

A SYSTEMATIC APPROACH TO SYNTACTIC SUGAR

BY

VANCE PALMER MORRISON

B.S., Northwestern University, 1986

B.S., Northwestern University, 1986

M.S., University of California, Berkeley, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

©Copyright by
Vance Palmer Morrison
1995

The first step in compiling any program is parsing the input text into an equivalent tree structured representation called the abstract syntax. While an impressive theory on parsing has been accumulated, most of it implicitly assumes that the character representation is the canonical form and syntax tree is simply a useful intermediate form. This paper explores the alternative of considering the abstract syntax the canonical representation and the character string format simply useful for input/output. While this general idea is not new, many of its implications have not been explored systematically. Here we attempt to do so by describing the design and use of a program called "BBFront". With this system a concrete syntax can be assigned to a given tree structure, making entering and displaying the tree structure easy. Since all other utilities that manipulate the program can operate on the tree structure, mechanical manipulations of programs as data also become much easier. The system has the additional advantage of allowing each user to define a different syntactic sugar for the same abstract syntax. Thus each user can enter and view programs in the way he likes best yet the code can be shared and exchanged easily since the abstract syntax is common to all.

TABLE OF CONTENTS

1	INTRODUCTION	1
2	THE PROBLEM	4
3	AN EXAMPLE	6
4	A NAIVE APPROACH	8
5	BBFRONT	11
	5.1 Totality & Ambiguity	11
	5.2 Tokenizing	12
6	AN EXAMPLE REVISITED	14
7	CONCLUSION	17
A	SOURCE CODE FOR BBFRONT	18
	BIBLIOGRAPHY	48

1 INTRODUCTION

Although the input to most compilers is a list of characters, it has been known for quite some time that a tree structure is a much more convenient data type for describing a program [9]. In fact, most compilers have the basic structure shown in Figure 1.1. The input string is immediately converted to a tree structure which the back end then processes into the final output.

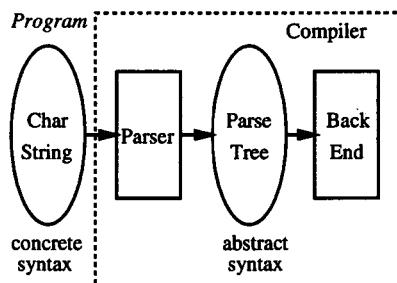


Figure 1.1: A typical compiler layout

The fact that the back end of the compiler only references the tree structure and not the original input string implies that the tree structure is as good (and in many ways better) at representing the desired program as the original input text was. The natural question arises as to which representation should be viewed as the canonical representation of the program, and which should be thought of as simply a convenient intermediate form.

Traditionally the most common operations on programs have been editing and reading by humans. Since text was well suited for editing and displaying, it only made sense to think of a program as a special type of text. With the advent of more sophisticated programming environments, however, this choice becomes dubious. Whereas traditionally the only mechanical operation on a program was compilation, now utilities like syntax directed editors, type checkers, all types of consistency checkers, code librarians, and theorem proving systems all need to manipulate code. Since a tree structure is a much better representation for all of these utilities, it seems better to consider the tree structure the canonical representation of the program and text as simply an input/output form. The resulting architecture is shown in Figure 1.2.

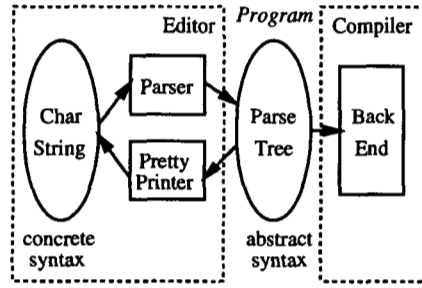


Figure 1.2: The new architecture

This change in philosophy has the immediate benefit of making the concrete syntax independent of the programming language. Now language designers need only concern themselves with the tree structure and not with the details of character based syntax (which always seemed irrelevant anyway).

This idea of treating a program as a tree structure instead of a character string is not new. Certainly the designers of LISP [6] understood the power of considering a program a tree structure. But LISP-like languages did away with character based syntax by adopting a single, regular syntax for representing tree structures. While this works, it does so only at the expense of human factors.

The human factors of a programming language are too important to be sacrificed in this way. Humans will be reading and manipulating programs for quite some time to come, so it only makes sense tailor the input and display of a language to the needs of the human rather than the needs of the machine. The designers of newer programming languages including Ada [2], C++ [3], and Haskell [5] understood this and provided mechanisms for overloading and redefining symbolic operators. Unfortunately the syntactic control offered in these languages is very primitive and ad-hoc. The architecture in Figure 1.2 carries the promise of separating out these human factor concerns from the rest of the programming language design. As an extra bonus, the model can be easily extended so that there are different concrete syntaxes for the same tree structure. Thus every user can read and modify code in the concrete syntax of his choice, even if that code was written using a different concrete syntax.

Consider the code in figure 1.3 which shows a program written in a C, Pascal and Lisp syntax.

<pre> struct List { int first; List* rest; }; List* append(List* x, List* y) { List* ret; if (x == 0) return(y); ret = new List; ret->rest = append(x->rest, y); ret->first = x->first; return(ret); } </pre>	<pre> TYPE List = RECORD first : INTEGER; rest : ^List; END RECORD; function append(IN x:^List IN y:^List):^List; VAR ret: ^List; BEGIN IF x = 0 THEN RETURN(y); END IF; ret := NEW List; ret^.rest := append(x^.rest, y); ret^.first := x^.first; RETURN(ret); END FUNCTION; </pre>	<pre> (defstruct List (first int) (rest (* List))) (defun append ((x (* List)) (y (* List))) (* List) ((local ret (* List)) (if (= x 0) (return y)) (set ret (new List)) (set (get rest ret) (append (get rest x) y)) (set (get first ret) (get first x)) (return ret))) </pre>
--	---	--

Figure 1.3: C, Pascal and Lisp syntax for the same program

These programs are equivalent in the sense that they all will parse to the same abstract syntax tree. Individual programmers, however, will typically have a strong preference toward only one (at most!) of the options and would find the other two syntaxes awkward.

As desirable as the architecture in Figure 1.2 is, it is not without its problems. In particular the new architecture implies the need for an *unparse* (pretty printer) as well as a *parse* function. More importantly, since the tree structure could be generated by other utilities besides the *parse* function, the *unparse* function must be able to accept *any* valid parse tree. In addition it is necessary for *parse* and *unparse* to be consistent with each other. These added constraints make it unclear that traditional parser-generator techniques [1] [4] can be used to solve the problem. Before attempting a solution, however, it is a good idea to state the problem as precisely as possible.

2 THE PROBLEM

Let us assume that we are given the abstract syntax of a programming language. This abstract syntax describes the set Γ of all possible tree structures that can represent legal programs. In addition we have the set Σ^* of all possible character strings. What we wish to define is a partial function $parse : \Sigma^* \dashrightarrow \Gamma$, and total function $unparse : \Gamma \rightarrow \Sigma^*$. These functions must have the following formal property.

$$\forall x \in \Gamma \ parse(unparse(x)) = x \tag{2.1}$$

In addition, it would be desirable that the functions have the following informal properties

1. The description that generated $parse$ and $unparse$ should be short and intuitive for “typical” programming languages.
2. For all $x \in \Gamma$ $unparse(x)$ produces something that is readable.
3. If Δx is a small perturbation of $x \in \Gamma$ then $unparse(\Delta x)$ is a small perturbation of $unparse(x)$. (If we could formalize what is meant by “small perturbation” the above simply says $unparse$ is continuous).

Notice it is not required that $parse$ and $unparse$ be true inverses of each other. In particular it is *not* required that $\forall x \in \Sigma^* \ unparse(parse(x)) = x$. This is simply too stringent a condition. It would require that $unparse$ be able to return the original program text down to irrelevant details like the whitespace between tokens. This is unacceptable. Instead $unparse$ can return a canonical form of the program and $parse$ is only required to be able to invert these canonical forms.

The first two informal properties are clearly useful properties. The third informal property attempts to capture the idea that the concrete syntax should mirror the abstract syntax and that $parse$ and $unparse$ should only do simple transformations. In particular $parse$ should *not* be used to macro expand a construct into a long sequence of more primitive constructs. While this seems like a very reasonable thing to want to do, it effectively destroys the ability for users to read code written with a different concrete syntax.

For example, suppose user A defines a concrete syntax so that his *parse* function converts a small construct into a long sequence of constructs. Everything is fine for user A since his *unparse* function will convert it back again for viewing. Unfortunately another user, user B, has defined a different concrete syntax and so in all likelihood his *unparse* function will unparse user A's code into some long (and probably much less readable) sequence. This is highly undesirable. If Property 3 holds this scenario can't happen.

On the other hand, Property 3 also prohibits seemingly legitimate uses of macro expansion. For example, the programming language ML [7] [8] is defined by first defining a small "core" language and then defining the full language as a set of macro expansions to the core. This would be impossible in a system enforcing Property 3. The view taken here is that macro expansion as used in defining ML is really a simple form of compilation. As such its proper place is *after* parsing. Thus it is perfectly reasonable to define extensions to a language via macro expansion, but the original and extended languages should be thought of as two distinct languages. Each should have its own tree structure and concrete syntax specification.

