JAMOOS*
A Domain-Specific Language for Language Processing

Joseph (Yossi) Gil† and Yuri Tsoglin‡

† Department of Computer Science, Technion Israel Institute of Technology, Haifa, Israel
‡ IBM Research Laboratory, Science Industrial Park (MATAM), Haifa, Israel

JAMOOS is a cohesive suite for quick definition of attribute grammars and generation of compilers, interpreters and other language processing tools. As a programming language, JAMOOS brings a new *tree computing metaphor* which unifies the notions of object creation, procedure call and reduction of rules in an attribute context-free grammar. JAMOOS has a rich object oriented-type system, with features such as genericity, lists, union and unit types. This type system serves also as a language for specifying context-free grammars.

Keywords: compilation, programming paradigms, object oriented languages, immutability, unified language

1. Introduction

Much research and practical work has revolved around the theme of *correspondence* between BNF grammars and object-oriented (OO) type hierarchies (see e.g., [50, 51, 5, 14, 21] as well as Koskimies's survey [28]). This correspondence, which was even captured in a reusable format as the INTERPRETER design pattern [12], stands upon three underlying principles:

1. *Nonterminal symbols of a grammar, henceforth called symbols, are classes.* A grammar production defining a symbol in a grammar is also thought of as a class definition that represents the type of all input specimens that may be reduced by this production.
2. *Sequence is record structure.* Production of the form $\mathsf{A} \rightarrow \mathsf{B}\ \mathsf{C}$ is understood as a definition of a class A, comprising two components of types B and C. This is because each occurrence of symbol A in the input has both a B and a C in it.
3. *Alternation is inheritance.* Production of the form $\mathsf{X} \rightarrow \mathsf{Y} \mid \mathsf{Z}$ defines an abstract class X with the two classes Y and Z inheriting from it. The intuition is that every input instance of the symbol X is either a Y or a Z.

Actual implementation of these principles must place some constraints on the grammar. For example, it is imperative that each symbol is defined by one production only, and productions which give rise to multiple inheritance should not be allowed. In the Mjølner BETA language processing system [26], one of the most prominent language processing systems using these principles, grammar productions are restricted [34] to be one of exactly three kinds: *alternation*, *composition*, or *list*. The inheritance structure in this system is also limited since, e.g., all the alternatives in an alternation must either be all compositions, or all lists. Similar restrictions are imposed by YOOCC [5].

This research also dwells on the correspondence between OO types and BNF grammars. Our motivation and main objective was to carry out a *unification* of the concepts of *language processing* and *OO programming* to the fullest pos-

* JAMOOS (Arabic) — Water Buffalo inhabiting the *Hula* swamp in northern Israel.
† contact author
‡ Work done while author was with the Technion

sible extent. The result took the form of the JAMOOS[1] programming language with its *inherent* dual nature. On the one hand, JAMOOS is a a general purpose OO programming language, whose unique features are the subject of this paper. On the other hand, it is a cohesive suite for quick definition of attribute grammars and generation of compilers, interpreters and other language processing tools. (This other aspect of JAMOOS is not discussed here in detail.) Thus, programs written in JAMOOS can be read both as a formal language specification, and as an OO program.

The pursuit of duality mutually benefited both the linguistic and the grammatical aspects of JAMOOS. As a grammar processor, JAMOOS makes a contribution by presenting a modern, practical and integrated tool with facilities for modular specification of grammars. From the linguistic point of view, JAMOOS steps forward in the visionary path outlined by Ole L. Madsen in the case he made for a unified programming language [32].

This paper describes two aspects of JAMOOS which make this dual interpretation possible: (i) a rich OO type system, which can be used also to specify context-free grammars, and (ii) a new *tree computing metaphor*, which unifies parsing with procedural- and OO- programming.

JAMOOS is a useful tool for a quick definition of domain-specific languages. Further, JAMOOS itself is an example of a domain-specific programming language, demonstrating an interesting language design principle: "identify the main kind of computation in the specific domain and unify this with familiar programming constructs". In language processing, the main computational task is in creating and processing syntax trees. As we shall shortly see, the tree computing metaphor unifies this with notions such as the tree of routine calling.

Some of the examples in this paper describe JAMOOS in its own syntax. Other examples are drawn from the JAMOOS specification of PASCAL [49]. JAMOOS was also used in prototyping prog2TEX, the program used in pretty printing the code excerpts in this paper.

Outline

The following Sec. 2 describes the tree computation metaphor and the process of creating and discarding objects. This section also explains how the computation is guided by *definitions*, each specifying a production rule, class, and procedure. Definitions are comprised of *fields*, which can each be thought of as a computational step, a data member, or an argument or internal variable in procedural programming. Fields are described in Sec. 3. Definitions are only part of the complete JAMOOS type system, which is the subject of sections 4 and 5. Sec. 6 explains how *internal definitions* make it possible to associate methods (in the OO sense) with productions. Sec. 7 describes abstract definitions and inheritance. A brief report on some of the other lingual features of JAMOOS is in Sec. 8. (More information is available in the JAMOOS definition document [48] or on the web[2].) Related work is the subject of Sec. 9. The paper ends with Sec. 10 which gives the conclusions and outlines some directions for future research.

2. Principles of Tree Computing

The *tree computation metaphor* brings together non-mutating procedural programming and the OO programming paradigms. (Non-mutating procedural programming is procedural programming in which variables are immutable; it is similar to non-high level functional programming.)

In this metaphor, computation is all about the construction of immutable objects, aggregating them into trees, and then discarding these trees. As soon as the construction of an object *b* is terminated, it is passed as an argument to the constructor of its containing object *a*. If *a* discards the reference to *b*, then *b* along with all other objects contained in it, is destroyed. If object *a* keeps a reference to *b*, then *b* becomes a part of *a*, and will only be destroyed when *a* is.

Objects can also be thought of as procedure activation records (sometimes called stack frames),

[1] Unlike many other made-up names beginning with *J*, JAMOOS has absolutely no relationship with the JAVA [2] programming language.

[2] `http://www.cs.technion.ac.il/Labs/ssdl/thesis/finished/2001/tyuri` or `http://www.cs.technion.ac.il/~tyuri/thesis`

while the aggregation relationship between them defines the program call tree. With this unification, a call to a routine is the same as the construction of the activation record of that routine. JAMOOS in fact restricts the computation in a routine to sequentially filling the slots in its activation record. Object immutability in procedural perspective means that once a routine is terminated, its activation record cannot be changed. However, unlike ordinary procedural programming, the activation record is not necessarily discarded when the routine finishes. Thus, tree computation generalizes the familiar call stack regime in that it is possible to access subtrees of activation records and discard them.

