The Project NoName

A functional programming language with its development environment

H. Braxmeier, D. Ernst, A. Mößle*
Universität Ulm, Fakultät für Informatik, Abt. Programmiermethodik und Compilerbau
E-mail: {hbraxmei,dietmar,andrea}@bach.informatik.uni-ulm.de

H. Vogler
TU Dresden, Fakultät Informatik, Lehrstuhl Grundlagen der Programmierung
E-mail: vogler@inf.tu-dresden.de

1 Introduction

Complex problems can often be described and solved in an easy and elegant way with functional programs. Well-known functional programming languages are, e.g., *LISP* [McC60], *ML* [Mil84], *Miranda* [Tur85], and *Gofer* [Jon93] (also cf. [Hud89] for a survey). In this paper we present a further functional programming language called *NoName* and its programming environment. The reason for this development is to be able to test evaluation strategies and transformation strategies in the field of different recursive program schemes. NoName has some helpful features which cannot be found in other languages in this composition:

Mathematical syntax: The syntax of the function definitions are based on the convention of mathematics. Thus, parentheses and commata together with a strict, monomorphic, higher-order typing discipline provide a clear and simple notation in contrast to the lambda calculus.

Evaluation strategies: For every function NoName allows the choice between the evaluation strategies call-by-name and call-by-value. Note that most of the declarative programming languages have a fixed strategy. Indeed, the only declarative programming language we know which supports different evaluation strategies is *Dactl* [GKS91].

Tree transducers: Special recursive program schemes are explicitely supported by simple NoName constructs. The program schemes are called top-down tree transducer [Rou70, Tha70], macro tree transducer [Eng80, CF82,

*The work of this author was supported by the "Deutsche Forschungsgemeinschaft (DFG)".

EV85], attributed tree transducer [Fül81], macro attributed tree transducer [KV94], and modular tree transducer [EV91]. A detailed description can be found in Section 2.3.

Module concept: NoName allows to structure programs and enables the possibility to import functions and data types from other programs.

The main part of the programming environment is the syntax driven editor of NoName that has been developed with the help of the *Synthesizer Generator*[1], a software tool from GrammaTech, Inc [RT89]. Such a syntax driven editor is tailormade for exactly one programming language and it simplifies the programmers work with the help of pull-down menus and transformations to construct a program in a top-down manner.

The NoName editor is based on an attribute grammar which determines both, the context-free syntax of NoName and the context-sensitive properties of NoName. While editing a NoName program, the NoName editor checks these properties by means of an incremental attribute evaluator.

The attribute grammar for NoName is specified in *SSL* (*S*ynthesizer *S*pecification *L*anguage) by the following parts: (i) the abstract syntax which describes the syntactical structure, (ii) the in- and output syntax, (iii) transformations to refine the program, (iv) attribute grammars for the context-sensitive properties, and (v) external functions for interfaces to other applications (for the implementation of the NoName editor see [EHM95]). The usage of the editor is described in Section 3.1 in more detail.

The further components of the NoName programming environment are based on the fact that, internally, the current program is represented as syntax tree which, e.g., can be easily modified. The integration of the components which are described later, was enabled by offered interfaces of the editor.

This paper is divided into four sections. In Section 2 we describe the language NoName by means of an example. The NoName programming environment is presented in Section 3. In Section 4 we provide a list of further research topics.

2 The language NoName

In this section we explain some basic concepts of the functional language NoName and describe the features of a NoName program: data types, functions, and tree transducers. The skeleton of a NoName program is as follows:

```
PROGRAM program_name;
  list of specifications
END_PROGRAM program_name.
```

[1] The Synthesizer Generator is a trade mark of GrammaTech^{TM}, Inc.

There are four different kinds of specifications:

- data type specification
- function specification
- tree transducer specification
- import specification

We will describe these kinds of specifications by constructing a NoName program. This NoName program is a type checker for SPL-programs, where the context-free part of the SPL-syntax is determined by the set of productions of the context-free grammar shown in Figure 1. The type checker checks the context-sensitive properties of SPL, i.e., (i) whether every identifier which occurs in a statement is also declared, and (ii) whether the type of the left-hand side of an assignment is equal to the type of its right-hand side. This example is derived from [Vog91].

```
prog  → Program decl; stmt End.
decl  → var ident: type
decl  → decl; ident: type
stmt  → ident := ident
stmt  → begin decl; stmt end
stmt  → stmt; stmt
ident → a
ident → b
type  → int
type  → bool
```

```
Program
  var
    a: int;
    b: int;
  begin
    var
      b: int;
    a := b
  end;
  b := a
End.
```

Figure 1: Context-free grammar of SPL and a valid SPL-program.

For a full description of the syntax and semantics of NoName see [EGH+94a].

2.1 Data type specifications

In NoName there are two main categories of data types: basic types, e.g., INTEGER, BOOLEAN, VOID (empty type), and algebraic data types. Already known data types can be composed via cartesian products or function types to new types called *data type synonyms*. Since basic types and data type synonyms are well known from other programming languages, we restrict our interest to algebraic data types.