3 AN EXAMPLE

The previous section defined the problem, but to make the following discussion more concrete it is useful to have a concrete example. Let us assume that we are trying to build a simple expression evaluator. The concrete syntax of the language can be described informally by the following context free grammar.

```
Exp  → Identifier
     → Number
     → Exp '*' Exp
     → Exp '-' Exp
     → '-' Exp
     → Exp 'mod' Exp
     → '-' Exp
```

Figure 3.1: An Informal Concrete Syntax

In order to describe the abstract syntax of the expression language we first need to agree on how to specify tree structures. For this we need a meta-language that can describe a wide variety of tree-like data structures. While in theory most general purpose programming languages could serve this purpose, it is very convenient to use a first order functional language with built in String and List types. In such a language we might define the abstract syntax as follows.

```
datatype Exp
  id(String)
  num(String)
  apply(String, List(Exp))
end
```

Figure 3.2: An Abstract Syntax

The above code is essentially a declaration of a variant record called “Exp” with three variants, “id”, “num”, and “apply”. The first two variants have one additional field that can hold a string. The last variant has two fields, one holding a string and another holding a list of “Exp” records.

Given the above datatype declaration, it is now possible to describe any tree structure in the syntax of the meta-language. For example, let us assume that in the meta-language lists are described by the notation [elem1,elem2,...]. Then the statement

```
apply('*',[apply('-',[num('3'),num('4')],apply('- -',[id('x')]))])
```

would describe the abstract syntax tree in Figure 3.3.

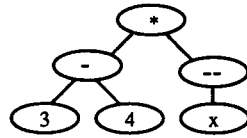


Figure 3.3: Graphical representation

Having given a description of the concrete and abstract syntax, we are in a position to finally look at the original problem. What is needed is a description that can generate *parse* and *unparse* functions so that among other things the following much more compact expression will be parsed into the tree structure in Figure 3.3.

$$(3 - 4) * --x$$

4 A NAIVE APPROACH

An initial attempt to solve the problem might be simply to do what YACC [1] [4] and other LALR parser generators do and simply attach action statements to each of the grammar rules describing the concrete syntax. If we used YACC's \$ convention for naming the value associated with grammar nonterminals, the description of a parser for the example from Figure 3.1 would look like.

```
Exp → Identifier      { id($1)          }
Exp → Number          { num($1)          }
Exp → Exp '*' Exp     { apply('mul', [$1, $3]) }
Exp → Exp '-' Exp     { apply('minus', [$1, $3]) }
Exp → '-' Exp         { apply('dec', [$2])   }
Exp → Exp 'mod' Exp   { apply('mod', [$1, $3]) }
Exp → '-' Exp         { apply('neg', [$2])   }
```

Figure 4.1: A Naive Parser Generator Description

This is not a bad start. Let us assume for the moment that the concrete syntax grammar is unambiguous LALR(1). In that case, the above description certainly defines a *parse* function since we can simply use standard parser-generator techniques to create it. Generating the *unparse* command can be more problematic. Essentially the problem is that functions defined by action statements need to be inverted. Traditional LALR parsers allow arbitrary code in action statements, making this impossible to do algorithmically. If the form of action statements is restricted, however, there is hope that they can be inverted.

Looking at the action statements in the above example, we notice that they consist only of variant descriptors (like "id", "num", and "apply"), string constants (like "mul", "dec" etc), list constants ([] operator), and \$ variables. The functional programming community has known for some time that expressions like these are really patterns that precisely describe how to take apart a data structure as well as how to build it. Thus if action statements are restricted to be patterns, functional programming techniques can be used to run the description backwards.

The process works like this. Each action pattern can be thought of as describing a set of Exp trees. For example, the third action pattern in Figure 4.1 describes the set of all Exp trees

that have a top node being an “apply” variant, the first field being the constant string “mul” and the second field being a list of length two containing arbitrary Exp trees named \$1 and \$3.

Now let us assume that *unparse* was given the tree structure in Figure 3.3. The *unparse* function would scan the list of action patterns and find that this particular tree matches the third action pattern in that grammar if

$$\begin{aligned} \$1 &= \text{apply}(\text{'-'}, [\text{num}(\text{'3'}), \text{num}(\text{'4'})]) \\ \$3 &= \text{apply}(\text{'-'}, [\text{id}(\text{'x'})]) \end{aligned}$$

Thus *unparse* will use the third grammar rule “Exp ‘*’ Exp” to guide the to pretty printing of the expression. First it calls itself recursively to print the multiplicand (bound to \$1) then it will print a “*”, and finally it will print the multiplier (bound to \$3).

So it seems that if action statements are restricted to be patterns, we can borrow well understood techniques to generate *unparse* as well as *parse* functions. In addition, *if the original grammar is unambiguous*, it can be shown that the *parse* and *unparse* functions do indeed have the required property $\forall x \in \Gamma \text{ parse}(\text{unparse}(x)) = x$.

Unfortunately, there are more problems to solve, including:

Insuring *unparse* is a total function: The first argument of “apply” can be any string, but the above description only describes what to do when the first argument is “mul”, “minus”, “dec”, “mod” or “neg”. If the tree structure is the canonical representation of a program, than the description must be augmented to describe what to do for *any* string in that position. Doing this becomes particularly problematic if identifiers can be arbitrary strings since unparsing something like `apply('j-34+*34', [])` must be possible.

Ambiguities in the grammar: The assumption that the grammar is unambiguous is really unacceptable in practice. The grammar in Figure 3.1, for example, is highly ambiguous, and yet this is the way we would like to specify the conversion. Precedence and associativity information can be added to remove the ambiguities, but the simplistic method of generating *unparse* must be modified to take precedence and associativity into account so disambiguating forms (like parentheses) are used when necessary.

Tokenizing: The grammar in Figure 4.1 assumes that the input stream is already tokenized into things like numbers, identifiers, and keywords. Since the input is really a string of characters, somewhere there must be a description of how to tokenize the input.

Ambiguities in Tokenizing: Tokenizing has its own set of ambiguities. For example, if the tokenizer sees the string “mod”, should it return an identifier or a keyword? The traditional method of simply excluding the keywords from the set of identifiers will not work since each user is free to choose his own set of keywords.

5 BBFRONT

While the above problems are not trivial, they are certainly solvable, and usually in more than one way. Here we present one possible way of arriving at a workable solution by describing a program called ‘BBFront’. BBFront is a parser-generator for a programming language ‘workbench’ called Building Block (hence the BB in BBFront). BBFront was designed to test the feasibility of abstracting concrete syntax from programming language design, as well as to act as a test case for the workbench itself (After all the description of the concrete syntax is a specialized type of programming language). Some of the design decisions made in BBFront may reflect BBFront’s limited scope, but most in fact are good general purpose solutions.

The key to solving many of the problems is simply to adhere closely to the philosophy that the tree structure “is” the language. The immediate implication of this is that it is the character string representation and not the tree structure that must “bend” to resolve any problems.

5.1 Totality & Ambiguity

Following this philosophy, BBFront partitions grammar rules into two types, “canonical” and “shorthand”. The canonical rules must have the following two properties:

1. The set of canonical rules must form an unambiguous, LALR(1) grammar. No precedence or associativity information is allowed.
2. Every possible abstract syntax tree must match the action pattern for at least one of the canonical rules.

Property 1 insures that the *parse* function is well defined by traditional LALR parsing techniques, and Property 2 insures that a *unparse* function is total. Both of these properties are decidable at the time the description is compiled.

While the restrictions imposed on canonical rules will insure a solution, they do so only by *severely* limiting the types of grammars allowed. To allow more flexibility, BBFront allows additional shorthand rules to be added. These rules can be ambiguous as long as precedence and associativity information is included to resolve the ambiguities. There are no restrictions on

the action patterns of shorthand rules other than the implicit restriction that they be patterns and not arbitrary code.

On input the shorthands are simply combined with the canonical grammar to form a larger LALR(1) grammar. If there is a conflict between a canonical and a shorthand rule, it is resolved in favor of the canonical rule. Since canonical rules are likely to be syntactically “ugly”, it is expected that enough shorthands will be defined so that “ugly” canonical forms will be needed rarely. In fact, most existing programming languages can be parsed by BBFront by defining some canonical form and making the entire original grammar a set of shorthands. The canonical form then only serves as a kind of a safety net that insures that every tree structure can be pretty printed unambiguously.

On output the presence of shorthands introduces the possibility that several action patterns will match the same tree structure. While this conflict could be resolved simply by always choosing the canonical pattern, the result is likely to be syntactically ugly and therefore unacceptable. Choosing a particular shorthand is also not acceptable since it may be ambiguous or illegal in the current context. What is done in BBFront is to order all grammar rules according to user friendliness. The *unparse* function keeps track of the current context and chooses the most user friendly form that will parse back into the same tree in the current context.

While it is clear that *unparse* can decide if a grammar rule can be used in a particular context by simply trying it, for LALR(1) parsers there is a better way. Roughly speaking, two things can go wrong that would cause the *parse* command not to return the same tree generated by unparsing a rule. The first possibility is that the parser will see the tokens composing the rule and not reduce it. The second possibility is that some other rule will reduce first and ‘steal’ tokens meant for the rule. In both cases what is of primary concern is the timing of when rules are reduced. For LALR parsers this information is stored in a state table in such a way that a relatively simple analysis can determine if tokens output by a rule will unambiguously parse back to the original tree structure.

