ((n ((lex "man")
            (cat noun)
            (number {subject number})))
        (proper no)
        (cat np)
        (pattern (det n))
        (det ((cat article) (lex "the")))))
 (verb ((v ((lex "eat") (cat verb) (number {verb number})))
        (cat vp)
        (number {subject number})
        (voice {voice})
        (pattern (v dots))))
 (goal ((n ((lex "meal")
            (cat noun)
            (number {object number})))
        (proper no)
        (cat np)
        (pattern (det n))
        (det ((cat article) (lex "the")))))
 (voice active)
 (subject {prot})
 (object {goal})
 (pattern (subject verb object)))
```

Figure 2.2.14: The ouput of the grammar from Figure 2.2.11 and the input at Figure 2.2.13

Chapter 3

Existing Implementations

3.1 FUF

FUF was developed by Elhadad[5] as an implementation and extension of the FUG formalism[17] — Functional Unification Grammar. The FUF interpreter is written in Common Lisp and extends FUG in many features, such as *types* and *external* that ease the grammar writing and *bk-class* which limits the search and thus improves performance.

3.2 CFUF

CFUF is a FUF interpreter written in C by Mark Kharitonov [19]. This implementation uses direct graphs to represent FD (as in Figure 2.2.4) and in particular it relies on passive merger rather than active merger as the process of making two nodes equivalent. CFUF, inspired by [2], is based on a set of commands generated by the parser for creation of an FD. These commands are interpreted by the *FD Virtual Machine* (FDVM).
Figure 3.2.2 shows the commands generated by the FD from Figure 3.2.1

```
;;  The FD.
((cat s)
 (voice passive)
 (verb ((v ((lex sell)))
 ({prot n lex} John)))
 (goal ((n ((lex car)
 (number plural))))))
```

Figure 3.2.1: An FD

```
;; Initialize
  ;; Add Leaf cat s
  ;; Add Leaf voice passive
  ;; Enter Subgraph verb
    ;; Enter Subgraph v
      ;; Add Leaf lex sell
    ;; Leave Subgraph
    ;; Enter Subgraph {^}
      ;; Enter Subgraph prot
        ;; Enter Subgraph n
          ;; Add Leaf lex john
        ;; Leave Subgraph
      ;; Leave Subgraph
    ;; Leave Subgraph
  ;; Leave Subgraph
  ;; Enter Subgraph goal
    ;; Enter Subgraph n
      ;; Add Leaf lex car
      ;; Add Leaf number plural
    ;; Leave Subgraph
  ;; Leave Subgraph
;; Realize
```

Figure 3.2.2: the FDVM commands that compile the FD from Figure 3.2.1

In addition to commands implementing the construction and modification of FDs, CFUF includes a set of commands to compile a grammar. These commands generate an and-or tree that guides the unification process of an FD with that grammar. The work presented in this thesis builds on Kharitonov's MSc thesis as a starting point.

Chapter 4

Objectives

The objective of this work is to find new approaches for the efficient compilation of FUF, and to explore optimizations based on these approaches.

During the process of creating the compiler, we shall create a machine that will execute the compiler's commands. We shall describe the commands executed by the machine in a low-level language, and see to it that they are as simple as possible. Eventually, this process results in a compiler and a virtual machine which provides a precise description of the FUF semantics in a low-level language. The compiler also provides for a much improved run-time efficiency when executing FUF programs.

Range of Optimizations to be Studied

To identify the most relevant optimizations, I have adopted the following strategy:

- Analyze existing optimization techniques from similar languages. This will cover the compilation techniques used in Logic Programming and the compilation techniques used in functional programming languages ([25]).
- Analyze the runtime behavior of the FUF virtual machine on a representative set of FUF input/grammar unification runs. From this empirical analysis, derive an accurate profile of the time spent by the virtual machine on different types of operations.

Work on the optimizations has been inspired by [2], [25] and [26]

Part II

Chapter 5

What does it mean to compile FUF

Our goal, when given a grammar, is to find a method for executing a given FD as fast as possible. That is, to produce the FD meaning in the grammar's language as fast as we can. In order to achieve this goal, we will try to describe two processes. The first process is a compiler that has the grammar (or the FD) as input. The compiler will generate machine instructions in a certain low level language found suitable for our purpose. The other process is a virtual machine $\mathcal{M} = \langle \mathcal{I}, \mathcal{D} \rangle$ consisting of $\mathcal{D}$, a data store and $\mathcal{I}$, a set of primitive instructions that acts on the machine data. During the process of searching for the desired abstract machine, we shall use the following principles:

abstract machine principle 1 keep the instructions as simple as possible.

abstract machine principle 2 use simple and economical data representation.

abstract machine principle 3 describe the machine in a very level language.

The need for such a tight abstract machine is noted at [23]. Aside from the decision about the data types and the commands of our machine (that is, the concrete representation) we have to choose an abstract model for our computation process.

An example for such a decision in the concrete level would be the choice between active and passive merger (See the discussion in 5.2 on page 40), whereas in the abstract level one could choose between a list or a direct graph representation for an FD.

The process followed to achieve this goal is:

- Find the appropriate abstract representation and a computation model.
- Find the data representation and the instruction set that will fit best our computation model.
- Design a compiler that will genarate the machine instructions.
- Optimize the compiler.

Regrettably, I didn't go that logical way. Hence, in order to give a comprehensible explanation, I explain the logical course actually taken in light of my later insights.

I considered two abstract computational models. In the first model, FDs are represented as lists and FD manipulations are implemented using list primitives. This representation was used by Michael Elhadad [5] to implement the first FUF interpreter. The second model represents an FD as a direct graph, as done by Mark Kharitonov [19] while implementing his CFUF. His choice was based on the original FUF interpreter profiling that showed that in the list representation, the most time consuming task was to find a sub FD specified by a path. For this operation, the graph represention is long known to be superior (see [20]). The computational model arising from this representation is that executing an FD is just building an image of the graph that represents the FD in the machine memory. Executing a grammar is a proccess in which we try to enlarge or verify this image. In this proccess we have choice points, points in which we have a choice of ways to go and we have to try those and backtrack to the latest choice point if we fail. Thus, it is most important that the effects of the machine instructions on the graph should be as easy to delete as it was easy to build them. It is crucial to understand this last demand because it will have a role in the decisions at the concrete level.

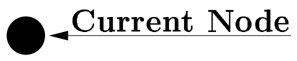
Since I chose the graph representation at the abstract level I had the following options at the concrete level:

- Use passive merger as was done in CFUF, or
- use active merger in which every node has a list of all its parents, or
- use active merger with indirections, similar to the one introduced in Warren's Abstract machine.

Before evaluating each of these three options, let us understand what are the issues that have been addressed by choosing the graph model. To this end, we will try to understand the process of building a graph that corresponds to the following FD.

`((a ((b c)))(d ((e f))) (a {d}))`

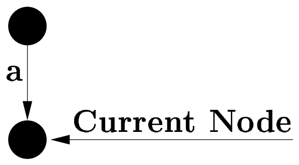
Our first observation regarding the graph is that it has two kinds of nodes: *internal* nodes in which there is no label can have outgoing arcs, and *leaf* nodes that do not have outgoing arcs but contain a label. In this work, I draw *internal* nodes as full small circles and *leaf* nodes as larger circles with their labels in on near them. At the begining, there is only the root of the graph and the current node is the root itself.



When the compiler sees

`((a ((b c))) ...)`

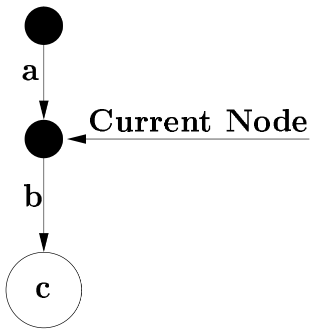
it understands that there should be an internal node that can be reached from the current node by the arc labeled a, hence it issues a command to do so. That command has the form `enter_subgraph(a)` and its meaning is to add a new internal node linked by an arc labeled *a* to the current node and change the current node to be the new node. After the execution of this command, the graph should look like this:



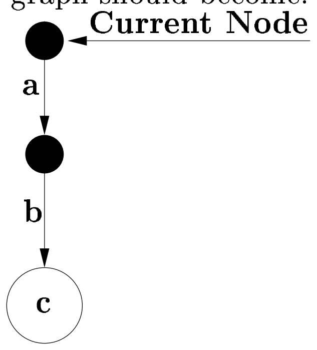
The compiler then continues to compile the inner expression

`(b c)`

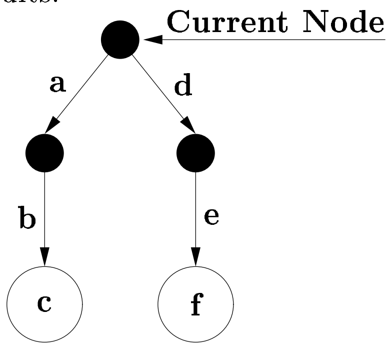
and since it is a leaf node in the graph the compiler generate the command `add_leaf(b,c)`, that is, a command that tells the machine to add to the current node the appropriate leaf. Thus, after the execution of that command the graph should look like this:



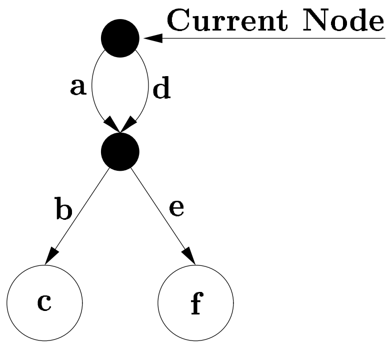
Upon finishing the compilation of the commands for that internal node the compiler must tell the machine to return to the currrent node which should be the previous current node (the one before the execution of `enter_subgraph(a)`). The command `leave_subgraph` serves this purpose and after its execution the graph should become:



The next step is executing the instructions issued by the compiler for
```
(d ((e f)))
```
The following graph results:



The last command issued by the compiler is triggered by the source code:
```
(a {d})
```
Its meaning is to merge the node at the end of the arc labeled *a* and the child of the root at the end of the arc labeled *d*, that is, to merge the two children below the root node. After the execution of this command the graph should become:



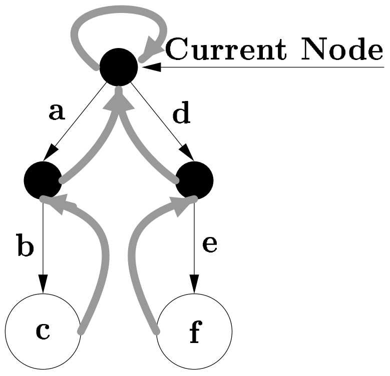
From the example above we see that if we want to choose an efficient graph representation we need to consider the time needed for the following tasks:

- find a leaf node type, that is, find the node label;
- find a specific child of a node;
- add a leaf to an internal node, or update the type of this leaf;
- merge two nodes together;
- and finally, undo all the above.

Note that the merge operation traverses subgraphs recursively. In the following sections, we consider three different approaches for the concrete representation of graphs. Next, we evaluate each representation with respect to the tasks defined above.

5.1 Direct Representation

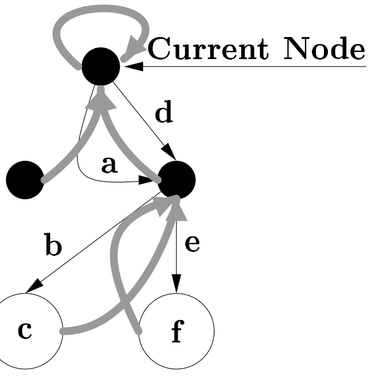
The direct representation is the simplest representation. In this representation an internal node contains two lists of pointers. The first pointing to its children and the second pointing to its parents. The list of parents is needed for the merge operation. Merging two nodes in that representation implies physically reomoving one of them and changing all its parents to point to the node merged with. In addition, in the case of leaf nodes it is needed to save the type as well – to enable the undo of a merge. Thus marking the parent pointers with a grey arrow we shall have the following picture for the FD: `((a ((b c)))(d ((e f))))`



Now, in order to merge {a} and {d} we have to choose one of them, say {a}, and execute the following:

- add to {a} every arc in {d} labeled by a label not found in {a}'s labels. This task includes scanning the parents lists and changing every parent of {d} to a parent of {a}.
- foreach arc that has the same label in {a} and {d} merge the nodes and add the results to {a}.
- add to the parent list of {a} all the parents of {d}.

In our example the graph will look like this:



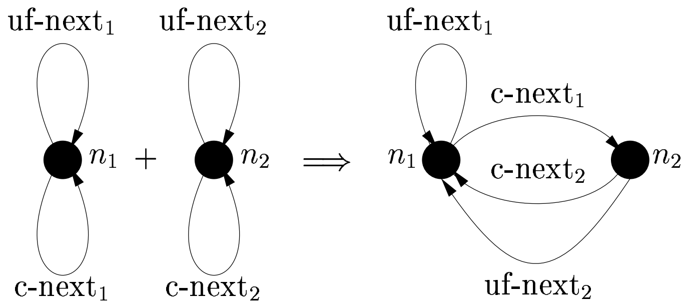
Note that the nodes that don't have a path from the root should be collected by the garbage collection or left alone in case you want to ease backtracking, since backtracking in this method is simply undoing all the recent changes. We must consider the cost of moving children from one node to the other. Clearly the cost of maintaining and updating the parent list is very high, and node merge and its undoing become very expensive in this method. Furthermore, managing two lists in internal nodes makes it a complicated srtucture.

5.2 Passive Representation

This method is mentioned at [20] as UNION/FIND. It tries to avoid the overhead of moving children from one node to another and to ease backtracking. To this end, the abstract machine has a global clock and an array that holds for each clock one bit that indicates whether the action taken at this clock is still valid (backtracking wasn't done yet) or not. This system with a special graph representation causes backtracking to be no more than an invalid clock entry in the clock's array. In that method, a node is actually a collection of nodes. The building block of a node is called basic node. It is a data structure containing two stacks: *uf-next* and *c-next*. Merging two nodes in this approach is done by:

- finding the representative of the two, that is, two basic nodes such that their uf-next points to themselves;
- choosing one of them to be the new representative of the new group, and change their links to indicate this;
- unifying the composite nodes' children recursively.

The following diagram demonstrates the situation:



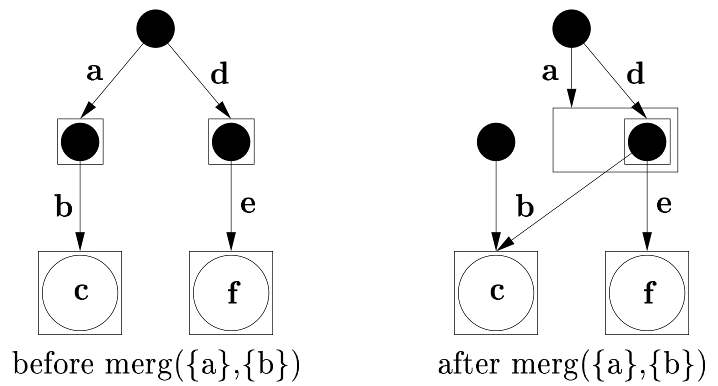
In the diagram, it can be seen that the two basic nodes n_1, n_2 merge into one composite node.

Now in order to find the new node data, (**e.g.,** if it is an internal node or a leaf node) we have to follow the valid uf-next until we find the node representative that carries that information. In this process, we have to check for every link we follow, whether it is still valid, that is, whether we have not yet backtracked from the instructions that created it. This is done by attaching to every link the clock of the action that changes it and pushing this pair onto the node stack.

This way, we can check in the global clock's array whether the link stack is valid or not and if it isn't, we just pop until we get a link with a valid clock. Similarly, when we look for a node's children (the children of all the basic nodes of the composite node) we should follow the c-next reference in a similar way.

5.3 Indirect Representation

This method tries to avoid the large drawback of the direct representation, namely, maintaining a parent list at every node. We solved this problem by introducing a level of indirection between the graph arcs and nodes. A good way to visualize it, is to think of every node as a box with the node inside it. Thus, using this approach an arc refers to a box and all the parents of a node refer to it by its box. The following diagram should help to understend this method, and its implications:



It is important to understand that the box pointed to by {a} on the left side is the same box that is pointed to by {a} on the right side, only the content has changed. Thus, following the path {a} is less efficient than following {d} after the merge, since the box pointed to by {d} is internal to the box pointed to by {a}. The instruction that gets the node inside a box is called `deref`.

Comparing the 3 options I found that:

The big advantage of the passive representation is that during a merge of two nodes the parents of these nodes are not affected. The passive representation of graphs is known to give the illusion of efficient lazy backtracking. The main reason I didn't choose it was that it brutally violates the first two abstract machine principles, namely: keep the commands and the data representation simple. It also turns out that because the final clock structure generated by the approach must be eventually "filtered" to produce a clean FD, the total work that must be expanded is not saved by a "lazy" approach, because eventually, all the work must be performed.

The direct representation has the disadvantage of managing the parent list in every node. This is not so bad when thinking about translating an FD into graph, but while executing the commands genarated by the compiled grammar, most of the commands' effects will eventually cancel in the worst case (this is because the resultant FD is much smaller than the grammar).

As for the indirect representation, it has the advantages that it is the simplest and it doesn't have to manage the parent list explicitly. One of its disadvantage is that it has an hidden factor of delay because of the dereferencing checks. However, this disadvantage appears in the passive merger too (option one).

I decided to adopt the active merger with indirection. For additional information on the comparison between these models, see [26] and [2].

With this architecture in mind, the next stage is to define what the target language for the compiler is. I chose the target language to consist of two kinds of commands. The commands that build the graph and control commands. Control commands are commands that control decision making, backtracking and the recursion on sub-constituents. In the following chapters, we define those commands and their results. Using the FD and the grammar as a starting point for their names and arguments, and using the low level data representation as a starting point for defining their effects. We shall describe our solution in stages describing compilers and their corresponding abstract machines in increasing complexity until a large subset of FUF is covered.

Chapter 6

Unification

We consider in this chapter a very simple language $\mathcal{L}_0$. In this language two entities can be specified: a *program* and a *query*. The semantics of $\mathcal{L}_0$ is to compute the most general FD more specific than both the program and the query. In $\mathcal{L}_0$ we don't have types hierarchy nor do we have special types (such as *given*, *none* ...).

6.1 FD representation

Let us first define an internal representation for FD. we will use a global array called `HEAP`. There are five sorts of data type stores in the `HEAP`:
$\langle$STR,$k\rangle$, $\langle$NEXT,$k\rangle$, $\langle$FEAT,constant$\rangle$,$\langle$TYPE,constant$\rangle$ and $\langle$CHIL,$k\rangle$ each of them has its type and its data. In our algorithms, we shall use the following notations:

- *type_of(*heap cell*)* to get the type of a heap cell.
- *data_of(*heap cell*)* to get the data of a heap cell.
- *type_p(*heap cell*)* to check if the heap cell has the type **type**

In $\mathcal{L}_0$ there are two sorts of structures on the heap: structures that represent a leaf of the FD and compound structures that represent an inner node. Since inner nodes tend to unify among themselves, we shall refer to them indirectly. When representing a leaf on the heap, the following information is required:

- the name of an arc leading to that node,
- and the type of the node.

we shall represent them on the heap as:

- $\langle$FEAT,f$\rangle$,
- and $\langle$TYPE,t$\rangle$.

respectively. For inner nodes, the situation is a little more complicated, because we need to save a reference to its children and we want to refer with a level of indirection, thus, we need:

- The name of an arc leading to that node,
- and reference to the node's children,

We use the following representations:

- $\langle$FEAT,f$\rangle$,
- and $\langle$CHIL,$n\rangle$,

respectively. (where n is a position on the `heap`) Since a large portion of the unification process is searching in an internal node for children, given an attribute we will need an easy way to move from a node to its brother. It is reasonable to add to each node a reference to its brother. We shall use the $\langle$NEXT,$i\rangle$ on `HEAP[`j`]` to indicate that the next brother is at `HEAP[`i`]` and $\langle$NEXT,$i\rangle$ on `HEAP[`i`]` if this is the last child.

Chapter 7

Compiling $\mathcal{L}_0$

A query or program[1] in $\mathcal{L}_0$ is a simple FD, that is, an FD without a path. According to $\mathcal{L}_0$'s semantics the execution of a query is to build an exemplar of the query (FD) on the heap. While doing this, a number of items must be processed:

1. a pair of two atoms;
2. a pair consisting of an atom with a compound FD.

To deal with these cases we define the instruction set $\mathcal{I}_0$ with the following three actions:

1. Push a new pair onto the `heap` (**i.e.,** unify the new pair with the old one if it exists).
2. Push a new str on the `heap` (**i.e.,** move into it if it already exists).
3. Go one level up.

Each of these actions will become a machine instruction, thus

1. `add_leaf` a,b
2. `enter_subgraph` a
3. `leave_subgraph`

As can be seen, the above instructions may extend the heap or may just check if the heap values agree with the instruction arguments. We will also need a global register `H` that will track the heap size for us, a global boolean

[1]in this chapter I will use query for meaning query or program.

register `FAIL` that will have, at the end of the unification, the value `True` or `False`. A global register `CN` that will mark our current postion on the heap and a global stack called `path_stack` to save our path along the heap.

With this information in mind, let us now define some of $\mathcal{M}_0$ instructions and look how they affect the heap. The first instruction we define, deals with the creation of a new structure on the heap and the positioning of the `CN` register on this new structure. For example, this situation occurs when we process the input `((a ((b c)(e f)))` and we want to add to the root of the graph a new internal node with two children. Thus, we first have to create the new structure on the heap and then to change our position in the graph from the root node to the new generated node.

The second instruction is generated when we see a simple pair in the input, for example, (a b), and its purpose is to add a leaf whose arc is *a* and type is *b* to the current node.

We now give an example of the effects of these instructions on the heap. Upon seeing the input:

```
  ((a ((b ((c d)
           (e f))))))
```

The compiler will generate the following instructions:

```
enter_subgraph a
  enter_subgraph b
    add_leaf c, d
    add_leaf e, f
  leave_subgraph
leave_subgraph
```

The heap initial state is:

0	CHIL	0	*

and the current node value is 0.

After executing `enter_subgraph a` the heap will become:

0	CHIL	1	
1	FEAT	a	
2	NEXT	2	
3	CHIL	3	*

and the current node value will become 3. Note that CHIL and NEXT at positions 2,3 refer to themselves. This means that the new structure is the last child of its father and that it has no children at all. After the second `enter_subgraph` command the heap will become:

0	CHIL	1	
1	FEAT	a	
2	NEXT	2	
3	CHIL	4	
4	FEAT	b	
5	NEXT	5	
6	CHIL	6	*

and the current node value will become 6. Note that adding a child is done by connecting the new child to the rear of the children list. It is done because we must scan the children list before adding a new child. The effect of `add_leaf c, d` on the heap is:

0	CHIL	1	
1	FEAT	a	
2	NEXT	2	
3	CHIL	4	
4	FEAT	b	
5	NEXT	6	
6	CHIL	7	*
7	FEAT	c	
8	NEXT	8	
9	TYPE	d	

Again, we added the new node to the rear of the list. The last operation on the list is done by the instruction `add_leaf e, f`. The resultant heap looks like this:

0	CHIL	1	
1	FEAT	a	
2	NEXT	2	
3	CHIL	4	
4	FEAT	b	
5	NEXT	6	
6	CHIL	7	*
7	FEAT	c	
8	NEXT	10	
9	TYPE	d	
10	FEAT	e	
11	NEXT	11	
12	TYPE	f	

The instruction `leave_subgreph` does not have an effect on the heap. It simply changes the current node. Thus, in order to maintain our position in the heap we use in $\mathcal{M}_0$ a stack that tracks our position (the path-stack PSL). Clearly it is `enter_subgraph`'s responsibility to push the new node onto PSL and `leave_subgraph`'s responsibility to pop PSL upon return.

7.1 $\mathcal{M}_0$ commands implementation

$\mathcal{M}_0$ commands are described in terms of primitive heap modification operations and a group of handy macros.

The level of indirection we introduced creates the possibility that reference chains will be formed. Therefore, `dereferencing` is performed by the macro *deref* which, when applied to an address, follows a possible reference chain until it reaches a non **CHIL** cell, which is the address it returns. deref's pseudo code is demonstrated in figure 7.1.1 on the facing page. The macros `CHILP` and `CHIL` test and get the data on the heap respectively.

The next macro is the macro `find_child(feat)` shown in Figure 7.1.2, on page 52. Its input is a register name *REG* and a feature *feat* where *REG* is the address of some internal node on the heap and *feat* is the feature we want to find. The result of this macro invocation is:

- FOUND = *false* if the internal node pointed by `REG` N_1 does not have a child with feat *feat* and `R1` points to the heap place of the last child of N_1.

- FOUND = *true* if the internal node pointed by `REG` N_1 has a child with feat *feat*, and `R1` will have in this case the heap address of this feature

Another macro is the macro `put_new_internal` (Figure 7.1.3, on page 53). Its input is a feature and a heap address a and its effect is to put a new internal node on the heap and to link it to address a with the given feature.

Similarly, we define the macro `put_new_leaf` (Figure 7.1.6, on page 54) that performs the same thing on new leaf nodes.

The last auxiliary macro is the macro `change_to_str` that gets a feature and the address of a node N_1 that used to be a leaf, and changes N_1 into an internal node that points to the new empty leaf N_2 under feature (Figure 7.1.4, on page 53).

The instruction `enter_subgraph` (Figure 7.1.5, on page 53) gets a feature and tries to find the relevant children from the current node `CN`. If this node doesn't exist it creates a new internal node on the heap, links it to the internal node pointed by `CN` and changes `CN` to the child position. If, on the other hand, the requested children are found, there are three cases to be considered:

- The new child is an internal node, in which case we change `CN` to its heap position.
- The new child has type *nil* in which case we just change its type to internal and adjust `CN`.
- The new child is a leaf and has type other than *nil*, in which case the unification fails.

The other instructions, `add_leaf` and `leave_subgraph` are simple and their pseudo-code appears below.