Generally an algebraic data type specification has the following form:

```
ALG_DEF
  data type_1 = constructor_{1,1}(list of types)
              | constructor_{1,2}(list of types)
              :
              | constructor_{1,m}(list of types)
  data type_2 = ...
              :
  data type_n = ...
END_ALG_DEF
```

We explain this concept by defining the context- free grammar of SPL (shown in Figure 1) as an algebraic data type in NoName. Since it suffices to specify the abstract syntax instead of the concrete syntax in the data type declaration of NoName, the terminal symbols are dropped completely. The production

$$prog \;\rightarrow\; \texttt{Program}\; decl\texttt{;}\; stmt\; \texttt{End.} \qquad (*)$$

is represented by the NoName data type declaration

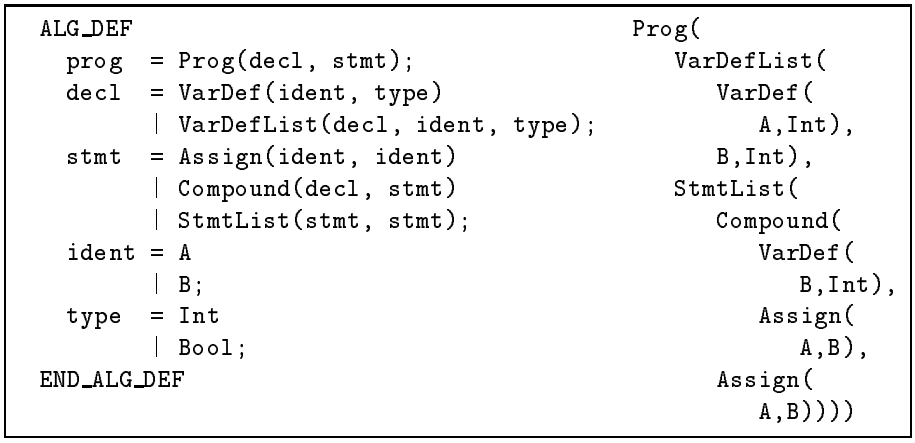
```
prog = Prog(decl,stmt);
```

where `prog` is the name of the data type, `Prog` is a constructor name of rank two, and the nonterminals `decl` and `stmt`, which correspond to *decl* and *stmt* in (∗), respectively, denote further data types. Note that each constructor name and the type of each object have to be uniquely determined in NoName. The complete NoName data type specification of the abstract syntax of SPL is shown in Figure 2. Additionally, this figure presents the abstract syntax tree of the SPL-program from Figure 1.

2.2 General functions

A function is specified by its declaration and its definition. The declaration determines the types of the input and output values. The definition fixes how the output values of the function depends on its input values. Note that a function must be declared before it is defined or used. The skeleton of a function specification is as follows:

```
FUNC_SPEC
  evaluation strategy (optional)
  list of function declarations
  list of function definitions
END_FUNC_SPEC
```



```
ALG_DEF                                          Prog(
  prog  = Prog(decl, stmt);                        VarDefList(
  decl  = VarDef(ident, type)                         VarDef(
        | VarDefList(decl, ident, type);                 A,Int),
  stmt  = Assign(ident, ident)                        B,Int),
        | Compound(decl, stmt)                     StmtList(
        | StmtList(stmt, stmt);                     Compound(
  ident = A                                            VarDef(
        | B;                                              B,Int),
  type  = Int                                          Assign(
        | Bool;                                          A,B),
END_ALG_DEF                                         Assign(
                                                       A,B))))
```

Figure 2: The NoName data type specification of the abstract syntax of SPL and the abstract syntax tree of the SPL-program shown in Figure 1.

For every function, an evaluation strategy (call-by-value or call-by-name) can be stated explicitely (for details cf. page 7 and [EGH+94a]). A function declaration has the following form:

function_name : input type -> output type

The input and output types are arbitrary; for example the input type can be `CARDINAL x INTEGER`, i.e., a cartesian product of types.

Generally a function definition has the following form:

function_name(t_1,...,t_k) = right-hand side

where the left-hand side has to be linear, i.e., no variable occurs twice. A function which has no parameters (denoted by `VOID` as input type in the declaration) needs empty parentheses. Obviously, the type of the right-hand side must be identical with the output type specified in the function declaration.

Now let us consider SPL and let us define a type checker for the conditions named before: every identifier has to be declared and the types of the two sides of an assignment have to be equal. For this purpose, we define a symbol table, called `env`, which is realized as a list of identifiers with their types. For its handling we need the function `update` which builds up and modifies the symbol table, and the function `look_up` to determine the type of a given identifier. Furthermore, we need the function `type_eq` which checks if two types are defined and equal. These functions are shown in Figure 3. The additional necessary algebraic data type definitions for `idtype` and `env` are shown in Figure 4.

```
FUNC_SPEC
  update : env x ident x idtype -> env;
  help_update : ident x idtype x env x ident x idtype -> env.
  update(Nil, id1, ty1)
        = Cons(id1, ty1, Nil)
  update(Cons(id1, ty1, tail), id2, ty2)
        = help_update(id1, ty1, tail, id2, ty2);
  help_update(id1, ty1, tail, id2, ty2)
        = Cons(id2, ty2, tail),
           IF id1 = id2,
        = Cons(id1, ty1, update(tail, id2, ty2)),
           OTHERWISE;
END_FUNC_SPEC

FUNC_SPEC
  look_up : env x ident -> idtype;
  help_look_up : ident x ident x idtype x env -> idtype.
  look_up(Nil, id1)
         = undef
  look_up(Cons(id2, ty, tail), id1)
         = help_look_up(id2, id1, ty, tail);
  help_look_up(id2, id1, ty, tail)
         = ty,
            IF id1 = id2,
         = look_up(tail, id1),
            OTHERWISE;
END_FUNC_SPEC

FUNC_SPEC
  type_eq : idtype x idtype -> BOOLEAN.
  type_eq(t1, t2) = ((t1 <> undef) and (t2 <> undef)) and (t1 = t2);
END_FUNC_SPEC
```

Figure 3: Functions of the type checker of SPL.

Let us explain the function `look_up` more detailed:
Assume that `update` has created a list where all identifiers together with their types of an SPL-program are listed. Now `look_up` checks if the list contains the given identifier, and if so, then it returns its type. Therefore we need a help function, called `help_look_up`, which returns the type if the identifier is found, or it checks if the identifier can be found in the rest of the list.

