Visual Meta-Programming Language[1]

Mikhail Auguston[2], Valdis Berzins, Barrett Bryant[3]
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943 USA
{auguston,berzins,bryant}@cs.nps.navy.mil

Abstract

This paper presents a relatively simple visual notation for meta-programming that spans multiple levels of abstraction. Two-dimensional data flow provides readable representations of meta-programs that expose potential parallelism. This work suggests visual notations for data structures, data flows, pattern matching, conditionals, iteration and synchronization. The framework provides encapsulation means for hierarchical rule design, data item associations that enable creation of arbitrary graphs, and default mapping rules to reduce screen real-estate requirements. The representation supports practical reuse of generic data structures for program representation, abstract syntax type definitions for common programming languages, and related default mappings (e.g. parsing and de-parsing, module dependency graphs, class diagrams, etc.).

1 Introduction and objectives

Meta-programs are programs that manipulate other programs. Typical applications include program generators, source code static analyzers and checkers, compilers, interpreters, and pretty-printers. Domain-specific language implementation and rapidly evolving generative programming [11],[8] are the latest examples of developments in this domain. The complexity and sophistication of meta-programs may be quite significant, particularly in the context of program generators that seek to produce product quality software. Readability, maintainability, and reliability become issues of concern in such contexts. The objective of this paper is to outline an approach for addressing the first two of these issues. The third issue is discussed in [6].

Compiler and program 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 many domain-specific software design templates have been accumulated in the literature. The following domain features are among the most common for language processor design and have contributed to our language.

- The architecture of a language processor in most cases can be represented as data flow between components (e.g., the famous compiler data flow diagram on page 13 in the "Dragon Book"[1]).
- Typically, the main components of a language processor are very hierarchical and organized according to the structure of data (recursive descent parsers are an excellent example of this feature).
- Context-free grammars are used to specify syntax and to serve as a basis for parser design.

[1] This research was supported in part by the U. S. Army Research Office under grant number 40473-MA-SP, and by U.S. Office of Naval Research under grant N00014-01-1-0746

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

[3] On leave from University of Alabama at Birmingham, USA

- Attributes associated with data items, and attribute dependency and propagation schemes are of 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).
- Commonly used data structures include trees, graphs, lists, stacks, tables, and strings.
- Tree and graph traversal and transformation are common for optimization and code generation tasks.
- Pattern matching (e.g., with respect to regular expressions or context-free grammars) is a useful control structure for this problem domain.

These considerations and experience with the compiler writing tools RIGAL[2][3], lex and yacc[16], Kodiyak [14] and ELI[13] contributed to this work. The data-flow paradigm is quite natural for this kind of software design, and consequently, graphical notation for data-flow diagrams could be appropriate. This should be integrated with visualization of typical data structures, pattern matching, and encapsulation to support well-structured, hierarchical programs. For example, railroad diagrams have been used to represent syntax of many programming languages since their initial application to PASCAL. Data-flow diagrams are most commonly used to represent dependencies between data and processes in visual programming languages, for instance, in LabVIEW[7] and Prograph[10]. Two-dimensional diagram notation could significantly improve readability of meta-programs. Some of these ideas have been explored in our previous work[4].

2 Main Language Concepts

We have extended traditional data flow to account for conditional data flow switches, repetition, pattern matching, and associations between objects. These themes are elaborated below. We use a simple example of a compiler from a small subset of Lisp language (called MicroLisp) to the C target language to illustrate our visual language constructs.

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 set of two-dimensional data flow diagrams that shows the dependencies between data and functional computations. Each diagram defines a single function called a rule, and rule calls may be recursive. Data flow diagrams support the possibility of parallel execution of threads within a rule.

The data flow paradigm is closely related to the functional programming paradigm [9] and shares with that paradigm referential transparency and good correspondence between the source code (the diagram) and the structure of rule execution.

Each diagram represents a single function with several inputs and outputs. A signature at the top of a diagram provides the rule name and types of its inputs and outputs. The body of the diagram is a graph containing several different kinds of nodes and edges, summarized in Figure 1. A diagram has one or more input ports and zero or more output ports.

