Visual Meta-Programming Notation[1]

Mikhail Auguston[2]

Department of Computer Science

Naval Postgraduate School

833 Dyer Road, Monterey, CA 93943 USA

auguston@cs.nps.navy.mil

Abstract

This paper describes a draft of visual notation for meta-programming. The main suggestions of this work include specialized data structures (lists, tuples, trees), data item associations that provide for creation of arbitrary graphs, visualization of data structures and data flows, graphical notation for pattern matching (list, tuple, and tree patterns, graphical notation for context free grammars, streams), encapsulation means for hierarchical rules design, two-dimensional data-flow diagrams for rules , visual control constructs for conditionals and iteration, default mapping rules to reduce real-estate requirements for diagrams, and dynamic data attributes.

Two-dimensional data flow diagrams improve readability of a meta-program. The abstract syntax type definitions for common programming languages and related default mappings (parsing and de-parsing) provide for a practically feasible reuse of those components.

1   Introduction and objectives

Meta-programs are programs manipulating other programs. Typical applications include compilers, interpreters, source code static analyzers and checkers, program generators, and pretty-printers. Domain-specific language implementation and rapidly evolving generative programming [9] are the latest examples of developments in this domain. The complexity and sophistication of meta-programs may be quite significant, so the readability and maintainability become an issue.

Compiler and generator design is a domain that has been studied extensively. There is a pretty good understanding of what to do and how to do it, especially for front-end design, and a lot of domain-specific software design templates are accumulated in literature. The following domain features are among the most common for language processor design.

- Use of context-free grammars to specify syntax and serve as a basis for parser design.

- Intermediate representation of the input in the form of an abstract syntax tree. The importance of different tree data structures is recognized in general for this problem domain.

- Typically, the main components of a language processor are very hierarchical and structured along the structure of data (recursive descent parser is an excellent example of this feature). In other words, language processors are heavily data-based applications.

- It appears that the most commonly used data structures include trees, lists, stacks, tables, and strings.

- The architecture of a language processor in most cases can be represented as a data flow between components (e.g., the famous compiler data flow diagram on the page 13 of the "Dragon Book"[1]).

- The notion of an attribute associated with the data item, and attribute dependency and propagation schemes are of a great relevance (the attribute grammar framework captures some of the essential static checking needs; the data flow analysis performed for the optimization stage in a compiler may be considered as an attribute propagation over the program graph).

---

[1] This research was supported in part by the U. S. Army Research Office under grant number 40473-MA-SP.

[2] On leave from New Mexico State University, USA

- Tree (and graph) traversal and transformation is a common template for optimization and code generation tasks.
- Pattern matching (e.g., with respect to regular expressions or context-free grammars) may be a useful control structure for this problem domain.

These considerations and experience with the compiler writing tools RIGAL[2][3], lex and yacc[11], and ELI[10] contributed to this work. Data-flow paradigm is quite natural for meta-programming domain since it is heavily data dependent, and consequently, the graphical notation for data-flow diagrams could be appropriate. This should be integrated with visualization of typical data structures, pattern matching, and encapsulation to provide for well-structured, hierarchical programs. Data-flow diagrams are most commonly used to represent dependencies between data and processes in visual programming languages, for instance, in LabVIEW[5] and Prograph[8].

Two-dimensional diagram notation could significantly improve readability of meta-programs. Some of these ideas have been explored in our previous work[4].

The main suggestions of this work are as follows:

- specialized data structures (lists, tuples, trees),
- data items associations that provide for creation of arbitrary graphs,
- visualization of data structures and data flows,
- graphical notation for pattern matching (list, tuple, and tree patterns; graphical notation for context free grammars and streams),
- encapsulation means for hierarchical rules design,
- two-dimensional data-flow diagrams for rules,
- visual control constructs for conditionals and iteration,
- default mapping rules to reduce screen real-estate requirements for diagrams,
- dynamic (Last #rule $attribute) and static (via associations) data attributes,
- data-flow notation that assumes potential parallelism in the data processing,
- abstract syntax type definitions for common programming languages and related default mappings (parsing and de-parsing) that provide for a practically feasible reuse of those components.

2   Constructs

This paper was not intended to give a complete and precise syntax and semantics of the visual language. At this point it is rather a notation that will be upgraded to programming language status after the implementation effort is completed. A (simplified) example of a compiler from a small subset of Lisp (called MicroLisp) to the C language will be used to present the main ideas. Figures 3– 7 present several annotated parsing and code generation rules of the MicroLisp to C compiler. Appendix A contains the MicroLisp context-free grammar and an example of a program.

2.1   Data flow diagrams

Detailed rationale for data-flow diagram notation and a survey of related work can be found in a previous paper[4]. Briefly, a meta-program is rendered as a two-dimensional data flow diagram that visualizes the dependencies between data and processes. Diagrams actually are similar to the notion of procedure in common programming languages. A diagram represents a single function called a rule, and rule calls may be recursive. The data-flow diagram supports the possibility of parallel execution of threads within the rule.