```
deref(a) ≡ while(CHILP(heap[a]))
                a ⟵ data_of(heap[a]);
```

Figure 7.1.1: The macro `deref`

7.2 Compiling $\mathcal{L}_0$ programs

In $\mathcal{L}_0$ a program is just an FD. Thus, compiling a program yields the same commands as if the program was a query. Note, however, that due to the

```
find_child(REG,feat) ≡
{
  R1 ⟵ REG;
  FOUND ⟵ true;
  if(R1 = data_of(heap[R1]))      /* empty of the list */
      FOUND ⟵ false;
  else
      R1 ⟵ data_of(heap[R1]);
      while(data_of(heap[R1]) ≠ feat))
          R1 ⟵ R1 + 1;
          if(R1 = data_of(heap[R1])) /* end of the list */
              FOUND ⟵ false;
              break;
          R1 ⟵ data_of(heap[R1]);
```

Figure 7.1.2: The macro `find_child(feat)` in $\mathcal{M}_0$

simplicity of $\mathcal{L}_0$, the graph represented by the heap image is a tree and not a general graph. Note also that a program in $\mathcal{L}_0$ executing by $\mathcal{M}_0$ should always halt. Thus, an execution of program in $\mathcal{M}_0$ is:

- Ended with the graph image (that is the result FD) is on the heap, or
- fails (as indicated by the value of register `FAIL`)

```
put_new_internal feat , addr a ≡
               HEAP[H + 1] ⟵ ⟨FEAT,feat⟩;
               HEAP[H + 2] ⟵ ⟨NEXT,H + 2⟩;
               HEAP[H + 3] ⟵ ⟨CHIL,H + 3⟩;
               ⟨tag,a⟩ ⟵ HEAP[a];
               heap[a] ⟵ ⟨tag,H + 1⟩;
               push(PSL,H + 3);
               /* push assign the top value to CN */
               H ⟵ CN;
```

Figure 7.1.3: The macro `put_new_internal` *feat*, `addr` *a* in $\mathcal{M}_0$

```
change_to_str feat , addr a ≡
HEAP[a] ⟵ ⟨CHIL,a⟩;
push(PSL,addr);
```

Figure 7.1.4: The macro `change_to_str` *feat*, `addr` *a* in $\mathcal{M}_0$

```
enter_subgraph feat ≡
                     find_child(CN,feat);
                     if(¬FOUND)
                       put_new_internal(feat,R1);
                     else
                         R1 ⟵ R1 + 2;
                         deref(R1);
                       if(type_of(heap[R1]) = CHIL)
                            push(PSL,R1);
                       else
                           if(type_of(heap[R1]) = TYPE and
                             (data_of(heap[R1]) = NIL))
                                change_to_str(feat,R1);
                                CN ⟵ R1;
                           else
                                 fail;
```

Figure 7.1.5: The instruction `enter_subgraph` in $\mathcal{M}_0$

```
put_new_leaf(feat ,type , addr a) ≡
                HEAP[H + 1] ⟵ ⟨FEAT,feat⟩;
                HEAP[H + 2] ⟵ ⟨NEXT,H + 2⟩;
                HEAP[H + 3] ⟵ ⟨TYPE,type⟩;
                ⟨tag,a⟩ ⟵ HEAP[a];
                heap[a] ⟵ ⟨type,H + 1⟩;
```

Figure 7.1.6: The macro put_new_leaf *feat*, *type*, addr *a* in $\mathcal{M}_0$

```
leave_subgraph ≡ CN ⟵ pop(PSL);
```

Figure 7.1.7: The instruction leave_subgraph in $\mathcal{M}_0$

```
add_leaf(feat ,type )≡
                     find_child(T1,feat);
                     if(FAIL = true)
                     then
                         FAIL ⟵ false;
                         put_new_leaf(feat,T1,type);
                     else
                         case H[T1 + 2] of
                             ⟨TYPE,t⟩:
                                 if (t ≠ type)
                                 then
                                     fail;
                                 endif
                             ⟨STR,n⟩:
                                 fail;
                     endif
```

Figure 7.1.8: The instruction add_leaf *feat*, *type* in $\mathcal{M}_0$

Chapter 8

Compiling $\mathcal{L}_1$ Handling Types and Paths

In this chapter, we extend $\mathcal{L}_0$ with two important constructs: *paths* and *types*[4]. The role of paths is to enable data sharing with the result FD in a manner similar to logic variables in Prolog. In this new situation, an FD is a direct graph and not a tree as we had until now. Furthermore, we now have to deal with a situation in which we know that a path a is the same as a path b and we have to record that information on the heap even though there is no value assigned to each of them yet[1].

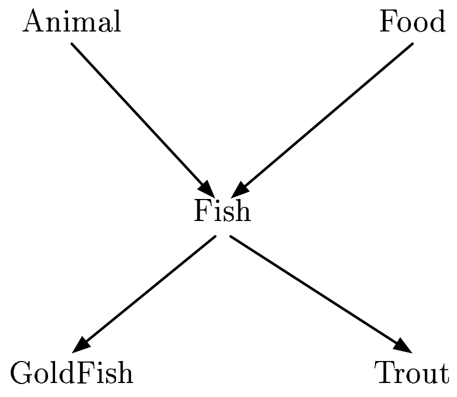
8.1 Handling Types

To ease the task of grammer writing, the pure FUF was enhanced with types[4], that is, a type hierarchy defined by the user and the unification of the graph leaves is taken into acount. For example, consider the following types hierarchy:

- Animal → Fish → GoldFish
- Food → Fish → Trout

This types hierarchy can be represented as the following graph:

[1]The reader can argue that by changing the order of the generated commands this situation can be avoided. Indeed it can be done in $\mathcal{L}_1$, but it will not be possible in a more general language such as $\mathcal{L}_2$



The execution of `((feat Animal)(feat Food))` has to leave on the heap an image of ((feat Fish)). Thus, in $\mathcal{M}_1$, the `add_leaf` ***`feat,type`*** can change the content of an already existing leaf heap cell. Consider the query `((a Animal)(b Food))` and the program `(({a} {b}))`. The semantic of $\mathcal{L}_1$ requires that the type at the end of the path a from the root and the type b be unified at the end of the execution. Furthermore, any further change to one of the paths should be reflected in the other. Clearly, we can solve this problem using the same technique we use to solve the structures unification problem, namely, adding a level of indirection where needed. This method of adding a level of indirection only to a group of nodes that need all to be changed when one of them changes (that is share their value) and not to all the graph nodes is, in fact, an optimization since a similar approach (which I followed in my prototype) is to add a level of indirection to every node and thus achieving a uniform node structure at a cost of heap space and execution time.

8.2 The Meaning of Left-Side Paths

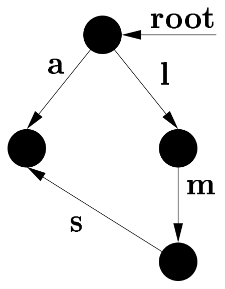
A left-side path appears on the left side of an expression and actually means that, standing at a certain point on the graph(called the current node `CN`) we now wish to go to another part (or branch) of the graph.

The Power of a Left-Side path

Why should we, standing at a certain point n the graph constrain another area in that graph? At this stage, this seems meanigless, but, later on, when we add the possibility of alternatives to our language it will be seen that this possibility gives us the power to choose a certain alternative, depending on what goes on at a different spot on the graph.

8.3 The Meaning of Right-Side Paths

A right side path (described at the right side of the expression) represents an equality of two paths in the graph. For example, the expression: `((a {l m s}))` which is represented by the following graph:



tells us that going from where we stand (the *current node*) to *a* is equivalent to going from the root following *l*,*m* and *s*. The commands created by the compiler in response to a right-side path should see to it that after their execution, the constraint on the graph will be that the above mentioned equality holds.

8.4 Left-Side Paths and the Macros Expander

The paths in $\mathcal{L}_1$ make $\mathcal{L}_1$ very different from $\mathcal{L}_0$. However, in return, we get a very expressive language. Paths are also very challenging to the compiler. As is the case for every complex problem, the most productive approach is to divide it in stages. First, I compile paths to a highly complex machine instruction in order to ease the compilation process and in order to understand better the semantics of paths. Then, in the following machines I break these complex commands into smaller commands suitable for a primitive and efficient machine.

It is greatly simplified to deal only with paths of the form `(a p)` where a is an atom and `p` is a path (unlike the general case where `a` can be a path too). In order to accomplish this, we introduce the macro expander which will expand an expression of the form `(p p)` to an equivalent expression that has the desired syntax (with no paths on the left). It is not hard to see that a pair of the form: `(`$\{p_1 \ldots p_n\}$ $\{p_{n+1} \ldots p_{n+m}\}$`)` can be transformed into: `((`p_1`((`$\ldots$ `((`p_n $\{p_{n+1} \ldots p_{n+m}\}$`))`$\ldots$`))))` Thus, after integrating the macro expander with the parser the compiler will no longer see a pair with a path on the left hand side. Now we can restrict our discussion to paths on the right side of pairs.

8.5 Why Merge Should be Avoided

When compiling paths, we may have to merge existing nodes. For example, when compiling the FD: `((a ((b 1))) (c ((d 2))) (a {c}))`, the compilation of the pair `(a {c})` leads to a merge of two existing nodes.

Merge is a graph operation costly in space and time. The first step of merging two nodes `a` and `b` is matching their children having the same arc label. Thus, the minimal number of operations is lower-bounded by twice the number of children of `a` and `b`. We then have to create a new node containing the children of `a` that are not connected to `a` with arc found in `b`'s arcs and the children of `b` that are not connected to `b` with arc that can be found in `a`'a arcs, followed by recursively merging the pairs we found at the first stage and adding them to the new node as well. Hence, we should avoid merge whenever possible.

8.6 Possible Approaches to Avoiding Merge

At first, it seems that we can avoid merge by reordering the sub FD. For instance the FD `((a ((a b)))(b ((b c))) (a {b}))` can be rearranged as `((a {b})(a ((a b)))(b ((b c))))`, thus eliminating the need for a merge operation. However, this solution cannot be adopted for two important reasons:

- Given: the given meta-fd has one interpretation (the "un-clean" one) which makes it depend on the order of features. For example, ((a 1) (a given)) would be ok under this interpretation but ((a given) (a 1)) would fail.
- Alternatives: although changing the orders of branches in an alt does not change the set of all possible answers accepted by a grammar, it does change the order in which answers are produced. Since applications rely on this order to encode the linguistic notion of "default" and "markedness", the compiler is not free to change the order of branches within alts.

The approach we follow instead is based on distinguishing between two different operations during the node creation process. The first is a preliminary check of the existing end of a path. The second is the creation of the child or the graph depending on the outcome of the preliminary check. Let us consider the program `((c ((d {g h}))))`. At the time the commands generated by the program fragment `(d {g h})` execute there can be four different situations to be considered.

8.7 Four Cases when Handling a Path

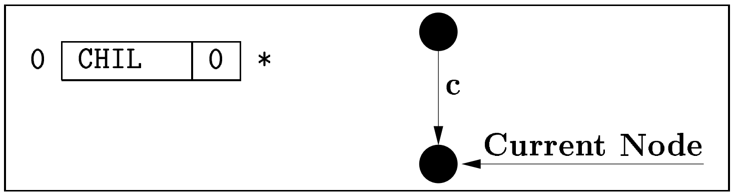
Case one: There is no child with feature `d` neither does the path {`g h`} exist on the heap;

Case two: The path {`g h`} doesn't exist on the heap, but the child d does;

Case three: The child `d` doesn't exist on the heap, but the path {`g h`} does, or

Case four: Both the path and the child exist on the heap.

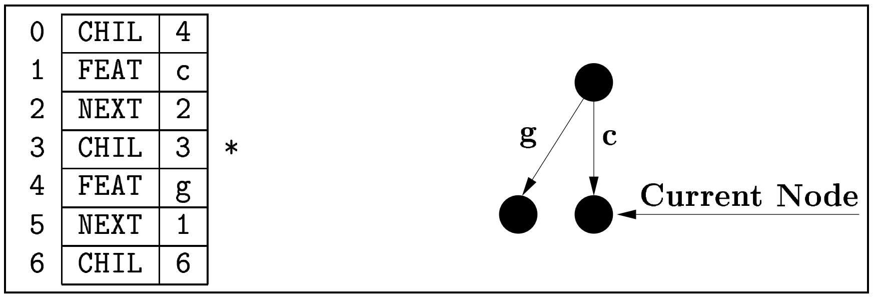
Case one In this case, the preliminary check will show that there is no child with feature `h` to the path {`g`} on the graph. Thus, we will create the child `d` from the current node if it does not exit yet. Only then will we create the arc between the end of path and the child `d` of the current node and label it with `h`. Consider the FD `((c ((d {g h}))))`



0	CHIL	0	*

Graph and heap representation before executing `(d {g h})`

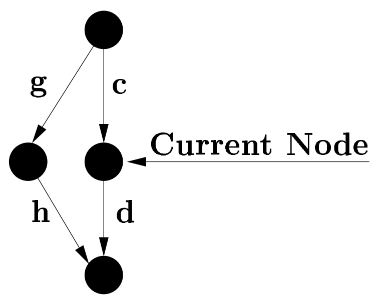
At this time the preliminary check is done for the path p. Since it finds that there is no such path, it creates the path, except for the last node, as shown in the diagram below.



0	CHIL	4	
1	FEAT	c	
2	NEXT	2	
3	CHIL	3	*
4	FEAT	g	
5	NEXT	1	
6	CHIL	6	

It turns now to create a child d to the current node, followed by connecting the node created last (during the preliminary check) and the new child – `d`, with an arc labeled `h`. See the diagram below.

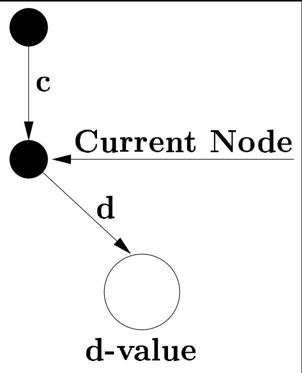
0	CHIL	4	
1	FEAT	c	
2	NEXT	2	
3	CHIL	7	*
4	FEAT	g	
5	NEXT	1	
6	CHIL	10	
7	FEAT	d	
8	NEXT	8	
9	CHIL	9	
10	FEAT	h	
11	NEXT	11	
12	NODE	9	



Notice the indirection: `heap[12]` directs us to `heap[9]`. We do that by creating a children heap cell that refers to itself (see line 9 in the above diagram).

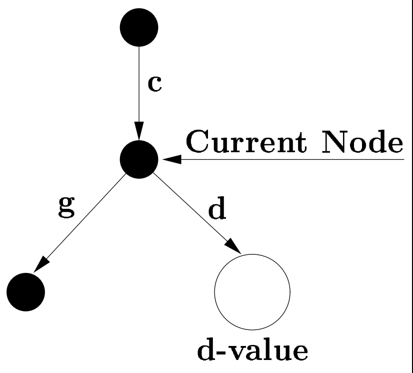
Case Two Let us take the example `((c ((d d-value)))(c d {^ g h}))`

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	4	*
4	FEAT	d	
5	NEXT	5	
6	TYPE	d-value	



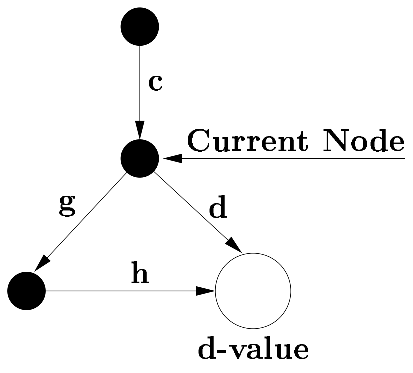
In this case the preliminary check will show that the end of the path `h` doesn't exist. The machine then turns to create the child `d` but it already exists as shown in the next figure.

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	7	*
4	FEAT	d	
5	NEXT	5	
6	TYPE	d-value	
7	FEAT	g	
8	NEXT	4	
9	CHIL	9	



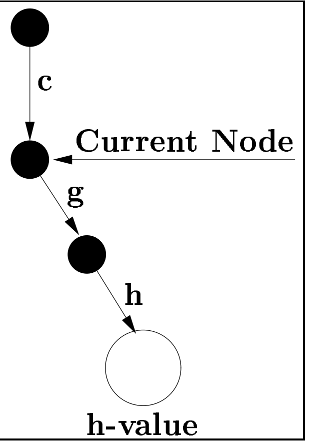
The only thing remaining is to connect the end of the path to the child *d*. This is illustrated in the next figure:

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	7	*
4	FEAT	d	
5	NEXT	5	
6	TYPE	d-value	
7	FEAT	g	
8	NEXT	4	
9	CHIL	10	
10	FEAT	h	
11	NEXT	11	
12	NODE	6	



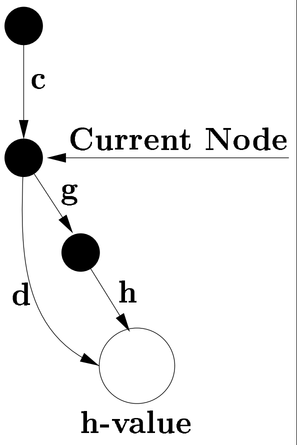
Case Three An example of this case is the input `((c ((g ((h h-value)))))({c d} {^ g h})))`. As in the previous cases we first preform the preliminary check (the check for the existance of the path {`g h`}) and find it exists in this FD. Thus, the graph looks like this:

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	4	*
4	FEAT	g	
5	NEXT	5	
6	CHIL	7	
7	FEAT	h	
8	NEXT	8	
9	TYPE	h-value	



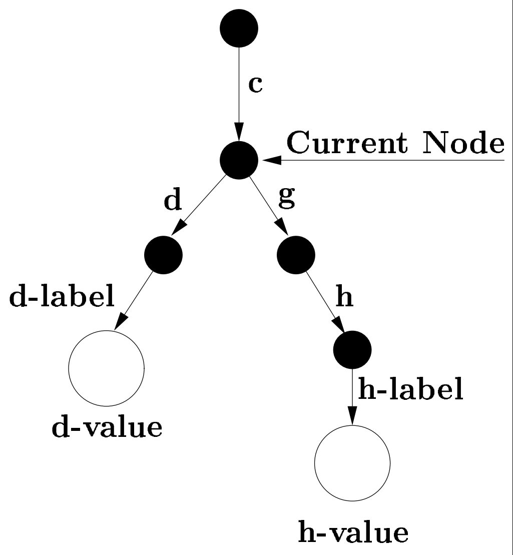
Now, since the preliminary check found that the node exits we must check to see if the current node child exists in order to distinguish between the third and the fourth cases. In our case this node doesn't exist and hence the machine just adds a link from the current node to the newly found node (the end of the path) and labels it `d` (see next figure)

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	10	*
4	FEAT	g	
5	NEXT	5	
6	CHIL	7	
7	FEAT	h	
8	NEXT	8	
9	TYPE	h-value	
10	FEAT	d	
11	NEXT	4	
12	NODE	9	



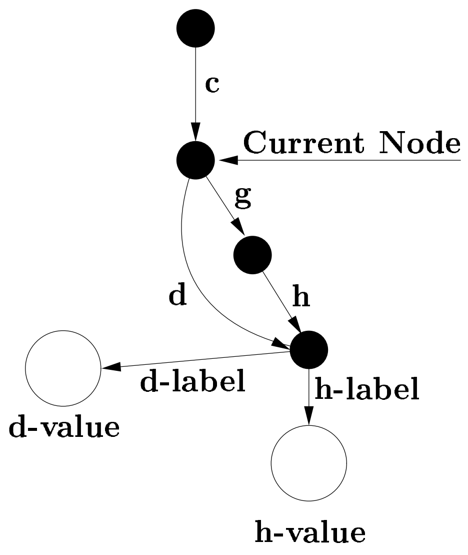
Case Four The last case is the most interesting because if both the path and the atom already exist on the heap before executing our program, we cannot avoid the merge operation. In this case we find the two nodes and merge them into a single node, if possible. Notice that this action will fail if the nodes are not mergeable. Let us demonstrate this idea using the example:
`((c ((g ((h ((h-label h-value)))))))(c ((d ((d-label d-value)))) (c d {^ g h}))`

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	4	*
4	FEAT	g	
5	NEXT	13	
6	CHIL	7	
7	FEAT	h	
8	NEXT	8	
9	CHIL	10	
10	FEAT	h-label	
11	NEXT	11	
12	TYPE	h-value	
13	FEAT	d	
14	NEXT	14	
15	CHIL	16	
16	FEAT	d-label	
17	NEXT	17	
18	TYPE	d-value	



After performing the preliminary check and finding that the path already exists we turn to check whether `d` exists too. Since this is our case we then have to merge the two nodes. The result is shown in the next figure.

0	CHIL	1	
1	FEAT	c	
2	NEXT	2	
3	CHIL	4	*
4	FEAT	g	
5	NEXT	13	
6	CHIL	7	
7	FEAT	h	
8	NEXT	8	
9	CHIL	10	
10	FEAT	h-label	
11	NEXT	16	
12	TYPE	h-value	
13	FEAT	d	
14	NEXT	14	
15	NODE	9	
16	FEAT	d-label	
17	NEXT	17	
18	TYPE	d-value	



8.8 Handling Paths by the Compiler

As mentioned previously, there are two basic kinds of operations needed when creating paths. One checks whether a certain child already exists on the graph and the other creates a new child to a given node. Different combination of those two basic operations are used by the compiler to create the code for the graph path-making, in an effort to minimize the execution time.

Additional features that were added to the machine and that will assist the compiler are labels, unconditional jumps and conditional jumps (explained later). Bear in mind that at compilation time, the compiler is unaware of the graph on which the path instruction code will be executed. Hence, the code issued by the compiler has to be designed to handle each of the different situations on page 59.

While creating a path in a graph it is possible that at a certain stage in the graph creation the path extends the graph. From this point on, it is unnecessary and inefficient to perform the check stage, it is time-saving to go straight ahead with creating the remaining path components.

In order to achieve that, the machine has to "remember" whether it extended the graph on a certain path or not. The naive solution seems to be having a special register set when the path first extends the graph, and defining the commands to perform the check only when this register is not set. There seems no reason why this shouldn't work, but a large drawback depicted when considering that paths may be arbitrary long, costing in execution time spent for repeatedly testing the special register.

Since being out of the graph (by extending the path beyond it) is not reversible[2] the idea of recording the event of extending the graph in the program code itself arises. This can be done using the program counter to record the same information recorded previously using a special register. This idea is illustrated in detail next. Consider the program `((t {l m s}))` the compiler will generate the following code for the part {l m} in its path:

```
set_relative_jmp 3;
try_path "l" ;
try_path "m" ;
goto l1: ;
enter_path "l" ;
enter_path "m" ;
l1:
```

It can be seen that there are two parallel sets of commands. The first set checks whether certain path components exist on the graph. The second set creates the new components to the graph without performing any preliminary checks.

The distance in the code segment between each pair, for example `try_path l` and `enter_path l` is constant, and equal to the length of the path. `set_relative_jmp` serves the role of informing the machine about this constant. The instruction `try_path l` first checks, as mentioned above, whether a child having arc label `l` exists on the graph. Depending on the result, it either commits itself to the new child found, or increments the program counter by the value set by `set_relative_jmp` in the case the child isn't found. The instruction `enter_path` creates a new node on the graph without performing any preliminary check. The idea of relative jump at the ISA level is mentioned in [28]. Following the complete compiled code for the program `((t {l m s}))`:

```
start_path_from_top ;
set_relative_jmp 3;
try_path "l" ;
try_path "m" ;
```

[2]This is because paths cannot go back after they start going forward, **e.g.,** there is no path like that {a b ˆ c d}.