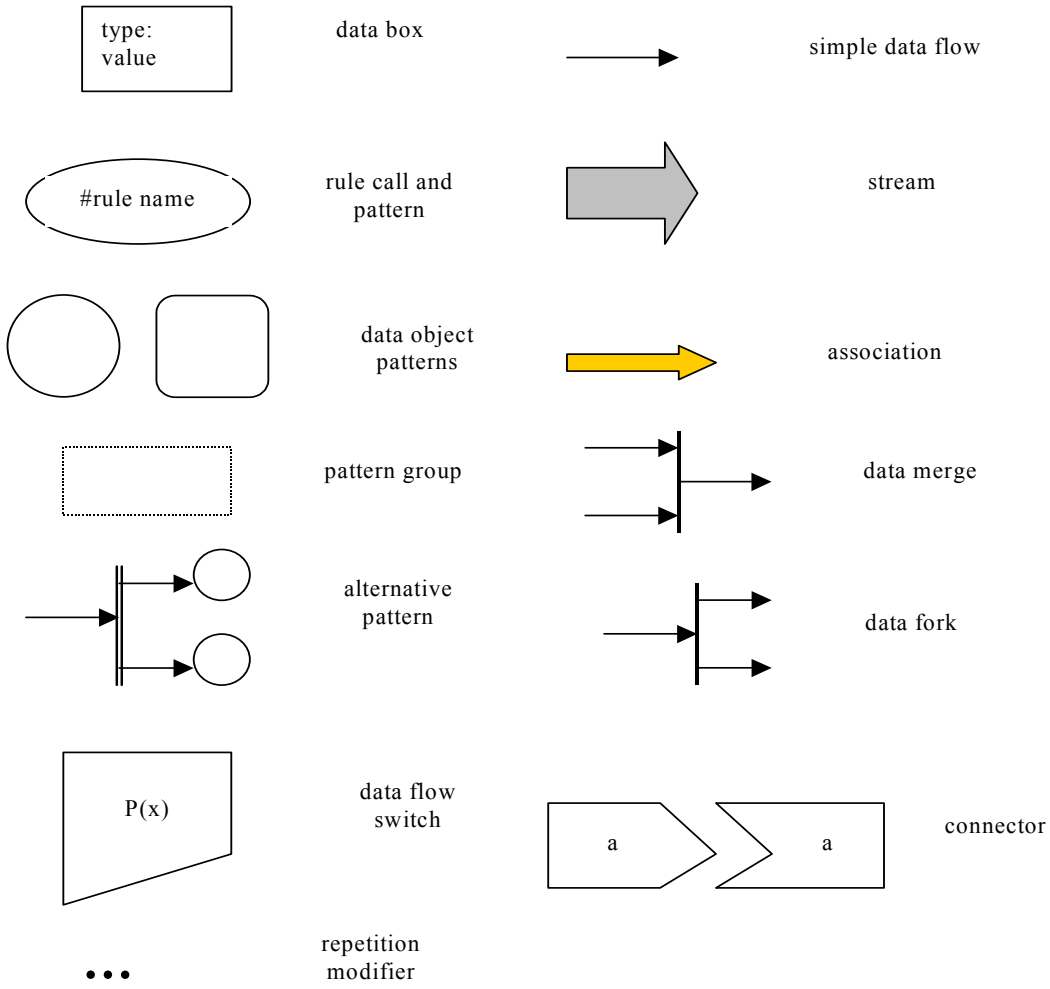

Figure 1. Basic icons for meta-programming language

The rectangular boxes in our notation denote values, and ovals denote patterns, that could be matched with data objects. A rectangular box representing a data item may have names associated with its input ports, those names can appear in the expression within the data box and are visible only within this box. A simple data flow delivers values one at a time. There is never more than one data item in a simple data flow channel between sender's output port and receiver's input port.