Note that `look_up` and `help_look_up` are defined by simultaneous recursion. Also note that `help_look_up` uses *conditional clauses* whereas `look_up` is defined by *pattern matching*. The conditional clause which is introduced by the keyword `IF`, must return a boolean value.

```
ALG_DEF
  idtype = int
         | bool
         | undef;
END_ALG_DEF

ALG_DEF
  env = Nil
      | Cons(ident, idtype, env);
END_ALG_DEF
```

Figure 4: The algebraic data types `idtype` and `env`

We demand that conditional clauses fulfill the following conditions:

- They have to be non-overlapping, i.e., for every input value at most one condition can match.
- They must be exhaustive, i.e., for every input value at least one condition must match.

If we use the conditional clause `OTHERWISE`, then it only can occur once, because the conditional expressions have to be non-overlapping.

In contrast to most of the functional programming languages, NoName allows to determine for every function one of the following evaluation strategies:

- call-by-name
- call-by-value.

Call-by-name is a strategy where the outermost function call is evaluated first, in contrast to the call-by-value strategy, where the innermost function call is evaluated first. The benefits and drawbacks of both strategies are well known.

The keyword `CALL_BY_NAME` or `CALL_BY_VALUE` which determines the strategy, has to be inserted between the keyword `FUNC_SPEC` and the list of function declarations (cf. Section 2.2, page 4).

Higher-order functions

In contrast to pattern matching, which is only allowed on inductive data types (i.e., algebraic data types, `CARDINAL`, and `BOOLEAN`), conditional expressions are not restricted to particular data types.

A more elegant way to define the type checker would be to define `look_up` and `update` in a higher-order manner. For this purpose the symbol table `new_env` is defined as a function of type `ident -> idtype`. Then `new_update` looks as shown in Figure 5.

```
FUNC_SPEC
  new_update : new_env x ident x idtype -> new_env.
  new_update(rho, id1, ty1) = lambda
                    WHERE FUNC_SPEC
                            lambda : ident -> idtype.
                            lambda(id2) = ty1,
                                          IF id1 = id2,
                                       = rho(id2),
                                          OTHERWISE;
                          END_FUNC_SPEC;
END_FUNC_SPEC
```

Figure 5: The function specification `new_update`.

2.3 Tree transducers

Tree transducers specify functions over algebraic data types. The language No-Name supports five different kinds of tree transducers, these are:

- top-down tree transducer [Rou70, Tha70],
- macro tree transducer [Eng80, CF82, EV85],
- attributed tree transducer [Fül81],
- macro attributed tree transducer [KV94], and
- modular tree transducer [EV91].

The most special one is the top-down tree transducer where each function is defined in a top-down way and has exactly one parameter called *recursion argument*. More precisely, these functions are *synthesized* functions, i.e., in the right-hand sides of their definitions only function calls (of a tree transducer function from the same specification block) with a successor of the recursion argument are allowed. Note that this restriction does not hold for general function calls.

A macro tree transducer is a generalization of a top-down tree transducer in such a way that its functions may contain further parameters additionally to the recursion argument.

An attributed tree transducer is a generalization of a top-down tree transducer in another direction: synthesized functions as well as *inherited* functions can be specified. Hereby in the right-hand side of an inherited function definition only function calls (of a tree transducer function from the same specification block) with a predecessor of the recursion argument are allowed.

A combination of the above enumerated concepts is realized by macro attributed tree transducers. That means we can use further parameters in addi-

tion to the recursion argument and we can use both, synthesized functions and inherited functions.

Modular tree transducers cluster several top-down tree transducers and macro tree transducers into a unit and allows for a tree shaped hierarchical arrangement of units.

For instance, the type checker of SPL can be specified by the following macro tree transducer:

```
BEGIN_MAC
  envir: SYN DESC decl x env -> env;
  check: SYN DESC stmt x env -> BOOLEAN.

  envir(<z:VarDef>, rho)
     =  update(rho, id(<z>.1), id(<z>.2));
  envir(<z:VarDefList>, rho)
     = envir(<z>.1, update(rho, id(<z>.2), id(<z>.3)));

  check(<z:Assign>, rho)
     = type_eq(look_up(rho, id(<z>.1)), look_up(rho, id(<z>.2)));
  check(<z:Compound>, rho)
     = check(<z>.2, envir(<z>.1, rho));
  check(<z:StmtList>, rho)
     = check(<z>.1, rho) and check(<z>.2, rho);
END_MAC
```

In contrast to general functions the left-hand side of a macro tree transducer function definition comprises a term of the form `<z:`*constructor*`>` which denotes a path in the input tree labeled by *constructor*. Note that this notation has to be used in recursion argument position, whereas the other argument positions have to be instantiated by variables.

The right-hand side of a tree transducer function definition is similar to that of a general function (cf. Section 2.2). If, in the right-hand side, we call a tree transducer function of the same specification block, then the recursion argument has the form

`<z>.`*j*

where *j* denotes the *j*-th subtree.

The functions `envir` and `check` are declared as synthesized functions by the lines

```
envir:  SYN DESC decl x env -> env;
```

and

```
check:  SYN DESC stmt x env -> BOOLEAN.
```

Note that the keywords `SYN DESC` marks a synthesized descent function. Since the data type `decl` has two constructors (`VarDef` and `VarDefList`), there are two function equations for `envir`.

The `id` function which occurs in the right-hand side of, e.g., `envir`, is a special construct for tree transducers, because `<z>.`j is the pointer to the j-th subtree and does not include the value of this subtree. The built-in function `id` is necessary to convert this pointer into the complete subtree.

The second function equation of `envir` recursively walks down the symbol table (by a recursive call of `envir`) until one identifier is left. Then the first function equation calls `update` and builds up the symbol table.

Additionally to the function `envir`, the macro tree transducer contains the function `check` which checks the type equality in the assignment statements of SPL.