```
goto l1: ;
enter_path "l" ;
enter_path "m" ;
goto l2: ;
l1:
try_long_path "s" ;
jf l2: ;
try_short_path "t" ;
jf l3: ;
merg_path ;
goto l4: ;
l3:
link_short_to_long "t";
goto l4: ;
l2:
enter_short_path "t" ;
link_long_to_short "s" ;
l4:
```

A few more specialized commands will be explained in order to fully follow the example

- The command `start_path_from_top` is responsible for telling the machine that the following commands refer to an absolute path (a path starting from the root).
- `try_long_path a` checks whether the end of the path exists, and if it does not it sets the `jump flag` to `true`.
- `jf label` is a command having a label as an input. It checks the `jump flag`, and if its value is `true` it jumps to that label.
- `try_short_path a` checks whether a child connected to the current node by an arc labeled a. If it doesn't finds this child it then sets the `jump flag` to `true`, otherwise, it sets the `jump flag` to `true`.
- `link_short_to_long a` assumes the case where the path exists but the child does not (case 3) and hence it adds the end of the path as a child to the current node and labels the arc connecting them with `a`.
- `merge_path` assumes that both ends existed on the graph and, hence, tries to merge them.

- `link_long_to_short a` assumes the case 2, that is, the child on the left exists but the end of the path does not. It adds the child on the left as a child of the last node in the path and labels the connecting arc with `a`.

- `enter_short_path a` receives a label and ensures that the child on the left exists, by creating it if needed.

Now, let us demonstrate how the compiled code works for each of the four different cases mentioned.

- Case one: Since neither the path nor the left child `t` exist on the graph, the second set of commands which create the path, excluding its last element {`e m`}, will be performed. Next, `goto l2:` will be executed with the result of jumping to the last two commands – `enter_short_path t` which creates the child `t` and `link_long_to_short s` which, as explained above, will add `t` as a child with arc label `s` to the node at the end of the path.

- Case two: since in this case, too, the path doesn't exist, the code will create it, except for the last node. Then, there will be a jump to `l2:`. With the only difference being that `enter_short_path t` this time will only verify the existence of `t` and will not have to create it.

- Case three: since the path already exists, the `try_path` commands are all executed leading to execution of `goto l1:`. At `l1:` the command `try_long_path s`, which checks for the existence of the end of the path, is executed with result of setting the `jump flag` to `false` (having found s on the graph) and consequently continuing without jumping, with `try_short_path t`. Since the child `t` doesn't exist in this case, the `jump flag` is set to `true` and the control is transferred to `l3:`. At `l3:` the end of the path is added as a child to the current end of the path, terminating with an unconditional jump to the end of the path sequence commands (`l4:`).

- Case four: This is the last case in which both the child at the left and the path do exist. Thus, the set of `try_path` commands are fully executed, and the control transferred to `l1:`. The commands `try_short_path a` and `try_long_path f` both set the `jump flag` to `false`, with the result of not jumping and arriving to `merg_path`.

8.9 Save path and restore path

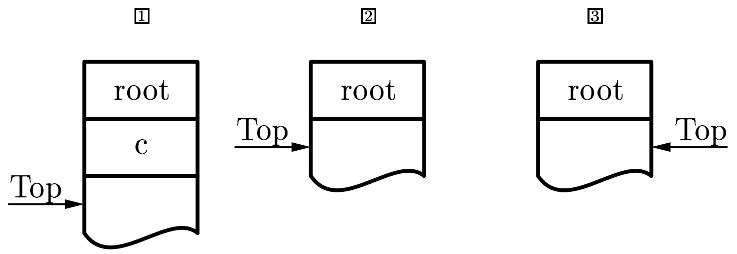
Since all left-side paths are transformed by the macro expander into ordinary FDs and thus translated by the compiler into $\mathcal{M}_0$ commands, executing the generated code can changes the state of the machine by changing the machine location on the graph (the machine `path_stack` array and the content of the current node register, `CN`). Since the meaning of the left-side path is, actually, to make a pause, enforce constraints on a different spot in the graph, and then come back to the place where we left, we must verify that the changes caused by executing those commands will be temporary. That is to say, that after the end of the execution, the machine state will be restored. Hence, in addition to the translation of the left-side path into $\mathcal{M}_0$ commands, we need to add two new commands, one at the beginning and one at the end of the new generated commands.

The first of these commands instructs the machine to save every element of $\mathcal{D}_1$ which might be changed as a result of executing the commands formed by the transformation of the left-side path. The second is a command that restores the state of the saved elements of $\mathcal{D}_1$ to their previous values (the values saved).

More specifically, two different situations must be considered.

- A relative left-side path encountered;
- and an absolute left-side path is encountered.

To illustrate the need for such saving, we will give an example, consider the following $\mathcal{L}_1$ program: `((c (({e r} ((a b)))(l m))))` after the parsing and macro expanding the compiler will be fed as input the parse tree corresponding to the following program:
`((c`$_{\boxed{1}}$ `(({}`$_{\boxed{2}}$ `((e ((r ((a b)))))))`$_{\boxed{3}}$ `(l m))))` where {} correspond to the root of the graph. The `stack_paths` corresponding to each of the stages are:



Clearly if we continue with the stack at $\boxed{3}$ we will get incorrect results since the last node `(l m)` will be added at the path {`root`} instead of the path {`root c`}

8.9.1 An Absolute Path

In order to recover $\mathcal{M}_1$'s state prior to the execution of an absolute left-side path (a left-side path starting form the root), we need to save the existing *path_stack* from the very top of the stack to a certain depth that the compiler decided. The command `save_all_path n` serves this purpose. It takes one arguments which tells it how many elements down the stack should be saved (recorded). It is the compiler's responsibility to minimize that number. The command `restore_all_path` restores the path stack to its previous state.

8.9.2 A Relative Path

In this case, in order to reconstruct the graph's state prior to the execution of the left-side path, the compiler must guide the machine to record a segment in the `path_stack` array: the beginning of the segment is at some place above the current location in the `path_stack` and its end is somewhere between this point and the current point on the `stack_path`. Thus, the appropriate command needs to receive two arguments. The first tells it where to start from (how many elements above the current node should the saving start from). The second argument tells it how many terms down must be saved and again this number is found at compilation time. Thus we have the command `save_path i j` for this purpose.

How does the compiler set the values for `save_path i j` ?

The first value is specified according to the number of climbs in relative path that is the number of climbs we must do in the `stack_path` above the current node. The second value is determined by the depth of the expression on the right and the path elements that are going down, but this value is also limited to the size of the first argument since we don't have to save elements that don't belong to the `stack_path` nodes below `CN`.

8.9.3 An example

The program `((c (({^ e r} ((a b)))(l m))))` will result in the following $\mathcal{M}_2$ code:

```
enter_subgraph  "c" ;
save_path 1 1 ;
enter_subgraph  "e" ;
enter_subgraph  "r" ;
```

```
add_leaf "a" "b" ;
restore_path 1 ;
add_leaf "l" "m" ;
up ;
end_p:
```

The second example is an absolute path example:
`((c (({e r} ((a b)))(l m))))`

```
enter_subgraph  "c" ;
save_all_path 2 ;
enter_subgraph  "e" ;
enter_subgraph  "r" ;
add_leaf "a" "b" ;
restore_all_path ;
add_leaf "l" "m" ;
up ;
end_p:
```

A somewhat more complex example is:
`((a ((b ((c (({^ 2 l} (({^ e} ((l m)({bgu} {cs}))))))))))))`

```
enter_subgraph  "a" ;
enter_subgraph  "b" ;
enter_subgraph  "c" ;
save_path 2 1 ;
enter_subgraph  "l" ;
save_path 1 1 ;
enter_subgraph  "e" ;
add_leaf "l" "m" ;
save_all_path 0 ;
start_path_from_top ;
try_long_path "cs" ;
jf l1: ;
try_short_path "bgu" ;
jf l2: ;
merg_path ;
goto l3: ;
l2:
link_short_to_long "bgu";
goto l3: ;
l1:
```

```
enter_short_path "bgu" ;
link_long_to_short "cs" ;
l3:
restore_all_path ;
restore_path 1 ;
restore_path 1 ;
leave_subgraph 3 ;
end_p:
```

8.10 $\mathcal{M}_1$ commands implementation

We now add an abstraction to the action of assigning a new value to a heap cell and instead of writing `HEAP[H+1]` $\longleftarrow$`⟨feat,type⟩`, we will use the term `BIND(HEAP + H,⟨feat,type⟩)`. This is for reasons that will be explained when dealing with backtracking. We will use this abstraction only when assigning a value to an old heap cell and not when creating a fresh cell.

```
start_relative_path_from (int n) ≡
  PATH_LONG_CN ⟵ path_stack[top_pathstack - n];
  PATH_SHORT_CN ⟵ CN;
```

Figure 8.10.1: The command `start_relative_path_from n` in $\mathcal{M}_1$

```
start_path_from_top () ≡
  PATH_LONG_CN ⟵ 0;
  PATH_SHORT_CN ⟵ CN;
```

Figure 8.10.2: The command `start_path_from_top` in $\mathcal{M}_1$

```
set_relative_jmp (int n) ≡
  PATH_RELJMP ⟵ n;
  FLAG ⟵ False;
```

Figure 8.10.3: The command `set_relative_jmp` in $\mathcal{M}_1$

```
enter_path(int feat) ≡
  put_new_internal_at_front_long(feat);
```

Figure 8.10.4: The command enter_path in $\mathcal{M}_1$

```
try_path (symbol feat) ≡
  R1 ⟵ PATH_LONG_CN;
  FIND_CHILD(R1,feat);
  if(¬FOUND)
      pc ⟵ pc + PATH_RELJMP;
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      if(type_of(HEAP[R1]) = CHIL)
          PATH_LONG_CN ⟵ R1;
      else
          if(type_of(HEAP[R1]) = TYPE)
              R2 ⟵ data_of(HEAP[R1]);
              switch (R2)
                 default: FAIL;
                 case NIL:
                   /*  change <type,nil> to <str,addr> */
                   change_to_str(feat,R1);
                   PATH_LONG_CN ⟵ R1;
                   break;
                 case ANY:
                   change_to_any_str(feat,R1);
                   PATH_LONG_CN ⟵ R1;
                   break;
            else
              FAIL;
```

Figure 8.10.5: The command try_path in $\mathcal{M}_1$

```
try_long_path(int feat) ≡
  R1 ⟵ PATH_LONG_CN;
  FIND_CHILD(R1,feat);
  if(¬FOUND)
      FLAG ⟵ True;
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      PATH_LONG_CN ⟵ R1;
      FLAG ⟵ False;
```

Figure 8.10.6: The command `try_long_path` in $\mathcal{M}_1$

```
int try_short_path(int feat) ≡
  R1 ⟵ PATH_SHORT_CN;
  FIND_CHILD(R1,feat);
  if(¬FOUND)
      FLAG ⟵ True;
  else
      R1 gassign R1 + 2;
      DEREF(R1);
      PATH_SHORT_CN ⟵ R1;
      FLAG ⟵ False;
```

Figure 8.10.7: The command `try_short_path` in $\mathcal{M}_1$

```
enter_short_path(int feat) ≡
  R1 ⟵ PATH_SHORT_CN;
  FIND_CHILD(R1,feat);
  if(¬FOUND)
    put_new_internal_short(feat);
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      PATH_SHORT_CN ⟵ R1;
```

Figure 8.10.8: The command `enter_short_path` in $\mathcal{M}_1$

```
jf(int address) ≡
  if(FLAG = True)
      pc ⟵ prog + address;
```

Figure 8.10.9: The command `jf` in $\mathcal{M}_1$

```
link_short_to_long(int feat) ≡
  HEAP[H+1] ⟵ ⟨feat,feat⟩ ;
  if(data_of(HEAP[PATH_SHORT_CN]) = PATH_SHORT_CN)
    HEAP[H+2] ⟵ ⟨next,H+2⟩;
  else
    HEAP[H+2] ⟵ ⟨next,data_of(HEAP[PATH_SHORT_CN])⟩;
  HEAP[H+3] ⟵ ⟨node,PATH_SHORT_CN⟩;
  ⟨tag,_⟩ ⟵ HEAP[PATH_SHORT_CN];
  BIND(HEAP + PATH_SHORT_CN,⟨tag,H + 1⟩);
  H ⟵ H + 3;
```

Figure 8.10.10: The command `link_short_to_long` in $\mathcal{M}_1$

```
link_long_to_short(int feat) ≡
  HEAP[H+1] ⟵ ⟨feat,feat⟩;
  if(data_of(HEAP[PATH_LONG_CN]) = PATH_LONG_CN)
    HEAP[H+2] ⟵ ⟨next,H+2⟩;
  else
    HEAP[H+2] ⟵ ⟨next,data_of(HEAP[PATH_LONG_CN])⟩;
  HEAP[H+3] ⟵ ⟨node,PATH_SHORT_CN⟩;
  ⟨tag,_⟩ ⟵ HEAP[PATH_LONG_CN];
  BIND(HEAP + PATH_LONG_CN,⟨tag,H + 1⟩);
  H ⟵ H + 3;
```

Figure 8.10.11: The command `link_long_to_short` in $\mathcal{M}_1$

```
merge(int heap-addr1 , int heap-addr2 ) ≡
  SAVESP ⟵ top;
  if (R1 = R2)
      return;
  tmp1 ⟵ heap-addr1; tmp2 ⟵ heap-addr2;
  PUSH(MERGE,R1); PUSH(MERGE,R2);
  while(top ≠ SAVESP)
      begin
      R2 ⟵ POP(MERGE); R1 ⟵ POP(MERGE);
      if(R1 = R2)
          continue;
      if(type_of(HEAP[R1]) = type ∧ type_of(HEAP[R2]) = type)
          t1 ⟵ data_of(HEAP[R1]);
          t2 ⟵ data_of(HEAP[R2]);
          R3 ⟵ unification_table[t1][t2];
          if(R3)
              if(R3 = type_of(HEAP[R1]))
                  continue;
              else
                  BIND(HEAP + R1,⟨type,R3⟩);
                  continue;
          else
              top ⟵ SAVESP;
              FAIL;
      else
```

Figure 8.10.12: The command merge in $\mathcal{M}_1$

```
        if(TYPEP(HEAP[R1]))
            FAIL;
        else
            if(TYPEP(HEAP[R2]))
                FAIL;
            else
                 INSERT_TO_MERGE_TABLE(R1,R3);
                 R3 ⟵ R1;
                 while(R2 != data_of(HEAP[R2]))
                     R2  ⟵ data_of(HEAP[R2]);
                     R1 ⟵ merge_table[data_of(HEAP[R2])];
                     if(R1)
                         MERGE_PUSH(R1);
                         R1 ⟵ R2;
                         FEAT_TO_NODE(R1);
                         MERGE_PUSH(R1);
                         R2 ⟵ R2 + 1;
                         continue;
                     else
                         R1 ⟵ R2;
                         R2 ⟵ R2 + 1;
                         /* The next field of this node */
                         if(R2 = data_of(HEAP[R2]))
                             BIND(HEAP + R3,⟨next,R1⟩);
                             break;
                         else
                             HEAP[H + 1] ⟵ HEAP[R2];
                             BIND(HEAP + R3,⟨next,R1⟩));
                             BIND(HEAP + R2,⟨next,R2⟩);
                             R2 ⟵ H + 1;
                             R3 ⟵ R1 + 1;
                             continue;
  end /* while */
BIND(HEAP + tmp2,⟨node,tmp1⟩);
```

Figure 8.10.13: The command `merge` in $\mathcal{M}_1$ (continue)

Chapter 9

$\mathcal{L}_2$ adding alternatives and recursion

In this chapter we will extend $\mathcal{L}_1$ by adding two important features, alternatives and recursion. The addition of alternatives creates for the first time a distinction between the input and the program, the latter representing the grammar. An alternative is a choice-point, representing a possible choice among several options. In this chapter, we will deal with the questions: what commands will the compiler create in order to help $\mathcal{M}_2$ handle alternatives and how does the machine execute these commands. In order to answer those questions, let us try to improve our understanding of the alternatives and of grammars containing alternatives.

```
((b 2)
 (c ((alt (((l 1))
           ((t 1))))))
 (alt (((m 1))
       ((f 2))))
 (e 1))
```

Figure 9.0.14: grammar g1

The grammar can be represented as an ond-or tree (see [15]), with circle nodes representing and-nodes and square nodes representing or-nodes. To every and-node a list of commands associated with it is attached. Every or-node has its children, which must be tried consecutively until one of them succeeds.

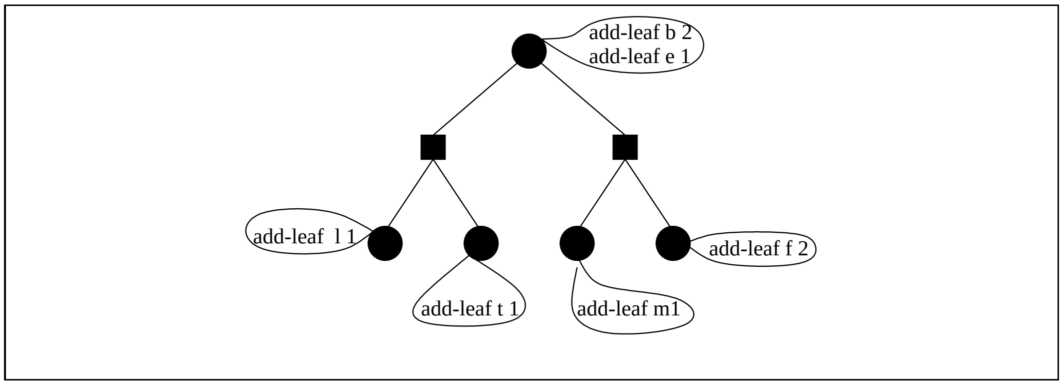


Figure 9.0.15: Grammar g1 as an and-or tree

9.1 How Unification Between Input and Grammar Can Be Done

An easy way to perform unification between input and grammar is to first create a graph representing the input in $\mathcal{M}_2$ memory and then try to execute the grammar commands on the same graph starting form the root. In this way, the result will be a graph obeying the grammar and the input constraints. In order to understand what are the grammar commands, we will look closer on the and-or tree model formed earlier. In this model, a computation of the grammar can be done by following four rules:

- The first node is the tree's root.
- When processing an and-node, first perform its commands and then process all its children going from left to right.
- When processing an or-node, try to process its children from left to right until one of them succeeds.
- If processing fails at any stage, return to the last successfully visited or-node, undo all the graph changes done after that or-node, reconstruct your position in the graph and continue to try its others children. (those that weren't tried before)

It should be noticed that when backtracking is performed, the state of the graph when that or-node was first visited should be recalled along with some of the machine state, for example the machine position in the graph[1]. Looking more closely on this computation model we discern that:

[1]That is because the point of the failure in the graph doesn't have to be the point we were at the time we first tried that or-node.

- It doesn't specify how to reconstruct the original state in the graph and how to recover the graph at backtrack time.
- While performing the unification according to this method, the number of times the question which is the next node to be visited is not bounded, in other words every time we fail we must do a computation in order to find where to go next.