JAMOOS programs comprise *definitions*, which give types to the nodes in the aggregation hierarchy. Each definition can be thought of as being simultaneously a routine or a function in procedural programming or an OO class definition. (An activation record captures the run time realization of the execution of a routine, in exactly the same way that an object captures the run time realization of a class.) An *evaluation* of a definition can be thought of as either a constructor call, or as an invocation of a routine.

There is yet a third unifying aspect to definitions, which can be thought of as a specification of a production rule in BNF. An evaluation is then a commitment to this rule at a certain point in the input during the parsing process.

Example 2.1.

A → B _:C ;

The above simple JAMOOS definition can be read as a parsing rule, where symbol A comprises B and C in sequence. It can also be thought of as a definition of a routine A, whose execution has two steps: execution of routine B and then routine C. Routines A and B can be also thought of as constructors of objects of classes A and B (respectively). Execution of routine B can be thought of as constructing the activation record *b* of class B. Once the creation of *b* is finished, it is passed as an argument to the constructor of its containing object of class A. The _: prefix of C in the definition means that the C object is not stored in an A object. In other words, the object C is created in the process of the construction of A. However, once this construction is terminated, object C is discarded. We say that C is *perishable*.

In the grammar-world, the object hierarchy is then both the *concrete-* and *abstract-* syntax trees, where only non-perishable objects take part in the *abstract* syntax, and a definition gives also a type to a node of the abstract syntax tree.

3. Fields

A definition is made as a series of *fields*. This section presents various kinds of fields through a series of examples.

Example 3.1.[3]

PascalProgram → **program** Name
decls:Declarations Body ;

In the OO perspective this is a definition of a class PascalProgram, consisting of four fields. The first field (the keyword **program**) is degenerate — it is of the familiar *unit* type (used in languages such as ML [42]), the type with exactly one value. The unit type is called **OK** in JAMOOS for reasons which will become clear in the following section. The other three fields are objects of types Name, Declarations and Body, respectively.

In the grammatical perspective, Examp. 3.1 is a grammar rule for the *symbol* PascalProgram.[4] Fields Name, Declarations and Body are symbols which must be defined elsewhere. Field **program** is a *token*, specifically, a keyword of the specified programming language, PASCAL in this case. Such tokens are pretty-printed in **bold**. Values of type **OK** need not be stored, and indeed, keywords are not stored in the abstract syntax tree. The distinction between different keywords is part of the parsing process only.

In the procedural perspective, the above definition is of procedure PascalProgram which is implemented as calls to three other procedures: Name, Declarations and then Body.

[3] We apply pretty printing of JAMOOS code throughout. The actual type written version is
`PascalProgram -> program Name decls:Declarations Body ;`

[4] Following Meyer's specification of EIFFEL symbols must be capitalized, and keyword tokens need not be quoted, as long as they begin with a lower case letter.

	Field Kind	Life Time		Has Initializer?	Routine Equivalent	Grammatical Meaning
		Begin	*End*			
Arguments	Component	constructor invocation	with object	No	IN-OUT argument	abstract syntax
	Perishable	inconstructor invocation	constructor return	No	IN argument	concrete syntax
Features	Attribute	initialization by constructor	with object	Yes	OUT argument	semantic feature
	Temporary	initialization by constructor	constructor return	Yes	Local variable	none

Table 1. The kinds of fields in JAMOOS .

decls is an identifier which names the field Declarations. The definition does not name the fields of type Name and Body. These fields can still be referenced using JAMOOS automatically generated names. This automatic mechanism makes it possible to refer in a definition to a field of type t simply as t, provided that there are no other fields in the definition whose type is t. We use this convention throughout this paper.

Example 3.2.

```
CondStatement → if Expression then
    then_part:Statement else else_part:Statement ;
```

In this example, the first field if has no names at all, and it cannot be referenced. The second field of type Expression can be referred to as Expression or as 1 because it is the first field which is not of type **OK**. Both these names are automatically generated by the JAMOOS naming mechanism. The fourth field can be referred to as then_part, Statement#1 (the *first* Statement) or 2. The sixth field also has three names: else_part, Statement#2 and 3.

In general, a field specification has four parts to it.

1. *Optional perishability prefix.* All fields prefixed by an underscore (_: as in Examp. 2.1) are discarded at the end of the evaluation of the definition.
2. *Optional name.* Anonymous fields receive their names using JAMOOS 's name deduction mechanism. Note that a field might have both perishability prefix and a name.
3. *Type.* This is often the name of another definition, but could be any one of JAMOOS primitive or compound types.
4. *Optional initializer.* Uninitialized fields are arguments to the constructor. Initialized fields can be thought of as computational steps.

As shown in Table 1, there are four kinds of fields, distinguished by their time of initialization and lifetime (both determined by the perishability prefix and the existence of initializer).

In examples 3.1 and 3.2 all fields are *components*, i.e., fields whose lifetime is the same as that of the containing object. In contrast, C in Examp. 2.1 is a *perishable*, i.e., a field which ceases to exist at the end of the constructor. All constructor calls of a certain class must pass as argument initial values to all components and perishables. Perishables and components are known collectively as *arguments*.

Attributes are fields whose value is computed at construction time from the components, perishables and other previously computed attributes. Attributes, which are just like attributes of *attribute grammars*, end their life with that of their containing object. The syntax of a full attribute definition is

Example 3.3.

```
AttributeDefinition→name:Id ":" Type ":=" Initializer ;
```

As long as the object exists, its attributes (just like its components) can be queried from outside. *Temporaries* are simply *immutable* local variables of the constructor; their life span is from initialization until the end of the constructor. Temporaries are defined by prefixing an attribute definition with _:.

Collectively, attributes and temporaries are called *features*. Features are commonly computed from arguments. In most cases features are defined after the arguments. However, this order

is not mandatory, and attributes may be arbitrarily mixed with arguments. In the grammatical perspective mid-rule actions [1], are a prime example of mixing attributes and arguments.

In mixing arguments and attributes with arguments, an interesting phenomenon occurs: some of the attributes must be evaluated before the arguments. A caller of a constructor cannot simply compute the arguments and then pass these along. Instead, in this case JAMOOS uses a *call by name* mechanism, in which the caller of a definition does not transfer values, but rather *thunks* to compute these.