5.2 Tokenizing

BBFront takes the traditional approach of defining a set of tokens and describing the parsing process in two stages. The primary problem with this is specifying the set of tokens, and in particular specifying what the keywords of the language are. What is desired is that every time

a quoted string is used in a grammar rule a new keyword is added to the token set. It seems the only way to do this is to treat keywords specially, and it is difficult to treat keywords in this special way unless the entire token set is built into the description language. This is the approach taken in BBFront. While it would be desirable to devise a more flexible approach, most programming languages tokenize characters in the same way so the built-in approach seems at least acceptable in practise.

Whether the tokenizer is built in or not, there is still the problem of ambiguities in the tokenizer. These ambiguities are handled in a very similar way to the way ambiguities are handled at the parsing level. A unambiguous canonical form is chosen for each token as well as a list of shorthands. The only difference is that these rules are built in instead of being user defined.

For example, the canonical form for an identifier is defined as the identifier text surrounded by parentheses. A shorthand of simply the identifier text is also defined. Thus for most identifiers parentheses are not necessary, but if there is a conflict with a keyword (like “mod”), a canonical form exists to fall back on (namely “(mod)”). In BBFront, whitespace is also used to disambiguate tokens and to insure canonical forms are unambiguous. For example the string of tokens consisting of a left parenthesis, the keyword “mod” followed by a right parenthesis would be printed by *unparse* as “(mod)”.

While the above approach can be used even if identifiers are allowed to be arbitrary strings, for readability reasons it is useful to restrict identifiers to be alphanumeric. BBFront supports this possibility by extending the tree structure declaration mechanism so that subsets are supported. This mechanism allows the language designer to state more precisely which syntax trees are truly well formed.

6 AN EXAMPLE REVISITED

Figure 6.1 shows the example from section 3 specified in BBFront’s description language.

```
datatype Exp
  id(Ident)
  num(Digits)
  apply(Ident, List(Exp))
end

Exp      → ident      { id(ident)      }
Exp      → digits     { num(digits)   }
Exp      → sExp       { sExp          }
Exp      → '(' sExp ')' { sExp         }
Exp      → ident '(' (Exp % ',') ')' { apply(ident, LIST(%, Exp)) }

sExp #a.1> → Exp.x '-' Exp.y { apply('minus', [x, y]) }
sExp #a.2> → Exp.x '*' Exp.y { apply('mul', [x, y])   }
sExp #a.2> → Exp.x 'mod' Exp.y { apply('mod', [x, y])   }
sExp #a.3> → '--' Exp.x      { apply('dec', [x])     }
sExp #a.4  → '-' Exp.x       { apply('neg', [x])    }
```

Figure 6.1: A BBFront Description

As was stated in the previous section, BBFront has a built in tokenizer. This tokenizer can return either a keyword, an identifier, or a number. A non-terminals “ident” is built into BBFront and is used in the grammar to represent identifier tokens. This non-terminal has type “Ident” which represents the set of all alphanumeric strings. A similar non-terminal “digits” of type “Digits” is used to represent tokens consisting of strings of digits.

The first part of a BBFront description is simply a description of the abstract syntax. This can be an arbitrarily complicated data structure declaration. Next is the actual grammar description. It is similar to the YACC-like description given in Figure 4.1, but there are differences.

Tags to represent values: Instead of \$1, \$2 ... notation for representing values in the action pattern, the name of the non-terminal is used instead. If the non-terminal name would be ambiguous, a tag can be attached to the non-terminal and the tag can be used instead.

Direct support for lists: Lists are common enough that they deserve special treatment. In particular the notation (Exp % ‘,’) describes a comma separated list of Exp structures. It would be equivalent to a new nonterminal “ExpList” defined as follows.

$$\begin{array}{l}
 \text{ExpList} \rightarrow \text{ExpList1} \quad \{ \text{ExpList1} \quad \} \\
 \text{ExpList1} \rightarrow \text{ExpList1} \quad \{ \text{ExpList1} \quad \} \\
 \text{ExpList1} \rightarrow \text{Exp} \quad \{ [\text{Exp}] \quad \} \\
 \text{ExpList1} \rightarrow \text{Exp} \text{ ‘,’ } \text{ExpList1} \quad \{ \text{Exp}:\text{ExpList1} \quad \}
 \end{array}$$

In addition to the % notation inside the grammar rule, lists also need additional support inside the action pattern. The *LIST* pattern is designed to work with the % term to provide this support. A *LIST* pattern has two arguments. The first argument is the name of a % term. Usually this is simply a % which is the default name for a % term, but a % term can be named to disambiguate it just like non-terminals. The second argument to *LIST* is a pattern whose scope is the body of the % term. (That is only names appearing in the body of the named % term can appear in the pattern). This notation makes specifying lists in the grammar almost as painless as specifying fixed length structures.

Precedence information: There is an optional precedence expression between the nonterminal and the right arrow. This expression consists of a pound sign, a precedence class identifier, a period, a number, and an optional less than or greater than sign. The precedence class identifier limits the scope of the precedence information. Only precedences in the same class are comparable. The number specifies the precedence where a larger number is a higher precedence. The optional less than or greater than sign describes the associativity of the rule. A > means expressions are evaluated left to right and < means expressions are evaluated right to left.

More important than the syntactic improvements of BBFront is the difference in the way the description is interpreted. Notice that action patterns for the first, second, and fifth rule for the non-terminal Exp cover all the possible node variants. These are the canonical rules for the Exp datatype and BBFront checks that canonical rules exist for each datatype. BBFront also insures that the grammar formed by these rules is unambiguous. In this example these checks do indeed hold so BBFront goes on to generate *parse* and *unparse* functions.

The *parse* function is generated using standard LALR(1) techniques. The *unparse* function produced works as follows. Given a *Exp* node it scans the action patterns in order looking for a match. For “id” and “num” variants one of the first two action patterns will match. For “apply” node variants, *unparse* first tries to display it as an unparenthesized infix expression (specified by the third line of the grammar). If that fails, it tries to display it as a parenthesized infix expression (specified by the fourth line). Finally if that fails, it uses standard functional notation (specified by the fifth line).

7 CONCLUSION

Programs can be viewed as characters strings or as tree structures. While it is traditional to view programs as strings, in most cases it is more convenient to consider them tree structures. Since humans will still need to enter and view the program as a string, if this latter viewpoint is taken then both *parse* and *unparse* functions are needed to perform the conversion. In addition *parse* and *unparse* must satisfy certain completeness and consistency properties. Traditional methods of generating parsers do not address these issues.

Here an attempt is made to do so as well as explain how the problems were solved in a program called “BBFront”. BBFront uses a combination of techniques including narrowing the description language, extensions to LALR(1) parser-generator techniques, and functional programming methods to arrive at a workable solution. The result is a program that can specify the concrete and abstract syntax of a wide variety of programming languages in a very compact and readable way. With BBFront a concrete syntax for a tree structure can be generated very quickly and users have the added benefit of tailoring the concrete syntax to their tastes. The result is that language design can take place at the abstract syntax level, reducing the overhead of experimentally inventing and combining languages. BBFront should prove to be an invaluable tool in an entire set of utilities comprising a complete programming language “workbench”.

A SOURCE CODE FOR BBFONT

```

/*****
/*
/*          env.h  env.h
/*
*****/

/* defines the internal form for types */

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91
*****/

#ifndef ENV_DEFINED
#define ENV_DEFINED

/*****
template<class ValueT>
class Env<ValueT> {
    List<Table<String, ValueT>> vars;

public:
    Env<ValueT>() : vars() {}
    Env<ValueT>(Table<String, ValueT>& tab) : vars(tab) {}

    ValueT* find(const String& key) const;
    void push(Table<String, ValueT>& tab) { vars.insert(tab); }

    Table<String, ValueT>& top()          { return(vars.first()); }
    void pop()                          { vars.pop(); }

    friend ostream& operator<<(ostream& out, const Env<ValueT>& env) {
        return(out << env.vars); }
};

/*****
template<class ValueT>
ValueT* Env<ValueT>::find(const String& key) const {

    ValueT* ret;
    ListIter<Table<String, ValueT>> curTab = vars;
    while(!curTab.isEnd()) {
        ret = curTab->find(key);
        if (ret != 0)
            break;

        curTab++;
    }
    return(ret);
}

#endif

/*****

```

```

/*          langinfo.c  langinfo.c          */
/*****

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
/*****

#include <iostream.h>
#include <langinfo.h>

/*****
const Bool nullable(RhsElem& elem, const LangInfo& info) {

    switch(elem.op().kind()) {
        case RhsOp::UNDEF:
            ASSERT(FALSE);
            break;
        case RhsOp::LITERAL:
            return(FALSE);
        case RhsOp::VAR: {
            NonTermDef* varDef = info.def(elem);
            if (varDef == 0)
                return(FALSE); // is a token
            return(info.nullable(*varDef));
        }
        case RhsOp::LIST:
            return(TRUE);
        case RhsOp::OPTIONAL:
            return(TRUE);
    }
}

/*****
/* returns the token associated with 'elem' or 0 if elem does not
   represent a token */

Token* LangInfo::token(const RhsElem& elem) const {

    switch(elem.op().kind()) {
        case RhsOp::UNDEF:
            ASSERT(FALSE);
            break;
        case RhsOp::LITERAL:
            return(&literal(elem.op().literal()));
        case RhsOp::VAR:
            return(tokenDesc.values.find(elem.op().var()));
        case RhsOp::LIST:
        case RhsOp::OPTIONAL:
            return(0);
    }
}

/*****
/* return true if the rest of the list 'elems' is nullable */

const Bool nullable(ListIter<RhsElem> curElem, const LangInfo& info) {