Furthermore we need the following function specification to complete the NoName specification of the type checker for SPL:

```
FUNC_SPEC
  check_prog : prog -> BOOLEAN.
  check_prog(Prog(z1,z2))
     = check(z2, envir(z1, Nil));
END_FUNC_SPEC
```

The table below presents an overview of the syntactic requirements of the different tree transducers (note that tr^2 abbreviates tree transducer). A more concise description can be found in [EGH+94a].

	top-down tr^2	macro tr^2	attributed tr^2	macro attr. tr^2
Begin	BEGIN_TOP	BEGIN_MAC	BEGIN_ATT	BEGIN_MAT
End	END_TOP	END_MAC	END_ATT	END_MAT
Functype	DESC	DESC	WALK	WALK
Synthesized func.	allowed	allowed	allowed	allowed
Inherited func.	—	—	allowed	allowed
More parameters	—	allowed	—	allowed

2.4 Import specification

NoName supports the use of library modules. Such modules have the same structure as other NoName programs and so we can use every NoName program as a library module. Functions, tree transducers, or data types from other NoName programs can be imported by the `IMPORT` specification. For example, the import specification

`FROM math.nnm IMPORT ack`;

imports from the NoName program `math.nnm` the function `ack` and, additionally, all functions and data types which are necessary for the specification of `ack` (even if, in its turn, all these functions and data types have been imported in `math.nnm` for `ack`).

2.5 Conclusion

With the help of the simple programming language SPL we have described the most important properties of NoName. The type checker, including the algebraic data type specifications, the functions `update`, `look_up` and `type_eq` (see Section 2.2, page 5), the macro tree transducer (described in Section 2.3, page 9) and the function `check_prog` (see Section 2.3, page 10) represents a complete NoName program to check the context-sensitive properties of SPL. The following sections explain how to use and work with the *NoName programming environment*, a tool to develop NoName programs in an effective, correct, and easy way and how to use the features of this environment.

3 The programming environment of NoName

The NoName programming environment was developed to support the user in writing NoName programs. The main part of this environment is a syntax driven editor for NoName (cf. Section 3.1). It was generated with the help of the *Synthesizer Generator*, a software tool designed by T. W. Reps and T. Teitelbaum. We have integrated an online help (cf. Section 3.2), different views of the current program (cf. Section 3.3), as well as a database connection (cf. Section 3.4) to maintain parts of NoName programs, e.g., standard functions (for a collection of standard functions see [May95]). For different purposes, e.g., optimization and user guidance, complex transformations (cf. Section 3.5) and a tool for representing the syntax tree of the current program graphically (cf. Section 3.6) complete our user-friendly environment. Additionally, the NoName programming environment includes two compilers (cf. Section 3.7). Figure 6

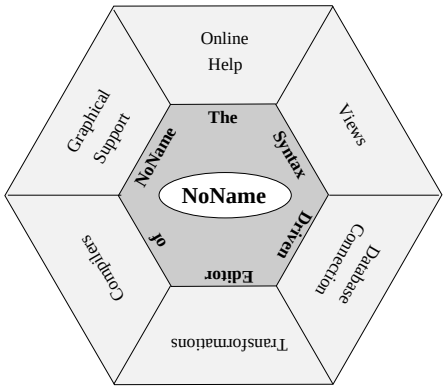


Figure 6: The programming environment of NoName and its features.

pictures this programming environment. Remember that the programming language NoName was described in Section 2.

3.1 The basics of the NoName editor



First we describe the basic concepts of the NoName editor. In contrast to text editors, the NoName editor provides syntactic correct NoName programs by accepting only syntactic correct modifications. Moreover it can react helpfully if any program errors are detected. Since a semantic analyser is also part of the NoName editor, the user is promptly informed about existing semantic defects.

With the help of a screenshot, shown in Figure 7, we describe how to work with the editor.

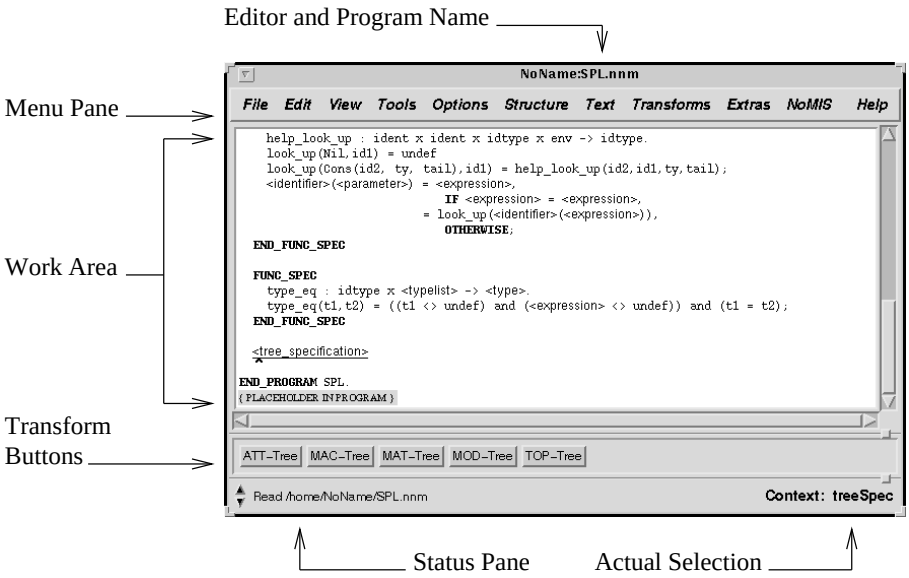


Figure 7: The programming environment of NoName.

The menu pane includes the following areas:

File: The menu File is used for file handling, i.e., to open, close, or save files etc. as usual.

Edit: This menu allows to execute several edit functions, i.e., to select, copy, paste, or delete parts of the actual program. Additionally to a text version of these edit functions, structure modifying functions are included, i.e., functions which only allow to modify complete subtrees of the actual program's syntax tree in a correct way.