Table 1 gives also the procedural perspective interpretation of the various kinds of fields. Input arguments are perishable fields—they must be passed as arguments and cease to exist when the evaluation terminates. Output arguments are attributes which are the exact converse: they are initialized during evaluation and can be examined from outside as long as the activation record exists. Input-output arguments are only approximated by components. They are passed to the definition from outside and retain their value as long as the activation record exists. The value of a component cannot be changed during evaluation. Finally, temporaries are just like local variables.

A field with the reserved name return makes it possible to think of the definition not only as a procedure but also as a *function* which returns this field's value. This unification is realized by having the return field the default value of a definition, i.e., the field to use when some object typed by this definition is addressed without any specific field selector.

Example 3.4.

```
Addition → Term "+" Term
    return:INTEGER := [[ return $Term#1
                                + $Term#2; ]] ;
```

A few comments are needed to understand the above example. The [[...]] construct switches from JAMOOS to C++; in $Term#1 and $Term#2 the parser switches back to JAMOOS code. **INTEGER** is a JAMOOS primitive type, which is initialized by a C++ expression.

Since access to the Term components does not include a field selector, the return field of these is chosen. Similarly, in any definition which has a field of type Addition, its return field can be accessed in the same manner.

A **return** field may not have a perishability prefix. In most cases, this field is initialized, but it may also be a component. The JAMOOS compiler is smart enough to use the context to resolve the inherent ambiguity of accessing the whole object vs. accessing the default return field.

From the grammar perspective, an object is a node in a syntax tree, where the arguments to the constructor are supplied once the parser commits to a certain rule.[5] Components are the descendants of the object in the abstract syntax tree. Perishables are elements of the concrete syntax, which are discarded in the creation of an abstract syntax tree. Attributes are the semantic properties of a node, which are computed from the syntactical elements. Temporaries serve just as local variables of the reduce procedure, and from the grammar point of view play no role at all.

The following JAMOOS code excerpt demonstrates some of the variety of field definitions. (For the purpose of illustration this example defines a simple PASCAL variant in which a program consists of variable declarations and body, with no procedures, functions, type definitions, etc., and where each variable must be declared separately.)

Example 3.5.

```
PascalProgram → program _:Name _:vdecl:VarDecls
Body
    vars:VarList := ...
      – Here is some computation of VarList field
    statements:StatementList := Body.statements:
;

Body → begin statements:StatementList end;

VarDecls → var VarDeclList;
      – VarDecLists is defined elsewhere as a list of VarDecl

VarDecl → Type VarName – A single variable declaration
    is_simple:BOOLEAN:=[[return
                  Type.is_simple: → is_simple;]]
;
```

[5] Although JAMOOS is implemented using bottom-up engine, it can also work with top-down parsers.

Note that JAMOOS uses ADA [43] style comments. **BOOLEAN** is another JAMOOS primitive type corresponding to C++ **bool**.

In response to the above definition, the JAMOOS compiler will generate the following C++ equivalent class definition of PascalProgram.

Example 3.6.

```
class PascalProgram {
public:
    const Body& body;
    const VarList& vars;
    const StatementList& statements;

    PascalProgram (const Name& arg1,
        const VarDecls& vdecl, const
        Body& arg3)
    {
        body = arg3;

        ...   // The computation of vars
        vars = ...;

        statements = body.statements;
    }
};
```

Note that *all* fields are public. To emphasize immutability, all data members and all constructor arguments are defined as **const**.

4. The Ok and Error Types

Keywords such as **program**, **if** and **begin** are of type **OK** . Consider for example the following definition of the syntax of **if** statement in C++:

Example 4.1.

If → **if** "(" Condition ")" Statement **else** Statement ;

Fields if and else are keyword tokens of type **OK** . Fields "(" and ")" are *literal string tokens*, whose type is also **OK** . Although technically components of the respective containing class, keyword and string literal fields do not take part in the abstract syntax and can be ignored for almost all practical purposes. This is because values of type **OK** require no storage and do not answer any queries.

Another important use of **OK** is to type imperative code fragments, which are written in C++:

Example 4.2.

Assignment → Variable ":=" Expression
  print_exp: **OK** := [[`cout` << $Expression.print();]]
;

In the example class Assignment has an attribute print_exp of type **OK** . The C++ code segment

```
cout << $Expression.print();
```

is invoked during the execution of the constructor of Assignment, as part of the initialization of this field. Since field names (computational steps in the procedural perespective), the above could have been written as:

Example 4.3.

Assignment → Variable ":=" Expression
    [[`cout` << $Expression.print();]]
;

which would amount to a definition of an anonymous field of type **OK** in class Assignment. In general, we see that it is not necessary to denote the type of attributes whose type is **OK** .

Much in the same way that keywords and fixed strings can show as separators, terminators and other concrete syntax aids in the definition of the argument list, any expression of type **OK** , and in particular code fragments, may appear anywhere in the arguments list of a call to a method or a constructor. If this should happen, these code fragments are executed in order during the constructor call.

What happens if an error is detected during the execution of a code fragment, taking care, say, of a semantical action? Such errors should not lead to a halt of the parsing process, in the same way as a failure to open a file in procedural programming should not abort the program. JAMOOS error types are used for an orderly handling of such failures. Nevertheless, if a failure is not captured by a matching error type, a runtime error will halt the program.

The type **OK**? in JAMOOS is an error type used for imperative code which might terminate in an error. Fields of type **OK**? are similar to *named assertions* (e.g., **check** statements in EIFFEL [37]). Alternatively, a code whose type

is **OK**? is similar to a procedure which might return abnormally with an exception.

More generally, any type has an error type variant. Error types may either store valid values or be *erroneous* (be in a *state of error*). Syntactically, an error type is defined by a ? character to the right of the type definition. For example, the following class definition:

Example 4.4.

```
PascalProgram → program _:Name
    Declarations Body? ;
```

means (in the grammatical perspective) that a recoverable parsing error might occur in processing the symbol Body. It is usually impossible to create a well-structured tree of type Body if a parsing error has occurred. However, a crucial characteristic of error types is that values which are in a state of error can be composed with other values to make compound values. In the example, the PascalProgram might exist and be processed even though the program body is erroneous.

The following example shows how error-prone classes are defined:

Example 4.5.

```
Procedure? → procedure name:Id
    FormalArgumentsList ";" Declarations Body ;
```

Error types can be thought of as a choice between some non-error type and a *bottom* or *none* type, i.e., the type with *no* legal values. From the language design perspective we have thus unified the notions of parsing errors and exceptions. This unification is not complete, since JAMOOS objects (activation records) might exist on the stack even after execution of the corresponding routine have terminated. There are therefore, JAMOOS mechanisms to test not only if a value is in error, but also whether it has any error components [48].