```

```

while(!curElem.isEnd()) {
    if (!nullable(*curElem, info)) {
        return(FALSE);
    }
    curElem++;
}
return(TRUE);
}

/*****
/* updates 'firstSet' so that it includes all the possible first tokens
of the element list 'elems'. It returns 'TRUE' if 'firstSet' is
changed */

const Bool first(RhsElem& elem, Set<Token>& firstSet, const LangInfo& info) {

    Bool ret = FALSE;
    switch(elem.op().kind()) {
        case RhsOp::UNDEF:
            ASSERT(FALSE);
            break;
        case RhsOp::LITERAL: {
            return(firstSet.insert(info.literal(elem.op().literal())));
        }
        case RhsOp::VAR: {
            NonTermDef* varDef = info.def(elem);
            if (varDef == 0)
                return(firstSet.insert(*info.token(elem)));
            return(firstSet.sum(info.first(*varDef)));
        }
        case RhsOp::LIST:
            ret = first(elem.op().list().body(), firstSet, info);
            if (elem.op().list().sep().isDefined())
                ret = first(*elem.op().list().sep(), firstSet, info) || ret;
            return(ret);
        case RhsOp::OPTIONAL:
            return(first(elem.op().optional(), firstSet, info));
    }
}

/*****
/* updates 'firstSet' so that it includes all the possible first tokens
of the element list 'elems'. It returns 'TRUE' if 'firstSet' is
changed */

const Bool first(ListIter<RhsElem> curElem, Set<Token>& firstSet,
const LangInfo& info) {

    Bool ret = FALSE;
    while(!curElem.isEnd()) {
        if (first(*curElem, firstSet, info))
            ret = TRUE;
        if (!nullable(*curElem, info))
            break;
        curElem++;
    }
}

```

```

    return(ret);
}

/*****
void LangInfo::compNonTermInfo(List<NonTermDef>& syntax) {

    // compute the nullable function for the variables
    // define initial attributes for all variables
    ListIter<NonTermDef> curNonTerm = syntax;
    while(!curNonTerm.isEnd()) {
        nonTermInfo.update(
            toObject(*curNonTerm),
            *(new NonTermInfo(*curNonTerm)));
        curNonTerm++;
    }

    Bool changed = FALSE;
    do {
        changed = FALSE;
        ListIter<NonTermDef> curNonTerm = syntax;
        while(!curNonTerm.isEnd()) {
            NonTermInfo* curInfo = nonTermInfo.find(toObject(*curNonTerm));
            ASSERT(curInfo != 0);

            ListIter<Rule> curRule = curNonTerm->rules();
            while(!curRule.isEnd()) {
                Bool newVal = ::nullable(curRule->rhs(), *this);
                if (newVal && ! curInfo->nullable) {
                    curInfo->nullable = newVal;
                    changed = TRUE;
                }
                if (::first(curRule->rhs(), curInfo->first, *this))
                    changed = TRUE;
                curRule++;
            }
            curNonTerm++;
        }
    } while(changed);
}

/*****
void LangInfo::makeList(RhsElem& elem, Pattern& pat) {

    ASSERT(elem.op().kind() == RhsOp::LIST);
    ASSERT(pat.kind() == Pattern::LIST);

    RhsElem& listRef = *(new RhsElem((new RhsOp)->setVar(newString("[1]"))));

    Rule& list0Rule1=(new Rule(*(new List<RhsElem>),*(new Pattern)));
    Rule& list0Rule2=(new Rule(*(new List<RhsElem>(listRef)),*(new Pattern)));

    Rule& list1Rule1 = *(new Rule(*(new List<RhsElem>),*(new Pattern)));
    list1Rule1.rhs().append(elem.op().list().body());

    Rule& list1Rule2 = *(new Rule(*(new List<RhsElem>),*(new Pattern)));
    list1Rule2.rhs().append(listRef);
}

```

```

if (elem.op().list().sep().isDefined())
    list1Rule2.rhs().append(*elem.op().list().sep());
list1Rule2.rhs().append(elem.op().list().body());

Type* listType = new Type;
NonTermDef& list0 = *(new NonTermDef(newString("[]"),
    *(new List<Rule>(list0Rule1, list0Rule2)),
    *listType)); // will be filled in by typechecking

NonTermDef& list1 = *(new NonTermDef(newString("[1]"),
    *(new List<Rule>(list1Rule1, list1Rule2)),
    *listType)); // will be filled in by typechecking

nonTerms.update(toObject(elem), list0);
nonTerms.update(toObject(listRef), list1);

actions.update(toObject(list0Rule1),
    *(new Action(list0, list0Rule1, 0, (new PatAction)->setList0())));
actions.update(toObject(list0Rule2),
    *(new Action(list0, list0Rule2, 1, (new PatAction)->setStack(0))));

actions.update(toObject(list1Rule1),
    *(new Action(list1, list1Rule1, list1Rule1.rhs().length(),
    (new PatAction)->setListCons(*(new ListAction(
    *makePatAction(pat.list().body(), *this),
    (new PatAction)->setList0()))))););

actions.update(toObject(list1Rule2),
    *(new Action(list1, list1Rule2, list1Rule2.rhs().length(),
    (new PatAction)->setListApp(*(new ListAction(
    *makePatAction(pat.list().body(), *this),
    (new PatAction)->setStack(list1Rule2.rhs().length()-1))))))););
}

/*****
void LangInfo::makeOptional(RhsElem& elem, Pattern& pat) {

    ASSERT(elem.op().kind() == RhsOp::OPTIONAL);
    ASSERT(pat.kind() == Pattern::OPTIONAL);

    Rule& optRule1=*(new Rule(*(new List<RhsElem>),*(new Pattern)));
    Rule& optRule2=*(new Rule(*(new List<RhsElem>),*(new Pattern)));
    optRule2.rhs().append(elem.op().optional());

    NonTermDef& optional = *(new NonTermDef(newString("<>"),
        *(new List<Rule>(optRule1, optRule2)),
        *(new Type)); // will be filled in by typechecking

    nonTerms.update(toObject(elem), optional);

    actions.update(toObject(optRule1),
        *(new Action(optional, optRule1, 0, (new PatAction)->setOpt0())));

    actions.update(toObject(optRule2),
        *(new Action(optional, optRule2, optRule2.rhs().length(),
        (new PatAction)->setOpt1(
        *makePatAction(pat.optional().body(), *this)))));

```

```

}

/*****
/* make auxillary nonterminal definitions (for list and optional productions)*/

void LangInfo::makeAux() {
    TableIter<Object<Pattern>, PatInfo> cur = patInfo;
    while(!cur.isEnd()) {
        Pattern& pat = fromObject(cur->key());
        switch(pat.kind()) {
            case Pattern::UNDEF:
            case Pattern::SUM:
            case Pattern::PRODUCT:
            case Pattern::VAR:
            case Pattern::LITERAL:
            case Pattern::TYPEDEC:
            case Pattern::LISTNIL:
            case Pattern::LISTCONS:
            case Pattern::OPTNIL:
            case Pattern::OPTVAL:
                break;
            case Pattern::LIST:
                makeList(cur->value().def(), pat);
                break;
            case Pattern::OPTIONAL:
                makeOptional(cur->value().def(), pat);
                break;
        }
        cur++;
    }
}

/*****
void LangInfo::makeRules(List<NonTermDef>& syntax) {

    ListIter<NonTermDef> curNonTerm = syntax;
    while(!curNonTerm.isEnd()) {
        ListIter<Rule> curRule = curNonTerm->rules();
        while(!curRule.isEnd()) {
            actions.update(toObject(*curRule),
                *makeAction(*curNonTerm, *curRule, *this));
            curRule++;
        }
        curNonTerm++;
    }
}

/*****
LangInfo::LangInfo(List<NonTermDef>& syntax, Env<Sort>& env,
    TokenDesc& tokDesc, GenStatus& status) : sorts(env), tokenDesc(tokDesc) {

    status.success(); // assume success

    defNonTerms(syntax, nonTerms, tokenDesc.values, status);
    if (!status.isSuccess())
        return;
}

```

```

    compNonTermInfo(syntax);

    defPats(syntax, patInfo, status);
    if (!status.isSuccess())
        return;

    typeCheck(syntax, *this, status);
    if (!status.isSuccess())
        return;

    makeAux();

    makeRules(syntax);
}

/*****
NonTermDef* makeTopRule(NonTermDef& start, LangInfo& info) {

    RhsElem* startElem = new RhsElem((new RhsOp)->setVar(start.name()));

    Rule* topRule = new Rule(*(new List<RhsElem>(*startElem)), *(new Pattern));

    NonTermDef* topNT = new NonTermDef(
        newString("$START$"),
        *(new List<Rule>(*topRule)),
        *(new Type));

    // link the new nonterminal into the info structure
    info.nonTerms.update(toObject(*startElem), start);
    info.actions.update(toObject(*topRule), *(new Action(*topNT, *topRule, 1)));

    return(topNT);
}

/*****
/*          langinfo.h          langinfo.h          */
/*****

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
/*****

#ifndef LANGINFO_DEFINED
#define LANGINFO_DEFINED 1

/*****
/*****
class NonTermInfo {
public:
    NonTermDef* def_;
    State* start_;
    Sort* sort;
    Bool nullable;
    Set<Token> first;
public:
    void setStart(State& aState)    { start_ = &aState; }
    State& start()                  { ASSERT(start_ != 0); return(*start_); }
}