The data-flow paradigm is closely related to the functional programming paradigm [7] and shares with that paradigm referential transparency and good correspondence between the source code (the diagram) and the order of program execution.

Each diagram represents a single function with several inputs and outputs. At the top of a diagram a signature of a rule provides the rule name and types of its inputs and outputs. Besides data items, the diagram may also contain control structures, such as other rule calls, conditional data flow switches, and iterative constructs [4]. All of those constructs are illustrated in the MicroLisp examples.

The rectangular boxes in our notation denote values, and circles and ovals denote patterns, that could be matched with data objects.

2.2   Types

Type represents a set of values (or objects). Basic predefined types include `char` (characters) and `int` (integers). There is also a universal type `ANY` (which is a super type for any type) and the minimal type `NULL` (which is a subtype of any other type and contains a single value `Null` representing also an empty list or tuple).

Aggregate types are ordered tuples of heterogeneous objects, which are useful for abstract syntax representation, and lists (sequences of homogeneous objects that could be dynamically augmented). Extended BNF notation may be used to define tuple types. To a large degree the type system is similar to the type mechanisms in VDM[13] and Refine[12].

Example of a tuple type definition.

```
prog::= function-def* expression
```

This establishes that an object of the type prog is a sequence of zero or more objects of the type function-def followed by an object of the type expression. This could be considered as an abstract syntax representation for the MicroLisp program level. Notice that ordered sequence of objects of the type function-def is nested within an object of the type prog.

Example of a list type definition.

```
text :: [char]
```

There is a predefined list type `id:: [char]`, which stands for a set of character strings that are valid identifiers.

Example of a type definition with several alternatives (union type).

```
expr :: int | id | simple-expression
```

This effectively declares that types int and id are subtypes of expr in the scope of this definition.

Appendix B presents some of the type definitions for the MicroLisp example.
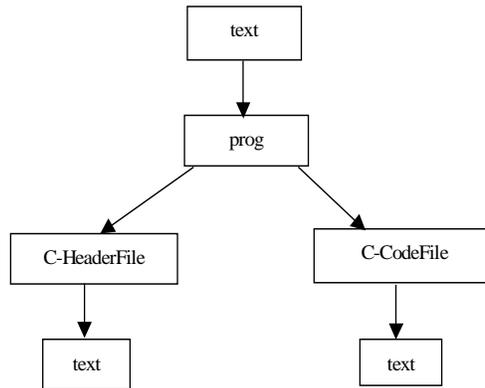
2.3   Default mappings

Figure 1. The top level data flow diagram for MicroLisp to C compiler

Certain rules may be declared as default mappings. It means that corresponding rule calls are optional in the diagrams, and input and output data boxes may be connected directly. This helps to save some screen real estate and to make diagrams less crowded and more readable. Typically default mappings may be introduced for text-to-abstract syntax (parsing) and for abstract syntax-to-text mappings (de-parsing, or abstract syntax-to-concrete syntax mappings).

Yet another kind of default mappings is associated with concatenation operations for tuples and sequences. In fact this is a composition of parsing and de-parsing default mappings applied in the context of (visualized) concatenation. See MicroLisp generation rules for examples (Figures 6-7).

Definitions of abstract syntax types for common programming languages and related parsing and de-parsing default mappings may be valuable assets for reuse.

Default mappings also open the road for "lightweight" inference. For example, suppose that type A is defined as follows:

```
A :: B | C
```

and there are default mappings B -> D and C -> D, then it is possible to derive a default mapping for A -> D. This example actually addresses the polymorphism issue in our lightweight type system. Similar inference rules could be developed for other aspects of type system based on transitivity of subtype relation.

2.4   Associations

Data objects may be associated with other data objects. Each of those objects may have other associations as well. Associations are not a necessary part of the type definition (although they could be included in the type definition as well) and are rather optional named attributes of particular objects. Associations may be used to create arbitrary graphs from objects. The following picture on Figure 2 illustrates the creation of a graph structure via associations from three data objects. Association is not symmetric. According to the following diagram object A has been associated with an attribute B via an association named ab, object B with C via bc, and C with A via ca.