View: In addition to the representation of the actual program as shown in Figure 7, other views are supported, e.g., the miranda view which shows the program in the syntax of the functional programming language Miranda [Tur85]. The different views are described in Section 3.3.

Tool: The menu Tool offers interfaces to the operating system.

Options: The menu Options allows to adjust some basic options like switching on and off the status pane.

Structure: With this menu the navigation through the syntax tree of the actual program can be performed.

Text: In contrast to the menu Structure the menu Text supports navigations through the program text.

Transforms: This menu has the same function as the transform buttons which are described below.

Extras: The menu Extras allows to start different compilers (cf. Section 3.7) and it is possible to edit NoName programs with any other editor.

NoMIS: The menu NoMIS is used to open and work with a database (see Section 3.4 and [Mül95]).

Help: The menu Help starts a detailed help environment, e.g., all transform buttons are explained precisely (see Section 3.2).

Note that the menus View and Help are interfaces offered by the *Synthesizer Generator* which we have adjusted for NoName. Additionally we have added the menus Extras and NoMIS which are explained later. For a more concise description of the other menus see [Gra96].

The actual program is shown in a pretty-printed version in the work area. Programs are built by refining so-called *templates*, i.e., predefined, formatted patterns for parts of NoName programs which contain placeholders. New templates can be inserted at the position of placeholders of already existing templates. By this method programs are constructed top-down. The underlined part of the program is the actual selection. In Figure 7 the template

`<tree_specification>`

is a placeholder where a new template can be inserted. Here, it can be refined to a certain tree transducer (see transform buttons in Figure 7). We refine this placeholder to a template of a macro tree transducer with the button MacTree. The result of this refinement is shown in Figure 8.

The available transformations depend on the current context, because only templates which generate a syntactic correct program are allowed.

Comments can be inserted at pre-defined places, which are automatically marked by the line `(% <comment> %)`. In Figure 7, e.g., there are two possible places where comments can be written. For more information see [EGH+94a].

The textual representation can also be edited by the keyboard. The return key finishes the editing mode and the text is parsed. Syntax errors are shown in the status pane and should be corrected immediately. Furthermore all other messages, as, e.g., the message Read /home/NoName/SPL.nnm in Figure 7 and the actual context are shown in the status pane.

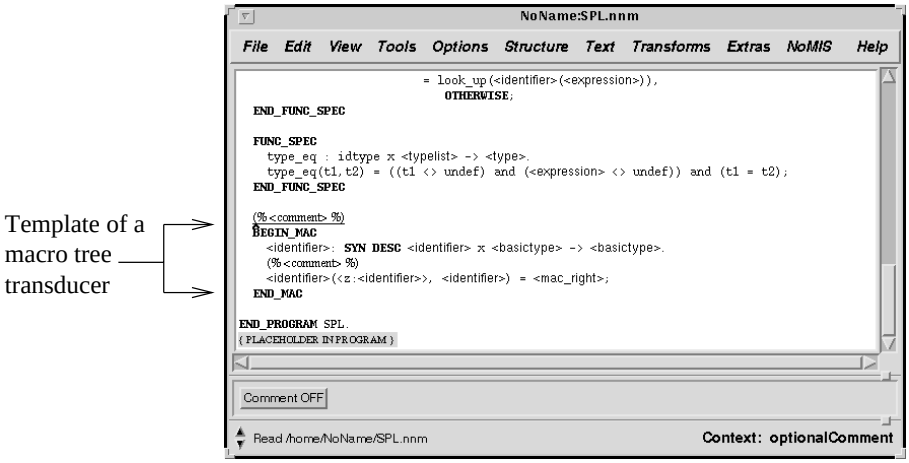


Figure 8: Part of the NoName specification of SPL: one step later as shown in Figure 7.

3.2 The online help of the editor



The online help of the editor offers support to the complete NoName environment. There are help functions to every executable command as, e.g., the file handling commands which can be found under the menu item File. Furthermore all transform buttons are described in detail. Let us show the online help for the transform button MacTree which was used in the previous section. To get this help page the item describe-transform of the menu Help has to be selected. The browser *eMosaic* is started with a page in *HTML* (*H*yper *T*ext *M*arkup *L*anguage) format which contains references to all help pages of the current enabled transforms buttons. As mentioned before we select the help page for the transform button MacTree. The result is shown in Figure 9. Underlined text phrases of the document are *links* which refer to other related topics. Here the underlined word synthesized is a link which refers to the topic *synthesized functions*. Note that all help pages for transform buttons consist of the following items where some of them may be omitted:

- USAGE:
  Describes which code is generated if we click the corresponding transform button.
- DESCRIPTION:
  Gives further details to the transform button.
- LITERATURE:
  Refers to other documents for further information on this topic.