```

```

const NonTermDef& def() const { return(*def_); }
NonTermDef& def() { return(*def_); }

NonTermInfo(NonTermDef& aNonT)
    : def_(&aNonT), start_(0), sort(0), nullable(FALSE), first() {}

friend ostream& operator<<(ostream& out, const NonTermInfo& info);
friend int cmp(const NonTermInfo& info1, const NonTermInfo& info2)
    { ASSERT(0); USE2(info1, info2); return(0); }

friend class LangInfo;
};

/*****
class LangInfo {
    Table<Object<RhsElem>, NonTermDef> nonTerms;
    Table<Object<NonTermDef>, NonTermInfo> nonTermInfo;
    Table<Object<Pattern>, PatInfo> patInfo;

    Table<Object<Rule>, Action> actions;
    Table<Object<Pattern>, Nat> sumands;

public:
    Env<Sort>& sorts;
    TokenDesc& tokenDesc;

private:
    void compNonTermInfo(List<NonTermDef>& syntax);
    void makeAux();
    void makeOptional(RhsElem& elem, Pattern& pat);
    void makeList(RhsElem& elem, Pattern& pat);
    void makeRules(List<NonTermDef>& syntax);

public:
    LangInfo(List<NonTermDef>& syntax, Env<Sort>& env, TokenDesc& tokDesc,
             GenStatus& status);

    Token& literal(const String& str) const {
        Token* ret = tokenDesc.literals.find(str);
        ASSERT(ret != 0);
        return(*ret); }

    Token* token(const RhsElem& elem) const;

    NonTermDef* def(const RhsElem& rhsElem) const {
        return(nonTerms.find(toObject(rhsElem))); }

    Sort& sort(const NonTermDef& def) const {
        NonTermInfo* ret = nonTermInfo.find(toObject(def));
        ASSERT(ret != 0);
        return(*ret->sort);
    }

    Sort& sort(const RhsElem& elem) const {

```

```

NonTermDef* elemDef = nonTerms.find(toObject(elem));
if (elemDef == 0) { // must be the name of a token
    Sort* ret = tokenDesc.types.find(elem.op().var());
    ASSERT(ret != 0);
    return(*ret);
}
else
    return(sort(*elemDef));
}

void setSort(NonTermDef& def, Sort& sort) {
    NonTermInfo* info = nonTermInfo.find(toObject(def));
    if (info == 0) {
        info = new NonTermInfo(def);
        info->sort = &sort;
        nonTermInfo.update(toObject(def), *info);
    }
    else
        info->sort = &sort;
}

RhsElem& def(const Pattern& pat) const {
    PatInfo* info = patInfo.find(toObject(pat));
    ASSERT(info != 0);
    return(info->def()); }

Nat rhsPosition(Pattern& pat) const {
    PatInfo* info = patInfo.find(toObject(pat));
    ASSERT(info != 0);
    return(info->pos()); }

Nat sumand(Pattern& pat) const {
    Nat* ret = sumands.find(toObject(pat));
    ASSERT(ret != 0);
    return(*ret); }

Action& action(Rule& rule) const {
    Action* ret = actions.find(toObject(rule));
    ASSERT(ret != 0);
    return(*ret); }

Set<Token>& first(NonTermDef& nonTerm) const {
    NonTermInfo* const ret = nonTermInfo.find(toObject(nonTerm));
    ASSERT(ret != 0);
    return(ret->first); }

const Bool nullable(NonTermDef& nonTerm) const {
    NonTermInfo* const ret = nonTermInfo.find(toObject(nonTerm));
    ASSERT(ret != 0);
    return(ret->nullable); }

friend NonTermDef* makeTopRule(NonTermDef& start, LangInfo& info);

friend ostream& operator<<(ostream& out, const LangInfo& info);

friend void typeCheckSum(Pattern& pat, Sort& expectSort,
    LangInfo& info, GenStatus& status);

```

```

};

/*****
const Bool nullable(ListIter<RhsElem> curElem, const LangInfo& info);
const Bool first(ListIter<RhsElem> elems, Set<Token>& firstSet,
    const LangInfo& info);

void typeCheck(List<NonTermDef>& syntax, LangInfo& info, GenStatus& status);

#endif

/*****
/*          main.c          main.c          */
/*****

/* AUTHOR: Vance Morrison
   DATE  : 10/25/91          */
/*****

#include <iostream.h>
#include <language.h>
#include <cllexer.h>
#include <literals.h>
#include <langinfo.h>

int main() {
    extern int newVerbose;

    cout << "In main\n";

    SyntaxDesc& myLang = *descLang();

    Set<String> myLiterals;
    literals(myLang.syntax(), myLiterals);
    cout << "***** Literals in the Language\n";
    cout << myLiterals << "\n";

    TokStatus tokStatus;
    Env<Sort>& sortEnv = *Sort::predefined();
    CTokenDesc myDesc(myLiterals, sortEnv, tokStatus);
    if(!tokStatus.isSuccess()) {
        cout << "Illegal literals\n";
        tokStatus.putErrorMsg(cout) << "\n";
        return(-1);
    }
    cout << "Description of Lexer-parser interface\n";
    cout << myDesc << "\n";
    cout.flush(); cout.flush();

    cout << "***** Defining Type Variables\n";
    SortStatus sortStatus;
    sortEnv.push(*(new Table<Ident, Sort>));
    defTypes(myLang.types(), sortEnv, sortStatus);
    if(!sortStatus.isSuccess()) {
        cout << "Type failure\n";
        sortStatus.putErrorMsg(cout) << "\n";
    }
}

```

```

        return(-1);
    }
    cout << "Success defining types, env = " << sortEnv << "\n";

    cout << "***** Making Langinfo structure\n";
    GenStatus status;
    LangInfo myInfo(myLang.syntax(), sortEnv, myDesc, status);
    if (!status.isSuccess()) {
        cout << "***** Failure making LangInfo\n";
        status.putErrorMsg(cout) << "\n";
        return(-1);
    }

    cout << "***** Success making Langinfo\n";
    Table<StateID, State> states;
    MachStatus machStatus;
    State* start = makeGrammar(myLang.syntax().first(), states, myInfo,
        machStatus);
    cout << "***** States" << "\n";

    if (!machStatus.isSuccess()) {
        cout << "***** Failure making States\n";
        machStatus.putErrorMsg(cout) << "\n";
        return(-1);
    }
    cout << "Success Start state = " << hex << (int) start << dec << "\n";

    cout << "Leaving main\n";
    return(0);
}

/*****
/*          nonterminfo.c          nonterminfo.c          */
/*****

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
/*****

#include <iostream.h>
#include <nonterminfo.h>

/*****
void checkNonTerms(const List<RhsElem>& elems,
    const Table<Ident, NonTermDef>& idTab,
    Table<Object<RhsElem>, NonTermDef>& elemTab,
    Table<Ident, Token>& tokens,
    GenStatus& status);

/*****
void checkNonTerms(RhsElem& elem,
    const Table<Ident, NonTermDef>& idTab,
    Table<Object<RhsElem>, NonTermDef>& elemTab,
    Table<Ident, Token>& tokens,
    GenStatus& status) {

    switch(elem.op().kind()) {

```

```

case RhsOp::UNDEF:
    ASSERT(0);
case RhsOp::LITERAL:
    break;

case RhsOp::VAR:
    NonTermDef* def = idTab.find(elem.op().var());
    if (def == 0) {
        if (tokens.find(elem.op().var()) == 0)
            status.unDefVar(elem.op().var());
        return;
    }
    elemTab.update(toObject(elem), *def);
    break;
case RhsOp::LIST:
    checkNonTerms(elem.op().list().body(), idTab, elemTab, tokens, status);
    break;

case RhsOp::OPTIONAL:
    checkNonTerms(elem.op().optional(), idTab, elemTab, tokens, status);
    break;
}
}

```

```

/*****/
void checkNonTerms(const List<RhsElem>& elems,
                  const Table<Ident, NonTermDef>& idTab,
                  Table<Object<RhsElem>, NonTermDef>& elemTab,
                  Table<Ident, Token>& tokens,
                  GenStatus& status) {

    ListIter<RhsElem> curElem = elems;
    while(!curElem.isEnd()) {
        checkNonTerms(*curElem, idTab, elemTab, tokens, status);
        if (!status.isSuccess())
            return;
        curElem++;
    }
}

```

```

/*****/
void defNonTerms(List<NonTermDef>& syntax,
                 Table<Object<RhsElem>, NonTermDef>& elemTab,
                 Table<Ident, Token>& tokens,
                 GenStatus& status) {

    status.success(); // assume success
    Table<Ident, NonTermDef> idTab;

    // define all the nonTerminal names
    ListIter<NonTermDef> curDef = syntax;
    while(!curDef.isEnd()) {
        NonTermDef* prevDef = idTab.find(curDef->name());
        if (prevDef != 0) {
            status.multDefVar(prevDef->name(), curDef->name());
            elemTab.clear();
        }
        return;
    }
}