Associated objects are retained when the host objects are the source and target in an identical transformation (plain arrow connecting data boxes of the same type) or are passed as inputs and outputs of rule calls. A special built-in rule #COPY creates a copy of an object but retains only those components declared in the type definition. Associated objects could be retrieved by pattern matching. For instance, on the right-hand diagram on Figure 2, object C (belonging to the associations established in the previous example) may be passed as input, and an access to objects B and A can be obtained via pattern matching (circles denote object patterns here). Notice that the direction of association arrow indicates the access path from the host object to the attribute object. The association mechanism may be useful to simulate attribute-grammar–like attribute propagation in ensembles of objects, to represent collections of objects as graphs, to implement symbol tables (where identifiers may be represented as associations names), and so on.
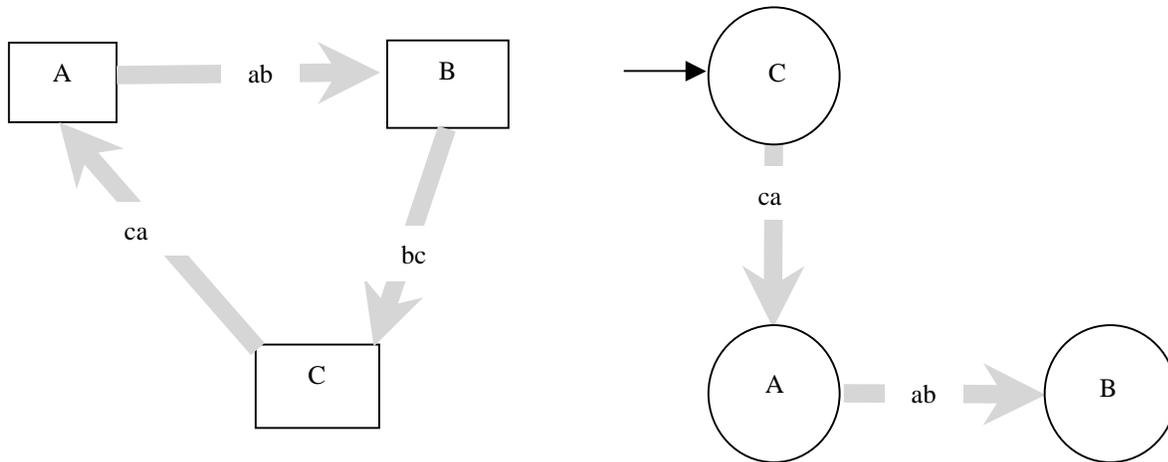

Figure 2. Construction of associations between objects and retrieval of them
using pattern matching

2.5   Patterns and streams

Data object patterns are used to visualize structure of objects in order to provide access to object components and associated objects. An object pattern may be placed in any part of the data flow and is matched with the object connected to the pattern input.

If pattern matching is successful the input object is passed downstream. If pattern matching fails, the entire diagram execution fails, and the diagram sends to its outputs a default value Null, unless the pattern has been provided with the 'Failed' output route. See MicroLisp rules in Figures 3-4 for examples.

If a rule's input is a list, patterns applied to this input may be chained in a sequence (using thick gray arrows) to be applied consecutively. This pattern sequence consumes as many objects from the stream as it can successfully match. The notion of stream corresponds to the sequence in RIGAL language[2][3], and semantics of pattern matching is derived from RIGAL's pattern matching semantics. See MicroLisp parsing rules for example (Figures 3-5).

Rules can create output streams of objects as well.

2.6   States and dynamic attributes

Rule may have states – objects that persist while rule instance is active and can be updated by assignment operators within the rule or from other rules called from this rule. This mechanism could be actually considered a macro extension for diagram notation when a corresponding state object is passed to the called rules as an additional parameter and returned back to the callee as an additional output. States have names starting with the $ symbol, e.g. $X. The reference to the rule's #A state $X has a form Last #A $X. When referred within the rule #A, the prefix Last #A can be dropped. See Figures 4-5 for examples.

3   Examples of MicroLisp to C compiler rules

The following diagrams present three top level parsing rules and two top level generation rules for MicroLisp -> C compiler. They illustrate most of the notations discussed above. Additional annotations provide more specific details and discussion. Those rules are deployed according to the data flow diagram on Figure 1 and default mappings in Appendix B.