Errors can be raised from JAMOOS code or from within embedded C++ code. Raising an error during execution will pop out objects from the execution stack until an object of an error type is encountered. This is possible since all code execution is always in the context of a nested constructor call.

5. The JAMOOS Type System

A class definition is a *named* type. Other JAMOOS types are constructed from named and primitive types. The six primitive types of JAMOOS are enumerated in Table 2, along with their C++ equivalents.

Grammatically, primitive types are nothing but tokens. There are predefined regular expressions to match the other five primitive types against the parsed input.

As a modern compiler-compiler JAMOOS supports regular expressions of terminals and symbols in the right-hand side of rules. Conversely, as a programming language, JAMOOS supports four kinds of type *generators* for *compound types*: *sequence*, *list*, *optional* and *choice*. These generators may be nested in a type definition. Thus, the right hand side of a JAMOOS definition can be any regular expression of terminals and symbols.

Type	C++ Equivalent	Default Regular Expression	Note
OK	**void**	none	Default for keyword / fixed string tokens
STRING	`std::string`[a]	⟨"[^"]*"⟩	Default for regular expression tokens
INTEGER	**int**	⟨(\+|\-|)[0-9]+⟩	
REAL	**double**	⟨(\+|\-|)[0-9]+\.[0-9]*⟩	
BOOLEAN	**bool**	⟨true|false⟩	
CHARACTER	**char**	⟨"."⟩	

[a] defined in the <string> module of C++ standard library

Table 2. Tokens and primitive types.

A *sequence* gives rise to a simple record structure. A *list* is a variable-sized collection of objects of the same type. In the following example, class VariablesDeclarations has a component-named vars which is a list of records, each storing a variable type and name.

Example 5.1.

```
VariablesDeclarations → var vars:
                    { (name:Id ":" type:Type ";") ... }+
```

A list is defined using curly brackets and ellipsis (...). The above definition is to say that VariablesDeclarations begins with the **var** keyword, followed by a list of pairs (denoted by parentheses) of Id and Type. The + symbol denotes that the list must have at least one element; this requirement is enforced at runtime, either during parsing or during a constructor call. A list definition might also include a start token, separators and an ending token:

Example 5.2.

```
FormalArgumentsList →
                    {"(" / FormalArgument ";" ... ")" }
```

In this example, the start token is the open parentheses character, the separator token is a semicolon ";", and the ending token is the close parentheses character. Note that a slash is used to separate the start token from the recurring list item. This syntax for specifying lists makes it clear that the start and the end token must be included in the input only in the case the list is not empty.

An *optional* type is a type whose value can be either an object of given type, or empty (denoted by a value of type **OK**). Square brackets are used to define optional types as in:

Example 5.3.

```
Conditional → if condition:Expression
                    then Statement [ else Statement ]
```

A *choice type* corresponds to alternations in regular expressions. The syntax borrows from ML `datatype` definitions:

Example 5.4.

```
For → for Variable ":=" Expression
      direction:(up OF to | down OF downto)
                        Expression do Statement
```

It is not possible to name the constituents of choice, optional, and list type generators. Any constituent can however be designated perishable. A type can only be named if it is a definition. JAMOOS uses structural equivalence, ignoring all perishable constituents, two different fields whose structure is the same are of the same unnamed type.

6. Internal Definitions and Methods

Fields can be thought of as data members. Methods in JAMOOS are just a special case of internal definitions. An *internal definition*, i.e., a definition made inside another definition, defines an *internal class* (a concept related to JAVA inner classes). The construction of an instance of an internal class is always carried out in the context of a containing object instantiated from the *containing class*.

Because the constructor of an internal class can access any field of the containing class, it can only be invoked through an instance of the containing class. The containing class is the only class which may have fields of the type of the internal class.

Some of these restrictions are similar to those placed on methods in traditional OO paradigm. A method can only be invoked in connection with an object of the class in which the method is defined. In fact, a constructor call of an internal class can be thought of as a method invocation. If the internal class has a default field, then that field can serve as the method return value, and the arguments are used as the methods arguments.

Here is an internal definition which can be used as method.

Example 6.1.

```
Procedure → procedure name:Id params:ParamList
Body
    ...
    PossibleCall → ExpressionList
        return:BOOLEAN := ...
        – Checking whether elements of
        – params is compatible to ExpressionList
    ;
;
```

Class Procedure represents procedures in PASCAL, and PossibleCall, when used as method, checks if a given list of expressions can be used to pass parameters to the procedure. A possible use of PossibleCall is:

Example 6.2.

```
ProcedureCall → name:Id params:ExpressionList
    legal:BOOLEAN := SEARCH(Procedures,
                    name.str).PossibleCall(params)
;
```

The **SEARCH** command here finds the procedure by its name in Procedures, which, as its name indicates, is a *dictionary* of procedures (see [48] for details).

In Example 6.1, the default field of the internal definition was used as the return value of the method. Here is an example of internal definition with two attributes.

Example 6.3.

```
VarDecl → VarName ":" Type
    ...
    Compatibility → Type
        can_assign:BOOLEAN := [[ ... ]]
        can_be_assigned:BOOLEAN := [[ ... ]]
        – code to check whether the given type is
        – assignable to the variable and vice versa
    ;
;
```

Given an object of type VarDecl, it is possible to call the constructor of the internal class Compatibility with a Type parameter. The returned object has two fields, each of which can be inspected. This is how the above definition can be used (assuming Variable has a field decl of type VarDecl referencing to the variable's declaration):

Example 6.4.

```
Assignment → Variable ":=" Expression
  is_legal:BOOLEAN := Variable.decl.Compatibility
                    (Expression.type).can_assign;
;
```

Internal classes can be used to type other fields in the containing definition, and hence can be thought of as a private type definition. Also, internal classes can be arbitrarily nested.

In general, an internal definition may access the components and attributes of the containing definition. Perishable and temporary fields of a definition cannot be accessed from an internal definition. The reason is that the constructor of an internal definition can, and usually will, be invoked after the containing object was entirely constructed.

The similarity of internal classes and methods is just a special case of duality between classes and routines. The main difference is that the constructor of internal classes can access fields of the containing object. Similarly, the difference between routines and methods is that a method is a routine which is always executed in the context of a receiving object.

7. Abstract Definitions and Inheritance

So far, all definitions were *concrete*, which means that, as classes, they could be instantiated and as routines, they could be invoked. In contrast, *abstract* definitions make *abstract classes* and outline a *pattern of execution* for concrete routines which inherit from an abstract definition. From the grammar perspective, an abstract definition is an alternation rule as in the following example.