```

```

    }
    idTab.update(curDef->name(), *curDef);
    curDef++;
}

// check to see if all reference are defined
curDef.reset(syntax);
while(!curDef.isEnd()) {
    ListIter<Rule> curRule = curDef->rules();
    while(!curRule.isEnd()) {
        checkNonTerms(curRule->rhs(), idTab, elemTab, tokens, status);
        if (!status.isSuccess()) {
            elemTab.clear();
            return;
        }
        curRule++;
    }
    curDef++;
}
}

/*****
/*          nonterminfo.h    nonterminfo.h          */
*****/

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
*****/

#ifndef NONTERMINFO_DEFINED
#define NONTERMINFO_DEFINED 1

/*****
void defNonTerms(List<NonTermDef>& syntax,
                 Table<Object<RhsElem>, NonTermDef>& elemTab,
                 Table<Ident, Token>& tokens,
                 GenStatus& status);

#endif
*****/

/*          patinfo.c    patinfo.c          */
*****/

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
*****/

#include <iostream.h>
#include <patinfo.h>

/*****
const String* defaultName(RhsOp& op) {

    switch(op.kind()) {
        case RhsOp::UNDEF:
            ASSERT(0);
            break;

```

```

    case RhsOp::LITERAL:
        return(0);
    case RhsOp::VAR:
        return(&op.var());
    case RhsOp::LIST:
        return(&string("[]"));
    case RhsOp::OPTIONAL:
        return(&string("<>"));
    }
}

/*****/
void defRhsElem(RhsElem& elem, Table<String, RhsElem>& tab) {

    const String* name;
    if (elem.tag().isDefined())
        name = &*elem.tag();
    else {
        name = defaultName(elem.op());
        if (name == 0)
            return;
    }
    tab.update((String&) *name, elem);
}

/*****/
void defRhsElems(List<RhsElem>& elems, Table<String, RhsElem>& tab) {

    ListIter<RhsElem> curElem = elems;
    while(!curElem.isEnd()) {
        defRhsElem(*curElem, tab);
        curElem++;
    }
}

/*****/
void defPat(Pattern& pat, List<RhsElem>& elems,
            Table<Object<Pattern>, PatInfo>& retTab, GenStatus& status);

/*****/
/* link up all the definitions of variables, list, and optionals with in
'pat' with the the RhsElem's in 'elems' they coorespond to */

void defPat(Pattern& pat, List<RhsElem>& elems,
            Table<String, RhsElem>& rhsTab, Table<String, String>& patTab,
            Table<Object<Pattern>, PatInfo>& retTab, GenStatus& status) {

    status.success(); // asume success

    String* name;
    switch(pat.kind()) {
        case Pattern::UNDEF:
            ASSERT(0);
            return;
        case Pattern::SUM:
            defPat(pat.sum().body(), elems, rhsTab, patTab, retTab, status);
            return;
    }
}

```

```

case Pattern::PRODUCT: {
    int ret = 0;
    ListIter<PatField> curField = pat.product();
    while(!curField.isEnd()) {
        defPat(curField->field(),elems, rhsTab, patTab, retTab, status);
        if (!status.isSuccess())
            return;
        curField++;
    }
    return;
}
case Pattern::LIST:
    if (pat.list().name().isDefined())
        name = &*pat.list().name();
    else
        name = &string("[]");
    break;
case Pattern::LISTNIL:
    return;
case Pattern::LISTCONS:
    defPat(pat.listCons().first(),elems,rhsTab,patTab,retTab,status);
    if (!status.isSuccess())
        return;
    defPat(pat.listCons().rest(),elems,rhsTab,patTab,retTab,status);
    return;
case Pattern::OPTIONAL:
    if (pat.optional().name().isDefined())
        name = &*pat.optional().name();
    else
        name = &string("<>");
    break;
case Pattern::OPTNIL:
    return;
case Pattern::OPTVAL:
    defPat(pat.optVal(),elems,rhsTab,patTab,retTab,status);
    return;
case Pattern::VAR:
    name = &pat.var();
    break;
case Pattern::LITERAL:
    return;
case Pattern::TYPEDEC:
    defPat(pat.typeDec().body(), elems, rhsTab, patTab, retTab, status);
    return;
}

String* prevDef = patTab.find(*name);
if (prevDef != 0) {
    status.multUses(*prevDef, *name);
    return;
}
RhsElem* elem = rhsTab.find(*name);
if (elem == 0) {
    status.unDefVar(*name);
    return;
}

```

```

Nat pos = elems.length();
ListIter<RhsElem> cur = elems;
for(;;) {
    --pos;
    ASSERT(!cur.isEnd());
    if (&*cur == elem)
        break;
    cur++;
}

retTab.update(toObject(pat), *(new PatInfo(*elem, pos)));
patTab.update(*name, *name);
rhsTab.remove(*name);

switch(pat.kind()) {
    case Pattern::UNDEF:
    case Pattern::SUM:
    case Pattern::PRODUCT:
    case Pattern::VAR:
    case Pattern::LITERAL:
    case Pattern::TYPEDEC:
    case Pattern::LISTNIL:
    case Pattern::LISTCONS:
    case Pattern::OPTNIL:
    case Pattern::OPTVAL:
        break;
    case Pattern::LIST:
        defPat(pat.list().body(), elem->op().list().body(), retTab, status);
        break;
    case Pattern::OPTIONAL:
        defPat(pat.optional().body(), elem->op().optional(), retTab, status);
        break;
}
}

/*****/
void defPat(Pattern& pat, List<RhsElem>& elems,
            Table<Object<Pattern>, PatInfo>& retTab, GenStatus& status) {

    Table<String, String> patTab;
    Table<String, RhsElem> rhsTab;

    defRhsElems(elems, rhsTab);

    defPat(pat, elems, rhsTab, patTab, retTab, status);
    if (!status.isSuccess())
        return;

    TableIter<String, RhsElem> rhsLeft = rhsTab;
    if (!rhsLeft.isEnd())
        status.notUsed(rhsLeft->value());
}

/*****/
void defPats(List<NonTermDef>& syntax, Table<Object<Pattern>, PatInfo>& patTab,
            GenStatus& status) {

```

```

status.success(); // asume success

ListIter<NonTermDef> curNonTerm = syntax;
while(!curNonTerm.isEnd()) {
    ListIter<Rule> curRule = curNonTerm->rules();
    while(!curRule.isEnd()) {
        defPat(curRule->action(), curRule->rhs(), patTab, status);
        if (!status.isSuccess())
            return;
        curRule++;
    }
    curNonTerm++;
}
}
/*****
/*          patinfo.h          patinfo.h          */
*****/

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91          */
*****/

#ifndef PATINFO_DEFINED
#define PATINFO_DEFINED 1

class ostream;
class PatInfo;

/*****
class PatInfo {
    RhsElem *def_;
    Nat pos_;
public:
    PatInfo(RhsElem& aDef, Nat aPos) : def_(&aDef), pos_(aPos) {}

    const RhsElem& def() const          { return(*def_); }
    RhsElem& def()                      { return(*def_); }
    const Nat pos() const              { return(pos_); }
    Nat pos()                          { return(pos_); }

    friend ostream& operator<<(ostream& out, const PatInfo& info);
    friend int cmp(const PatInfo& info1, const PatInfo& info2)
        { ASSERT(0); USE2(info1, info2); return(0); }
};

void defPats(List<NonTermDef>& syntax,
             Table<Object<Pattern>, PatInfo>& patTab,
             GenStatus& status);

#endif

/*****
/*          sort.c          sort.c          */
*****/

/* defines the internal form for types */

```

```

/* AUTHOR: Vance Morrison
   DATE  : 12/16/91           */
/*****

#include <iostream.h>
#include <sort.h>
#include <obj.h>

/*****
Env<Sort>* Sort::predefined() {

    static Env<Sort>* ret = 0;

    if (ret == 0) {
        Table<Ident, Sort>* tab = new Table<Ident, Sort>;

        Sort& stringSort = (new Sort)->setString();
        tab->update(newString("String"), stringSort);

        Sort& identSort = (new Sort)->setSubset(
            *(new SortSubset(stringSort, identFilter)));
        tab->update(newString("Ident"), identSort);

        Sort& digitsSort = (new Sort)->setSubset(
            *(new SortSubset(stringSort, digitsFilter)));
        tab->update(newString("Digits"), digitsSort);

        ret = new Env<Sort>(*tab);
    }

    return(ret);
}

/*****
void checkVars(List<TypeField>& fields, const Env<Sort>& env, SortStatus& status){

    // check that field names are unique
    ListIter<TypeField> curField = fields;
    while(!curField.isEnd()) {
        ListIter<TypeField> curCheck = curField;
        if (curField->name().isDefined()) {
            curCheck++;
            while (!curCheck.isEnd()) {
                if (curCheck->name().isDefined()) {
                    if (*curCheck->name() == *curField->name()) {
                        status.multFields(*curField->name(), *curCheck->name());
                        return;
                    }
                }
                curCheck++;
            }
        }
        curField++;
    }

    // check the interior fields