```
((alt(((a 1))((a 2))))
 (alt(((a 3))((a 2)))))
```

Figure 9.1.1: grammar g2

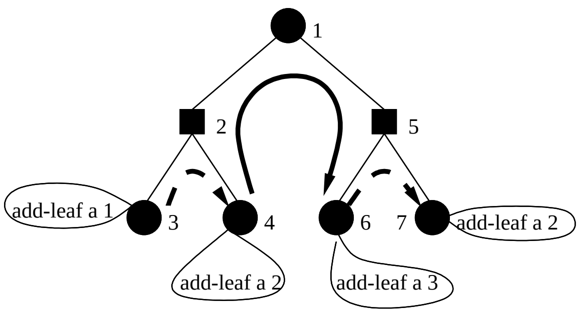


Figure 9.1.2: grammar g2 as and-or tree

Note that in the case of Figure 9.1.2 in case of failure at node 3 the next node is node 4 (dashed arrow) and in case of success at node 3 the next node is 6 which is the next node in case of success to every child of 2. This example demonstrates that the next node to be visited is not necessarily a brother of the node we are at, in fact it doesn't even have to be on the same level on the and-or tree.

9.2 How to avoid repeated computation of the next node

It can be seen from Figure 9.1.2 that we can actually determine the following nodes in case of success or failure for most of the nodes, in advance. Hence, it follows that there is no need to perform this computation at run time. The next visited node in the case of success or in the case of failure for an and-node[2] seems to be determined by the structure of the grammar itself and except for the special case where a failure occurred at the last child of an or-node it should not be affected by runtime parameters.

Let us define for each and-node, a function which carries out the commands correlated with it. This function should transfer the control to the address at the top of the control stack if one of its commands fails. Similarly, we will define for each or-node a function which pushes the fail address in a stack and then calls its first or-function. If the or-function succeeds, it returns to its caller, otherwise, the control transfers to the next and-function.

We will now explain this idea in more detail using an example: 9.2.2

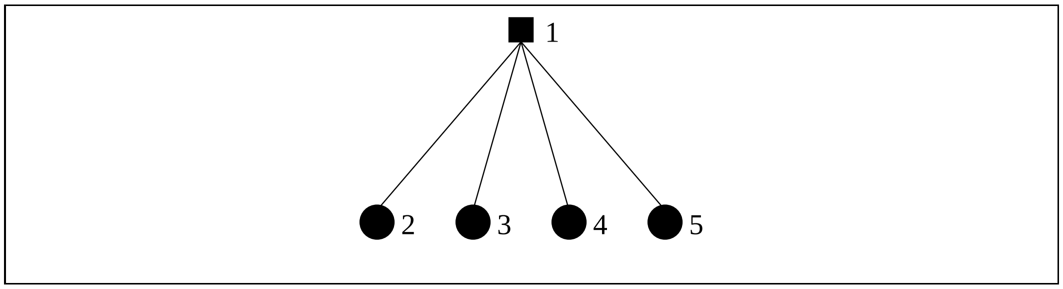


Figure 9.2.1: An Alternative With Four Children

From the example, we can see that three cases must be considered:

- Call to the first and-node of an or-node: we push the failure address (the destination to take in case of failure) of the first node (that is, the address of the second node of the or-node) on the stack and then execute the first and-node.

- Call to a middle and-node: instead of pushing again the failure address of the next node, we simply update the address on top of the stack to the new address of failure (the address of the next and-node under the same or-node). We also need to restore the graph to its previous state (this is done by the undo command).

- Call to the last and-node: we pop the address of failure from the stack and restore the graph *before* trying the last branch. This, in fact, gives us an effect similar to the last call optimization in [2]

[2]The next visited node is determined by the success or failure at the and-nodes, since or-nodes do not include commands and serve only to call upon the and-node

```
proc or1
  push l1:;
  call and2;
  goto end_or1:
l1:
  undo;
  stack_top = l2:;
  call and3;
  goto end_or1:
l2:
  undo;
  stack_top = l3:;
  call and4;
  goto end_or1:
l3:
  undo;
  pop;
  call and4;
  goto end_or1:
end_or1:
endproc;
```

Figure 9.2.2: The code for Figure 9.2.1

The main point of this optimization is that the compiler can statically determine the address of failure within an or-node except for the case of the last and-node; the success-continuation address can always be determined statically.

If all the children failed, the proc *or1* will transfer control to the address that was on top of the control stack prior to its invocation. This method has the following advantages:

- The flow of control in case of success is encoded into the code itself, that is, it is the natural linear flow of control.
- The flow of control in the case of failure is computed by the compiler and pushed at runtime into the control stack, thus, its computation costs one pop of the control stack.
- Before the last branch, the machine will pop the control stack, and thus achieve two advantages: the control stack will be as compact as

possible, and the failure address will always be the real target in case of failure (not an indirect address containing a pop and goto instructions to the address of failure).

In order to deal with program failure, the compiler must add an implicit alt around the program, that is, when given a program p the compiler should compile the program `((alt (p fail-proc)))` where fail proc is a procedure that issues a fail message and halts the machine.

9.3 Adding Recursion on the Sub-Constituents

In this section, we examine how to integrate recursion on the sub-constituents into a program p without recursion. For this purpose we define $\mathcal{M}_2$ with the following structures:

- A stack named cset-stack which serves the role of saving the address of the sub-constituents not yet processed.
- A command `find_all_cset` which searches and finds all the sub-constituents that are children of the current node and pushes them onto the cset-stack.
- A command `get_next_cset address` which gets an address as an argument and jumps to this address if the cset-stack is empty, otherwise pops the cset-stack into the current-node.

We also make `find_all_cset` mark on the heap every sub-constituent it found in order to prevent repetitions. Figure 9.3.1 demonstrates how the compiler uses the above features to generate a program that recurses on the sub-constituents given a program p which doesn't.

9.4 Further Abstraction of the Compiled Code

To prepare for the correct handling of undo during backtracking, we introduce the following abstractions in the instruction set of $\mathcal{M}_2$:

- `try_me_else` *address* $\equiv$
  `push_control` *address*.

```
init fail:;
start:
get_next_cset success:;

code for p

find_all_cset;
goto start:;
success:;
success_proc;
fail:;
fail_proc;
```

Figure 9.3.1:

- `retry_me_else` *address* ≡
  `undo`;
  `push_control` *address*.
- `trust_me` ≡
  `undo`;
  `pop_control`.
- `leave_alt` *address* ≡
  `goto` *address*.

To avoid the cost of function call, we inline all the functions generated by the compiler (the and-functions and the or-functions). To this end, we must modify the instruction `fail` in a way that instead of halting the machine, it now transfers the control to the address on the top of the control-stack, that is, `fail` ≡ `goto stack_top`. The result code can be seen on Figure 9.4.1, on page 96

9.5 The Trail and Backtracking

As seen in Section 9.1, the state of the graph should be recovered upon backtracking. In this section, we elaborate on how this can be done efficiently.

As mentioned earlier, the address of the continuation of the computation in the case of failure was recorded in a choice point. Now, we wish to add to the choice point information that will help us restore the graph to its previous state. For this purpose we introduce:

- A stack named `trail` is added to $\mathcal{M}_2$, where every entry in the `trail` is a pair (address, cell value).
- The address of the top of the `trail` is kept in a register `t`.
- The structure of a choice point is broadened to include the value of `t` at the choice point creation time, in addition to the address of the fail continuation.

Note: The choice point is created as a result of `alt` expressions in the grammar, by the command `try_me_else fail_address`.

In order to make the backtracking process more efficient and to minimize the length of the trail, we noticed that there is actually no need to reconstruct the graph beyond the point in the heap that was active at the creation time of the choice point. That is, any node allocated after the choice point has been activated does not need to be "trailed".

To implement this optimization, we introduce a register `save_h` which keeps the value of the heap register `h` at the time of choice point creation, and further broaden the choice point to encompass the value of the heap register `h`[3].

To maintain the `trail`, let us alter the command `bind` which updates a cell in the heap (see Section 8.10) so that, prior to changing the heap, it first updates the trail if necessary. That is, if the address `bind` received is above the value of the register `save_h`, the update is not trailed.

To avoid the cost of this verification, we must ensure that the use of `bind` can be avoided in all cases where it is clear that the value of the address on the heap to be changed is always larger than the value of the register `save_h`. For example, see Figure 9.5.1 on page 97 for the command `put_new_internal`. Note that we use `bind` only for changing the connection between the father and the new child.

9.5.1 Undoing

The undoing process is executed simply by going through the trail, from its top down to the value recorded on the choice point and restoring all values passed on that path – a value for each address. In addition to that, it must be remembered that the value of the register `h` has to be restored to the value saved at the choice point creation time.

[3]we use the register `save_h` for efficiency reasons since its value should be always consistent with the value of `h` as it is kept on the last choice point

9.5.2 Restoring the Machine State While Backtracking

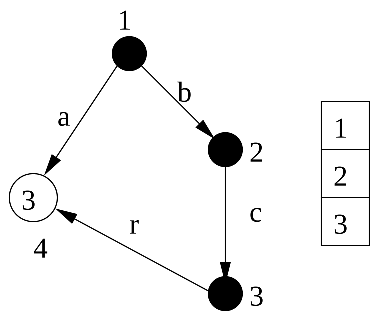
As seen in the previous section, the `trail` enables us to restore the graph state. In addition to the graph's state, we must also restore parts of the machine's state during the backtracking. Thus, among others, the current node (`cn`) or the place where we were while creating the choice point, and the path we followed to reach this place, specified by the `path_stack` must be restored as well.

9.5.3 Why Restoration of the `path_stack` Is Needed ?

It may not be obvious why the register `path_stack` must be protected upon backtracking. The best way to demonstrate this point is by using an example. Let us look at the program for the following grammar:

```
(⁰({a} {b c r})
 (b ((c ((alt (((r 3)¹)
               (³(r {^ 1}))
               (e g)))))))
 (a 2)²))
```

We intend to look into three states of the machine while this program is executing, marked with boxed $\boxed{1}$, $\boxed{2}$, $\boxed{3}$.

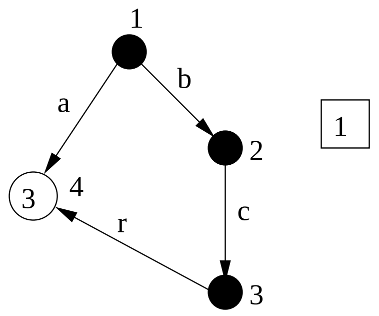


The graph and `path_stack` state at $\boxed{1}$

The previous diagram represents the respective states of the graph and the `path_stack`[4] at state $\boxed{1}$. The `path_stack` contains the nodes 1, 2 and 3, while the current node is 3.

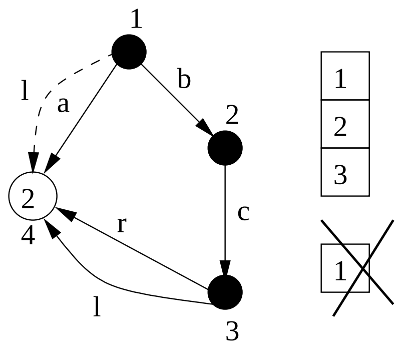
At state $\boxed{2}$, execution fails, and at the time of failure the `stack_path` contains only the node 1. This is demonstrated in the next diagram.

[4]in real life the heap address of this node will be on the `path_stack`.



the graph and `path_stack` state at $\boxed{2}$

After failing, backtracking is performed, and the graph is restored to its previous state using the `trail`. Now the second branch of the alternative must be tried, that is trying to add the child `(r {^ l})`, doing this without restoring the `path_stack` will result in the incorrect dashed arc shown in the next diagram:



The graph and `path_stack` state at $\boxed{3}$

Thus, we have shown the reason for the need to restore the path stack at backtracking time. In the next paragraph, we will see how to maintain this information efficiently.

9.5.4 How To Record the `path_stack`

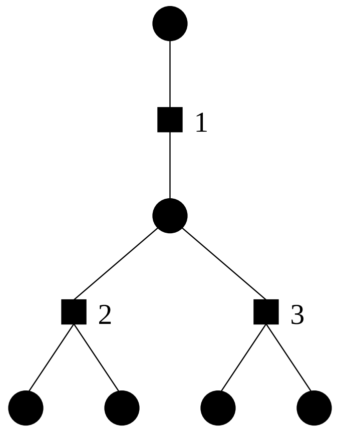
The naive approach would be to record the entire `path_stack` at every choice point on a stack of stacks. While it is correct, this approach falls short of our goals because of two major drawbacks. Since it causes $\mathcal{M}_2$ to record the same information many times, it takes up a lot of memory space and it takes a long time to save the information and retrieve it. Considering those drawbacks, we took the approach of recording the `stack_path` incrementally, that is, saving only part of the information at the choice point. While backtracking, we have to gather and associate the information taken from a chain of choice points.

9.5.5 How To Identify the Choice Points Needed to Reconstruct the `path_stack` ?

As mentioned above, since restoring the `path_stack` is incremental and involves gathering information from a set of choice points, we have to develop a way of recognizing those choice points that, combined, will give us the full information about the `path_stack`. In order to understand the solution to this problem, let us first look at the structure of a typical program and its matching and-or tree and define a relation on its alts. The program and its companion and-or tree are shown below:

```
((a ((b ((alt (((c ((d ((alt (((a 1))
                              ((a 2))))
                         (alt (((b 1))
                              ((b 2)))))))))
                 ((m e))))))))))
```

Definition: child alternative – an alternative is a child of another alternative if, lexically speaking, the child appears within another alternative (to be called the father), provided that there is no other (third) alternative appearing within the father and containing lexically the child.



The And-Or Tree for the Previous Program.

In the and-or tree diagram above, it can be easily seen that 2 and 3 are children of 1.

9.5.6 Recovering the `path_stack`

During backtracking, we wish to reconstruct the `path_stack` trough the choice points, starting from the choice point we are at that moment (the choice point corresponding to the innermost alternative we're in) and ending at the choice point at the head of the `control_stack` (the choice point corresponding to the alt we are about to give another try), provided that all the choice points we are about to traverse are father-child couples.

While traversing the father-child path the `path_stack` is reconstructed accordingly and so is the `dynamic_stack`[5]. In order to reconstruct the path according to the description given above[6], we must know:

- How to recognize the choice point in which failure occurred, that is the choice point we are in, and
- Were to go down the path, that is given a choice point find its children and decide who will lead us to the choice point at the top of the `control_stack`

The solutions[7] are, respectively those:

- Introduce a stack named `dynamic_stack` in which a pointer to the current choice point is kept all the time. The current choice point is the choice point that corresponds to the alternative we are in at this time. Thus, the `dynamic_stack` is changed by the commands:
  - `leave_alt` ***`argument`*** which pops the `dynamic_stack` *argument* time.
  - `trust_me`, which pops once the `dynamic_stack`.
  - `try_me_else` ***`addr`***, which pushes the new choice point onto the `dynamic_stack`.
- Hold in every choice point a pointer pointing to its father, this is the choice point at the top of the `dynamic_stack` at the child choice point creation time.

Thus a choice point structure includes:

```
path stack element number 1
.
.
```

[5]this stack will be explained in the following pages.

[6]we must reconstruct the `path_stack` moving on the choice point in the order specified, that is from father to child and not from child to its father, although the second approach is simpler to implement.

[7]this problem is very similar to the problem of implementing dynamic-wind in language with call/cc, (see [12])

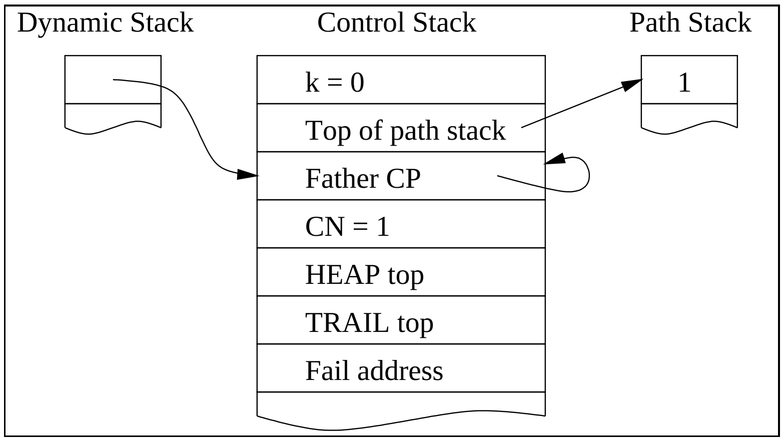
```
   .
   path stack element number k
   k
   top of path_stack
   pointer to father choice point
  current node (cn)
  trail pointer
  fail address
```

Having provided the answers to the above mentioned difficulties, we now describe the reconstruction process.

9.5.7 How Reconstruction is Done

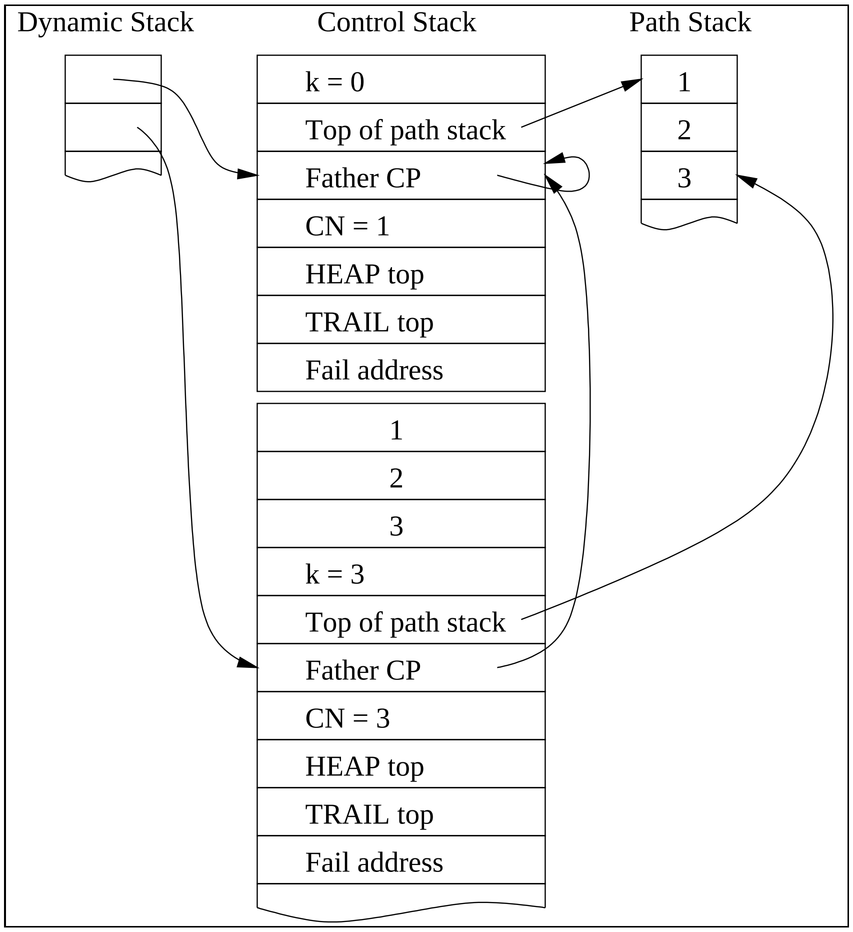
Reconstruction is performed in two consecutive steps. First, going upward in the `control_stack` from the choice point at the head of the stack, through the father pointers, until reaching the choice point at the head of the `dynamic_stack`. During that journey, we push a reference to every choice point we have found onto a stack[8] until reaching the choice point at the head of the dynamic stack, that is, the current choice point.

Next, going down to the bottom using the path recorded at the first step, we reconstruct the `path_stack` and the `dynamic_stack` as we descend. The next three diagrams show $\mathcal{M}_2$ stacks at various states and correspond to the program shown in Section 9.5.3

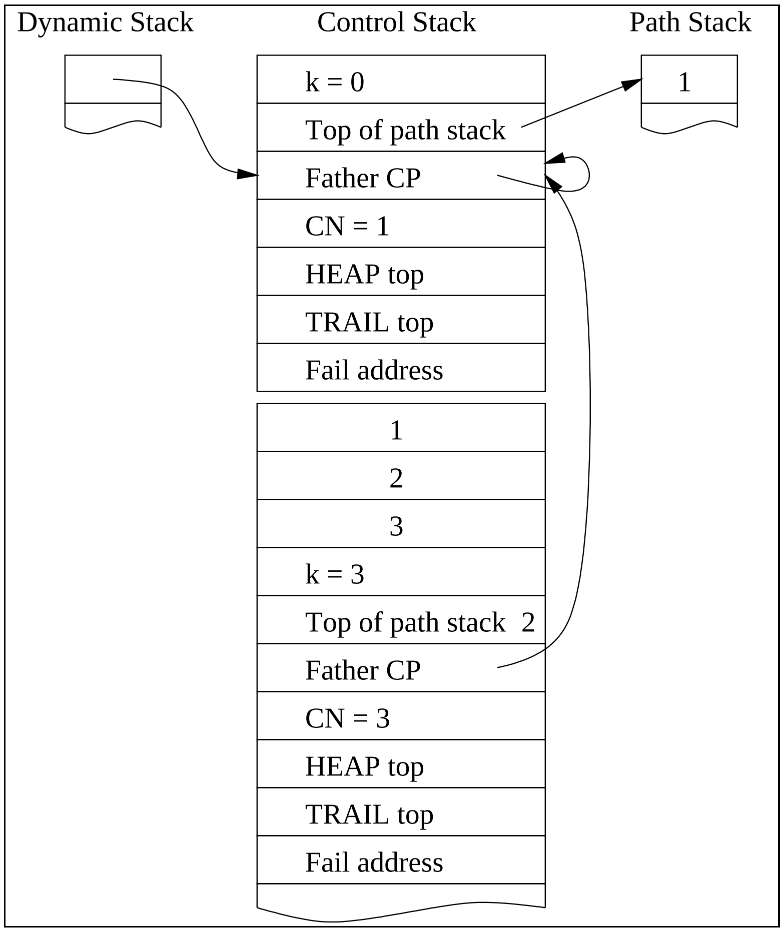


$\mathcal{M}_2$ stacks at $\boxed{0}$ of the program at 9.5.3

[8]We can use any of the machine free stacks, for example the `save_path` stack.



$\mathcal{M}_2$ stacks at $\boxed{1}$ of the program at 9.5.3

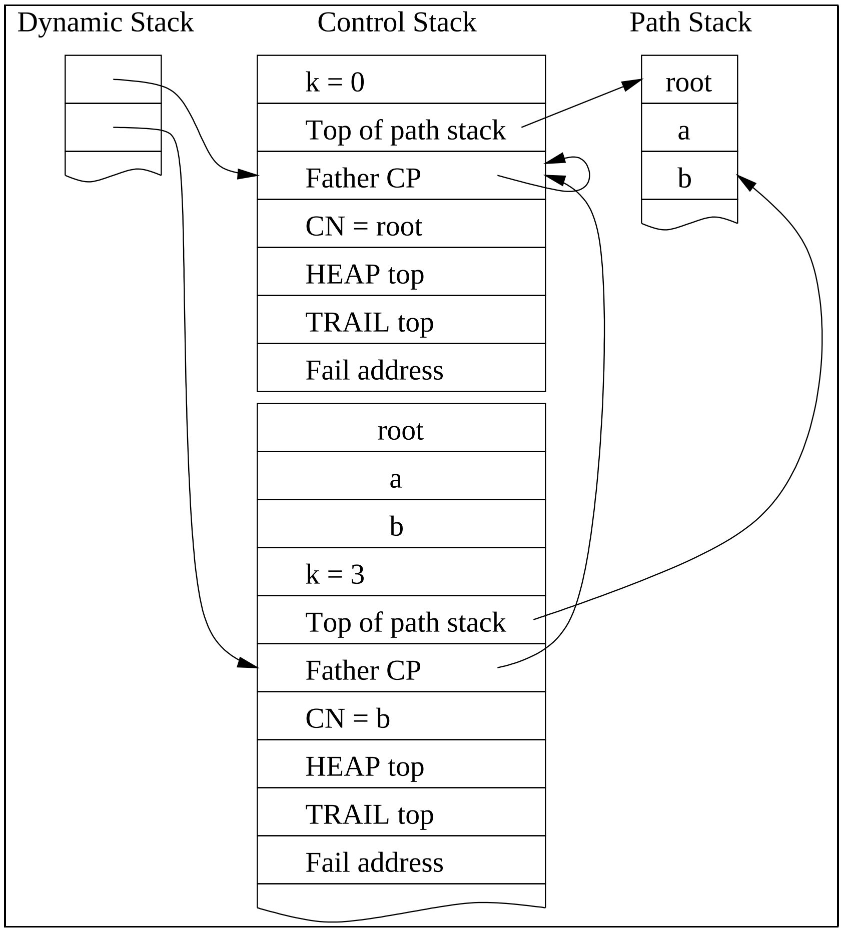


$\mathcal{M}_2$ stacks at $\boxed{3}$ of the program at 9.5.3

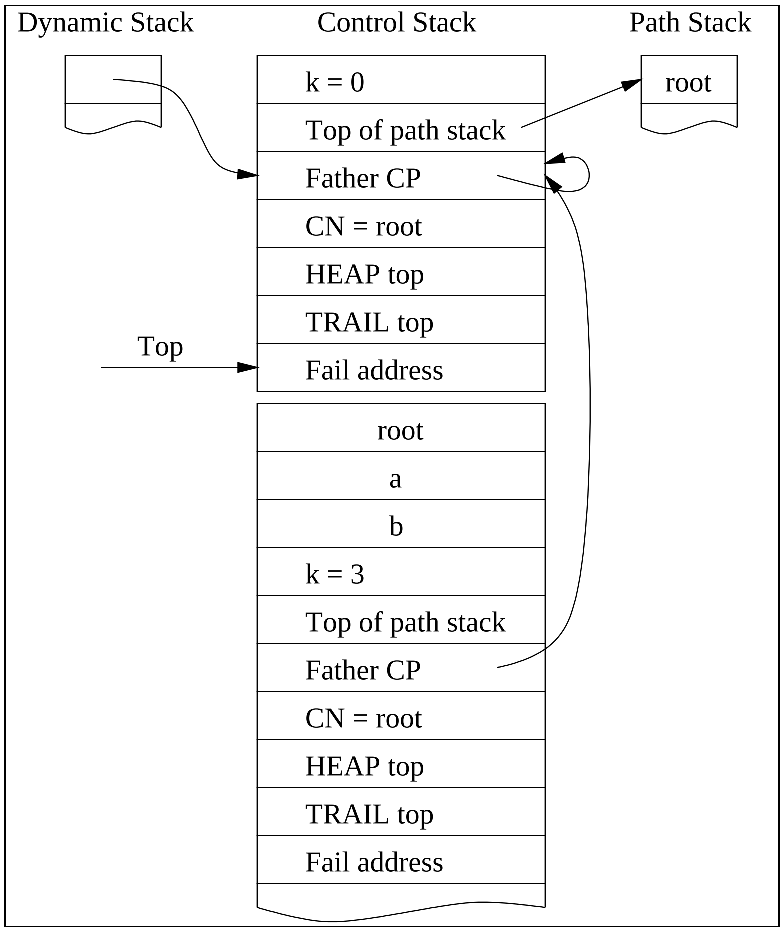
Finally, a special case to be considered is the case of an alternative having in turn an alternative as its last branch. The problem which arises is this: since prior to arriving at the last branch the command `trust_me` is executed, the father choice point is then removed form the `control_stack`. Thus, reconstructing the `path_stack` for its children will no longer be possible. The following program and its companion compiled code (figure 9.5.2, on page 98) demonstrates this point:

```
((a ((b ((alt (((a 1)$_{\boxed{0}}$)
               ($_{\boxed{1}}$(c ((d ((alt ($_{\boxed{2}}$((b 1))
                                  ((b 2)))))))))))))))
 ({a b c a} 2)))
```

The line in the compiled code corresponding to the $\boxed{1}$ is 10. At the time of its execution, the choice point of the father alt (enclosing alt) is removed from the `control_stack`. The machine stacks are shown next at $\boxed{0}$ and $\boxed{1}$ respectively:

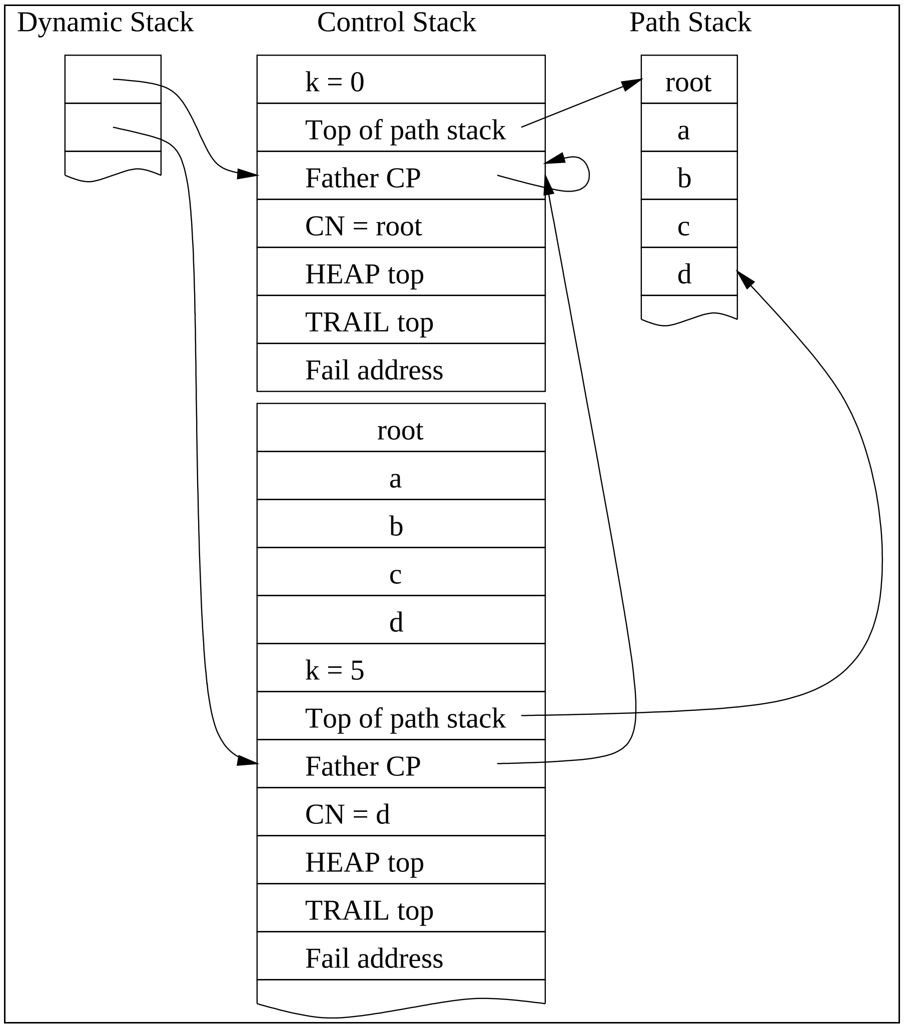


$\mathcal{M}_2$ stacks at $\boxed{0}$



$\mathcal{M}_2$ stacks at $\boxed{0}$

The solution to this problem is to add a new command to $\mathcal{M}_2$, which exploits the fact that the command `trust_me` didn't erase the father choice point from the `control_stack` but only moved the head of the stack, and the data of the father choice point was not overwritten by another choice point. The new command is named `in_tail_try_me_else`. It receives an additional argument (compared with the arguments of `try_me_else`) which tells $\mathcal{M}_2$ the size of the path stack portion saved at the father choice point. The command then forms a new choice point that attaches the father portion of the path stack to its own choice point. The next diagram shows this last stage.



$\mathcal{M}_2$ stacks at $\boxed{2}$

9.5.8 Can We Avoid the Use of the `dynamic_stack` ?

Apparently, the use of the `dynamic_stack` is redundant, since we can have the compiler add another argument to the command `try_me_else`. This argument serves to denote which is the father alternative at any given time. The compiler can determine this argument because the father definition is a lexical definition and can be determined at compile time.

But it turns out that the `dynamic_stack` is essential, after all, since knowing who is the father alternative doesn't tell us where is its corresponding choice point, and where it can be found on the `control_stack`. This can only be found at run-time due to the command `trust_me` that pops a choice point out of the stack when we are still in its last branch. If we hadn't introduced the last call optimization, the situation would have been different. The tradeoff is to decide between last call optimization and maintenance of the dynamic stack. We decided that the benefits of last call optimization would override the cost of the stack maintenance.

Further investigation, however, showed that the whole approach of storing the `path_stack` – with all the corresponding inefficiencies – could be avoided altogether. We explore this optimization route in the next chapter.

```
#(define p3
#  '((alt (((a ((b ((alt (((a c))
#                          ((e b))
#                          ((r t)))))))))
#          ((m ((c ((r d)))))))))
.program
.labels 8
init fail:;
start:
get_next_cset success: ;
try_me_else l2:;
enter_subgraph  "a" ;
enter_subgraph  "b" ;
try_me_else l4:;
add_leaf "a" "c" ;
leave_subgraph 2 ;
leave_alt end_p:;
l4:
retry_me_else l5: ;
add_leaf "e" "b" ;
leave_subgraph 2 ;
leave_alt end_p:;
l5:
trust_me ;
add_leaf "r" "t" ;
leave_subgraph 2 ;
leave_alt end_p:;
l3:
leave_subgraph 2 ;
leave_alt end_p:;
l2:
trust_me ;
enter_subgraph  "m" ;
enter_subgraph  "c" ;
add_leaf "r" "d" ;
leave_subgraph 2 ;
leave_alt end_p: 0 ;
end_p:
find_all_cset ;
goto start: ;
success:
success_proc ;
fail:
fail_proc ;
```

Figure 9.4.1: Compiled Code With Abstractions

```
int put_new_internal(int feat_value,int addr) ≡
  heap[h+1] ⟵ ⟨feat,feat_value⟩;
  heap[h+2] ⟵ ⟨next,h + 2⟩;
  heap[h+3] ⟵ ⟨chil,h + 3⟩;
  ⟨tag,_⟩ ⟵ heap[addr];
  bind(heap + addr,⟨tag,h + 1⟩);
  push(path,h + 3);
  h ⟵ cn;
```

Figure 9.5.1: The Use of `bind` in `put_new_internal`

```
init fail:;
start:
get_next_cset success: ;
enter_subgraph  "a" ;
enter_subgraph  "b" ;
try_me_else l2: ;
add_leaf "a" "1" ;
leave_alt l1: 1 ;
l2:
trust_me ;
enter_subgraph  "c" ;
enter_subgraph  "d" ;
in_tail_try_me_else l5: 2 ;
add_leaf "b" "1" ;
leave_subgraph 2 ;
leave_alt l1: 0 ;
l5:
trust_me ;
add_leaf "b" "2" ;
leave_subgraph 2 ;
leave_alt l1: 0 ;
l4:
leave_subgraph 2 ;
leave_alt l1: 0 ;
l1:
leave_subgraph 2 ;
save_all_path 3 ;
enter_subgraph  "a" ;
enter_subgraph  "b" ;
enter_subgraph  "c" ;
add_leaf "a" "2" ;
restore_all_path ;
end_p:
find_all_cset ;
goto start: ;
success:
success ;
fail:
fail ;
```

Figure 9.5.2: compile code sample

Chapter 10

$\mathcal{M}_3$ – A More Efficient Approach

Or transferring computations from the machine to the compiler

The most disturbing aspect of the solution given to the backtracking problem in $\mathcal{M}_2$ is the high cost of the backtracking, due to the need to reconstruct the entire `path_stack`. As mentioned in the previous chapter, the use of the incremental approach was substantially more efficient than the naive approach (*i.e.,* to reconstruct the whole stack path at each choice point). But we were still unsatisfied with the overall result.

Additional problems associated with that approach were the need to hold and manage the `dynamic_stack`, the relatively costly creation of choice points and the non-constant size of the choice point.

The last two attributes arose since a certain part of the `path_stack` had to be recorded in the choice point. But as said above, reconstructing the `path_stack` during backtracking was the most serious problem, since we aspired to make the backtracking as cheap as possible. We were even ready to consider raising the cost paid during the creation of the choice point as a trade-off for lowering the cost during backtracking in order to improve the runtime for the bad cases.

With this in mind, we contemplated and tried to think what essentially we use the `path_stack` for. The `path_stack` is necessary to process relative paths in the grammar. Specifically, we discerned that there are two cases where the `path_stack` is used:

- when executing the command `leave_subgragph`: the stack is popped.

- for the path commands

First Case A `leave_subgraph` command is followed by further commands which extend the graph. These commands must be executed at the same location we were in before the corresponding `enter_subgraph` was performed. For this purpose, we have to leave the subgraph we were in, in order to continue the execution of the following commands.

Second Case It should be noted that every path has two parts: a rising part (possibly empty) and a descending part. The rising part is the first, and its semantics is to go up **the same way you went down**. This is important because the node we're in may have several fathers. It was necessary to record the path we came down through in the `path_stack`, in order to know where to ascend and start executing the descending part.

The key to finding a good compilation solution for this problem is the observation that the characteristics of the nodes where we need to ascend from is a lexical characteristic, that is, looking at an FD segment and a path within it, one can determine precisely where the ascending part of the path refers in the FD.

`((a ((b ((c ((l m))))))`
` `$\boxed{1}$`({l f} {a b})`
` (l ((f ((c (`$\boxed{2}$`({^ r} t)))))))`

Figure 10.0.3: The Lexical Characteristic of Paths

An example for this path property is given in Figure 10.0.3. Note that the path at $\boxed{1}$ is not interesting, since it is an absolute path: we know that we must ascend to the root of the graph. When one looks at $\boxed{2}$, we can easily see that the path at that point should ascend to {l f} and not to {a b}. Because of the lexical property of the position of the raising part of paths, the compiler can know for each position in the graph whether it is needed for a path as its raising point. When a node is the target of a raising path, the compiler issues a command to save that node in a memory area for that use while going down. Later, when the path ascending to this location is encountered, the compiler instructs the machine to start the descending part from the position saved before.

As for the first case, too, the compiler can determine at compile time which are the points were the present location should be recorded, and to which points among the recorded nodes we should return.

Hence, if only we thought of a way to enable the compiler to instruct $\mathcal{M}_3$ to save and restore certain positions, we could get rid of the need to use `path_stack` in $\mathcal{M}_3$. This would give rise to a number of advantages:

1. the dynamic_stack will not be needed anymore,
2. the path_stack will be also redundant,
3. the process of creating a choice point will become much simpler,
4. the size of a choice point will become fixed,
5. the backtracking process will become much more efficient, since the reconstruction of the right place in the graph needs only involve saving the position of the alternative, and reconstructing it (as compared to saving a chunk of the stack path),
6. and finally the command enter_subgraph becomes cheaper since the need to push the cn on the path_stack is no longer needed.

In order to implement this approach, we endow $\mathcal{M}_3$ with a new memory area called the environment, and add a few more commands to $\mathcal{M}_3$ enter_node, save, restore and restore_path_start.

We will explain their effects using the following examples:

```
((a ((b ((c ((d e)
             (f g)))))
     (h i))))

save 0;
enter_node "a" ;
save 1;
enter_node "b" ;
enter_node "c" ;
add_leaf "d" "e" ;
add_leaf "f" "g" ;
restore 1;
add_leaf "h" "i" ;
restore 0;
```

The machine instruction save i instructs the machine to save the current node at cell i of the environment. The new instruction enter_node is different from the old instruction enter_subgraph in that it no more pushes the cn onto the control_stack and the instruction restore i loads the register cn with the value at the environment cell i. The next example shows how the compiler deals with paths:

```
(({c d e f} {^ ^ g h})))

save 0;
enter_node "c" ;
enter_node "d" ;
save 1;
enter_node "e" ;
restore_path_start 1;
set_relative_jmp 2;
try_path "g" ;
goto l5: ;
enter_path "g" ;
goto l6: ;
l5:
try_long_path "h" ;
jf l6: ;
try_short_path "f" ;
jf l7: ;
merg_path ;
restore 0;
goto l8: ;
l7:
link_short_to_long "f";
restore 0;
goto l8: ;
l6:
enter_short_path "f" ;
link_long_to_short "h" ;
restore 0;
l8:
```

In this example, we can see that at the position {`c d`} the compiler instructs $\mathcal{M}_3$ to save the content of `cn` at `environment[1]`. Later on, when the time comes to access the path, it will then issue the command `restore_path_start 1` to tell the machine that the next path will start its descent from `environment[1]`.

The last example is a full program, and here we have one more hazard to deal with, namely the recursion on sub-constituents. The problem is that unification with a sub-constituent can change the `environment` and prohibit us from executing backtracking.

One solution is to record the change done to the `environment` on the `trail`. This solution, although it would work, would make every `save` instruction more expensive. We adopt, therefore, another solution, in which all reference to the `environment` is done relatively to some other `base` register. Now, all we have to do is to advance the value of `base` at the time any new recursion starts. This change must be recorded on the `heap` so that it can be undone upon backtracking. To manipulate the `base` register, $\mathcal{M}_3$ includes the command `assign env i`.

The environment must also be manipulated each time a branch of an alternative is entered. The following principles apply:

- All the branches of an alternatives start to use the `environment` from the same location
- The alternative branches use the top of the environment, that is, the environment is manipulated in a stack-like discipline as branches are traversed according to the static organization of alts in the grammar. (The stack is pre-computed by the compiler and environment access is translated into direct access into a linear array.)

```
((a ((b ((alt (((alt (((c d)) ((e ((r ((d f))) (m c)))) ((g h)))))
              ((1 2))))))
     (at ((end ((it ((is {^ ^ getting complicated}))))))))))

.labels 12
.code 58
init fail:;
start:
get_next_cset success: ;
save 0;
enter_node "a" ;
save 2;
enter_node "b" ;
try_me_else l10: ;
try_me_else l13: ;
add_leaf "c" "d" ;
restore 2;
assign env 3;
goto l9: ;
l13:
retry_me_else l14: ;
enter_node "e" ;
save 3;
```

```
enter_node "r" ;
add_leaf "d" "f" ;
restore 3;
add_leaf "m" "c" ;
restore 2;
assign env 4;
goto l9: ;
l14:
trust_me ;
add_leaf "g" "h" ;
restore 2;
assign env 3;
l12:
goto l9: ;
l10:
trust_me ;
add_leaf "1" "2" ;
restore 2;
assign env 3;
l9:
enter_node "at" ;
enter_node "end" ;
save 1;
enter_node "it" ;
restore_path_start 1;
set_relative_jmp 2;
try_path "getting" ;
goto l16: ;
enter_path "getting" ;
goto l17: ;
l16:
try_long_path "complicated" ;
jf l17: ;
try_short_path "is" ;
jf l18: ;
merg_path ;
restore 0;
goto end_p: ;
l18:
link_short_to_long "is";
restore 0;
```

```
goto end_p: ;
l17:
enter_short_path "is" ;
link_long_to_short "complicated" ;
restore 0;
end_p:
find_all_cset ;
goto start: ;
success:
success ;
fail:
fail ;
```

With this new representation, we can simplify the commands `try_me_else label`, `retry_me` and `trust_me`. This is because of the change in the choice point structure. In $\mathcal{M}_3$ a choice point consists of:

- `cn`,
- `h`,
- `trail_pointer`,
- and fail address.

The size of a choice point is now constant, whereas in $\mathcal{M}_2$ it was variable as it stored a stack. As a consequence, the commands manipulating a choice point do not need to have the extra-argument of the choice point size.

```
try_me_else (address addr) ≡
  push(cn);
  push(h);
  push(t);
  push(addr);

retry_me_else (address addr) ≡
  push(addr);
  untrail(stack[sp - 1]);
  h ⟵ (stack[sp - 2]);
  cn ⟵ (stack[sp - 3]);

trust_me ≡
```

```
untrail(stack[sp]);
h ⟵ (stack[sp - 1]);
cn ⟵ (stack[sp - 2]);
sp ⟵ sp - 3;
```

10.1 compiling $\mathcal{L}_2$ for $\mathcal{M}_3$

```
((a ((b ((c ((d e)
             (f g)))
         ({^ ^ m v } w)))
     (l d))))
```

Figure 10.1.1: an example FD

In this section we explain how the compiler issues the commands `save` and `restore`.

Instead of creating the code directly from the parse tree, as was the case for $\mathcal{M}_1$ and $\mathcal{M}_2$, we first go over the parse tree and add information to it. The parse tree has two types of nodes: leaves and internal nodes. Going over the parse tree, we attach two data structures (called position) to each node. Those positions, denoted with hexagons in Figure 10.1.3, represent positions in the graph. One of the positions (the one linked to the left side of the tree's node) represents the node's position in the graph, while the other represents the position from which the computation should continue after the generated code corresponding to the sub-graph rooted at that node is completed. The position to be reached is denoted in Figure 10.1.3 with an arrowed broken line, and the node position is denoted by a finer doted line.

The function that performs the above has tree arguments: first, the parse tree, second, a stack full of positions, that actually imitates $\mathcal{M}_2$'s `path_stack`; third a position that is the position in the graph that should be reached following the execution of the parse tree portion.

Every time the function enters a child in the parse tree, it creates a new position for it and pushes it onto the stack. Every time it leaves a child, it pops this stack.

The compiler must determine in which cases, when visiting a node in the parse tree, code should be generated to record and subsequently restore positions. Obviously, this should not be done for all nodes, but only for those which are targets of a path or those where the control flow should return after an excursion to a different position.

The solution to the first case is quite simple: finding the appropriate position in the positions stack and marking it as needed to be saved, as well as saving a reference to it in the path's parse tree (a wide arrow in Figure 10.1.3).

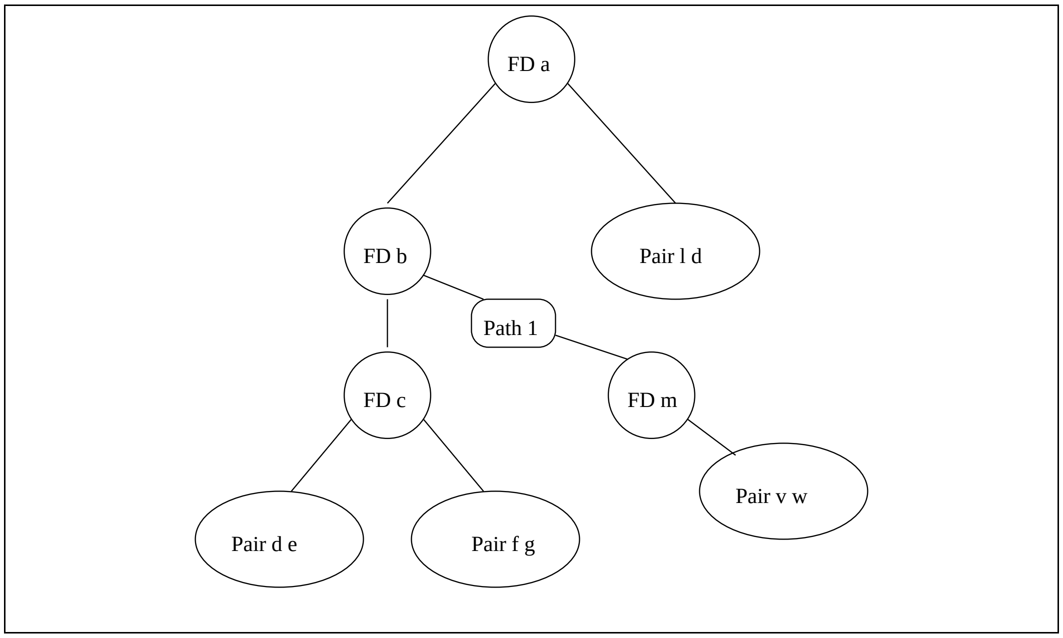


Figure 10.1.2: parse tree for Figure 10.1.1

For the second case, we use the target position (the third argument): each time we descend into a sub parse tree, we transfer the father target as the target for the sub parse tree call if it (the sub parse tree) is the leftmost (last) child of the father. If it isn't last, the father's position is transferred to it as its target position, since it will have to return to the father position, in order to continue the computation of the other children.

The above holds for internal nodes. We must also handle the leaves. Upon a call to a leaf node, the function compares the target position arguments to the top position in the position stack: when these positions are equal, nothing has to be done (assign to this node the null position), that is since we are already in the position needed to be reached after the execution of the node's commands and leaf nodes doesn't change the machine position.

If, however, the target position doesn't equal the position at the top of the positions stack, we must mark the target position as needed to be saved and add a reference to it in the leaf node.

After this preprocessing of the parse tree, we only have to assign numbers to the positions that are marked as need to be saved. These numbers will be their position in the environment. This assignment is done by traversing the parse tree and assigning numbers according to the and-or tree in a manner that for and nodes the commands are numbered first (get the lowest numbers) and the and node children (or nodes) are assigned last. This order results in a more compact environment.

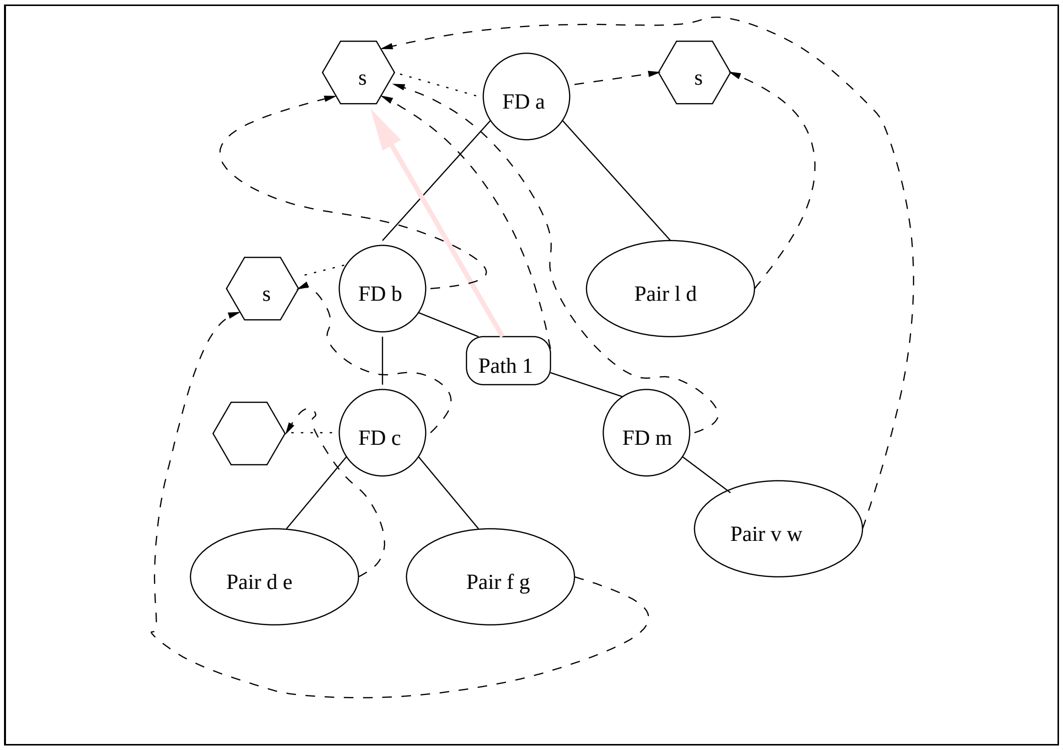


Figure 10.1.3: Annotated Parse Tree for Figure 10.1.1

Part III

Chapter 11

Evaluation

The best benchmark would be to run the three system with SURGE (a large grammar for English written in Fuf) as a grammar and its huge collection of inputs. However, since my system (the compiler and $\mathcal{M}_3$) doesn't support some of features found in surge (patterns, explicit cset, and fset), I cannot perform this benchmark. Instead I choose to test the system's behavior on some situations that seem important to test the assumptions underlying the system's design.

The test cases come in two groups: the first group is tests cases 1-3. The basis for these tests is the following grammar:

```
((alt
  (([1](cat one)
    (a a a a a a a a a a stop)
    (a ((cat two))))

   ([2](cat two)
    (alt (([3](a ((cat two))))

          ([4](a stop))))))))
```

It is tested with the input `((cat one))`. This gramar is intended to check the system in states where there is many recursion on sub-constituents. For each system and each case (1-3), there are 10 variations of the case in which the number of a's in the path is different: it starts from 10 and goes up to 100 in step 10. Thus, for each test case and system, there is a graph in which the x axis represents the number of a's and the y axis is the time measured in seconds.

Test cases 1 - 3:

- Test case 1 is just the bare grammar appearing previously. Its purpose is to help us understand the next two cases.
- Test case 2 is test case 1 with an FD (of about 700 lines) inserted at $\boxed{1}$, $\boxed{2}$, $\boxed{3}$ and $\boxed{4}$.
- Test case 3 is similar to test case 1, but a large FD consisting of paths is added at $\boxed{2}$.

The rest of the test case is based on the following variations:

```
((alt
    (
     ((alt (
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))
            ((they are))

            ((they think))))
      (alt (
            ($\boxed{1}$(i {they}))
            ($\boxed{2}$(i {they}))
            ($\boxed{3}$(i {they}))
            ($\boxed{4}$(i {they}))
            ($\boxed{5}$(i {they}))
            ($\boxed{6}$(i {they}))
            ($\boxed{7}$(i {they}))
            ($\boxed{8}$(i {they}))
            ($\boxed{9}$(i {they}))
            ($\boxed{10}$(i {they}))
            )))
     ((b1 failed)))))
```

with the input `((i think))`, the numbers on the x axis represent the number of `((they are))`, in the first alternatives. Those tests cases check the system's performance under exhaustive backtracking.

- Test case 4 is just the prototype given above.
- Test case 5 is the prototype with a large FD (the same large FD from case 2) added at $\boxed{1}$ - $\boxed{10}$.
- Test case 6 is the prototype with paths added at $\boxed{1}$ - $\boxed{10}$.

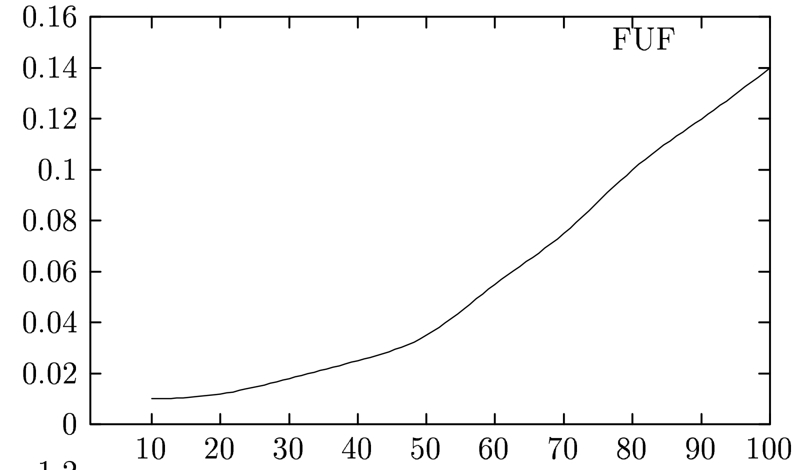
Test case 7 is different, in that it checks the efficiency of the system's in reconstructing the positions in the graph while backtraking. Thus, case 7 is:

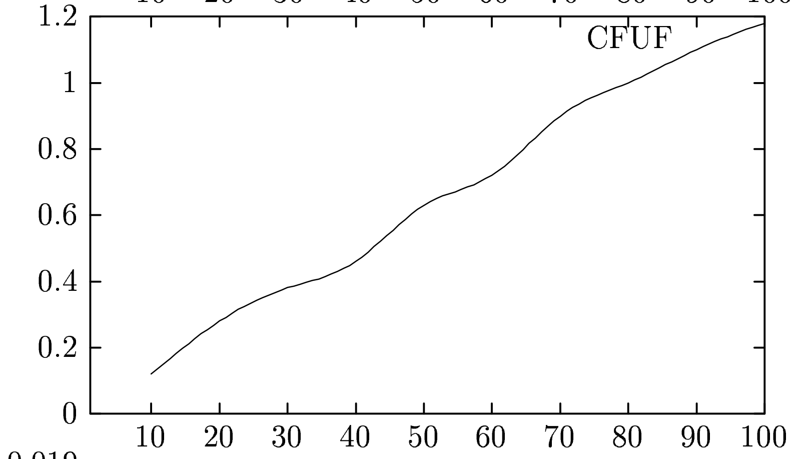
```
((alt
     (
      ((r ((f ((c ((m ((l ((t ((v ((u ((f ((c ((alt (
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} are))
             (({they} think))))))))))))))))))))))))
       (v ((p ((c ((t ((q ((f ((alt (
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     (({i} {they}))
                                     )))))))))))))))
      ((b1 failed)))))
```

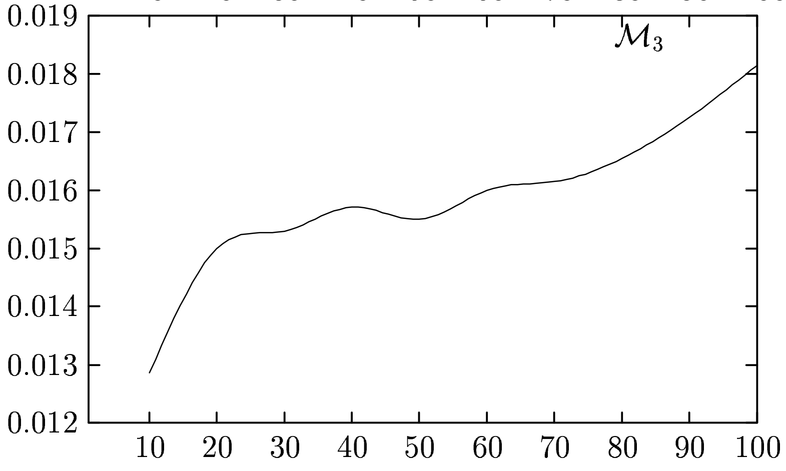
test case 7 was run with the same input as Test cases 4-6.

It is worthwhile to mention that the times for $\mathcal{M}_3$ include the time it takes to do scanning, parsing and linking along with the execution time. At Appendix B, I show a profile file of $\mathcal{M}_3$ for one of the test cases.

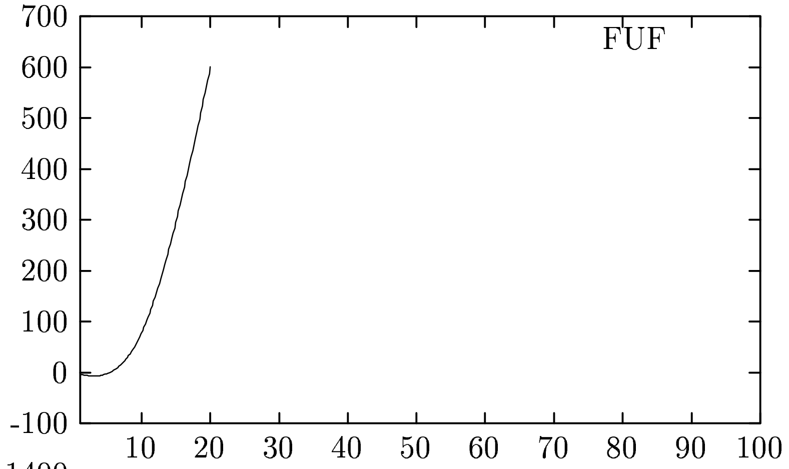
11.1 Test Case 1

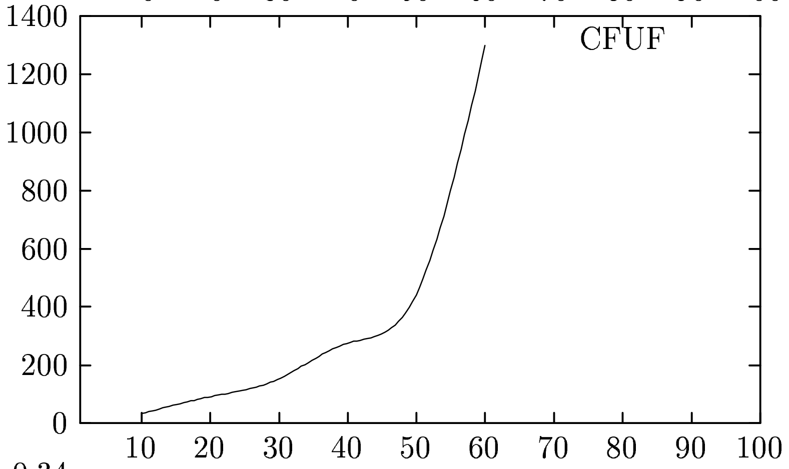


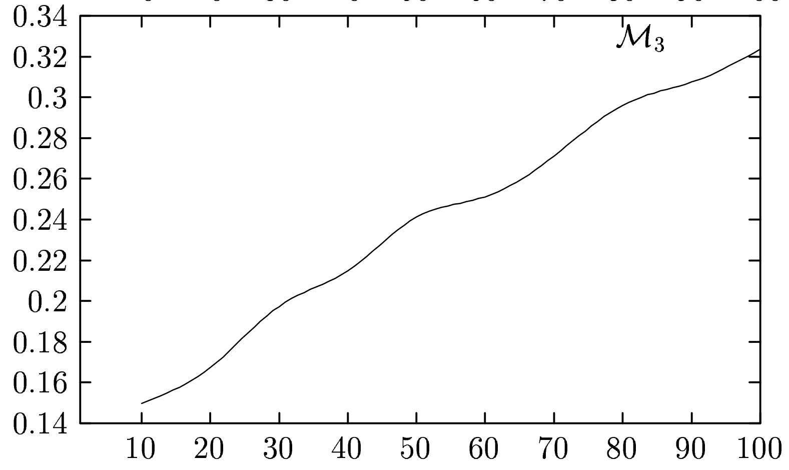




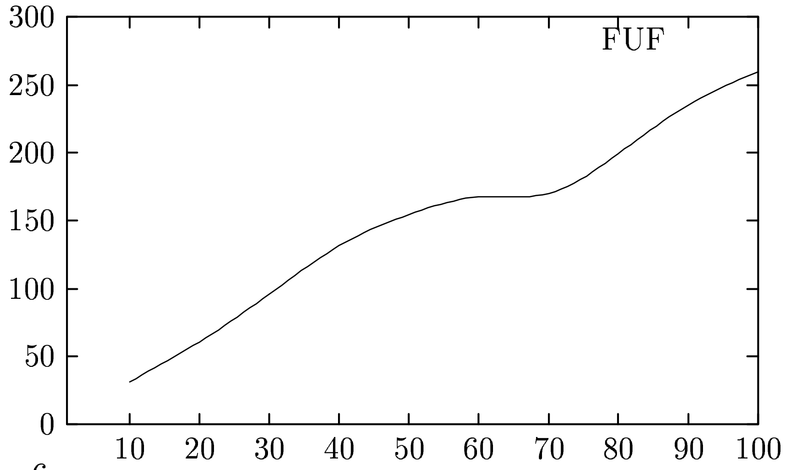
11.2 Test Case 2

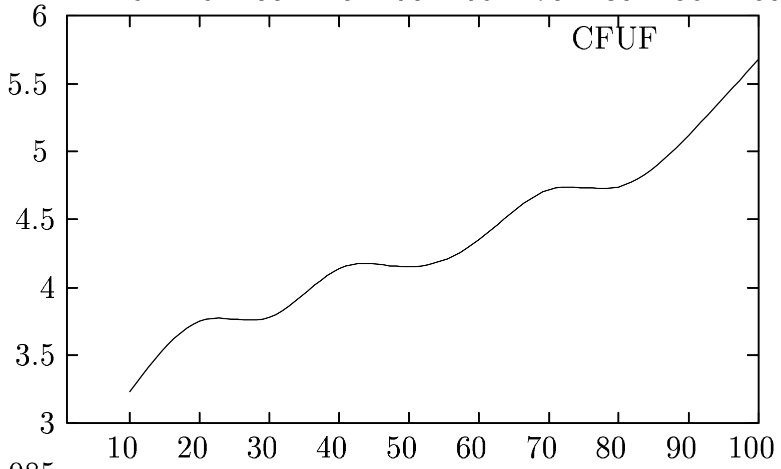


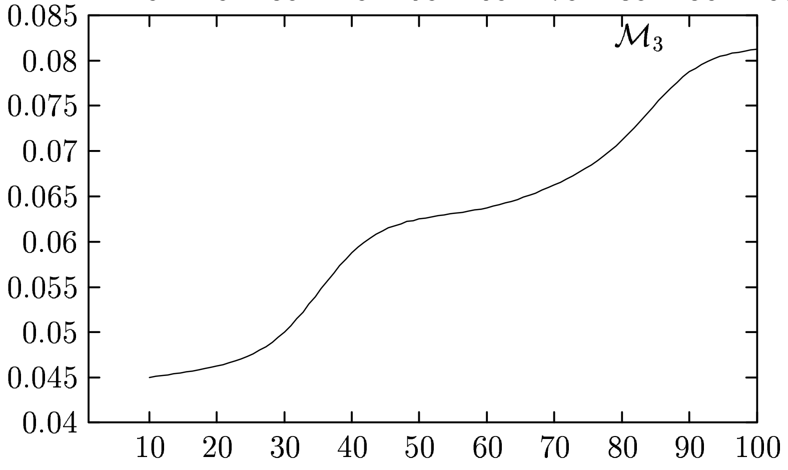




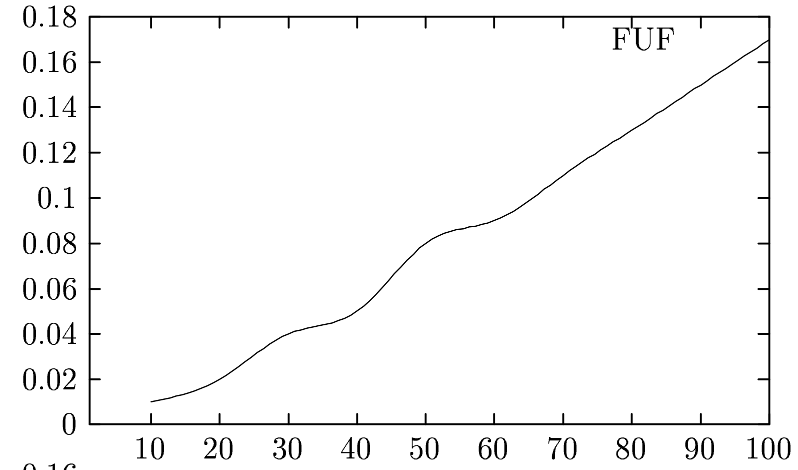
11.3 Test Case 3

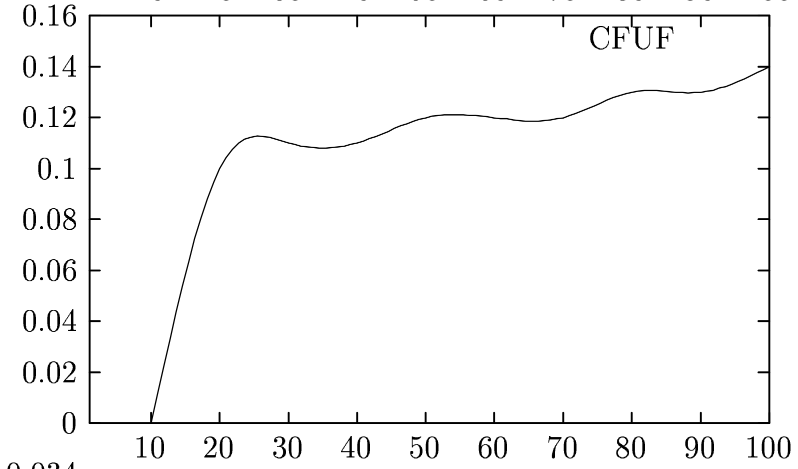


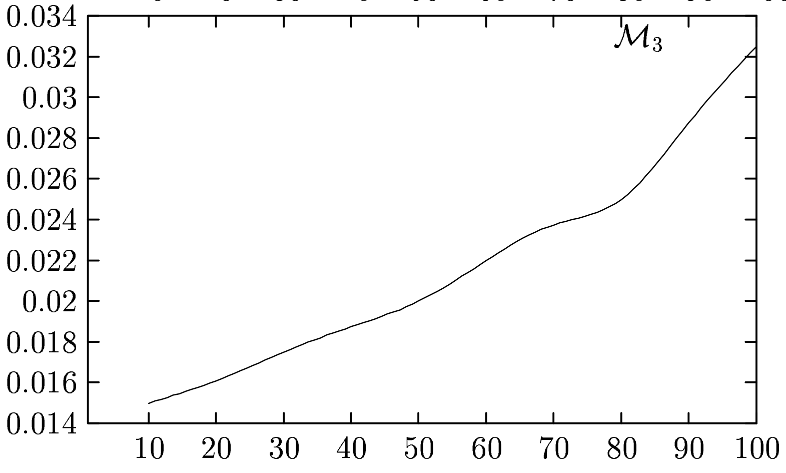




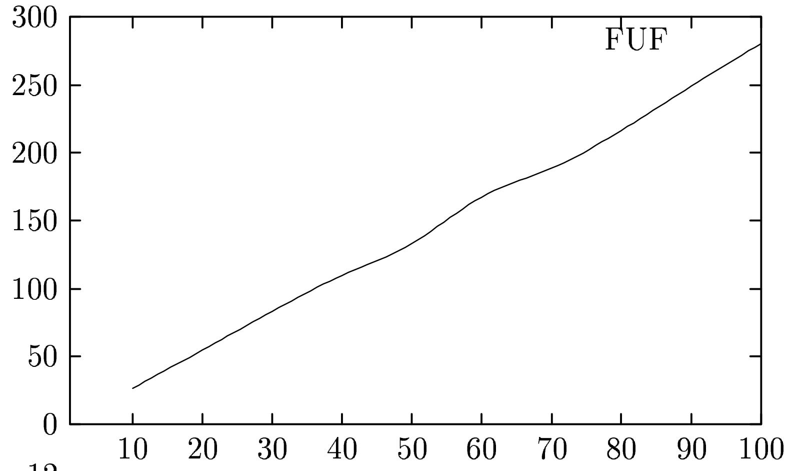
11.4 Test Case 4

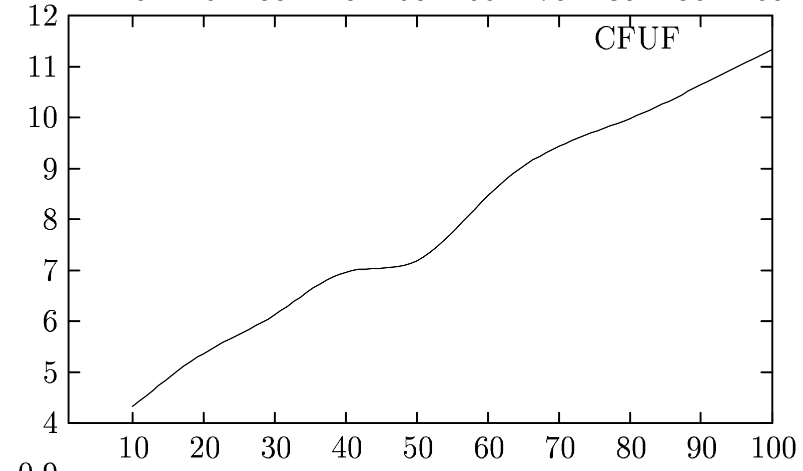


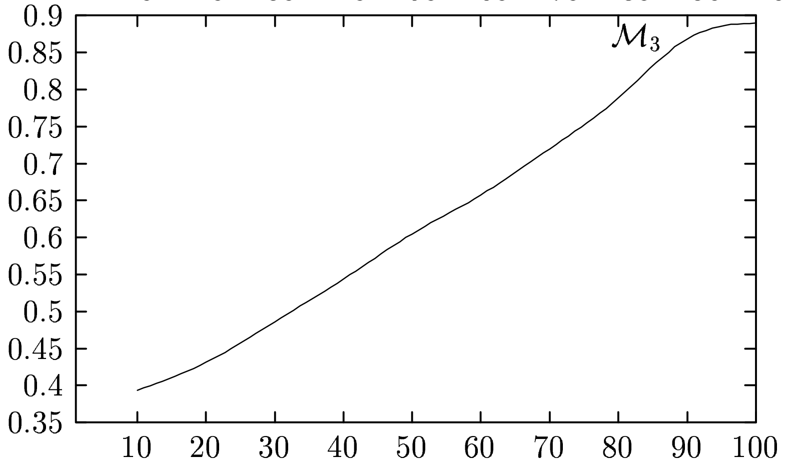




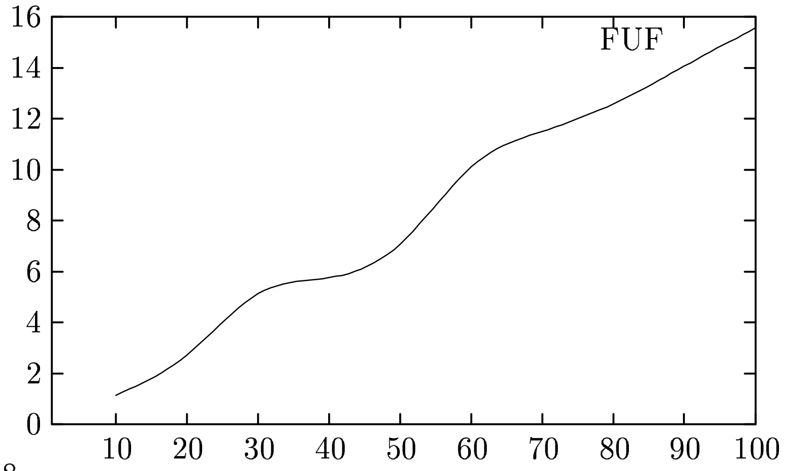
11.5 Test Case 5

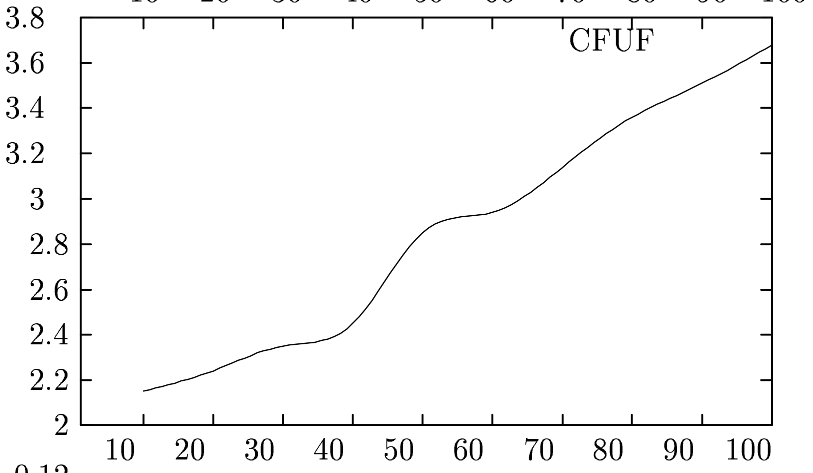


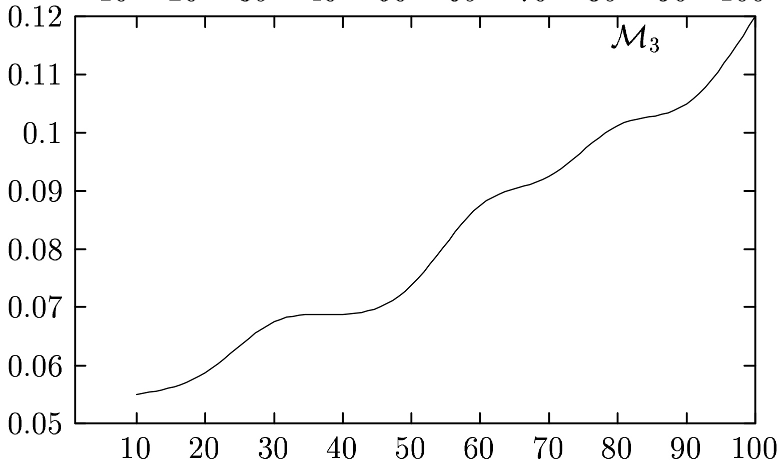




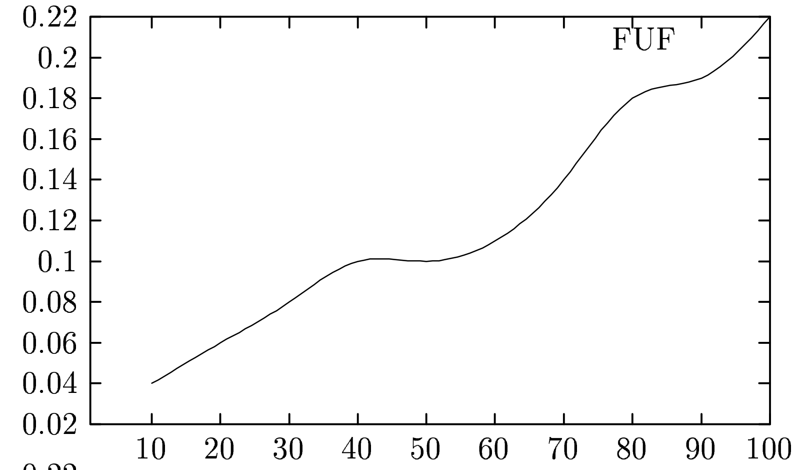
11.6 Test Case 6

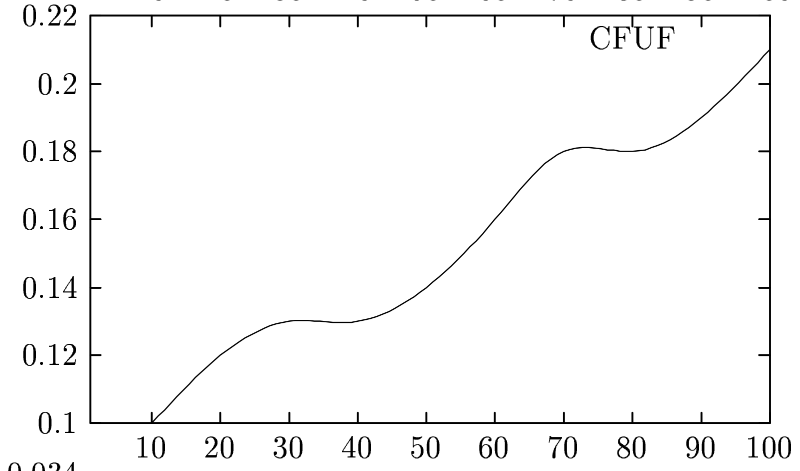


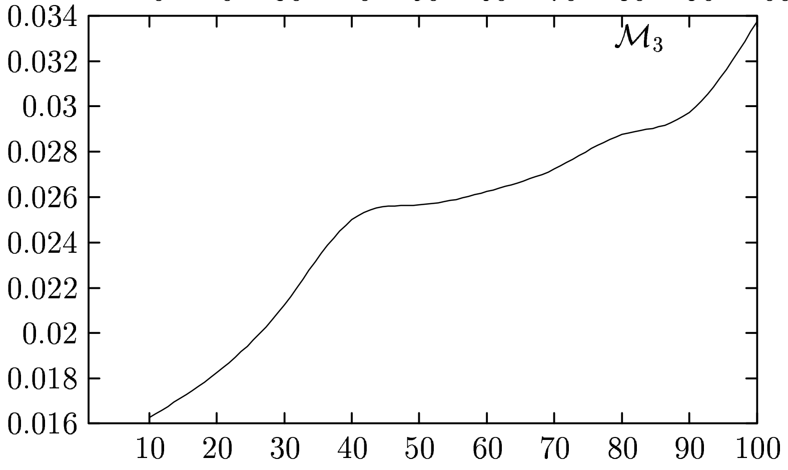




11.7 Test Case 7







Part IV

Chapter 12

Conclusion

The starting point of this research was a FUF (Functional Unification Formalism) interpreter. My aim was to design an efficient compiler for this language. The problems I was facing are listed below:

1. Understanding the meaning of compiling FUF.
2. Finding a way to represent the graph structures that FUF creates.
3. Figuring out a way to efficiently handle paths and type hierarchies.
4. Compiling alternatives and paths efficiently.
5. Solving the problem of recursion on the sub-constituent.
6. Streamlining the backtracking process.

I have presented a series of compilers and virtual machines $\mathcal{M}_0$ to $\mathcal{M}_3$ in an increasing order of complexity, where each compiler–machine–language triplet deals with a growing part of the problems listed above.

At first, I demonstrated a language $\mathcal{L}_0$ which is a simplified language which lacks paths, types, alternatives and recursion on sub-constituents. From there, I proceeded by presenting a more complete language $\mathcal{L}_1$, which adds to $\mathcal{L}_0$ types and paths. I then showed how to compile $\mathcal{L}_1$.

Lastly, the language $\mathcal{L}_2$ which includes alternatives and recursion on sub-constituents was presented and I showed two methods for compiling and executing this language, described in Chapter 9 and Chapter 10, respectively.

In Chapter 10, the relation between procedure application and alternative branches in FUF is revealed and the goal of transforming the compilation of FUF into the more familiar problem of imperative language compilation is achieved.

The key ideas and achievements of this research are:

- A compact representation of the graph in the machine memory.
- Type compilation in a compact data-structure that enables fast type unification.
- Specific treatments of path construction for each of the four possible cases faced at runtime when unifying two graph locations (commands `try_path`, `enter_path`, `try_long_path`, `try_short_path`, `link_short_to_long`, `enter_short_path`, `link_long_to_short` and `merg_path` presented in Chapter 8.8).
- Use of conditional relative jump to enhance path execution speed: when at runtime the graph is traversed along a given path, the question whether to extend the graph with new arcs or to follow existing arcs must be asked for each arc of the path. With the new method, this test is not performed anymore from the moment a new arc is created. This optimization makes the construction of large graphs much more efficient.
- Introduction of a model of the generated code where a grammar is encoded as an and-or tree of procedures, each procedure corresponding to a branch in an alt in FUF.
- Compilation of the and-or tree structure of a grammar in the form of success-fail continuations, where success is encoded in the program's flow of control, and fail is saved on the machine stack.
- Last call optimization when executing the procedures corresponding to alt branches.
- Keeping the trail smaller, by saving only those changes performed on the old part of the heap. (The trail keeps track of the history of changes performed on the graph so that the graph can be restored to its original state upon backtracking.)
- Introducing environments as a way of dealing with paths and alternatives: the basic intuition is that paths in FUF serve the same purpose as free variables in logic programs; the cost of keeping runtime information to resolve paths destination can be avoided by encoding paths at compile time as references in a lexically scoped environment. The mapping of the FUF concepts of branches and paths to the classical concepts of variables and environments is described in Chapter 10.

In the last chapter of this work, I describe various tests performed in order to compare the run times of the three systems - FUF, CFUF, and the compiler - machine product of my research. A table summarizing run time ratios is presented below:

Test case	1	2	3	4	5	6	7
$\mathcal{M}_3$	1	1	1	1	1	1	1
CFUF	60	∞	600	5	12	32	6.5
FUF	8	∞	25,000	5	300	133	6.5

As demonstrated by these results, the compiler provides significant speed improvements especially in the cases where the existing implementations suffered: handling of large FDs with heavy structure sharing (that is, many paths) and handling of heavy backtracking.

In addition, note that a "real life" implementation of the compiler would further enhance these results significantly, because:

1. $\mathcal{M}_3$'s run times include scanning, parsing and linking, as is the case for FUF. CFUF times were measured for unification time only.
2. The compiled code is interpreted, which means that every $\mathcal{M}_3$ command is a call to a C function. If the code were implemented as native code, the overhead of C function-call would be avoided.

Chapter 13

Suggested Future Study

This research has demonstrated the possibility to compile a very high level language to efficient machine code. Building on the models introduced here, one could investigate the following routes:

- Compiling FUF to native assembly language code instead of machine commands. This can be done without losing portability by defining the current compiler command's code as an intermediate code and using an existing compiler's back end to translate it to the assembly level (a natural candidate would be gcc's backend).
- Compiling indexes.
- Adding support for explicit cset and patterns in FUF.
- Adding additional optimizations, such as common sub-expression elimination, changing the commands order[1] and grammar analysis[2].

[1]As was done in [19]

[2]This is partly done in $\mathcal{M}_3$ but there is more to do

Appendix A

$\mathcal{M}_3$ commands implementation.