Example 7.1.

```
Loop → ConditionalLoop | ControlVariableLoop
FEATURES
    body: Statement;
END;
```

An abstract definition does not enumerate arguments for its constructor. Instead, the abstract definition of Loop lists two definitions ConditionalLoop and ControlVariableLoop which inherit from it. The definitions for ConditionalLoop and ControlVariableLoop, which could be either abstract or concrete, must be provided elsewhere. JAMOOS uses a single inheritance model. Therefore, ControlVariableLoop and ConditionalLoop cannot show up in the definition header of any other abstract definition.

The only kind of fields allowed in an abstract definition are features: attributes and temporaries. All features of abstract definitions must be declared within a **FEATURES . . . END** block.

An abstract definition does not have a constructor and hence is not allowed to have arguments.

Field body of type Statement, which is the only field of Loop, is called an *abstract attribute* since it contains no initializer. Abstract attributes, which are similar to abstract methods in the OO paradigm, are only allowed in abstract classes.

A subclass, which may be concrete or abstract, *inherits* the attributes of its abstract superclass. An abstract class may provide an initializer to any of its features, and override any inherited initializer. A concrete subclass *must* provide an initializer for every abstract attribute for which an initializer was not provided by any of its superclasses. A concrete definition may not be inherited.

An abstract attribute can be turned into an argument by a concrete subclass, whereby delegating the responsibility of initialization to the caller of the constructor. To denote that an inherited field is turned into argument, it is preceded by an at-sign character (@). The following example continues Example 7.1.

Example 7.2.

```
Loop → ConditionalLoop | ControlVariableLoop
FEATURES
    body: Statement;
END;
ConditionalLoop → WhileLoop | RepeatLoop
FEATURES
    cond: Expression;
END;
ControlVariableLoop →
    for Variable ":=" from:Expression
                 to to:Expression do body;
WhileLoop → while cond do body;
RepeatLoop → repeat _:stmnts:{ Statement ";" ... }
                 until cond
    body := ... – Code converting the list of statements
               – into a single (block) statement
;
```

In this example, fields cond in ConditionalLoop and body in Loop are abstract features. While cond is defined by both subclasses as arguments, body in RepeatLoop is an attribute computed from the list of Statements stmnts. Note that stmnts itself is defined as perishable, because as soon as body is computed, there is no need to store stmnts any more.

As internal definitions can be used as methods, a mechanism is necessary to override them. Fortunately, the above mechanism for overriding fields works exactly in the same way for internal definitions. Any subclass may override any number of the field initializers of an internal definition. Since the internal definition fields' initializers can collectively be thought of as the method body, such overriding redefines only parts of the method body, and can be viewed as its overriding. Thus, internal definitions are more general than ordinary methods in that it is possible to selectively override sections of their body.

In Example 7.3 we illustrate overriding of internal methods.

Example 7.3.

```
ConditionalLoop → WhileLoop | RepeatLoop
FEATURES
    cond: Expression;
    Optimize →
        pre_computation: { MachineStat ...} := {}
        – empty pre_computation by default
        body: { MachineStat ... } := [[ ... ]]
        – C++ code optimizing the body
        – This code may add statements to pre_computation
        ...
    ;
END;

WhileLoop → while cond do body;

RepeatLoop → repeat _:stmnts:{ Statement ";" ... }
                 until cond
    body := ... – Code converting the list of statements
               – into a single (block) statement
    Optimize.pre_computation := [[ ... ]]
    – The actual invocation of the algorithm is done here
END;
```

In the example, Optimize is an internal definition of ConditionalLoop used in an optimizing compiler to generate optimized code. A RepeatLoop overrides only the initializer of **pre_computation** in this internal definition. Both subclasses of ConditionalLoop will override this internal definition, but only partially.

In an abstract definition header, each subclass name can be surrounded by any number of tokens, which are used as aid to parsing. The following abstract definition specifies the syntax of compound types in JAMOOS:

Example 7.4.

```
CompoundType →
        "(" Sequence ")" | Optional | List | Choice;
```

The parentheses around Sequence play only syntactic role.

Determining the order of initialization of inherited fields of a definition (and an internal definition) is tricky, since the inherited and the inheriting definitions may enumerate these fields in different order. JAMOOS tries to consolidate constraints of initialization due to the fact that an initializer of one field may use another. If these constraints cannot be consolidated, then an error is reported. The detailed algorithm is described in [48].

8. Other Language Features

Some unique elements of JAMOOS , lying outside the scope of this paper, are briefly described below. An interested reader is referred to [48] for a more detailed description of these.

1. *Genericity and Modular language specification.* JAMOOS extends the correspondence between type systems and grammar definition in two more ways. First, modular structuring of software is equated with modular grammar specification, which supports advanced concepts in parsing such as reusable grammars and language embedding. This feature made it possible to use JAMOOS to generate a compiler for itself, since the JAMOOS language can be thought of as an embedding of C++ code in pure JAMOOS code and vice versa to any depth.
   Secondly, *genericity*, as a typing mechanism, found its equivalent in the world of the specification of formal languages as a powerful means for defining generic grammars, i.e., grammars which expect parameters. Modular parameterized grammars make it possible, for example, to define the syntax of arithmetical expressions independently of the definition of literals in the language.
2. *Environmental acquisition.* Environmental acquisition is a newly proposed lingual mechanism [15] by which objects can "inherit" properties from their containers. Inner classes in JAVA [2] can be thought as a special and restricted case of environmental acquisition. In JAMOOS , environmental acquisition is used to allow objects to use *dictionaries*, which can be thought of as *symbol tables*, and more generally, "inherited attributes" in the parsing realm, or as LISP's A-lists in the programming world.
3. *Seamless integration with host language.* As in other OO programming languages, classes in JAMOOS are the main building blocks out of which software is made. Accordingly, JAMOOS is all about type declarations, with limited support for imperative commands and expressions. Indeed, expressions and commands are supported using embedded C++ code, which in its turn may contain embedded JAMOOS code. Escapes from JAMOOS to C++ and vice versa are smooth by clear-cut division of responsibilities. JAMOOS code provides all the type declarations and nothing other than that, while the evaluation of expressions as well as control structure are done in C++.
4. *Automatic field name deduction.* A unique feature of JAMOOS is a sophisticated mechanism for automatically assigning names to fields. With this mechanism, programmers are not obliged to name fields in class declarations as in the awkward, but typical following C++ code:

```
class Date {
public:
    Day aDay;
    Month aMonth;
    Year aYear;
};
```

9. Related Work

As a *language processor*, JAMOOS uses an extended BNF for productions and integrates lexical and grammatical definitions. In this sense, it is more powerful than the famous and very popular LEX [30] and YACC [22] tools pair, although learning how to use it may take an extra conceptual burden. JAMOOS is also different from many of the successors of LEX and YACC [17, 11, 19, 18, 50, 44], many of which use better than vanilla BNF, and stronger integration of tokenization and parsing, since it is

an essentially declarative language for the very specific task of language design, similar to the FNC [23] system with its full blown functional language OLGA.