```

```

    curField.reset(fields);
    while(!curField.isEnd()) {
        checkVars(curField->type(), env, status);
        if (!status.isSuccess())
            return;
        curField++;
    }
}

/*****/
void checkVars(List<TypeSumand>& sums,const Env<Sort>& env, SortStatus& status){

    // check that field names are unique
    ListIter<TypeSumand> curSumand = sums;
    while(!curSumand.isEnd()) {
        ListIter<TypeSumand> curCheck = curSumand;
        curCheck++;
        while (!curCheck.isEnd()) {
            if (curCheck->name() == curSumand->name()) {
                status.multFields(curSumand->name(),curCheck->name());
                return;
            }
            curCheck++;
        }
        curSumand++;
    }

    // check the interior sums
    curSumand.reset(sums);
    while(!curSumand.isEnd()) {
        checkVars(curSumand->type(), env, status);
        if (!status.isSuccess())
            return;
        curSumand++;
    }
}

/*****/
void checkVars(Type& type, const Env<Sort>& env, SortStatus& status) {

    switch(type.kind()) {
        case Type::UNDEF:
            break;

        case Type::SUM:
            checkVars(type.sum(), env, status);
            break;

        case Type::PRODUCT:
            checkVars(type.product(), env, status);
            break;

        case Type::LIST:
            checkVars(type.list(), env, status);
            break;

        case Type::OPTIONAL:

```

```

        checkVars(type.optional(), env, status);
        break;

    case Type::VAR:
        Sort* def = env.find(type.var());
        if (def == 0) {
            status.unDefVar(type.var());
            return;
        }
        break;
    }
}

/*****
void defTypes(List<TypeDef>& defs, Env<Sort>& env, SortStatus& status) {

    status.success();                // assume success

    // define all the type names
    ListIter<TypeDef> curDef = defs;
    while(!curDef.isEnd()) {
        Sort* def = env.top().find(curDef->name());
        if (def != 0) {
            status.multDefVar(curDef->name(), curDef->name());
            return;
        }
        env.top().update(curDef->name(), (Sort&) curDef->type());
        curDef++;
    }

    // check to see if all reference are defined
    curDef.reset(defs);
    while(!curDef.isEnd()) {
        checkVars(curDef->type(), env, status);
        if (!status.isSuccess()) {
            return;
        }
        curDef++;
    }
}

/*****
/* tries to write some sort of error message to 'out'. Unfortunately,
the object at the focus of the error message may be hard to print
or ambiguous, so this routine should only be used when no other
alternative is possible */

ostream& SortStatus::putErrorMsg(ostream& out) {

    out << message();
    switch(kind_) {
        case SortStatus::UNDEF:
        case SortStatus::SUCCESS:
            break;

        case SortStatus::MULTFIELD:
        case SortStatus::MULTDEFVAR:

```

```

        case SortStatus::UNDEFVAR:
            out << " " << *((String *) object1_) << " ";
            break;
        }
    return(out);
}

/*****
const char* const SortStatus::messages[] = {
    "Undefined Error",
    "Success",
    "Multiply defined Variable",
    "Multiply defined Field",
    "Undefined variable"
};

/*****
/*
sort.h sort.h
*/
/*****

/* defines the internal form for types */

/* AUTHOR: Vance Morrison
DATE : 10/23/91 */
/*****

#ifndef SORT_DEFINED
#define SORT_DEFINED

/*****
class SortStatus {
public:
    enum SortStatusKind { UNDEF, SUCCESS, MULTDEFVAR, MULTFIELD, UNDEFVAR };

private:
    SortStatusKind kind_;

    const void* object1_;
    const void* object2_;
    const void* object3_;

    static const char* const messages[];
public:
    SortStatus() { kind_ = UNDEF; }
    void success() { kind_ = SUCCESS; }
    Bool isSuccess() { return(kind_ == SUCCESS); }

    void multDefVar(const String& firstDef, const String& curDef)
        { object1_ = (const void *) &firstDef;
          object2_ = (const void *) &curDef;
          kind_ = MULTDEFVAR; }

    void multFields(const String& firstDef, const String& curDef)
        { object1_ = (const void *) &firstDef;
          object2_ = (const void *) &curDef;
          kind_ = MULTFIELD; }

```

```

void unDefVar(const String& name)
    { object1_ = (const void *) &name;
      kind_ = UNDEFVAR; }

    // returns just the error message (no object or position info
const String& message() {return(string(messages[kind_])); }

ostream& putErrorMsg(ostream& out); // writes a complete error message
friend class GenStatus;
};

#endif

/*****
class Sort {
public:
    enum SortKind { UNDEF, SUM, PRODUCT, LIST, OPTIONAL, VAR,
                   STRING, SUBSET };

private:
    SortKind kind_;
    union {
        List<SortSumand>* sum_;
        List<SortField>* product_;
        Sort* list_;
        Sort* optional_;
        SortSubset* subset_;
        Ident* var_;
    };

public:
    Sort() { kind_ = UNDEF; }

    Sort& setSum(List<SortSumand>& aSum) { kind_ = SUM;
                                         sum_ = &aSum;
                                         return(*this); }

    Sort& setProduct(List<SortField>& aProduct) { kind_ = PRODUCT;
                                                product_ = &aProduct;
                                                return(*this); }

    Sort& setList(Sort& aList) { kind_ = LIST;
                                list_ = &aList;
                                return(*this); }

    Sort& setOptional(Sort& aOptional) { kind_ = OPTIONAL;
                                       optional_ = &aOptional;
                                       return(*this); }

    Sort& setVar(Ident& aVar) { kind_ = VAR;
                               var_ = &aVar;
                               return(*this); }

    Sort& setString() { kind_ = STRING;
                       return(*this); }

    Sort& setSubset(SortSubset& aSubset) { kind_ = SUBSET;
                                          subset_ = &aSubset;
                                          return(*this); }

    SortKind kind() const { return(kind_); }

```

```

const List<SortSumand>& sum() const      { ASSERT(kind_ == SUM);
List<SortSumand>& sum()                  { ASSERT(kind_ == SUM);
const List<SortField>& product() const  { ASSERT(kind_ == PRODUCT);
List<SortField>& product()              { ASSERT(kind_ == PRODUCT);
const Sort& list() const               { ASSERT(kind_ == LIST);
Sort& list()                           { ASSERT(kind_ == LIST);
const Sort& optional() const           { ASSERT(kind_ == OPTIONAL);
Sort& optional()                       { ASSERT(kind_ == OPTIONAL);
const Ident& var() const               { ASSERT(kind_ == VAR);
Ident& var()                           { ASSERT(kind_ == VAR);
const SortSubset& subset() const       { ASSERT(kind_ == SUBSET);
SortSubset& subset()                   { ASSERT(kind_ == SUBSET);

friend void checkVars(Type& type, const Env<Sort>& env, SortStatus& status);

friend Sort& toSort(Type& def, const Env<Sort>& env, SortStatus& sortStatus){
    sortStatus.success();           // assume success
    checkVars(def, env, sortStatus);
    return((Sort&) def);
}

friend void checkType(Type& def, Env<Sort>& env, SortStatus& status);
friend void defTypes(List<TypeDef>& defs, Env<Sort>& env,
    SortStatus& status);

static Env<Sort>* predefined();

friend ostream& operator<<(ostream& out, const Sort& typeDef);
friend int cmp(const Sort& type1, const Sort& type2)
    { ASSERT(0); USE2(type1, type2); return(0); }
};

/*****/
typedef Bool (*FilterType)(Obj&);

/*****/
class SortSubset {
    Sort* sort_;
    FilterType filter_;

public:
    SortSubset(Sort& aSort, Bool (*filter)(Obj&)
        : sort_(&aSort), filter_(filter) {}

```

```

    const Sort& sort() const           { return(*sort_); }
    Sort& sort()                       { return(*sort_); }

    FilterType filter() const         { return(filter_); }

public:
    friend ostream& operator<<(ostream& out, const Sort& sort);
    friend class Sort;
};

/*****
class SortSumand {
    Ident* name_;
    Sort* sort_;

public:
    SortSumand(Ident& aName, Sort& aSort) : name_(&aName), sort_(&aSort) {}

    const Ident& name() const          { return(*name_); }
    Ident& name()                      { return(*name_); }
    const Sort& sort() const           { return(*sort_); }
    Sort& sort()                       { return(*sort_); }

    friend ostream& operator<<(ostream& out, const SortSumand& sum);
    friend int cmp(const SortSumand& sum1, const SortSumand& sum2)
        { ASSERT(0); USE2(sum1, sum2); return(0); }
};

/*****
class SortField {
    Optional<Ident> name_;
    Sort* sort_;

public:
    SortField(Sort& aSort) : name_(), sort_(&aSort) {}
    SortField(Ident& aName, Sort& aSort)
        : name_(aName), sort_(&aSort) {}

    const Optional<Ident>& name() const { return(name_); }
    Optional<Ident>& name()              { return(name_); }
    const Sort& sort() const           { return(*sort_); }
    Sort& sort()                       { return(*sort_); }

    friend ostream& operator<<(ostream& out, const SortField& field);
    friend int cmp(const SortField& field1, const SortField& field2)
        { ASSERT(0); USE2(field1, field2); return(0); }
};

/*****
/*                               typecheck.c   typecheck.c                               */
/*****

/* AUTHOR: Vance Morrison
   DATE  : 10/23/91                */
/*****

#include <iostream.h>

