

Incremental Programming Language Development

Marjan Mernik^{a,*} Viljem Žumer^a

^a *University of Maribor,
Faculty of Electrical Engineering and Computer Science,
Institute of Computer Science,
Smetanova 17, 2000 Maribor
Slovenia*

Phone: (+386 2) 220 7455

Fax: (+386 2) 251 1178

Abstract

One of the well known properties of software systems is that they are subject to changes. Incremental software development enables making such program changes in a non-destructive manner. In the area of programming language definition the language designer/implementer wants to include new language features incrementally as the programming language evolves. In the paper our approach to incremental programming language development is presented. The proposed approach has been successfully used in the development of real programming languages, which confirms feasibility in practice.

Key words: Attribute grammars; compiler-compilers; modular, reusable and extensible language definition.

1 Introduction

In the position statements of the working group on Programming Languages [1], held as a part of the ACM workshop on Strategic Directions in Computing Research at MIT, Boston, in June 1996, the participants identified five strategic directions. Among them were *Incrementality, Modularity and Abstraction*.

* Corresponding author.

Email addresses: `marjan.mernik@uni-mb.si` (Marjan Mernik),
`zumer@uni-mb.si` (Viljem Žumer).

It was stated that *“A challenge for future language design is to support modularity and abstraction in a manner that allows incremental changes to be made as easily as possible. Object-oriented concepts have much to offer and are the topic of much on-going investigation.”* Although that direction was oriented toward general software development we are interested in a more specialized topic, namely in the specification of programming languages. Let us stress this difference a little bit more. While building an application, a programmer should use a programming language that supports incremental change. Such incremental development has many benefits [2] and current object-oriented programming languages already sufficiently support it. In a similar way, when specifying a programming language, general-purpose or domain-specific [3], a language designer should be able to use specifications of incremental changes to existing languages.

But have we reached this ambitious goal in the area of programming language specifications? The syntax and semantics of a programming language should be specified in a formal way using one of the formal methods for programming language description, such as operational semantics, attribute grammars, denotational semantics, action semantics, algebraic semantics, abstract state machines, etc. The advantages of formal specification of programming languages are well known: the meaning of a program is precisely and unambiguously defined, and it offers a unique possibility for automatic generation of compilers or interpreters. The programming languages that have been designed with one of the various formal methods have a better syntax and semantics, less exceptions and are easier to learn. Moreover, researchers recognized the possibility that many other language-based tools could be generated from formal language specifications [4] [5]. Therefore, many language implementation systems not only automatically generate a compiler/interpreter but also complete language-based environments including editors, type checkers, debuggers, various analyzers, animators, etc. Despite their usefulness formal language specifications are not popular [6] since they are hard to understand, modify and maintain. Yet worse, small modifications of some parts in the specifications have widespread effects on the other parts of the specifications. Therefore specifications are not modular, extensible and reusable. Compared to modern programming languages, such as object-oriented or functional languages, language specification languages are far less advanced, specifically concerning provisions for abstraction, modularization, extensibility and reusability. A good survey of existing attribute grammar-based specification languages, at the time it was written, is [7].

Ideally, a language designer would like to build a language simply by reusing different language definition modules, such as modules for expressions, declarations, etc., regardless of different formal methods (as common in component-based programming where components can be simply plug-in), and afterwards straightforwardly extends them to reflect language design changes. This can-

not be done now, even if we restrict ourselves to just one of the formal methods since different compiler-compilers used different and incompatible specification languages (e.g. despite that Eli [8] and FNC-2 [9] both rely on attribute grammars one can not exchange language definition modules written in the other system). Moreover, the same is usually true even in the case of the same specification language since syntax constructs (non-terminals and terminals) and semantic constructs (e.g. attributes and semantic rules in the case of attribute grammars) are not constituents of the hidden part of the module, nor are the parameters of language definition modules (an attempt was Composable Attribute Grammars [10]). Up to now, many language definitions are modularized so the closely related language constructs are collected into a separate module (e.g., name analysis, type checking, code generation, etc.), but these modules cannot be easily reused or extended in other language specifications. Therefore, the language designer has difficulties in integrating new concepts into the language in an easy way. This is especially true in developing domain-specific languages which change more frequently [11] than the general-purpose programming languages do. Language designers need a formal method which enables them to make incremental changes of programming language specifications. If incremental language development is not supported, then the language designer has to define languages from scratch or by scavenging old specifications.

In this paper our approach to incremental language development is presented. Inheritance is commonly regarded as central feature of object-oriented programming. By analogy to object-oriented programming languages inheritance has been incorporated into attribute grammars.