JAMOOS has strong *self-descriptive properties* and a primary case of language design in JAMOOS was its bootstrapping. The reflective software architecture used in the construction of JAMOOS is reported in [31]. Arguably, the expressive power of a language can be measured at the ease of reflectively writing its own language processor. Much of the power of LISP is due to the simplicity of writing a LISP processor in LISP. PROLOG, and to a lesser extent SMALLTALK, exhibit similar characteristics.

The *small and cohesive computing metaphor* of JAMOOS might be compared with languages such SMALLTALK [16], LISP [45] and PROLOG [9]. However, unlike this trio, cohesiveness in JAMOOS does not compromise strong typing.

The *unification of class and procedure*, and *the tree computing metaphor* in JAMOOS are reminiscent of the BETA [33] programming language. What is called a "pattern" in BETA can be viewed both as a class and as a procedure definition. BETA, however, is a general purpose programming language, objects in it are mutable, and procedures may have conditional and iterative constructs. This limits the degree of similarity between objects and procedures since objects have no iteration and it is not clear what the semantics would be for an assignment to a component of a procedure. JAMOOS, on the other hand, is designed for grammar-based architectures [27], which makes it possible to take a more pure and unifying, even austere, approach, in restricting the computation to construction of immutable fields and allowing only sequential or recursive computation. With these restrictions, JAMOOS can offer a stronger unification of the class and procedure concepts.

Internal definitions in JAMOOS are quite similar to their BETA counterparts. Inner classes in JAVA are similar to those of JAMOOS in that the inner object can refer to fields of the containing object. JAMOOS internal definitions are different from nested classes in C++. In C++, the only implication of defining a class within another class is inserting the scope of its class name into the name space of the containing class. Other than that, nested class definitions in C++ have no effect on the behaviour of the contained class: it is *not* connected to any object, is visible from outside the containing class, and has no more access to the containing class' members than defined by the latter's protection labels.

The JAMOOS restriction to sequential computation lead to a *step-wise refinement semantics*, i.e., the ability of an inheriting definition to selectively replace any number of named steps of computation in the overridden definition. In contrast, most programming languages use replacement semantics, in which the overriding method completely subsumes the overridden method. It is usually possible for the overriding method to use a refinement semantics by calling their overridden counterpart, but not mandated. (In old versions of EIFFEL [37], it was even impossible to call an overridden method.) From a modeling perspective, refinement, which adds to the semantics, is almost always preferable to overriding. In response to this preference various lingual mechanisms have been proposed, including "before" and "after" daemons in languages such as FLAVORS and generic methods in languages such as CLOS [3]. Notable are SIMULA [39] and BETA for their **inner** construct, which specifies that no method can be overridden. JAMOOS offers a greater flexibility of selective overriding. The price, however, is in restricting the computation to sequence of named steps.[6]

Paakki's seminal survey [40] gives a broader perspective on the *unification of programming language paradigm and processing attribute grammars*. The concept unification of JAMOOS captures the following "equations" offered by Paakki:

1. **Production = Block**
   and since there is one-to-one correspondence between productions and non terminal boxes, also **Nonterminal = Block**.
   The equality holds since a production may have local entities (the internal definitions), whose visibility and accessability are controlled by normal scope rules.

[6] Note that the complete algorithm for consolidating the constraints on order of computation placed by the overriding and the overridden method offers some, but not unlimited flexibility of changing this order.

2. **Nonterminal = Procedure**.
   The equality here holds in that every nonterminal can be thought of as a procedure receiving arguments, and even returning a value.
3. **Noterminal = Class**
   and
   Production = Class.
   Just like the Mjølner/Orm [35] system, a nonterminal in JAMOOS in a grammar corresponds to a class. An important difference is that JAMOOS definition encompasses both the abstract and the concrete aspects of a grammar. In contrast, in Mjølner, the user must provide a mapping between these two grammars.

A related work *applying the programming language metaphor to the task of grammar specification* is that of Koster [29]. His CDL language builds upon the close relationship between LL grammars and procedural recursive programming. The underlying concept is that a production is unified with the (potentially recursive) procedure which conducts its parsing. With this unifying view one may prescribe, within a production, loops and even **goto**s for its corresponding procedure. Other extensions to the concepts of a BNF production include parameter passing and value return, macros and the addition of logical predicates. The JAMOOS approach puts limitations on the kinds of operations permitted in procedural computation, allowing only sequential computation. Loops and conditionals are captured by the type-system constructs: lists and choice types.

Comparing JAMOOS to the LISA system [36], we find that LISA, much in the spirit of Mjølner/Orm environment, is a full blown interactive environment for language definition, developed using OO techniques [52]—but not structuring grammar around an OO class hierarchy.

An interesting related work which *combines OO with language processing* is that of Jan Bosch [6, 7, 8] who uses an extension of the traditional object model, an OO architecture and design principles for a framework around which a compiler is constructed. This approach is similar to that of LISA which uses the equation

Attribute Grammar = Class

at the specification level. However, Bosch's slant on marriage of the OO paradigm and compiler construction does not concentrate on new principles of language design.

A syntax specification in JAMOOS includes both the *concrete and the abstract syntax*, where elements of the concrete syntax, which do not serve in the abstract syntax, are represented as *perishable* objects, i.e., objects which are discarded after their construction is completed. In contrast, Kadhim and Waite's MAPTOOL [24] allows the user to partially specify each syntax as long as their sum has complete information. JAMOOS is limited, however, in that the abstract syntax is obtained from the concrete one way of elimination. Thus, the class hierarchy of the abstract syntax is a subset of that of the concrete syntax. Computed fields in JAMOOS , which are really attributes of attribute grammars, allow adding more fields to the production of an abstract syntax. Still, it is not possible in JAMOOS to restructure the abstract grammar — a limitation which proved to be sometimes annoying in actual language definition.

Reflecting on the *choice of host language*, we note that JAMOOS is translated into C++ [46] (just as YACC translates to C [25]) — with a direct correspondence between the fields of a JAMOOS definition and C++ data types. Most of JAMOOS primitive types have their C++ counterparts. The more elaborate types make an extensive use of the STL library [38]. The Mjølner system translates grammars to BETA, However, a rule in Mjølner cannot be as general as a BETA pattern, since it is restricted to three different kinds as described above.