```

```

#include <langinfo.h>

void typeCheck(Pattern& pat, Sort& mySort, LangInfo& info, GenStatus& status);

/*****
Sort& normalize(Sort& sort, Env<Sort>& env)
{
    Sort* curSort = &sort;

    for (int i=0; i < 256; i++) {
        if (curSort->kind() != Sort::VAR)
            return(*curSort);

        Sort* nextSort = env.find(curSort->var());
        if (nextSort == 0)
            return(*curSort);

        curSort = nextSort;
    }

    // 256 is arbitrary (and technically wrong), but it is quite
    // unlikely that there will be direct sort synonym chains longer
    // than 256
    ASSERT(0);
    return(*curSort);
}

/*****
const Bool equal(Sort& sort1, Sort& sort2, Env<Sort>& env) {

    Sort& mySort1 = normalize(sort1, env);
    Sort& mySort2 = normalize(sort2, env);

    if (&mySort1 == &mySort2)           // if point to same definition
        return(TRUE);

    if (mySort1.kind() != mySort2.kind())
        return(FALSE);

    switch(mySort1.kind()) {
        case Sort::UNDEF:
            return(TRUE);
        case Sort::SUM:
        case Sort::PRODUCT:
            return(FALSE);           // FIX: Not really correct, but easy
                                     // and almost always OK.
        case Sort::LIST:
            return(equal(mySort1.list(), mySort2.list(), env));
        case Sort::OPTIONAL:
            return(equal(mySort1.optional(), mySort2.optional(), env));
        case Sort::VAR:
            return(FALSE);           // should not happen
        case Sort::STRING:
            return(TRUE);
        case Sort::SUBSET:
            if (mySort1.subset().filter() != mySort2.subset().filter())
                return(FALSE);
    }
}

```

```

    return(equal(mySort1.subset().sort(),mySort2.subset().sort(),env));
}
}

/*****/
void typeCheckVar(Pattern& pat, Sort& expectSort,
    LangInfo& info, GenStatus& status) {

    ASSERT(pat.kind() == Pattern::VAR);

    RhsElem& patRhsElem = info.def(pat);
    switch(patRhsElem.op().kind()) {
        case RhsOp::UNDEF:
            ASSERT(0);
            break;
        case RhsOp::LITERAL:
            if (expectSort.kind() != Sort::STRING)
                status.notExpectedSort(pat, expectSort);
            break;
        case RhsOp::VAR: {
            Sort& varSort = info.sort(patRhsElem);
            if (!equal(expectSort, varSort, info.sorts))
                status.typeConflict(pat, varSort, expectSort);
        } break;
        case RhsOp::LIST:
            status.badRhsElem(pat, patRhsElem);
            break;
        case RhsOp::OPTIONAL:
            status.badRhsElem(pat, patRhsElem);
            break;
    }
}

/*****/
void typeCheckSum(Pattern& pat, Sort& expectSort,
    LangInfo& info, GenStatus& status) {

    ASSERT(pat.kind() == Pattern::SUM);
    ASSERT(expectSort.kind() == Sort::SUM);

    ListIter<SortSumand> curSumand = expectSort.sum();
    Nat sumTag = 1;          // sumands start with 1 (0 is left for undefined)
    for (;;) {
        if (curSumand.isEnd()) {
            status.noSuchSumand(pat.sum().constr(), expectSort);
            return;
        }
        if (curSumand->name() == pat.sum().constr())
            break;

        sumTag++;
        curSumand++;
    }

    typeCheck(pat.sum().body(), curSumand->sort(), info, status);
    if (!status.isSuccess())
        return;
}

```

```

    info.sumands.update(toObject(pat), *(new Nat(sumTag)));
}

/*****
void typeCheckProd(Pattern& pat, Sort& expectSort,
                  LangInfo& info, GenStatus& status) {

    ASSERT(pat.kind() == Pattern::PRODUCT);
    ASSERT(expectSort.kind() == Sort::PRODUCT);

    Nat numPatFields = pat.product().length();
    Nat numSortFields = expectSort.product().length();
    if (numPatFields != numSortFields) {
        status.wrongNumFields(pat, numPatFields, numSortFields);
        return;
    }

    ListIter<PatField> curField = pat.product();
    ListIter<SortField> curSortField = expectSort.product();
    while(!curField.isEnd()) {
        if (curField->name().isDefined()) {
            if (!curSortField->name().isDefined()) {
                status.wrongFieldName(*curField, *curField->name(), string(""));
                return;
            }
            if (!(*curField->name() == *curSortField->name())) {
                status.wrongFieldName(*curField,
                                      *curField->name(), *curSortField->name());
                return;
            }
        }
        typeCheck(curField->field(), curSortField->sort(), info, status);
        if (!status.isSuccess())
            return;

        curField++;
        curSortField++;
    }
}

/*****
void typeCheck(Pattern& pat, Sort& rawSort,
              LangInfo& info, GenStatus& status) {

    Sort& expectSort = normalize(rawSort, info.sorts);
    switch(pat.kind()) {
    case Pattern::UNDEF:
        ASSERT(0);
        break;
    case Pattern::SUM:
        if (expectSort.kind() != Sort::SUM) {
            status.notExpectedSort(pat, expectSort);
            return;
        }
        typeCheckSum(pat, expectSort, info, status);
        break;
}

```

```

case Pattern::PRODUCT:
  if (expectSort.kind() != Sort::PRODUCT) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  typeCheckProd(pat, expectSort, info, status);
  break;
case Pattern::LIST: {
  if (expectSort.kind() != Sort::LIST) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  typeCheck(pat.list().body(), expectSort.list(), info, status);

  RhsElem& patRhsElem = info.def(pat);
  if (patRhsElem.op().kind() != RhsOp::LIST)
    status.badRhsElem(pat, patRhsElem);
  } break;
case Pattern::LISTNIL:
  if (expectSort.kind() != Sort::LIST) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  return;
case Pattern::LISTCONS:
  if (expectSort.kind() != Sort::LIST) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  typeCheck(pat.listCons().first(), expectSort.list(), info, status);
  if (!status.isSuccess())
    return;
  typeCheck(pat.listCons().rest(), expectSort, info, status);
  return;
case Pattern::OPTIONAL: {
  if (expectSort.kind() != Sort::OPTIONAL) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  typeCheck(pat.optional().body(), expectSort.optional(), info, status);

  RhsElem& patRhsElem = info.def(pat);
  if (patRhsElem.op().kind() != RhsOp::OPTIONAL)
    status.badRhsElem(pat, patRhsElem);
  } break;
case Pattern::OPTNIL:
  if (expectSort.kind() != Sort::OPTIONAL) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  return;
case Pattern::OPTVAL:
  if (expectSort.kind() != Sort::OPTIONAL) {
    status.notExpectedSort(pat, expectSort);
    return;
  }
  typeCheck(pat.optVal(), expectSort.optional(), info, status);

```

```

    return;
case Pattern::VAR:
    typeCheckVar(pat, expectSort, info, status);
    break;
case Pattern::LITERAL:
    if (expectSort.kind() == Sort::STRING)
        return;
    if (expectSort.kind() == Sort::SUBSET) {
        SortSubset& subset = expectSort.subset();
        if (subset.sort().kind() != Sort::STRING)
            status.notExpectedSort(pat, expectSort);
        if (!(subset.filter())((Obj&) pat.literal()))
            status.notExpectedSort(pat, expectSort);
    }
    status.notExpectedSort(pat, expectSort);
    break;
case Pattern::TYPEDEC: {
    SortStatus sortStatus;
    Sort& patSort=toSort(pat.typeDec().type(),info.sorts,sortStatus);
    if (!sortStatus.isSuccess()) {
        status.set(sortStatus);
        return;
    }
    if (!equal(expectSort, patSort, info.sorts)) {
        status.typeConflict(pat, patSort, expectSort);
        return;
    }
    typeCheck(pat.typeDec().body(), expectSort, info, status);
    break;
}
}
}

```

```

/*****
void typeCheck(List<NonTermDef>& syntax, LangInfo& info, GenStatus& status) {

```

```

    ListIter<NonTermDef> curDef = syntax;
    while(!curDef.isEnd()) {
        SortStatus sortStatus;
        Sort& ruleSort = toSort(curDef->type(), info.sorts, sortStatus);
        if (!sortStatus.isSuccess()) {
            status.set(sortStatus);
            return;
        }
        info.setSort(*curDef, ruleSort);
        curDef++;
    }

```

```

    curDef.reset(syntax);
    while(!curDef.isEnd()) {
        ListIter<Rule> curRule = curDef->rules();
        while(!curRule.isEnd()) {
            typeCheck(curRule->action(), info.sort(*curDef), info, status);
            if (!status.isSuccess())
                return;
            curRule++;
        }
    }

```

```
    curDef++;  
  }  
}
```

BIBLIOGRAPHY

- [1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977. ISBN 0-201-00022-9.
- [2] ANSI. The programming language ada reference manual. *Lecture Notes in Computer Science*, 155, 1983.
- [3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [4] Allen I. Holub. *Compiler Design in C*. Prentice Hall, 1990. ISBN 0-13-155045-4.
- [5] Paul Hudak et al. Report on the programming language haskell, a non-strict, purely functional language. Technical report, Yale University, 1991. Available via FTP from nebula.cs.yale.edu in pub/haskell-report.
- [6] Guy L. Steele Jr. *Common LISP, The Language*. Digital Press, 1984. ISBN 0-9327376-41-X.
- [7] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. ISBN 0-262-13255-9.
- [8] Chris Reade. *Elements of Functional Programming*. Addison-Wesley, 1989. ISBN 0-201-12915-9.
- [9] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown Publishers, 1986. ISBN 0-697-06849-8.