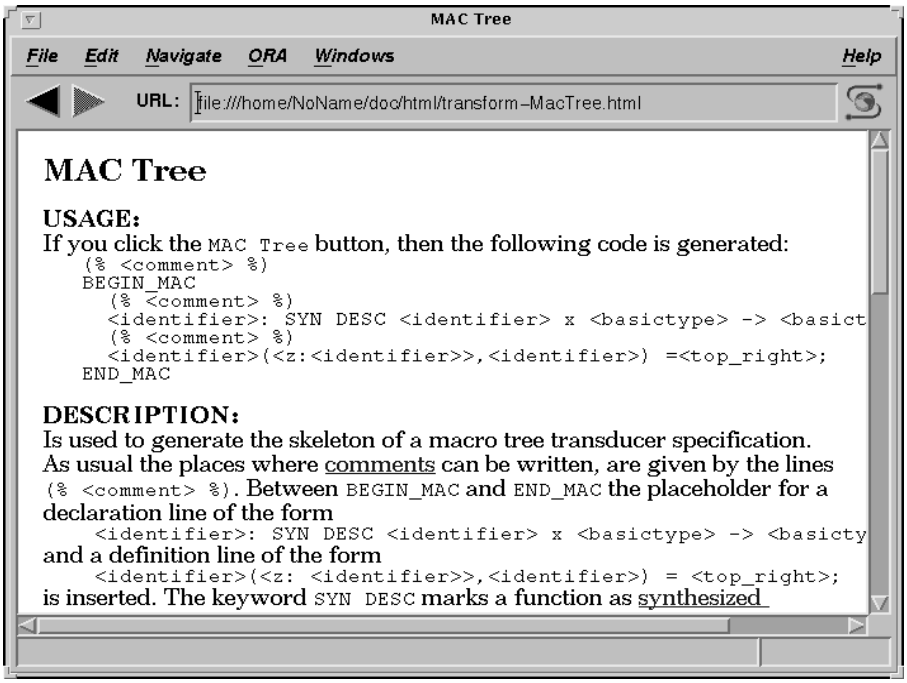


Figure 9: The NoName online help for the MacTree transform button.

- EXAMPLE:
  Pictures the usage of the transform buttons.
- ATTENTION:
  Gives hints for important peculiarities.
- SEE ALSO:
  Links to other similar transform buttons.

Moreover our online help contains references to the language description of NoName [EGH+94a] and to the manual for the NoName programming environment [EGH+94b].

3.3 Views



The NoName environment supports four different views: the *baseview* and the views *miranda*, *decl*, and *clickpoint*. The baseview shows the program in the standard view as shown in Figure 7 and 8. The miranda view presents NoName programs in miranda syntax. The decl view only shows the declarations of a NoName program and the clickpoint view marks all positions where templates

can be inserted. Views can be shown simultaneously, e.g., the baseview and the miranda view can be shown in two different buffers at the same time. Note that changes made in one view are adapted in the other views immediately. Consequently, with slight modifications the NoName editor can be expanded in such a way that programs can also be written in miranda syntax. For more information see [EHM95].

3.4 Database connections



The NoName programming environment includes a relational mSQL database (cf. [Hug96]) to administrate constructs of NoName programs interactively. To use the database we have to open the connection to it with the help of the item db-connect of the menu item NoMIS. The database itself allows to retrieve, store, and delete specifications of data types, algebraic data types, functions, and tree transducers as well as templates of such specifications (see the items db-retrieve, db-store, and db-delete). Each saved construct is determined by a name and a version and it can be accompanied by a comment. Search criterions called *topics* simplify the administration of constructs. They are determined by their name (comments to describe them more precisely are allowed). We can create new topics, delete them, or view all topics (see the item topic-handling). Topics can be associated to constructs and vice versa. The item db-output allows the user to generate different *LATEX*-files which give an overview about stored data in the database. The following files can be generated:

- constructs.tex: Gives an overview over all stored constructs.
- topic.tex: Gives an overview about all topics.
- assign.tex: Lists all associations between constructs and topics.

We use the item db-disconnect to end the connection with the database and return to the NoName programming environment. If we use the item db-exit-all we disconnect the database and close the NoName programming environment.

For a complete description of the database connection see [Mül95]. There, the backgrounds, the basics, the implementation, and the usage of the database etc. are described in a very detailed form. A lot of examples complete the description in that document.

3.5 Transformations



In Section 3.1 we have described some basic transformations to refine templates. Now we present more powerful transformations which allow complex code modifications. Since the class of tree transducer is well structured, many theoretical results are known for this class (cf., e.g, [Eng75, Eng77, Fül81, Eng80, CF82, EV85, EV86, EV88, FHVV93, KV94]). Some of them are realized in the NoName programming environment. With the help of transformations it is possible

to transform one kind of tree transducer into another kind of tree transducer, e.g., a modular tree transducer can be split into several top-down and macro tree transducers; hereby external functions are resolved, context parameters are substituted, and simultaneous recursion is eliminated (the underlying technique can be found in [Küh90]).

Obvious syntactic transformations from one class into the other are enabled if the conditions pictured in the following table are given.

Furthermore it is possible to improve the efficiency of the underlying program by applying the *tupling strategy* to macro tree transducer (cf. [MV95]). Indeed this tupling strategy is a combination of some transformations which are based on the idea to avoid recomputations of values and multiple traversals of the input argument. This transformation is fully automatic and it is offered with the transform button tupling when a macro tree transducer is marked.

To support the programmer in the usage of tree transducers other helpful transformations are integrated (cf. [Ern95]), e.g., the necessary left-hand sides of tree transducer definitions can be inserted automatically.

To simplify the programmers work several transformations exist as, e.g., the transform button PM -> IF which allows to change the distinction of cases by pattern-matching into IF-clauses.

Now let us explain helpful transformations of tree transducers by an example. We demonstrate the usage of the transform buttons Create Defs and Make Compact with the following template of the macro tree transducer from Section 2.3 (page 9).

```
BEGIN_MAC
  envir: SYN DESC decl x env -> env;
  check: SYN DESC stmt x env -> BOOLEAN.
END_MAC
```

If the functions of a tree transducer are completely declared as shown above, then it is possible to determine the corresponding definitions except their right-hand sides because of the restricted syntax of tree transducers. More precisely it is known that for every constructor function equations must exist of the form:

functionname (<z:*constructor*>) = right-hand side

Clicking the transform button Create Defs generates these equations automatically and the result is as follows:

```
BEGIN_MAC
  envir: SYN DESC decl x env -> env;
  check: SYN DESC stmt x env -> BOOLEAN.
  envir(<z:VarDefList>, y1) = <mac_right>;
  envir(<z:VarDef>, y1) = <mac_right>;
  check(<z:Assign>, y1) = <mac_right>;
  check(<z:Compound>, y1) = <mac_right>;
```

```
  check(<z:StmtList>, y1) = <mac_right>;
END_MAC
```