Our choice can also be compared with that of the more modern LISA, which chose JAVA as the host programming language for the portability of the graphical user interface. Such constraints were not a factor in the design of JAMOOS, which is a command line tool. Other than type-safety of containers as offered by the `template` mechanism, performance considerations were also a factor in our choice of C++.

Another aspect in which LISA and JAMOOS differ from each other is that LISA favors incremental and interactive language development. A LISA user can interactively add new productions and attribute computation rules. In

JAMOOS, the specified language cannot be extended without recompilation of the specification file.

It is also interesting to compare the handling of non-syntactical aspects of a language, i.e., attributes as in attribute-grammars in JAMOOS with the Doors [20] system of Mjølner. Doors are similar to attributes, but use reference semantics, instead of the usual copy-semantics of attributes. Thus a door system is more suitable for multi-pass compilation systems.

In contrast, JAMOOS is in principle a single-pass system. The reason is that computation follows that of a program call tree; the next conceptual step of multiple-passes along a program call tree is more daring, and is left to further research. The single-pass restriction is ameliorated by two special features: internal definitions, which allow to revisit a node, and by a built-in environmental acquisition [15], used for managing symbol tables. A node *a* can therefore refer to the objects represented by nodes in the abstract syntax subtree whose root is *a*, and even invoke methods (internal definitions) of these objects.

10. Conclusions and Further Research

Elegance and ease of use were prime concerns in the design of JAMOOS . From the compiler-compiler perspective, we have endeavored to abstract JAMOOS away from many details of parsing, lexical analysis, and intricacies of symbol table management. A software engineer using JAMOOS to create a language-processing tool should be able to concentrate on the language design, rather than on the architecture of the language processor.

This paper concentrated on the description of JAMOOS as a new programming language, offering an unusual merge in its tree computation metaphor of several concepts and paradigms: grammar specification, OO programming, and procedural programming. Ideally, the reader will find this perspective as esthetically pleasing as we did.

From the language definition perspective we leave a few problems open. Below we briefly describe some of them and give some possible directions for dealing with them.

Multi-dispatch and the Traversal Engines

Language processors often traverse the parse tree executing a recurring operation on each node. Consider for example the task computing the number of appearances of the `if` keyword in a program. In most languages, the programmer is required to define many "engine methods" throughout the entire class hierarchy, whose role is not any useful computation but rather the chore of propagating the traversal along the tree. Implementation of only one method in one class will actually do the counting. Such engine method must be added and entire class hierarchy should be changed any time a new such traversal is needed.

Although there are design pattern solutions (see the VISITOR pattern [13] and the ensuing literature), it might be interesting and useful to integrate traversals into the language. In our vision, the traversal engines would be augmented by a hierarchy of visitors and and a hierarchy of receivers. The actual operation at each visit is determined in a multi-dispatch fashion, depending on the visitor and on the receiver.

Producing Output and Compiling

Currently, JAMOOS is well suited for consuming input, but has no built-in mechanism for producing output. JAMOOS would be extended by adding support for output grammars. The traversal engine mentioned above might be useful in this task. An ambitious objective would be a tree grammars language for describing a family of trees into another such family, similar to RIGAL [4] and PCCTS/ANTLR [41]. Such a mechanism would also address a limitation of JAMOOS in which the abstract syntax can only be obtained from the concrete one, only by the elimination of some components, with the same basic class structure.

Statements and Expressions in JAMOOS

Currently, JAMOOS relies on C++ , its host language, for writing statements and expressions. As a result, JAMOOS must be translated into this host language. Native expressions and statements in JAMOOS will make it possible to translate it into other languages, such as JAVA.

Acknowledgments

We are grateful to Görel Hedin for inspiring comments on an earlier version of this paper. Meticulous reading and thoughtful comments of the anonymous reviewers and of the guest editors were indispensable!

References

[1] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] K. ARNOLD AND J. GOSLING, *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[3] G. ATTARDI, C. BONINI, M. R. BOSCOTRECASE, T. FLAGELLA, AND M. GASPARI, Metalevel programming in CLOS. In S. Cook, editor, *ECOOP '89*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

[4] M. AUGUSTON, RIGAL — programming language for compiler writing. In *Baltic Computer Science Selected Papers*, number 502 in Lecture Notes in Computer Science, pages 529–564. Springer Verlag, 1991.

[5] J. AVOTINS, C. MINGINS, AND H. SCHMIDT, Yes! an object-oriented compiler compiler YOOCC. In *Proceedings of TOOLS'94*, 1994.

[6] J. BOSCH, *Layered Object Model investigating paradigm extensibility*. PhD thesis, Department of Computer ScienceLund University, Oct. 1995.

[7] J. BOSCH, Parser delegation — an object-oriented approach to parsing. In *Proceedings of the 16th International Conference on Technology of Object-Oriented Languages and Systems* [47], pages 55–67.

[8] J. BOSCH, Delegating compiler objects — an object--oriented approach to crafting compilers. In Cointe [10].

[9] W. F. CLOCKSIN AND C. C. MELLISH, *Programming in Prolog*. Springer Verlag, Berlin, 3rd edition, 1987.

[10] P. COINTE, editor, *6th International Conference on Compiler Construction, CC'96*, number 1060 in Lecture Notes in Computer Science, Linköping, Sweden, Apr. 1996. Springer Verlag.

[11] M. EULENSTEIN, POCO compiler generator user manual. Technical Report A2/85, Universität des Saarlandes, 1985.

[12] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, Design patterns: Abstraction and reuse of object-oriented design. In O. M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 406–431, Kaiserslautern, Germany, July 26–30 1993. ECOOP'93, Springer Verlag.

[13] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[14] J. GIL AND D. H. LORENZ, SOOP — A synthesizer of an object-oriented parser. In *Proceedings of the 16th International Conference on Technology of Object-Oriented Languages and Systems* [47], pages 81–96.

[15] J. GIL AND D. H. LORENZ, Environmental Acquisition — A new inheritance-like abstraction mechanism. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct. 6–10 1996. OOPSLA'96, Acm SIGPLAN Notices 31(10) Oct. 1996.

[16] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[17] G. GOOS AND J. HARTMANIS, editors, *GAG: A Practical Compiler Generator*, number 141 in Lecture Notes in Computer Science. Springer Verlag, 1981.

[18] 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*, 35(2):121–131, Feb. 1992.