A stream is similar to a simple data flow, except that it contains an unbounded sequence of values, and can deliver as many of those values as needed to match the guarding patterns of a rule. A data box fires when and only when the following conditions are satisfied:

- all input values are delivered to the corresponding input ports;
- all simple data flow output values produced at the previous execution cycle are consumed by input ports of the connected nodes downstream (output streams are always available for output).

A fork node fires when its input port receives a value and all output values from previous cycle have been consumed by input ports downstream. Semantics is similar to the transition node in Petri net[17]. A

merge node fires when at least one input port receives a value and passes it to the output port. A merge node preserves the temporal order of data items arriving at its input ports. This fairness property guarantees that each input value will be processed. A merge node may be considered as an analog of a place in Petri net[17].

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 output object is passed downstream. If pattern matching fails (or all branches of an alternative pattern fail) then 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. In the event of a rule failure all input values are restored to the input channels, regardless of whether they are simple values or streams.

If a rule's input is a stream, 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 4-6).

Rules can create output streams of objects as well.

2.2 Types

A 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, set or tuple).

Aggregate types are trees (ordered tuples of heterogeneous objects), which are useful for abstract syntax representation, sets 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[19] and Refine[18]. Types can be parameterized.

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 :: [alphanumeric],` 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 Associations

Data objects may be associated with other data objects. Associations are optional named attributes of particular objects that can be single valued or multiple valued. Associations may be used to create arbitrary graphs from objects. Figure 2(a) illustrates the creation of a graph structure via associations from

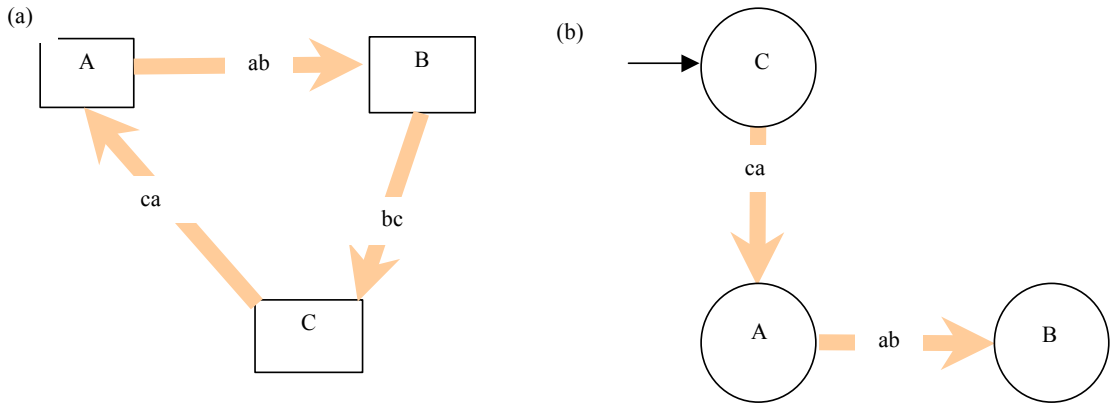

Figure 2. Construction of associations between objects and retrieval via pattern matching

three data objects. Association is not symmetric. According to the following diagram object A has been associated with an object 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 that retains only those components declared in the type definition. Associated objects can 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. We illustrate this mechanism on graph processing examples in sec. 3.1 and 3.2.

2.4 Default mappings

Selected rules may be declared as default mappings. There can be at most one default mapping from the given source types to the given target types. The type system uses name equivalence, so that it is easy to create many disjoint but isomorphic subtypes of a given type. This facilitates control of 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 are introduced for text-to-abstract syntax (parsing) and for abstract syntax-to-text mappings (de-parsing, or abstract syntax-to-concrete syntax mappings).