3.1   Parsing

The source code of MicroLisp program is represented as a stream of characters. It is assumed that there is a lexical component that filters out comments, spaces, tabs, end-of-line characters from the stream before it is fed to the parsing rules.

```
#program: Stream [char]-> prog, Stream [message]
    state $func-list: [id]  -- updated by #func-def
```
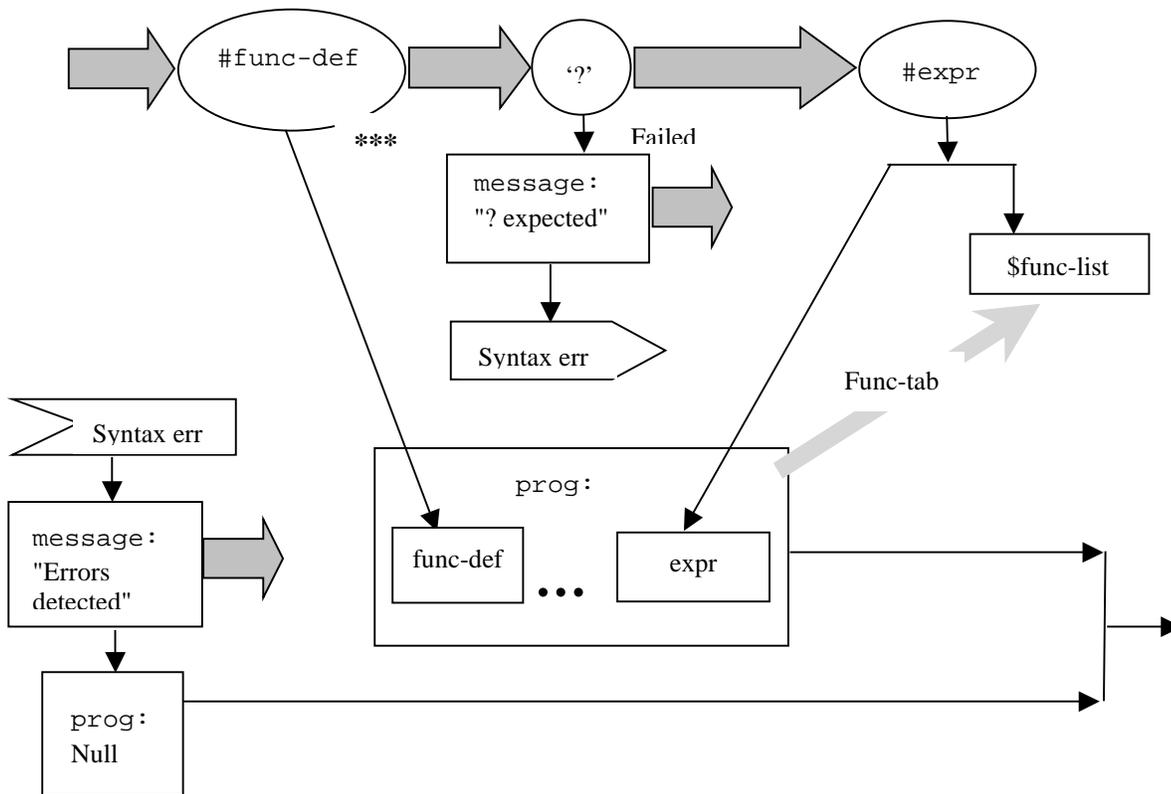
#func-def

'?'

#expr

***

Failed

message:
"? expected"

$func-list

Syntax err

Func-tab

Syntax err

message:
"Errors
detected"

prog:

func-def ... expr

prog:
Null

Figure 3. Parsing rule for the grammar rule
        program ::= func-def * '?' expression

Annotations for the rule #program

- This rule has a state $func-list which will be gradually updated by the rule #func-def calls (see Figure 4). At the end of parsing, object $func-list will be added as an attribute (via association with the name Func-tab) to the resulting object of the type prog. The box containing $func-list has a dummy input of the type ANY, which is activated when the last pattern #expr terminates with success. This ensures the timing when the state value is picked up for the association operation.

- The rules #func-def and #expr are used as patterns. If pattern matching encapsulated in these rules is successful, the rules also are successful and return values, which are used to assemble the return value of the rule #program.

- If pattern matching for the pattern '?' fails, the entire rule #program also fails and returns object Null, but before it happens two messages will be sent to the output stream. Markers labeled 'Syntax err' are used to prevent a mess with arrow intersections.

- A data flow fork denotes duplication of the data item sent to two or more threads.

- Nesting boxes and forwarding output of pattern rules of the types func-def and expr inside the resulting box of the type prog provide an intuitive visualization for the tuple constructor.