This can also be done by refining templates by hand. Hereby the case may arise that the functions of a tree transducer are not defined compactly, that means, it is not guaranteed that the function equations are defined in the right order like shown in the following example:

```
BEGIN_MAC
  envir: SYN DESC decl x env -> env;
  check: SYN DESC stmt x env -> BOOLEAN.
  envir(<z:VarDef>, rho) = <mac_right>;
  check(<z:Assign>, rho) = <mac_right>;
  check(<z:Compound>, rho) = <mac_right>;
  check(<z:StmtList>, rho) = <mac_right>;
  { NOT COMPACT }  envir(<z:VarDefList>, rho) = <mac_right>;
END_MAC
```

The error message { `NOT COMPACT` } shows that this tree transducer is not defined compactly, i.e., the program is not correct. If the actual tree transducer is marked, then this error can be corrected automatically with the help of the transform button Make Compact:

```
BEGIN_MAC
  envir: SYN DESC decl x env -> env;
  check: SYN DESC stmt x env -> BOOLEAN.
  envir(<z:VarDefList>, rho) = <mac_right>;
  envir(<z:VarDef>, rho) = <mac_right>;
  check(<z:StmtList>, rho) = <mac_right>;
  check(<z:Compound>, rho) = <mac_right>;
  check(<z:Assign>, rho) = <mac_right>;
END_MAC
```

For more information cf. [Ern95].

3.6 Graphical support



The NoName programming environment supports a graphical representation of NoName programs. To use this feature select the item Enable Graphical Support from the menu item Options. An extra window is opened in parallel to the NoName editor to show the syntax tree of the corresponding program. All modifications in this program are shown immediately in the syntax tree and the actual selection is highlighted (see Figure 10).

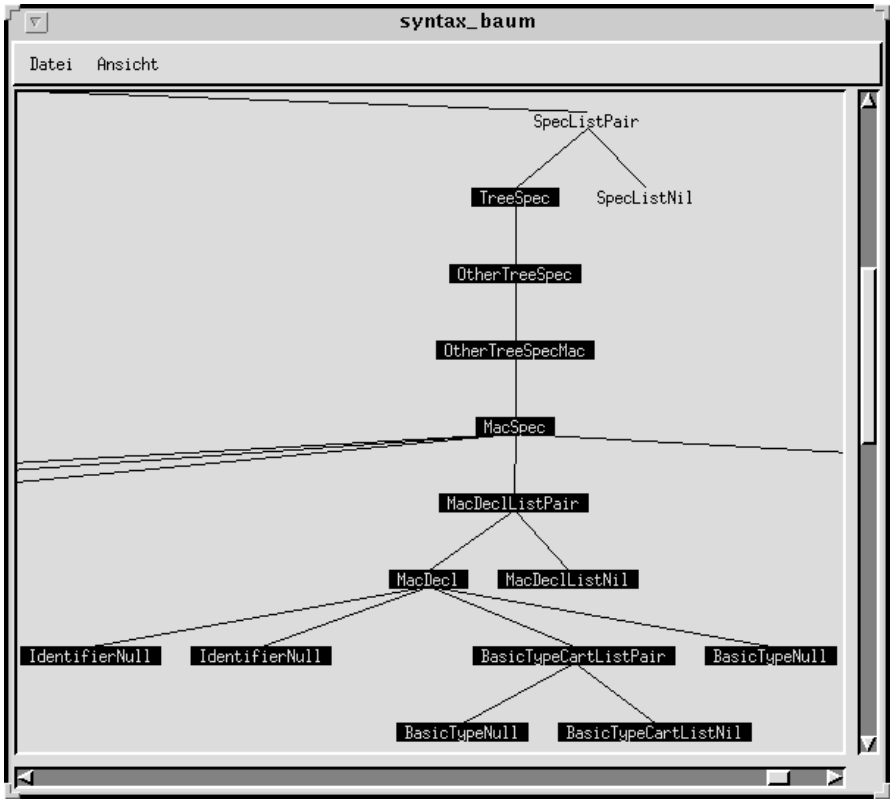


Figure 10: Part of the syntax tree of our SPL program.

3.7 Compilers



NoName offers two different compilers: The NoName compiler and the miranda compiler. As we have described in Section 3.3 the miranda view shows a NoName program as miranda program. By this method it is possible to evaluate NoName expressions with the help of an integrated miranda compiler by transforming them also in miranda expressions. More precisely, for the evaluation of a concrete expression *expr* we select the menu item Execute Miranda from the menu Extra. A dialog box is opened where the user can type in *expr*. By selecting the Start button, *expr* is automatically transformed in a miranda expression $expr'$ and evaluated with the associated miranda program. The result is back transformed to NoName and shown in the message pane. This process is pictured in Figure 11. Most parts of NoName programs are easy to translate, except tree transducers and different evaluation strategies because Miranda does not support them. So each tree transducer is represented in the corresponding

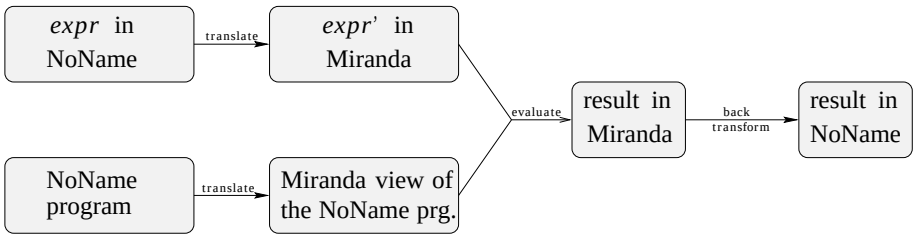


Figure 11: The Miranda compiler.