```
put_new_internal_short(int feat) ≡
  HEAP[H+1] ⟵ ⟨feat,feat⟩ ;
  if(data_of(HEAP[PATH_SHORT_CN]) = PATH_SHORT_CN)
    HEAP[H+2] ⟵ ⟨next,H+2⟩ ;
  else
    HEAP[H+2] gassign ⟨next,data_of(HEAP[PATH_SHORT_CN])⟩;
  HEAP[H+3] gassign ⟨chil,H+3⟩;
  ⟨tag,_⟩ ⟵ HEAP[PATH_SHORT_CN];
  BIND(HEAP + PATH_SHORT_CN,⟨tag,H + 1⟩);
  PATH_SHORT_CN ⟵  H + 3;
  H ⟵ H + 3;

put_new_internal_at_front_long(int feat) ≡
  HEAP[H+1] ⟵ ⟨feat,feat⟩;
  if(data_of(HEAP[PATH_LONG_CN]) = PATH_LONG_CN)
    HEAP[H+2] ⟵ ⟨next,H+2⟩;
  else
    HEAP[H+2] ⟵ ⟨next,data_of(HEAP[PATH_LONG_CN])⟩;
  HEAP[H+3] ⟵ ⟨chil,H+3⟩;
  ⟨tag,_⟩ ⟵ HEAP[PATH_LONG_CN];
  BIND(HEAP + PATH_LONG_CN,⟨tag,H + 1⟩);
  PATH_LONG_CN ⟵ H + 3;
  H ⟵ H + 3;

put_new_internal_at_front_short(int feat) ≡
  HEAP[H+1] ⟵ ⟨feat,feat⟩;
  if(data_of(HEAP[PATH_SHORT_CN]) = PATH_SHORT_CN)
    HEAP[H+2] ⟵ ⟨next,H+2⟩;
  else
```

```
    HEAP[H+2] ⟵ ⟨next,data_of(HEAP[PATH_SHORT_CN])⟩;
  HEAP[H+3] = ⟨chil,H+3⟩;
  ⟨tag,_⟩ ⟵ HEAP[PATH_SHORT_CN];
  BIND(HEAP + PATH_SHORT_CN,⟨tag,H + 1⟩);
  PATH_SHORT_CN ⟵ H + 3;
  H ⟵ H + 3;


save(int envaddress) ≡
  env[env_start+envaddress] = CN;

enter_node (int f) ≡
  FIND_CHILD(CN,f);
  if(¬FOUND)
    put_new_internal(f,R1);
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
    if(type_of(heap[R1]) = CHIL)
        CN ⟵ R1;
    else
        if(type_of(heap[R1]) = TYPE)
            R2 ⟵ data_of(heap[R1]);
            switch (R2)
              case ⟨TYPE,NIL⟩:
                change_to_str(feat,R1);
                CN ⟵ R1;
                break;
              case ⟨TYPE,NIL⟩:
                change_to_any_str(feat,R1);
                CN ⟵ R1;
                break;
              default: FAIL();
          else
            FAIL();

restore_path_start (int envindex) ≡
  PLONGCN ⟵ env[env_start + envindex];
  PSHORTCN ⟵ CN;

restore (int envindex) ≡
```

```
  CN ⟵ env[env_start + envindex];
```

assign_env `(int baseindex) ≡`

```
  env_next ⟵ baseindex;
```

set_relative_jmp `(int reljump) ≡`

```
  PRELJMP ⟵ reljump;
  FLAG ⟵ 0;
```

try_path `(int f) ≡`

```
  R1 ⟵ PLONGCN;
  FIND_CHILD(R1,f);
  if(¬FOUND)
      pc ⟵ pc +  2 * (PRELJMP - 1);
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      if(type_of(heap[R1]) = CHIL)
          PLONGCN ⟵ R1;
      else
          if(type_of(heap[R1]) = TYPE)
              R2 = data_of(heap[R1]);
              switch (R2)
                case ⟨TYPE,NIL⟩:
                  change_to_str(f,R1);
                  PLONGCN ⟵ R1;
                  break;
                case ⟨TYPE,ANY⟩:
                  change_to_any_str(f,R1);
                  PLONGCN ⟵ R1;
                  break;
                default: FAIL();
            else
              FAIL();
```

enter_path `(int f) ≡`

```
  put_new_internal_at_front_long(feat);
```

try_long_path `(int f)`

```
  R1 ⟵ PLONGCN;
  FIND_CHILD(R1,feat);
  if(¬FOUND)
      FLAG ⟵ 1;
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      PLONGCN ⟵ R1;
      FLAG ⟵ 0;
```

try_short_path (int f) ≡

```
  R1 ⟵ PSHORTCN;
  FIND_CHILD(R1,f);
  if(¬FOUND)
      FLAG ⟵ 1;
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      PSHORTCN ⟵ R1;
      FLAG ⟵ 0;
```

enter_short_path (int f) ≡

```
  R1 ⟵ PSHORTCN;
  FIND_CHILD(R1,f);
  if(¬FOUND)
    put_new_internal_short(f);
  else
      R1 ⟵ R1 + 2;
      DEREF(R1);
      PSHORTCN ⟵ R1;
```

jf (int address) ≡

```
  if(FLAG = 1)
      pc ⟵ base + address
```

merg_path ≡

```
  R1= PLONGCN;
  R2 = PSHORTCN;
  merge();
```

link_short_to_long (int f) ≡

```
  heap[H+1] ⟵ ⟨FEAT,f⟩;
  if(data_of(heap[PSHORTCN]) = PSHORTCN)
  heap[H+2] ⟵ MAKNEXT(H+2);
else
  ⟨_,ADDR⟩ ⟵ heap[PSHORTCN];
  heap[H+2] ⟵ ⟨NEXT,ADDR⟩;
heap[H+3] ⟵ ⟨NODE,PLONGCN⟩;
⟨TAG,_⟩ ⟵ heap[PSHORTCN];
BIND_DEBUG(heap + PSHORTCN,⟨TAG,H + 1⟩);
H = H + 3;
```

link_long_to_short (int f) ≡

```
heap[H+1] ⟵ ⟨FEAT,f⟩;
if(data_of(heap[PLONGCN]) = PLONGCN)
  heap[H+2] ⟵ ⟨NEXT,H+2⟩;
else
  ⟨_,data⟩ ⟵ heap[PLONGCN];
  heap[H+2] ⟵ ⟨NEXT,data⟩;
heap[H+3] ⟵ ⟨NODE,PSHORTCN⟩;
⟨tag,_⟩ ⟵ heap[PLONGCN];
BIND_DEBUG(heap + PLONGCN,⟨tag,H + 1⟩);
H ⟵ H + 3;
```

insert_if_have_children_with_cat ≡

```
int save_R1 ⟵ R1;
FIND_CHILD_DEBUG(R1,CAT);
if(FOUND)
    BIND(heap+save_R1,heap[save_R1]);
    heap[save_R1] ⟵ MARK_DONE(heap[save_R1]);
    trail[++t] ⟵ csetsp;
    trail[++t] ⟵ address of csetsp;
    if(csetsp ≠ -1)
        trail[++t] ⟵ csetstack[csetsp];
        trail[++t] ⟵ (address of csetstack) + csetsp;
    CSETPUSH(save_R1);
```

find_all_cset ≡

```
R2 ⟵ CN;
while(R2 ¬ data_of(heap[R2]))
    R2 ⟵ NEXT(heap[R2]);
    R1 ⟵ R2;
    FEAT_TO_NODE(R1);
```

```
        if(CHILP_UN_DONE(heap[R1]))
            insert_if_have_children_with_cat();
        R2 ⟵ R2 + 1;
```

get_next_cset (int address) ≡

```
  if(csetsp ≠ 0)
        trail[++t] ⟵ csetsp;
        trail[++t] ⟵ address of csetsp;
        trail[++t] ⟵ csetstack[csetsp];
        trail[++t] ⟵ (address of csetstack) + csetsp;
        next_cset ⟵ CSETPOP;
        DEREF(next_cset);
        CN ⟵ next_cset;
  else
        trail[++t] ⟵ env_start;
        trail[++t] ⟵ address of env_start;
        env_start ⟵ env_start +  env_next;
        pc ⟵ base + address;
```

program_init (int address) ≡

```
  if(¬seen_input)
    init();
  env_start ⟵ 0;
  base ⟵ prog;
  POP;
  sp ⟵ 0;
  PUSH(address);
  csetsp ⟵ 0;
  CSETPUSH(0);
```

success

```
  pc ⟵ prog + end_p;
  program_ended_successfully ⟵ 1;
```

fail

```
  if(prog = base)
    pc ⟵ prog + end_p;
  else
    pc ⟵ prog + end_fd;
  program_ended_successfully ⟵ 0;
```

```
goto_inst (int address)
  pc ⟵ address;

FAIL ≡  backtrack;

init
  sp ⟵ 0;
  csetsp ⟵ 0;
  base ⟵ fdprog;
  env_start ⟵ 0;
  heap[0] ⟵ ⟨CHIL,0⟩;
  CN ⟵ 0;
  H ⟵ 0;
  saved_h ⟵ heap;
  PUSH(end_fd);

put_new_internal (int f,int addr)
  heap[H+1] ⟵ ⟨FEAT,f⟩;
  heap[H+2] ⟵ ⟨NEXT,H + 2⟩;
  heap[H+3] ⟵ ⟨CHIL,H + 3⟩;
  ⟨tag,_⟩ ⟵ heap[addr];
  BIND(heap + addr,⟨tag,H + 1⟩);
  H ⟵ H + 3;
  CN ⟵ H;

change_to_str (int feat,int addr) int ≡
  BIND(heap + addr,⟨CHIL,addr⟩);
  CN ⟵ addr;

change_to_any_str(int feat,int addr)
  BIND(heap + addr,⟨ANY,H + 1⟩;
  CN ⟵ addr;

put_new_leaf(int feat, int type, int addr)
  heap[H+1] ⟵ ⟨FEAT,feat⟩;
  heap[H+2] ⟵ ⟨NEXT,H + 2⟩;
  heap[H+3] ⟵ ⟨TYPE,type⟩;
  ⟨tag,_⟩ ⟵ heap[addr];
  BIND_DEBUG(heap + addr,⟨tag,H + 1⟩);
  H ⟵ H + 3;
```

```
add_any (int feat) ≡
  FIND_CHILD(CN,feat);
  if(¬FOUND)
    put_new_leaf(feat,ANY,R1);
  else
      FEAT_TO_NODE(R1);
      if(type_of(heap[R1]) = TYPE)
          R2 = data_of(heap[R1]);
          R3 = unify(R2,ANY);
          if(R3 ≠ -1)
              if(R3 ≠ R2)
                  BIND(heap + R1,⟨TYPE,R3⟩);
          else
              FAIL();
      else
          if(type_of(heap[R1]) = CHIL)
              ⟨_,data⟩;
              BIND(heap + R1,⟨ANY,data⟩);
          else
              if(type_of(heap[R1]) = ANY)
                  exit(1);

add_leaf (int feat, int type) ≡
  FIND_CHILD_DEBUG(CN,feat);
  if(¬FOUND)
    put_new_leaf(feat,type,R1);
  else
      FEAT_TO_NODE(R1);
      if(type_of(heap[R1]) = TYPE)
          R2 ⟵ type_of(heap[R1]);
          R3 ⟵ unify(R2,type);
          if(R3 ≠ -1)
              if(R3 ≠ R2)
                  BIND(heap + R1,⟨TYPE,R3⟩);
              else
                FAIL();
          else
          FAIL();

try_me_else (int address) ≡
```

```
  PUSH(CN);
  PUSH(H);
  saved_h ⟵ heap + H;
  PUSH(t);
  PUSH(address);
```

retry_me_else (int address) ≡

```
  PUSH(address);
  UNTRAILG(stack[sp -1]);
  H ⟵ (stack[sp - 2]);
  CN ⟵ (stack[sp - 3]);
  saved_h ⟵ heap + H;
```

trust_me

```
  UNTRAIL(stack[sp]);
  H ⟵ (stack[sp - 1]);
  CN ⟵ (stack[sp - 2]);
  saved_h ⟵ heap + H;
  sp ⟵ sp - 3;
```

backtrack ≡

```
  pc = base+POP;
```

unify (int old, int new) ≡

```
  if(old == new)
    return old;
  if((UNIFICATION_TABEL_SIZE + SPECIAL_TYPES_NUMBER ≤ new) and
     (UNIFICATION_TABEL_SIZE + SPECIAL_TYPES_NUMBER ≤ old))
    return -1;
  if((new < UNIFICATION_TABEL_SIZE) and
     (old < UNIFICATION_TABEL_SIZE))
    return unification_table[old][new];
  switch (old)
    case NIL:
      return new;
      break;
    case ANY:
      return new;
      break;
    case NONE:
      return -1;
  break;
case GIVEN:
  return new;
  break;
case CAT:
  CAT == new ? new : -1;
  break;
```

Appendix B

Profiling For r59

```
/*
 * copyright (c) 1993 by sun microsystems, inc.
 */

#pragma ident   "@(#)gprof.callg.blurb  1.8     93/06/07 smi"


call graph profile:
          the sum of self and descendents is the major sort
          for this listing.

          function entries:

index     the index of the function in the call graph
          listing, as an aid to locating it (see below).