- The application of pattern #func-def may be repeated zero or more times (indicated by the ellipsis '***'), and it is synchronized with the tuple constructor (as the box of the type func-def in the resulting prog box is also accompanied by an ellipsis).

```
#func-def: Stream [char]-> Func-def, Stream [message]
           state $param-list: [id]    -- used in #expr
```
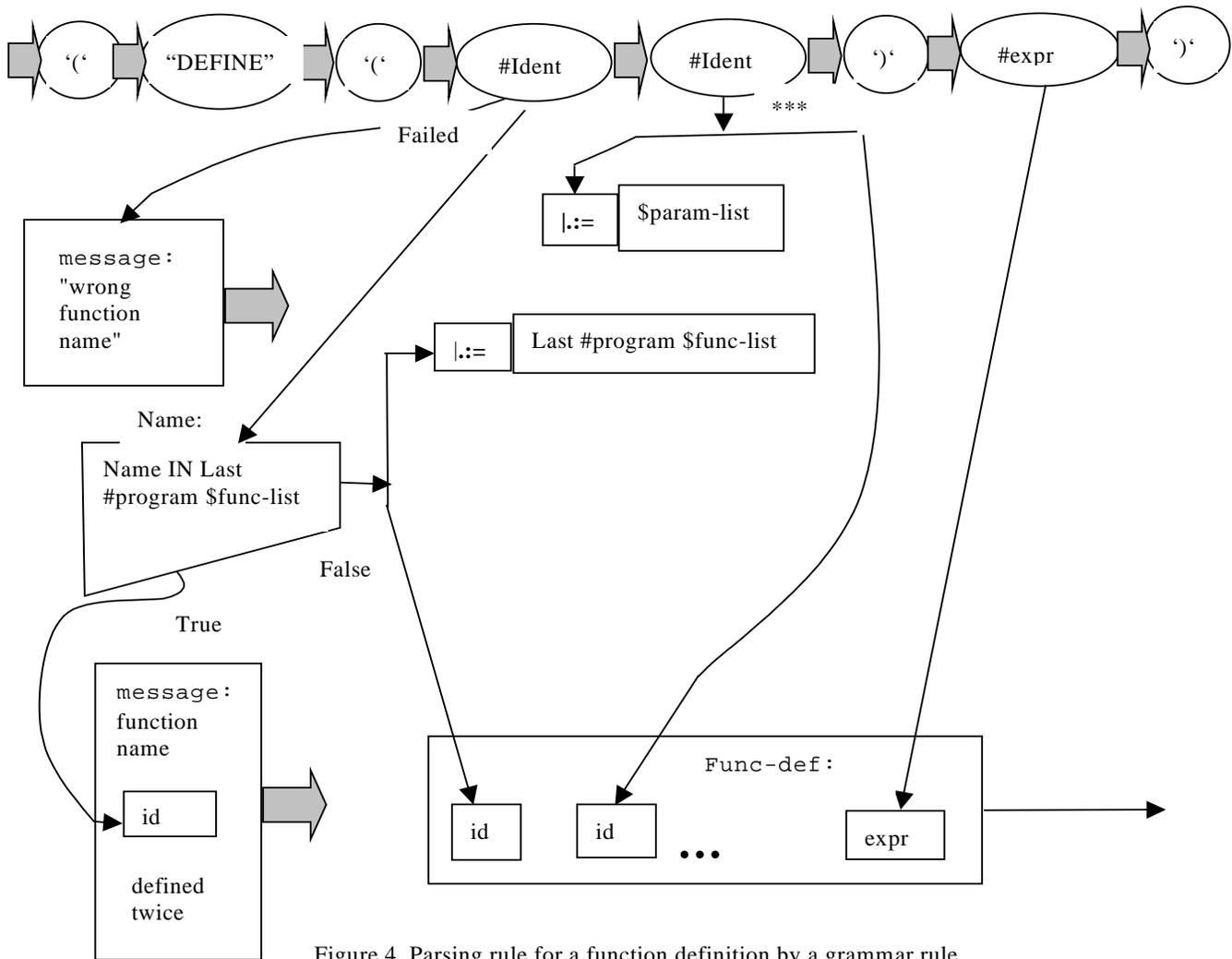

Figure 4. Parsing rule for a function definition by a grammar rule
Function-definition ::= '(' DEFINE '(' Name Param * ')' Expression ')'

Annotations for the rule #func-def

- Built-in rule #Ident matches a character string that is an identifier. When successful, this identifier (an object of the type id) is input to the conditional data flow switch to check whether the function name is already on the list. If true, the id item is forwarded to the message output stream. If false, it goes to the resulting tuple constructor.

- A function name is also sent to update state $func-list in the current instance of rule #program. |.:= stands for the operation to append an element to the end of list. This assignment operation updates the state Last #program $func-list.

- The entire sequence of patterns in this rule consumes part of the input stream delegated from the calling rule #program.

- Parameter names are appended to the state variable $param-list. All state variables are initialized by Null, which stands for empty list in this case.

```
#expr : Stream [char] -> expr, Stream [message]
```
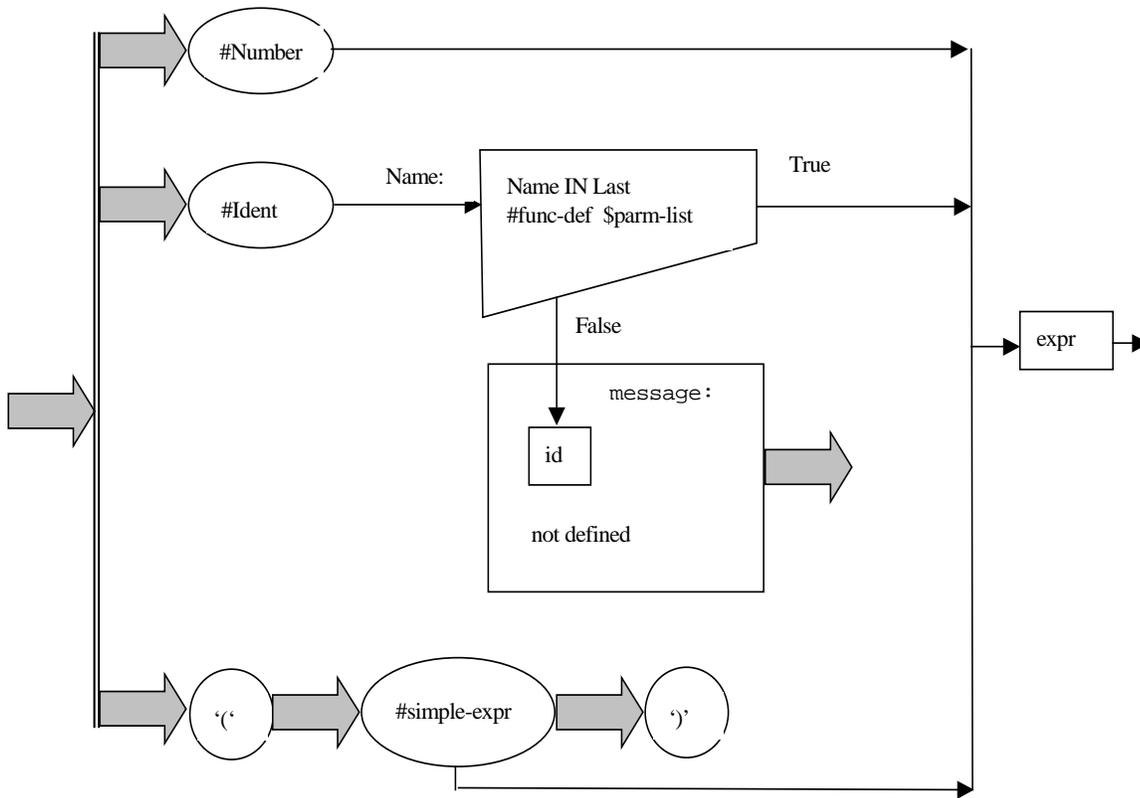