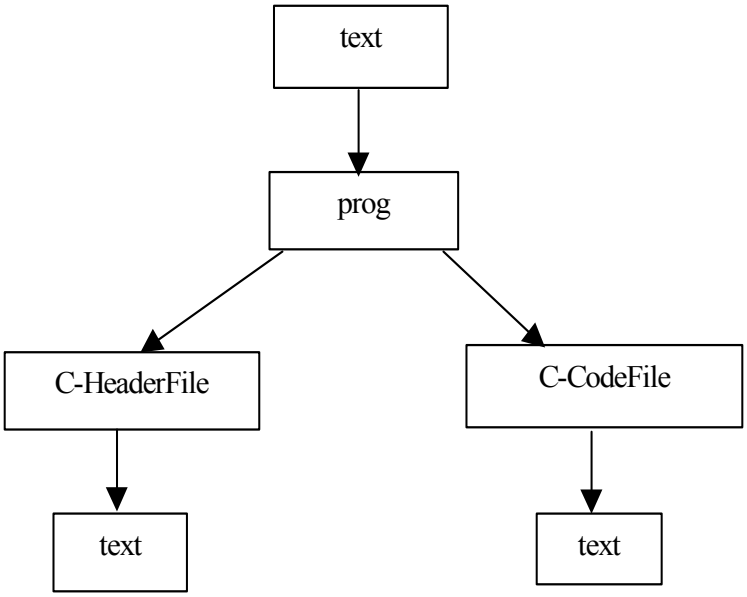

Figure 3. The top level data flow diagram for MicroLisp to C compiler

Yet another kind of default mapping is associated with construction and concatenation operations for tuples, sets and lists. See MicroLisp generation rules for examples (Figures 7-8).

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 the subtype relation.

3 Examples of source to source translation

This section illustrates source-to-source translation in the context of a simple MicroLisp -> C translator. It illustrates most of the concepts and notations discussed above as well as many issues that arise in higher level applications of program generators. Rules are deployed according to the data flow diagram in Figure 2 and default mappings in Appendix B. Appendix A contains the context-free grammar for MicroLisp and an example of a program.

3.1 Parsing

This section shows that the same notations that we use for higher level meta-programming tasks can also be used to define the rules for a parser generator. The source code of MicroLisp program is represented as a stream of strings representing tokens. It is assumed that there is a lexical module that filters out comments, spaces, tabs, end-of-line characters from the stream before it is fed to the parsing rules.

Remarks for the rule #program

- A rule pattern can consume arbitrary many data items from a stream.
- When a pattern succeeds, the matching data items are tentatively removed from the stream.
- When a pattern fails, all data items that were matched by parts of the pattern are restored to the stream, including indirectly invoked sub-patterns.
- 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

```
#program: Stream [string]-> prog, Stream [message]
```