Miranda program as set of functions and given evaluation strategies are ignored. For further information see [EHM95].

At the time being the NoName compiler which is based on a runtime stack machine can only evaluate first-order NoName programs (that means, NoName programs without higher-order functions). The evaluation can be started by selecting the menu item Execute NoName. For detailed information see [Hou95].

3.8 Conclusion

In this section we have presented the NoName programming environment with all its features: The integrated online help, the different views, the database connection, the transformations with their various functions, the graphical support to show the syntax tree of a NoName program, and the different compilers. A detailed description of the programming environment can be found in [EGH+94a] and further background information is enclosed in the quoted articles.

4 Further research topics

After having described the language NoName and the programming environment of NoName, we give an overview about other desirable features.

One major point is to introduce the concept of lists in NoName. It would be useful to have an extra notation for lists.

Furthermore we want to support the concept of polymorphism, that means, a function which is required for different data types should be implemented only once with the help of type variables. Note that the introduction of polymorphism causes great changes in the NoName programming environment and in the underlying type checker for NoName.

Obviously some of the existing features should be generalized. For example we want to expand the possibility to correct syntactic and semantic errors automatically. Clearly this can not be done for all errors which can occur in a program.

Other program transformation techniques could be senseful as, e.g., deforestation ([Wad90]) and partial evaluation ([DGT96]), and should be integrated in the NoName editor. The ability to compare and test transformation techniques, a possibility to protocol the number of evaluated function calls, and other measure criteria shall be introduced.

References

[CF82] B. Courcelle and P. Franchi–Zannettacci. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17:163–191 and 235–257, 1982.

[DGT96] O. Danvy, R. Glück, and P. Thiemann. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer–Verlag, 1996.

[EGH+94a] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mößle, E. Müller, S. Sablatnög, H. Vogler, and E. Wieser. NoName — Eine funktionale Sprache mit Unterstützung der primitiven Rekursion. Universität Ulm, 1994.

[EGH+94b] D. Ernst, D. Gluche, F. Houdek, A. Kühnemann, A. Mößle, E. Müller, S. Sablatnög, H. Vogler, and E. Wieser. Handbuch zum NoName–Editor, Version 1.0. Universität Ulm, 1994.

[EHM95] D. Ernst, F. Houdek, and A. Mößle. Technische Dokumentation zum NoName–Editor, Version 1.0. Universität Ulm, 1995.

[Eng75] J. Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Mathematical Systems Theory*, 9:198–231, 1975.

[Eng77] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10, 1977.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R.V. Book, editor, *Formal language theory; perspectives and open problems*, pages 214–256. New York, Academic Press, 1980.

[Ern95] D. Ernst. Entwicklung und Implementierung eines syntaxgesteuerten Editors für die Sprache NoName. Master's thesis, Universität Ulm, 1995.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Science*, 31:71–146, 1985.

[EV86] J. Engelfriet and H. Vogler. Pushdown machines for the macro tree transducer. *Theoretical Computer Science*, 42(3):251–369, october 1986.

[EV88] J. Engelfriet and H. Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26:131–192, 1988.

[EV91] J. Engelfriet and H. Vogler. Modular tree transducers. *Theoretical Computer Science*, 78:267–303, 1991.

[FHVV93] Z. Fülöp, F. Herrmann, S. Vagvölgyi, and H. Vogler. Tree transducers with external functions. *TCS*, 108:185–236, 1993.

[Fül81] Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[GKS91] J. Glaubert, R. Kennaway, and R. Sleep. Dactl: An experimental graph rewriting language. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 378–395, 1991.

[Gra96] GrammaTech. *The Synthesizer Generator 5.0 Reference Manual*, 1996.

[Hou95] F. Houdek. Implementation von first–level NoName auf einer Runtime–Stack Maschine. Master's thesis, Universität Ulm, 1995.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.

[Hug96] Hughes Technologies Ltd. On–line manual of mini sql 2.0, 1996. Available via *http://Hughes.com.au/library/msql2/manual*.

[Jon93] M.P. Jones. Gofer — an introduction to Gofer, 1993. Included as part of the standard Gofer distribution.

[Küh90] A. Kühnemann. Transformation strukturell-rekursiver Programme in Normalform. Master's thesis, RWTH Aachen, 1990.

[KV94] A. Kühnemann and H. Vogler. Synthesized and inherited functions — a new computational model for syntax–directed semantics. *Acta Informatica*, 31:431–477, 1994.

[May95] R. Mayer. Beschreibung der Standardfunktionen der Sprache NoName. Universität Ulm, 1995.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3:184–195, 1960.

[Mil84] R. Milner. A proposal for Standard ML. In *ACM Conference on LISP and Functional Programming*, pages 184–197. ACM, 1984.

[Mül95] E. Müller. Entwicklung eines Modul–Informationssystems für den NoName–Editor. Master's thesis, Universität Ulm, 1995.

[MV95] A. Mößle and H. Vogler. Efficient call-by-value evaluation of primitive recursive program schemes. In M. Takeichi and T. Ida, editors, *Functional and Logic Programming*. World Scientific Publishing Co. Pte Ltd., 1995.

[Rou70] W.C. Rounds. Mappings and grammars on trees. *Mathematical Systems Theory*, 4:257–287, 1970.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator — A system for constructing language–based editors*. Springer–Verlag, 1989.

[Tha70] J.W. Thatcher. Generalized^2 sequential machine maps. *Journal of Computer and System Science*, 4:339–367, 1970.

[Tur85] D. Turner. Miranda: A non–strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Functional programming languages and computer architecture*, volume 201 of *Lecture notes in computer science*, pages 1–16. Springer Verlag, September 1985.

[Vog91] H. Vogler. Functional description of the contextual analysis in block-structured programming languages: a case study of tree transducers. *Science of Computer Programming*, 16:251–275, 1991.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.