Figure 5. Parsing rule for MicroLisp expression for the grammar rule
```
expression::= integer | parameter-name | '(' SimpleExpression ')'
```

Annotations for the rule #expr

- A pattern may have several alternatives. The alternatives are applied in order of appearance, if the first alternative fails, the pattern matching backtracks in the input stream and the next alternative is applied until one of alternatives is successful. If all alternatives fail, the entire alternative pattern also fails.

- The built-in rules #Number and #Ident, when successful, return objects of the types `int` and `id`, correspondingly. Since the type expr is defined as a supertype for `int` and `id`, the data flow to the resulting object of the type expr is consistent.

3.2   Code generation

Code generation rules take as input a MicroLisp abstract syntax object and output C abstract syntax objects. Target code template representation in the diagrams is based on default mappings for C abstract and concrete syntax and visual representation of append operation as nested boxes.

Annotations for the rule #gen-program

- The input is of the type prog (abstract syntax object for MicroLisp) and a pattern for this object provides an access to the component retrieval. Since func-def components may be repeated zero or more times, the ellipsis in the pattern represents the iterative traversal.

```
#gen-program: prog -> C-HeaderFile, C-CodeFile
```
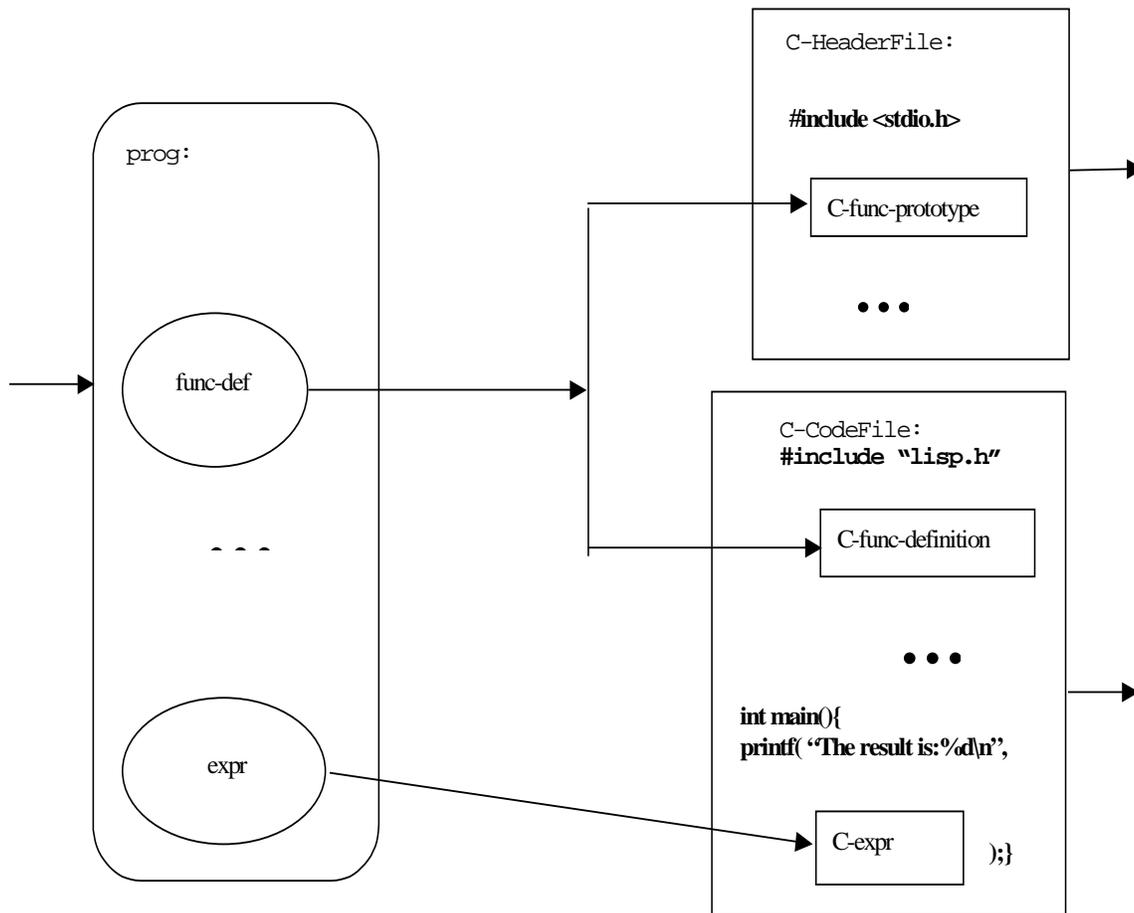