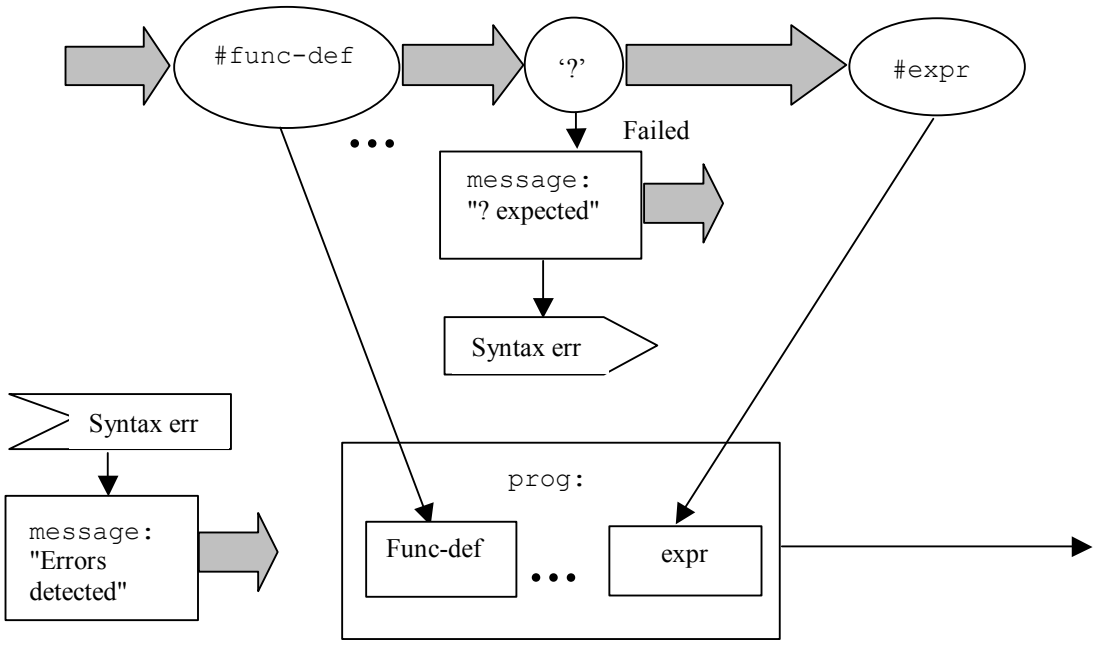

Figure 4. Parsing rule for the grammar rule
`program ::= func-def * '?' expression`

Null, but before it happens two messages are sent to the output stream. Connectors labeled 'Syntax err' are used to prevent a visual mess with arrow intersections. They have no semantic significance, other than connecting the incoming arrow to the arrow leaving the matching connector.

- 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).

Remarks for the rule #func-def

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

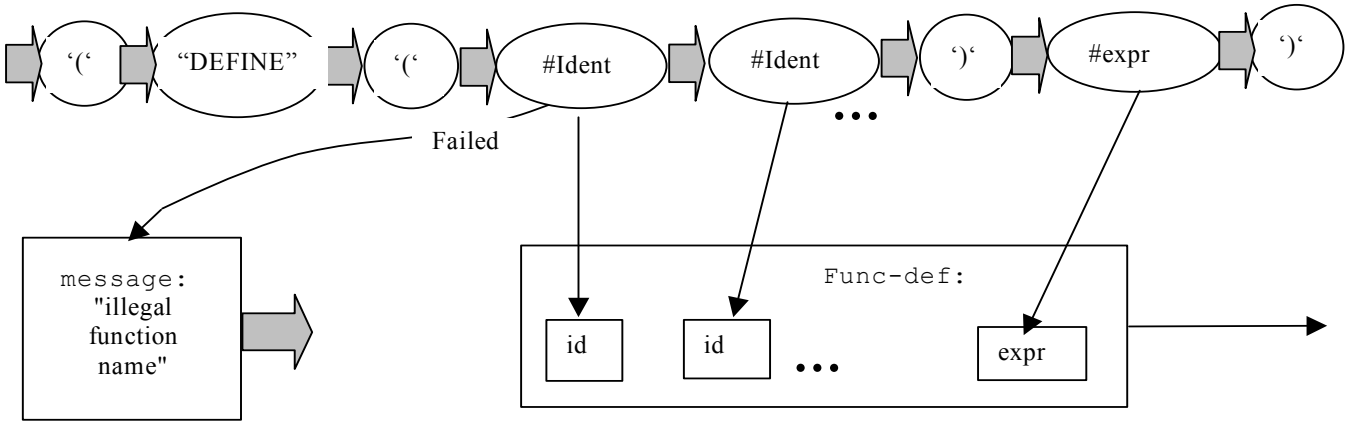

Figure 5. Parsing rule for a function definition by a grammar rule
Function-definition ::= ‘(‘ DEFINE ‘(’ Name Param * ‘)’ Expression ‘)’

- The built-in rule #Ident matches a character string that is an identifier.
- The entire sequence of patterns in this rule consumes part of the input stream delegated from the calling rule #program.