%time     the percentage of the total time of the program
          accounted for by this function and its
          descendents.

self      the number of seconds spent in this function
          itself.

descendents
          the number of seconds spent in the descendents of
          this function on behalf of this function.

called    the number of times this function is called (other
          than recursive calls).

self      the number of times this function calls itself
          recursively.

name      the name of the function, with an indication of
          its membership in a cycle, if any.

index     the index of the function in the call graph
          listing, as an aid to locating it.



          parent listings:

self*     the number of seconds of this function's self time
```

```
          which is due to calls from this parent.

descendents*
          the number of seconds of this function's
          descendent time which is due to calls from this
          parent.

called**  the number of times this function is called by
          this parent.  this is the numerator of the
          fraction which divides up the function's time to
          its parents.

total*    the number of times this function was called by
          all of its parents.  this is the denominator of
          the propagation fraction.

parents   the name of this parent, with an indication of the
          parent's membership in a cycle, if any.

index     the index of this parent in the call graph
          listing, as an aid in locating it.



          children listings:

self*     the number of seconds of this child's self time
          which is due to being called by this function.

descendent*
          the number of seconds of this child's descendent's
          time which is due to being called by this
          function.

called**  the number of times this child is called by this
          function.  this is the numerator of the
          propagation fraction for this child.

total*    the number of times this child is called by all
          functions.  this is the denominator of the
          propagation fraction.

children  the name of this child, and an indication of its
          membership in a cycle, if any.

index     the index of this child in the call graph listing,
          as an aid to locating it.



          * these fields are omitted for parents (or
          children) in the same cycle as the function.  if
          the function (or child) is a member of a cycle,
          the propagated times and propagation denominator
          represent the self time and descendent time of the
          cycle as a whole.

          ** static-only parents and children are indicated
          by a call count of 0.
          cycle listings:
          the cycle as a whole is listed with the same
          fields as a function entry.  below it are listed
          the members of the cycle, and their contributions
          to the time and call counts of the cycle.



granularity: each sample hit covers 4 byte(s) for 0.91% of 1.10 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                0.00        0.67       1/1           _start [2]
[1]     60.9    0.00        0.67       1         main [1]
                0.01        0.51       1/1           run_machine [3]
                0.00        0.15       2/2           loadfd [8]
                0.00        0.00       2/2           link [40]
                0.00        0.00       1/1           allocate_symbol_tabel [45]
                0.00        0.00       1/1           allocate_argumets_tabel [44]
                0.00        0.00       1/1           hashinit [46]

-----------------------------------------------

                                                     <spontaneous>
[2]     60.9    0.00        0.67                 _start [2]
                0.00        0.67       1/1           main [1]

-----------------------------------------------

                0.01        0.51       1/1           main [1]
[3]     47.3    0.01        0.51       1         run_machine [3]
                0.13        0.32  628730/628730      add_leaf [4]
                0.03        0.00   59059/59059       enter_node [10]
                0.01        0.00   52052/52052       restore [12]
                0.01        0.00    6007/6007        save [13]
                0.01        0.00    2002/2002        jf [14]
                0.00        0.00    1001/1001        merg_path [16]
                0.00        0.00       1/1           find_all_cset [18]
                0.00        0.00    1001/1001        try_short_path [23]
                0.00        0.00    1001/1001        try_long_path [22]
                0.00        0.00    1001/1001        restore_path_start [21]
                0.00        0.00     899/899         retry_me_else [28]
                0.00        0.00     103/103         try_me_else [31]
                0.00        0.00     102/102         goto_inst [32]
                0.00        0.00     101/101         trust_me [33]
                0.00        0.00       2/2           get_next_cset [39]
                0.00        0.00       1/1           init [47]
                0.00        0.00       1/1           success [49]
                0.00        0.00       1/1           program_init [48]

-----------------------------------------------

                0.13        0.32  628730/628730      run_machine [3]
[4]     40.7    0.13        0.32  628730         add_leaf [4]
                0.29        0.00  628730/628731      FIND_CHILD_DEBUG [6]
                0.03        0.00  628730/688790      BIND_DEBUG [11]

-----------------------------------------------

                                                     <spontaneous>
```

```
[5]     39.1    0.43        0.00                 internal_mcount [5]

-----------------------------------------------

                0.00        0.00       1/628731      insert_if_have_children_with_cat [19]
                0.29        0.00  628730/628731      add_leaf [4]
[6]     26.4    0.29        0.00  628731         FIND_CHILD_DEBUG [6]

-----------------------------------------------

                0.15        0.00   30207/30207       yyparse [9]
[7]     13.6    0.15        0.00   30207         yylex [7]
                0.00        0.00   13406/13406       getsymbolkey [20]
                0.00        0.00      55/55          yy_get_next_buffer [34]
                0.00        0.00      53/53          yy_get_previous_state [35]
                0.00        0.00       1/1           yy_create_buffer [50]
                0.00        0.00       1/3           yy_load_buffer_state [38]

-----------------------------------------------

                0.00        0.15       2/2           main [1]
[8]     13.6    0.00        0.15       2         loadfd [8]
                0.00        0.15       2/2           yyparse [9]

-----------------------------------------------

                0.00        0.15       2/2           loadfd [8]
[9]     13.6    0.00        0.15       2         yyparse [9]
                0.15        0.00   30207/30207       yylex [7]
                0.00        0.00     263/263         add_label [29]
                0.00        0.00     145/145         add_address [30]
                0.00        0.00       2/2           set_labels_number [41]

-----------------------------------------------

                0.03        0.00   59059/59059       run_machine [3]
[10]     3.0    0.03        0.00   59059         enter_node [10]
                0.00        0.00   59059/59059       put_new_internal [15]

-----------------------------------------------

                0.00        0.00    1001/688790      merge [17]
                0.00        0.00   59059/688790      put_new_internal [15]
                0.03        0.00  628730/688790      add_leaf [4]
[11]     2.7    0.03        0.00  688790         BIND_DEBUG [11]

-----------------------------------------------

                0.01        0.00   52052/52052       run_machine [3]
[12]     0.9    0.01        0.00   52052         restore [12]

-----------------------------------------------

                0.01        0.00    6007/6007        run_machine [3]
[13]     0.9    0.01        0.00    6007         save [13]

-----------------------------------------------

                0.01        0.00    2002/2002        run_machine [3]
[14]     0.9    0.01        0.00    2002         jf [14]

-----------------------------------------------
```

```
                0.00        0.00   59059/59059       enter_node [10]
[15]     0.2    0.00        0.00   59059         put_new_internal [15]
                0.00        0.00   59059/688790      BIND_DEBUG [11]

-----------------------------------------------

                0.00        0.00    1001/1001        run_machine [3]
[16]     0.0    0.00        0.00    1001         merg_path [16]
                0.00        0.00    1001/1001        merge [17]

-----------------------------------------------

                0.00        0.00    1001/1001        merg_path [16]
[17]     0.0    0.00        0.00    1001         merge [17]
                0.00        0.00    1001/688790      BIND_DEBUG [11]
                0.00        0.00    1001/1001        unify [24]
                0.00        0.00    1000/1000        FAIL [25]

-----------------------------------------------

                0.00        0.00       1/1           run_machine [3]
[18]     0.0    0.00        0.00       1         find_all_cset [18]
                0.00        0.00       1/1           insert_if_have_children_with_cat [19]

-----------------------------------------------

                0.00        0.00       1/1           find_all_cset [18]
[19]     0.0    0.00        0.00       1         insert_if_have_children_with_cat [19]
                0.00        0.00       1/628731      FIND_CHILD_DEBUG [6]

-----------------------------------------------

                0.00        0.00   13406/13406       yylex [7]
[20]     0.0    0.00        0.00   13406         getsymbolkey [20]

-----------------------------------------------

                0.00        0.00    1001/1001        run_machine [3]
[21]     0.0    0.00        0.00    1001         restore_path_start [21]

-----------------------------------------------

                0.00        0.00    1001/1001        run_machine [3]
[22]     0.0    0.00        0.00    1001         try_long_path [22]

-----------------------------------------------

                0.00        0.00    1001/1001        run_machine [3]
[23]     0.0    0.00        0.00    1001         try_short_path [23]

-----------------------------------------------

                0.00        0.00    1001/1001        merge [17]
[24]     0.0    0.00        0.00    1001         unify [24]

-----------------------------------------------

                0.00        0.00    1000/1000        merge [17]
[25]     0.0    0.00        0.00    1000         FAIL [25]
                0.00        0.00    1000/1000        backtrack [27]
```

```
-----------------------------------------------

                0.00        0.00     101/1000        trust_me [33]
                0.00        0.00     899/1000        retry_me_else [28]
[26]     0.0    0.00        0.00    1000         UNTRAIL_DEBUG [26]

-----------------------------------------------

                0.00        0.00    1000/1000        FAIL [25]
[27]     0.0    0.00        0.00    1000         backtrack [27]

-----------------------------------------------

                0.00        0.00     899/899         run_machine [3]
[28]     0.0    0.00        0.00     899         retry_me_else [28]
                0.00        0.00     899/1000        UNTRAIL_DEBUG [26]

-----------------------------------------------

                0.00        0.00     263/263         yyparse [9]
[29]     0.0    0.00        0.00     263         add_label [29]

-----------------------------------------------

                0.00        0.00     145/145         yyparse [9]
[30]     0.0    0.00        0.00     145         add_address [30]

-----------------------------------------------

                0.00        0.00     103/103         run_machine [3]
[31]     0.0    0.00        0.00     103         try_me_else [31]

-----------------------------------------------

                0.00        0.00     102/102         run_machine [3]
[32]     0.0    0.00        0.00     102         goto_inst [32]

-----------------------------------------------

                0.00        0.00     101/101         run_machine [3]
[33]     0.0    0.00        0.00     101         trust_me [33]
                0.00        0.00     101/1000        UNTRAIL_DEBUG [26]

-----------------------------------------------

                0.00        0.00      55/55          yylex [7]
[34]     0.0    0.00        0.00      55         yy_get_next_buffer [34]
                0.00        0.00       2/2           yyrestart [43]

-----------------------------------------------

                0.00        0.00      53/53          yylex [7]
[35]     0.0    0.00        0.00      53         yy_get_previous_state [35]

-----------------------------------------------

                0.00        0.00       3/3           yy_init_buffer [37]
[36]     0.0    0.00        0.00       3         yy_flush_buffer [36]

-----------------------------------------------

                0.00        0.00       1/3           yy_create_buffer [50]
                0.00        0.00        2/3           yyrestart [43]
[37]     0.0    0.00        0.00        3         yy_init_buffer [37]
                0.00        0.00        3/3           yy_flush_buffer [36]

-----------------------------------------------

                0.00        0.00        1/3           yylex [7]
                0.00        0.00        2/3           yyrestart [43]
[38]     0.0    0.00        0.00        3         yy_load_buffer_state [38]

-----------------------------------------------

                0.00        0.00        2/2           run_machine [3]
[39]     0.0    0.00        0.00        2         get_next_cset [39]

-----------------------------------------------

                0.00        0.00        2/2           main [1]
[40]     0.0    0.00        0.00        2         link [40]

-----------------------------------------------

                0.00        0.00        2/2           yyparse [9]
[41]     0.0    0.00        0.00        2         set_labels_number [41]

-----------------------------------------------

                0.00        0.00        2/2           yy_create_buffer [50]
[42]     0.0    0.00        0.00        2         yy_flex_alloc [42]

-----------------------------------------------

                0.00        0.00        2/2           yy_get_next_buffer [34]
[43]     0.0    0.00        0.00        2         yyrestart [43]
                0.00        0.00        2/3           yy_init_buffer [37]
                0.00        0.00        2/3           yy_load_buffer_state [38]

-----------------------------------------------

                0.00        0.00        1/1           main [1]
[44]     0.0    0.00        0.00        1         allocate_argumets_tabel [44]

-----------------------------------------------

                0.00        0.00        1/1           main [1]
[45]     0.0    0.00        0.00        1         allocate_symbol_tabel [45]

-----------------------------------------------

                0.00        0.00        1/1           main [1]
[46]     0.0    0.00        0.00        1         hashinit [46]

-----------------------------------------------

                0.00        0.00        1/1           run_machine [3]
[47]     0.0    0.00        0.00        1         init [47]

-----------------------------------------------

                0.00        0.00        1/1           run_machine [3]
[48]     0.0    0.00        0.00        1         program_init [48]
```

```
-----------------------------------------------

                0.00       0.00      1/1          run_machine [3]
[49]     0.0    0.00       0.00      1        success [49]

-----------------------------------------------

                0.00       0.00      1/1          yylex [7]
[50]     0.0    0.00       0.00      1        yy_create_buffer [50]
                0.00       0.00      2/2          yy_flex_alloc [42]
                0.00       0.00      1/3          yy_init_buffer [37]

-----------------------------------------------



/*
 * Copyright (c) 1993 by Sun Microsystems, Inc.
 */

#pragma ident   "@(#)gprof.flat.blurb   1.8     93/06/07 SMI"


flat profile:

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.



granularity: each sample hit covers 4 byte(s) for 0.89% of 1.12 seconds

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 38.4       0.43     0.43                             internal_mcount [5]
 25.9       0.72     0.29   628731     0.00     0.00  FIND_CHILD_DEBUG [6]
 13.4       0.87     0.15    30207     0.00     0.00  yylex [7]
 11.6       1.00     0.13   628730     0.00     0.00  add_leaf [4]
```

```
  2.7       1.03     0.03   688790     0.00     0.00  BIND_DEBUG [11]
  2.7       1.06     0.03    59059     0.00     0.00  enter_node [10]
  1.8       1.08     0.02                             _mcount (164)
  0.9       1.09     0.01    52052     0.00     0.00  restore [12]
  0.9       1.10     0.01     6007     0.00     0.00  save [13]
  0.9       1.11     0.01     2002     0.00     0.00  jf [14]
  0.9       1.12     0.01        1    10.00   520.00  run_machine [3]
  0.0       1.12     0.00    59059     0.00     0.00  put_new_internal [15]
  0.0       1.12     0.00    13406     0.00     0.00  getsymbolkey [20]
  0.0       1.12     0.00     1001     0.00     0.00  merg_path [16]
  0.0       1.12     0.00     1001     0.00     0.00  merge [17]
  0.0       1.12     0.00     1001     0.00     0.00  restore_path_start [21]
  0.0       1.12     0.00     1001     0.00     0.00  try_long_path [22]
  0.0       1.12     0.00     1001     0.00     0.00  try_short_path [23]
  0.0       1.12     0.00     1001     0.00     0.00  unify [24]
  0.0       1.12     0.00     1000     0.00     0.00  FAIL [25]
  0.0       1.12     0.00     1000     0.00     0.00  UNTRAIL_DEBUG [26]
  0.0       1.12     0.00     1000     0.00     0.00  backtrack [27]
  0.0       1.12     0.00      899     0.00     0.00  retry_me_else [28]
  0.0       1.12     0.00      263     0.00     0.00  add_label [29]
  0.0       1.12     0.00      145     0.00     0.00  add_address [30]
  0.0       1.12     0.00      103     0.00     0.00  try_me_else [31]
  0.0       1.12     0.00      102     0.00     0.00  goto_inst [32]
  0.0       1.12     0.00      101     0.00     0.00  trust_me [33]
  0.0       1.12     0.00       55     0.00     0.00  yy_get_next_buffer [34]
  0.0       1.12     0.00       53     0.00     0.00  yy_get_previous_state [35]
  0.0       1.12     0.00        3     0.00     0.00  yy_flush_buffer [36]
  0.0       1.12     0.00        3     0.00     0.00  yy_init_buffer [37]
  0.0       1.12     0.00        3     0.00     0.00  yy_load_buffer_state [38]
  0.0       1.12     0.00        2     0.00     0.00  get_next_cset [39]
  0.0       1.12     0.00        2     0.00     0.00  link [40]
  0.0       1.12     0.00        2     0.00    75.00  loadfd [8]
  0.0       1.12     0.00        2     0.00     0.00  set_labels_number [41]
  0.0       1.12     0.00        2     0.00     0.00  yy_flex_alloc [42]
  0.0       1.12     0.00        2     0.00    75.00  yyparse [9]
  0.0       1.12     0.00        2     0.00     0.00  yyrestart [43]
  0.0       1.12     0.00        1     0.00     0.00  allocate_argumets_tabel [44]
  0.0       1.12     0.00        1     0.00     0.00  allocate_symbol_tabel [45]
  0.0       1.12     0.00        1     0.00     0.00  find_all_cset [18]
  0.0       1.12     0.00        1     0.00     0.00  hashinit [46]
  0.0       1.12     0.00        1     0.00     0.00  init [47]
  0.0       1.12     0.00        1     0.00     0.00  insert_if_have_children_with_cat [19]
  0.0       1.12     0.00        1     0.00   670.00  main [1]
  0.0       1.12     0.00        1     0.00     0.00  program_init [48]
  0.0       1.12     0.00        1     0.00     0.00  success [49]
  0.0       1.12     0.00        1     0.00     0.00  yy_create_buffer [50]


Index by function name

  [11] BIND_DEBUG           [47] init                 [49] success
  [25] FAIL                 [19] insert_if_have_chil  [33] trust_me
   [6] FIND_CHILD_DEBUG      [5] internal_mcount      [22] try_long_path
  [26] UNTRAIL_DEBUG        [14] jf                   [31] try_me_else
 (164) _mcount              [40] link                 [23] try_short_path
  [30] add_address           [8] loadfd               [24] unify
  [29] add_label             [1] main                 [50] yy_create_buffer
   [4] add_leaf             [16] merg_path            [42] yy_flex_alloc
  [44] allocate_argumets_t  [17] merge                [36] yy_flush_buffer
  [45] allocate_symbol_tab  [48] program_init         [34] yy_get_next_buffer
  [27] backtrack            [15] put_new_internal     [35] yy_get_previous_sta
  [10] enter_node           [12] restore              [37] yy_init_buffer
```

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* McGraw Hill, Cambridge, Mass., second edition, 1996.

[2] Hassan Aït-Kaci. Warren's abstract machine: A tutorial reconstruction. In Koichi Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 939–939. MIT, June 1991.

[3] Victor Eijkhout. Wynter Snow, *TEX for the Beginner. TUGboat*, 13(4):486–487, December 1992.

[4] M. Elhadad. Types in functional unification grammars. -380.

[5] M. Elhadad. *the Universal Unifier User Manual*, June 1993.

[6] Michael Elhadad. FUF: THE UNIVERSAL UNIFIER – USER MANUAL, VERSION 2.0. Technical Report CUCS-489-89, University of Columbia, 1989.

[7] Michael Elhadad. Constraint-based text generation: Using local constraints and argumentation to generate a turn in conversation. Technical Report CUCS-003-90, University of Columbia, 1990.

[8] Michael Elhadad. FUF user manual - version 5.0. Technical Report CUCS-038-91, University of Columbia, 1991.

[9] Michael Elhadad and Jacques Robin. Control in functional unification grammars for text generation. Technical Report CUCS-020-91, University of Columbia, 1991.

[10] Elhadad,M. and and Robin ,J. An overview of SURGE: a reusalble comprehensive syntactic realization component. April 1996.

[11] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer: Second Edition.* Science Research Associates, Inc., Palo Alto, California, 1986.

[12] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages.* McGraw-Hill Book Co., New York, N.Y., 1992.

[13] Simon L. Peyton Jones. *Parallel Graph Reduction*, chapter 24. Prentice-Hall, 1987. ISBN 0-13-453325-9.

[14] Lauri Karttunen. Features and values. *COLING-84*, pages 28–33, 1984.

[15] Robert T. Kasper. A unification method for disjunctive feature descriptions. In *Proceedings of the 25th Annual Meeting*, Stanford, CA, July 1987. Association for Computational Linguistics. Also available as report ISI/RS-87-187, USC/Information Sciences Institute, Marina del Rey, CA.

[16] Martin Kay. Parsing in functional unification grammar. In Karen Sparck-Jones Barbara J. Grosz and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 125–138. Morgan Kaufmann, Los Altos, 1982.

[17] Martin Kay. Functional unification grammar: a formalism for machine translation. *COLING-84*, pages 75–78, 1984.

[18] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1984.

[19] M. Kharitonov. Cfuf. Master's thesis, Ben Gurion university, 1999.

[20] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[21] Donald Ervin Knuth. *The TEXbook*, volume A of *Computers and typesetting.* Addison-Wesley, Reading, MA, USA, 1986.

[22] Kiyoshi Kogure. Strateguc kazy incremental copy graph unification. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 223–228, 1994.

[23] Michael Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards.* PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999. submitted.

[24] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1991.

[25] S. L. Peyton Jones and D. R. Lester. *Implementing Functional Languages: A Tutorial.* Prentice-Hall, Hemel Hempstead, 1992.

[26] Peter Van Roy. 1983–1993: The wonder years of sequential prolog implementation. *Journal of Logic Programming*, 19,20:385–441, 1994.

[27] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford University, Stanford, CA, 1986.

[28] Andrew S. Tanenbaum. *Structured Computer Organisation.* Prentice Hall, Englewood Cliffs, 3 edition, 1990.

Index