Figure 6. Generation rule for the MicroLisp program level

- The iteration of the input is synchronized with the iterative generation of objects in two outputs. The transformations itself are carried by default mappings `func-def -> C-func-prototype` and `func-def -> C-func-definition`. The rule #gen-function-prototype in the next example gives the algorithm for the first of these default mappings. Since the template provides particular concrete syntax for parts of the C code, those text segments will be stored with corresponding C abstract syntax objects. The resulting parse tree for include and printf will contain objects of the type id and text-string that hold values, such as "int", "printf", and other. These concrete syntax values are retrieved by default mappings when pretty-printing corresponding C abstract objects.

- The rule #gen-program constructs the target C code in the abstract syntax form. The mapping from abstract syntax to the text will be done according to the main diagram in Figure 1 by corresponding de-parsing default mappings for the C language. Both the abstract syntax definitions and default parsing and de-parsing mappings for the C language may be reused for any other meta-program that uses C as a target.

Annotations for the rule #gen-function-prototype

- This rule provides the flavor of hierarchical structure of generation templates.
- The first appearance of the string "int" in the target object C-func-prototype object will be converted by the C default parsing mapping into object C-type and the string "int" will be associated with it as a value. The same is true also for the iteration of "int" in the parameter list.
- Box around the second instance of "int" is needed to indicate the binding with the iteration of id in the source object func-def.

```
#gen-function-prototype: func-def -> C-func-prototype
```
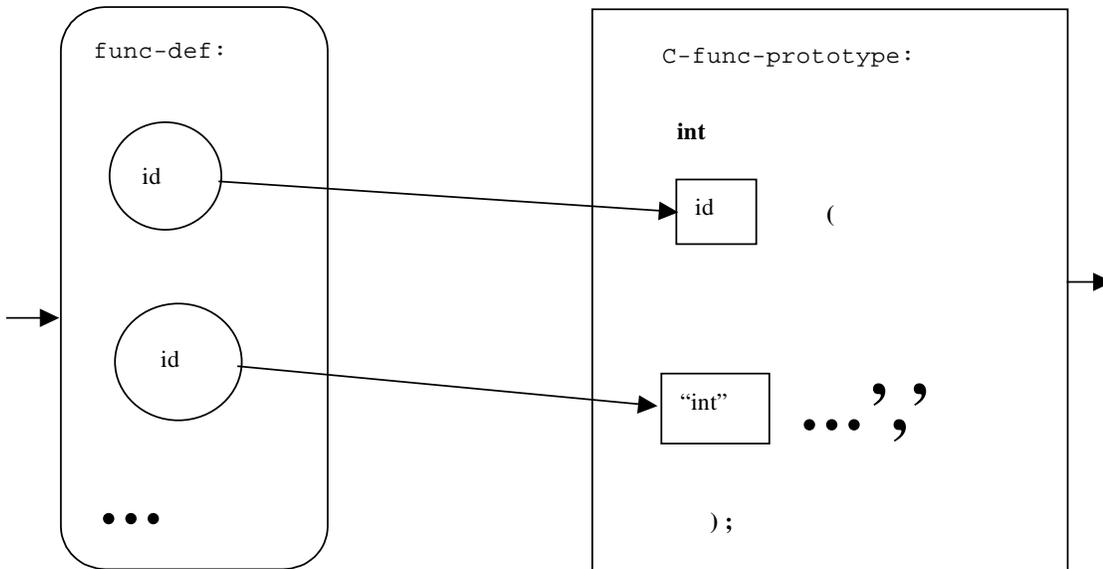

Figure 7. Generation rule for C function prototype.

- Parentheses, semicolon, and comma (as a separator between iterated elements; in the graphical interface there should be a way to indicate that comma is related to the iteration ellipsis) in the target object are optional, and if present, will be consumed by corresponding C default parsing mappings. The resulting object is still an abstract syntax object.

4   Preliminary conclusions

This paper presents very preliminary results on the visual notation for meta-programming. Work continues on the language itself, case studies, and implementation issues. At the moment of this writing the interpreter for the core of data-flow language is already implemented, and work is in progress on the graphical editor and advanced features like default mappings and tuple pattern matching. In it current form, the concepts presented may be used as a useful supplement to the meta-program design documentation. We expect the advantages of this approach to be as follows.