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

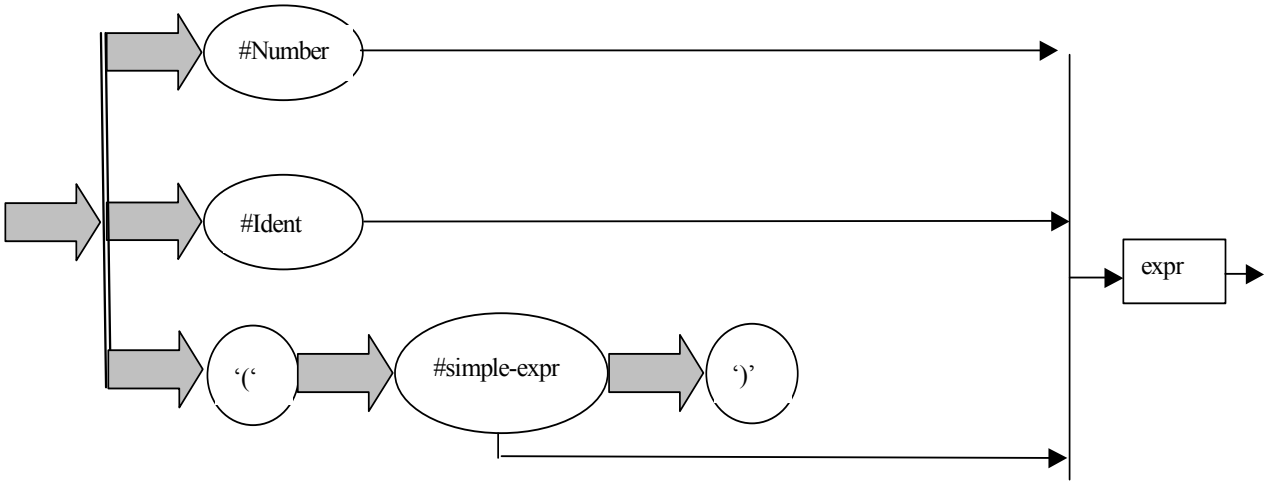

Figure 6. Parsing rule for MicroLisp expression for the grammar rule
expression::= integer | parameter-name | ‘(‘ SimpleExpression ‘)’

Remarks for the rule #expr

- A pattern may have several alternatives. The alternatives are applied in order of appearance, if the first alternative fails, the input stream is restored to its state before the first alternative 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 with the type system.

3.2 Code generation

The code generation rules take a MicroLisp abstract syntax object and produce 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 operations as nested boxes.