[19] J. GROSCH AND H. EMMELMANN, A tool box for compiler construction. In G. Goos and J. Hartmanis, editors, *Compiler Compilers 3rd International Workshop, CC'90*, number 477 in Lecture Notes in Computer Science, pages 106–116. Springer Verlag, 1990.

[20] G. HEDIN, Using door attribute grammars for incremental name analysis. In *Object-Oriented Environments, The MJØLNER Approach* [26], chapter 33, pages 497–510.

[21] E. JÄRNVALL, K. KOSKIMIES, AND M. NIITTYMÄKI, Object-oriented language engineering with TaLE. *Object-Oriented Systems*, 2:77–98, 1995.

[22] S. C. JOHNSON, Yacc — yet another compiler compiler. Technical Report Computing Systems Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[23] M. JOURDAN, D. PARIGOT, C. JULIE, O. DURIN, AND C. L. BELLEC, Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Proceeding of the Conference on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as ACM SIGPLAN Notices, 25(6).

[24] M. B. KADHIM AND W. M. WAITE, Maptool—supporting modular syntax development. In Cointe [10].

[25] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language*. Software Series. Prentice-Hall, 2nd edition, 1988.

[26] J. L. KNUDSEN, M. LÖFGREN, O. L. MADSEN, AND MAGNUSSON, *Object-Oriented Environments, The MJØLNER Approach*. The Object-Oriented Series. Prentice-Hall, 1993.

[27] J. L. KNUDSEN, E. SANDVAD, AND S. MINOÖR, Grammar-based architectures—introduction. In *Object-Oriented Environments, The MJØLNER Approach* [26], pages 259–273.

[28] K. KOSKIMIES, Object orientation in attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute grammars, Applications and Systems, Proceedings of the International Summer School SAGA*, volume 545 of *LNCS*, pages 297–329, Prague, Czechoslovakia, June 1991. Springer Verlag.

[29] C. A. H. KOSTER, Using the CDL compiler-compiler. In F. L. Bauer and J. Eickel, editors, *Compiler Construction—an advanced Course*, number 21 in Lecture Notes in Computer Science, pages 366–395. Springer-Verlag, 1974.

[30] M. E. LESK, Lex — a lexical analyzer generator. Technical Report Computing Systems Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[31] D. H. LORENZ, Tiling design patterns — a case study using the interpreter pattern. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Acm SIGPLAN Notices 32(10) Oct. 1997.

[32] O. L. MADSEN, Towards a unified programming language. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, Sophia Antipolis and Cannes, France, June 12–16 2000. ECOOP 2000, Springer Verlag.

[33] O. L. MADSEN, B. MØLLER-PEDERSEN, AND K. NYGAARD, *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[34] O. L. MADSEN AND C. NØRGAARD, An object-oriented metaprogramming system. In *Object-Oriented Environments, The MJØLNER Approach* [26], chapter 19, pages 283–296.

[35] B. MAGNUSSON, The mjølner orm system. In *Object-Oriented Environments, The MJØLNER Approach* [26], chapter 1, pages 11–23.

[36] M. MERNIK, M. LENIČ, E. AVDIČAUŠEVIĆ, AND V. ŽUMER, Compiler/interpreter generator system LISA. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Jan. 2000.

[37] B. MEYER, *EIFFEL: The Language*. Object-Oriented Series. Prentice-Hall, 1992.

[38] D. R. MUSSER AND A. SAINI, *STL Tutorial and Reference Guide. C++ Programming with the Standart Template Library*. Addison-Wesley, 1996.

[39] K. NYGAARD AND O.-J. DAHL, Simula 1967. In R. L. Wexelblat, editor, *History of Programming Languages*. ACM, 1981.

[40] J. PAAKKI, Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[41] T. PARR AND J. LILLY, *ANTLR 2.10 Reference Manual*, 1997. http://www.antlr.org/doc/index.html.

[42] L. C. PAULSON, *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[43] I. C. PYLE, *The* ADA *Programming Language*. Prentice-Hall, 2nd edition, 1985.

[44] M. SASSA, H. ISHIZUKA, AND I. NAKATA, Rie, a compiler generator based on a one pass-type attribute grammar. *Software — Practice and Experience*, 25(3):229–250, Mar. 1995.

[45] G. STEELE, *Common Lisp the language*. Digital, 1990.

[46] B. STROUSTRUP, *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[47] TOOLS 95 Europe Conference, *Proceedings of the 16th International Conference on Technology of Object-Oriented Languages and Systems*, Versailles, France, Mar. 6–10 1995. Prentice-Hall.

[48] Y. TSOGLIN, The JAMOOS programming language. Master's thesis, Technion—Israel Institute of Technology, Technion City, Haifa 32000, Israel, Dec. 2000.

[49] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[50] P.-C. WU AND F.-J. WANG, An object-oriented specification for compiler. *ACM SIGPLAN Notices*, 27(1):85–94, Jan. 1992.

[51] P.-C. Wu and F.-J. Wang. Applying classification and inheritance into compiling. *ACM OOPS Messengers*, Oct 1993.

[52] V. ŽUMER, N. KORBAR, AND M. MERNIK, Automatic implementation of programming languages using object-oriented approach. *Journal of System Architecture*, 43(1–5):203–210, 1997. ISSN 1318-7621.

Received: July, 2001
Revised: October, 2001
Accepted: November, 2001

Contact address:

Joseph (Yossi) Gil
Department of Computer Science
Technion Israel Institute of Technology
Haifa 32000, Israel
e-mail: yogi@cs.technion.ac.il

Yuri Tsoglin
IBM Research Laboratory
Science Industrial Park (MATAM)
Haifa 31905, Israel
e-mail: tyuri@cs.technion.ac.il

The academic titles of JOSEPH (YOSSI) GIL were conferred by the Hebrew University of Jerusalem: B.Sc. in Physics, Mathematics and Computer Science (1984, suma cum laude), M.Sc. in Computer Science (1986, suma cum laude), and Ph.D, (1990), under the supervision of Prof. Avi Wigderson. Dr. Gil teaches at the faculty of the department of computer science at the Technion—Israel Institute of Technology. He was a member of the program committee of conferences such as TOOLS, ECOOP and OOPSLA, and he chaired TOOLS USA'98. His research interests include object oriented programming languages and compiler construction.

YURI TSOGLIN is a member of the research staff at the IBM Haifa Research Lab where he is currently engaged in research on compilation. Yuri obtained his B.Sc. (1995) and M.Sc. (2001) degrees from the Department of Computer Science at the Technion. His dissertation was about the design, definition and implementation of the Jamoos programming language. His research interests include compilation theory, programming languages, object oriented programming.