- Visualization of data and data flow provides for better readability and uncovers parallelism in data processing.
- The tuple type provides for a precise, disciplined, and flexible way to define abstract syntax.
- The simple association mechanism provides a natural way to introduce data attributes and opens the road for processing of arbitrary graphs without cluttering the language with additional means.
- Pattern matching notation covers in a uniform way data objects, rule calls, associations, and extended BNF notation for parsing.
- The language provides for systematic and consistent correspondence between constructors and patterns.

- The dynamic attributes (states) are actually macro extensions of pure functional paradigm (may be considered as additional inputs and outputs for diagrams referring to the states), provide for more efficiency, and make the data flow diagram simpler and less cluttered.

- Default mappings may be very convenient for generation templates, provide basis for lightweight type inference, and rule reuse.

- Data streams and patterns give a flexible and expressive framework for parsing rules supporting extended BNF notation, support reasonable and informative parsing error messages.

- Control mechanism, such as data flow switch, iteration and recursion fit well with data-flow notation and provide for transparent and expressive language to define different kinds of meta-programming algorithms.

References

[1] A.Aho, R.Sethi, J.Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986

[2] M.Auguston, "RIGAL - a programming language for compiler writing", Lecture Notes in Computer Science, Springer Verlag, vol.502, 1991, pp.529-564.

[3] M.Auguston, "Programming language RIGAL as a compiler writing tool", ACM SIGPLAN Notices, December 1990, vol.25, #12, pp.61-69

[4] M.Auguston, A.Delgado, Iterative Constructs in the Visual Data Flow Language, in Proceedings of IEEE Symposium on Visual Languages, Capri, Italy, 1997, pp.152-159

[5] E.Baroth, C.Hartsough, Visual Programming in the Real World, in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.21-42

[6] D.Batory, Gang Chen, E.Robertson, Tao Wang, Design Wizards and Visual programming Environments for GenVoca Generators, IEEE Transactions on Software Engineering, Vol. 26, No 5, May 2000, pp.441-452

[7] R. Bird, T. Scruggs, M. Mastropieri ,Introduction to Functional Programming, Prentice Hall, 1998

[8] P.T.Cox, F.R.Gilles, T. Pietrzykowski, "Prograph", in Visual Object-Oriented Programming, Concepts and Environments, (ed. M.Burnett, A.Goldberg, T.Lewis), Manning 1995, pp.45-66

[9] K.Czarnecki, U.Eisenecker, Generative Programming, Methods, Tools, and Applications, Addison Wesley, 2000, pp.832, ISBN 0-201-30977-7

[10] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A Complete, Flexible Compiler Construction System, Communications of the ACM, 35(2):121-131, February 1992.

[11] J. Levine, T.Mason & D.Brown, lex & yacc, 2nd Edition, O'Reilly, 1992

[12] Reasoning Systems, "Refine User's Guide", Palo Alto, 1992

[13] The Vienna Development Method: The Meta-Language, D. Bjorner et al, eds, LNCS 61, Springer 1978

Appendix A. Syntax of MicroLisp language and an example of a program

```
Program ::= Function-definition* '?' Goal-Expression

Goal-Expression ::= Expression

Function-definition ::= '('DEFINE'('Function-name Parameter-name*')' Expression ')'

Expression ::= Integer | Parameter-name |'(' SimpleExpression ')'

SimpleExpression ::= BinOperation Expression Expression | UnOperation Expression |
                     Function-name  Expression* |COND Branch + | READ_NUMBER

Branch::= '('Expression Expression ')'

BinOperation ::= ADD | SUB | MULT | DIV | MOD | EQ | LT |GT | AND | OR

UnOperation ::=  MINUS |  NOT
```

```
Function-name   ::= Identifier
Parameter-name    ::= Identifier
```

Example of a MicroLISP program.

```
( DEFINE ( gcd  x y)
   (COND (EQ x y)  x )
  ( (GT x y) ( gcd  (SUB x y)  y ) )
  (   1 ( gcd  x (SUB y x)  ) ) )  )
? (gcd  (READ_NUMBER)  (READ_NUMBER) )
```

Appendix B. Type definitions for MicroLisp -> C compiler

```
message::  [ char ]
program:: ( func_def* expr)| NULL
        attribute func_tab: [id]
func_def:: id id* expr
expr:: number | id |(op expr expr)|(op expr)|read_num | cond | function-call
function-call:: id  expr*
cond:: (expr expr)*
```
default mappings
```
 #prog: [ char ] -> prog
 #gen_program: prog -> C-HeaderFile, C-CodeFile
 #gen-function-prototype: Func-def -> C-func-prototype
 #gen-function-def: Func-def -> C-func-definition
 #pretty_print_prog: prog -> [ char ]
 ........
```
This is a sketch of a (over)simplified version of C abstract syntax.
```
 C_CodeFile::  include-statement * C-func-definition +
 C_HeaderFile:: include-statement C-func-prototype *
 C_func_prototype:: C-type func-name C-type *
 C-type:: id
 C_func_definition:: ..........
 C_expr:: .....
```
Default mappings include parsing rules and pretty-printing rules (abstract syntax to text mappings).