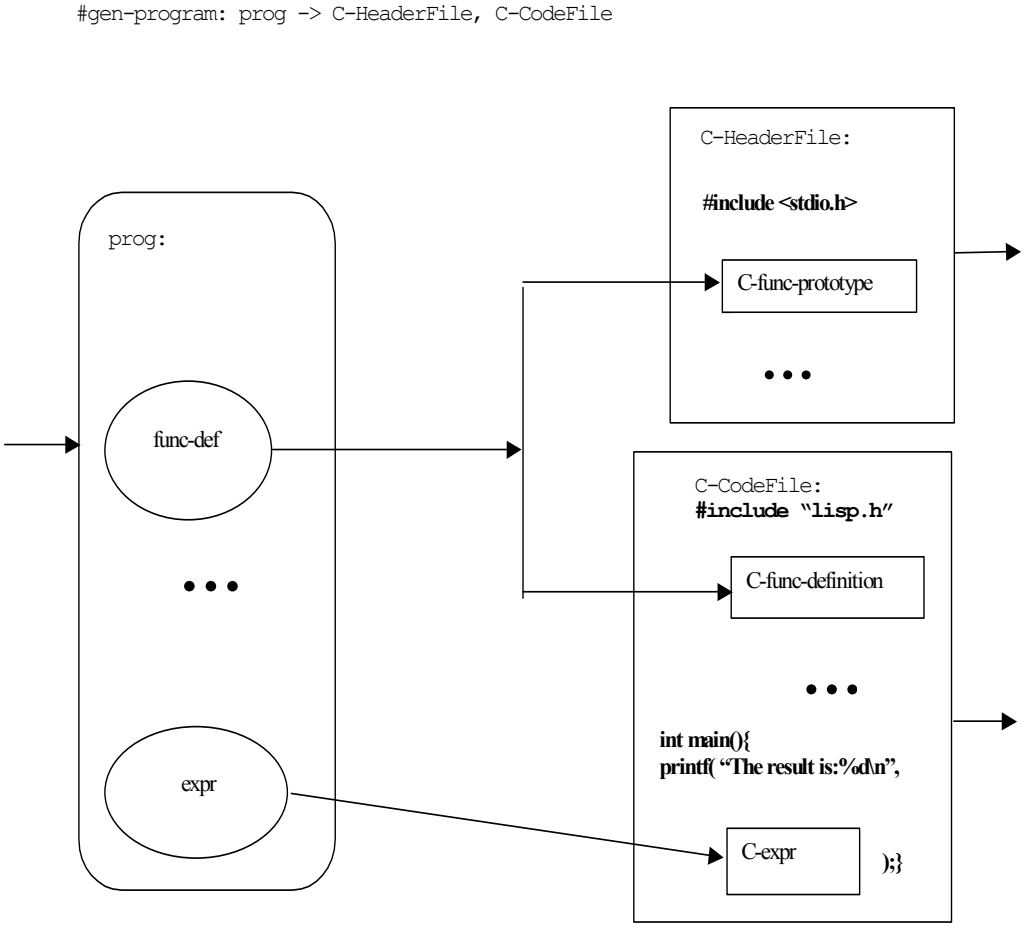

Figure 7. Generation rule for the MicroLisp program level

Remarks 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 its subcomponent retrieval. Since func-def components may be repeated zero or more times, the ellipsis in the pattern represents an iterative traversal.
- The iteration of the input is synchronized with the iterative generation of objects in two outputs. The transformations are carried by default mappings `func-def -> C-func-prototype` and `func-def -> C-func-definition`. The rule #gen-function-prototype in Figure 8 gives the algorithm for the first of these default mappings.
- The constant parts of the generation templates are shown using the concrete syntax of the target language in the visual notation since this is more compact and readable than abstract syntax. This is a shorthand for an abstract syntax constructor that is generated by the processor for the visual language.
- The final source file is produced by a default mapping that produces a pretty-printed text representation from the abstract syntax tree, as shown in Figure 3.
- 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.

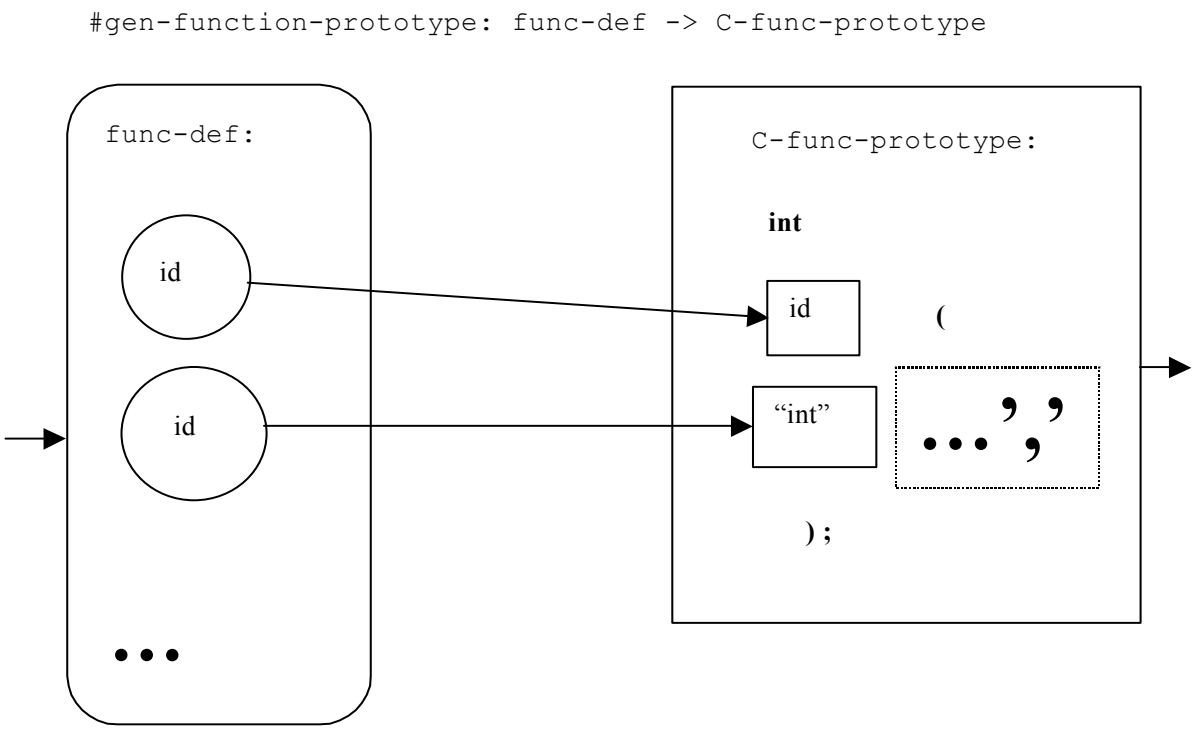

Figure 8. Generation rule for C function prototype. This is a default mapping for `func-def -> C-func-prototype`

Remarks 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 a C-type object 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 synchronization with the iterated id pattern in the func-def pattern.
- Parentheses, semicolon, and comma (as a separator between iterated elements; the dotted grouping box indicates that the comma is part of 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 Conclusions and work in progress

This paper presents a visual meta-programming language for program generation. A first draft of this paper appears in [5]. The language supports parsing, domain-specific operations (abstract syntax processing, attribute evaluation, generation templates) and error messages, code optimization, and code generation, all of which may be used in conventional compiling or program generation from high-level specifications. The readability of program generation templates is facilitated through the visual representation. Readability is enhanced further by the use of pattern matching which combines declarativeness with clear and transparent data flow.

We expect the advantages of this approach to be as follows.

- Visualization of data and data flow provides better readability and uncovers parallelism in data processing.
- The tuple type provides 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 systematic and consistent correspondence between constructors and patterns.
- 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 mechanisms, such as data flow switch, iteration and recursion well fit with data-flow notation and provide a transparent and expressive language to define different kinds of meta-programming algorithms.

The examples in section 3 illustrate how the proposed notation performs for defining program generation rules at a variety of levels of abstraction, ranging from graph models to parsing. Although the space

does not permit a detailed example, the same notation applies naturally also to lexical analysis at the level of individual characters.

Work is continuing 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 the progress on the graphical editor and advanced features like default mappings.

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 , IEEE Computer Society Press, 1997, pp.152-159

[5] M.Auguston, Visual Meta-Programming Notation, in Proceedings of the Monterey Workshop "Engineering Automation for Software Intensive System Integration", Monterey, California, June 18-22, 2001.

[6] V. Berzins, "Static Analysis for Program Generation Templates", 2000 ARO/ONR/NSF/DARPA Workshop on Modeling Software System Structures in a Fastly Moving Scenario, Santa Margherita Ligure, Italy, June 12-16, 2000.

[7] 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

[8] 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

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

[10] 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

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

[12] Glaser H., Smedley T., PSH - the next generation of command line interface, in Proceedings of the 11th International Symposium on Visual Languages, VL'95, IEEE Computer Society Press, 1995, pp.29-36

[13] 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.

[14] R. Herndon, V. Berzins, "The Realizable Benefits of a Language Prototyping Language", IEEE Transactions on Software Engineering, June 1988, pp. 803-809.

[15] T.D.Kimura, Object-Oriented Dataflow, in Proceedings of the 11th International Symposium on Visual Languages, VL'95, IEEE Computer Society Press, 1995, pp.180-186

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

[17] J.Peterson, Petri net theory and the modeling of systems, Prentice-Hall, 1981

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

[19] 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* '?' 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 2. Type definitions for MicroLisp -> C compiler

```
message::  [ char ]
prog:: ( func_def* expr)| NULL
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
```

Default mappings include parsing rules and pretty-printing rules (abstract syntax to text mappings).