The organization of the paper is as follows. The quest for extensible programming language definition and our contribution to this quest (multiple attribute grammar inheritance) are presented in section 2. Our approach of incremental language development is shown on a small but interesting example in section 3. After that, examples of design and implementation of real programming languages are briefly described in section 4. In section 5 related work is described. Concluding remarks are given in section 6.

2 Extensible Programming Language Definition

The challenge in programming language definition is to support modularity and abstraction in a manner that allows incremental changes to be made as easily as possible. This was our goal and to achieve it we soon recognized that inheritance can be very helpful since it is a language mechanism that allows new definitions to be based on the existing ones. A new specification can inherit the properties of its ancestors, and may introduce new properties that

extend, modify or defeat its inherited properties. In object-oriented languages the properties that consist of instance variables and methods are subject to modification. But what are the corresponding properties in language definition? First, we restrict ourselves to just one formal method. We chose attribute grammars because we were most experienced in them and also because high quality language-based tools (compilers, editors, etc.) can be generated automatically from attribute grammar specifications [5]. An attribute grammar consists of:

- A context-free grammar $G = (T, N, S, P)$, where T and N are the set of terminal symbols and nonterminal symbols; $S \in N$ is the start symbol, which does not appear on the right side of any production rule; and P is the set of productions. Now set $V = T \cup N$.
- A set of attributes $A(X)$ for each nonterminal symbol $X \in N$. $A(X)$ is divided into two mutually disjoint subsets, $I(X)$ of inherited attributes and $S(X)$ of synthesized attributes. Now set $A = \bigcup A(X)$. Let $Type$ denote a set of semantic domains. For each $a \in A(X)$, $a : type \in Type$ is defined which is the set of possible values of a .
- A set of semantic rules R . Semantic rules are defined within the scope of a single production. A production $p \in P, p : X_0 \rightarrow X_1 \dots X_n$ ($n \geq 0$) has an attribute occurrence $X_i.a$ if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of semantic rules R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a, 1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, \dots, y_k), k \geq 0$, where $y_j, 1 \leq j \leq k$, is an attribute occurrence in p and f is a semantic function. In the rule $X_i.a = f(y_1, \dots, y_k)$, the occurrence $X_i.a$ depends on each attribute occurrence $y_j, 1 \leq j \leq k$. Now set $R = \bigcup R_p$. For each production $p \in P, p : X_0 \rightarrow X_1 \dots X_n$ ($n \geq 0$) the set of defining occurrences of attributes is $DefAttr(p) = \{X_i.a | X_i.a = f(\dots) \in R_p\}$. An attribute $X.a$ is called synthesized ($X.a \in S(X)$) if there exists a production $p : X \rightarrow X_1 \dots X_n$ and $X.a \in DefAttr(p)$. It is called inherited ($X.a \in I(X)$) if there exists a production $q : Y \rightarrow X_1 \dots X \dots X_n$ and $X.a \in DefAttr(q)$.

Therefore, an attribute grammar is a triple $AG = (G, A, R)$ which consists of a context free grammar G , a finite set of attributes A and a finite set of semantic rules R . Since semantic rules in attribute grammars are tightly coupled with particular production rules, properties in attribute grammars consist of:

- lexical regular definitions,
- attribute definitions,
- rules which are generalized syntax rules that encapsulate semantic rules, and
- operations on semantic domains.

Taking into account above thoughts our attribute grammar-based specification language has following parts:

```

language  $L_1$  [extends  $L_2, \dots, L_N$ ] {
  lexicon {
    [[P] overrides | [P] extends] R regular expr.
    :
  }
  attributes type  $A_1, \dots, A_M$ 
  :
  rule [[Y] extends | [Y] overrides] Z {
    X ::=  $X_{11} X_{12} \dots X_{1p}$  compute {
      semantic functions }
    :
    |
       $X_{r1} X_{r2} \dots X_{rt}$  compute {
        semantic functions }
    ;
  }
  :
  method [[N] overrides | [N] extends] M {
    operations on semantic domains
  }
  :
}

```

In this manner we can extend the lexical, syntax and semantic parts of the programming language specification. Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications.

In our approach the attribute grammar as a whole is subject to inheritance. We call this multiple attribute grammar inheritance. The formal definition of multiple attribute grammar inheritance is described in [12], while the informal presentation is based on examples presented in section 3. With our approach, the language designer/implementer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications. Multiple attribute grammar inheritance was successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0 [13]. The tool LISA is a compiler generator with the following features:

- LISA is platform independent since it is written in Java.

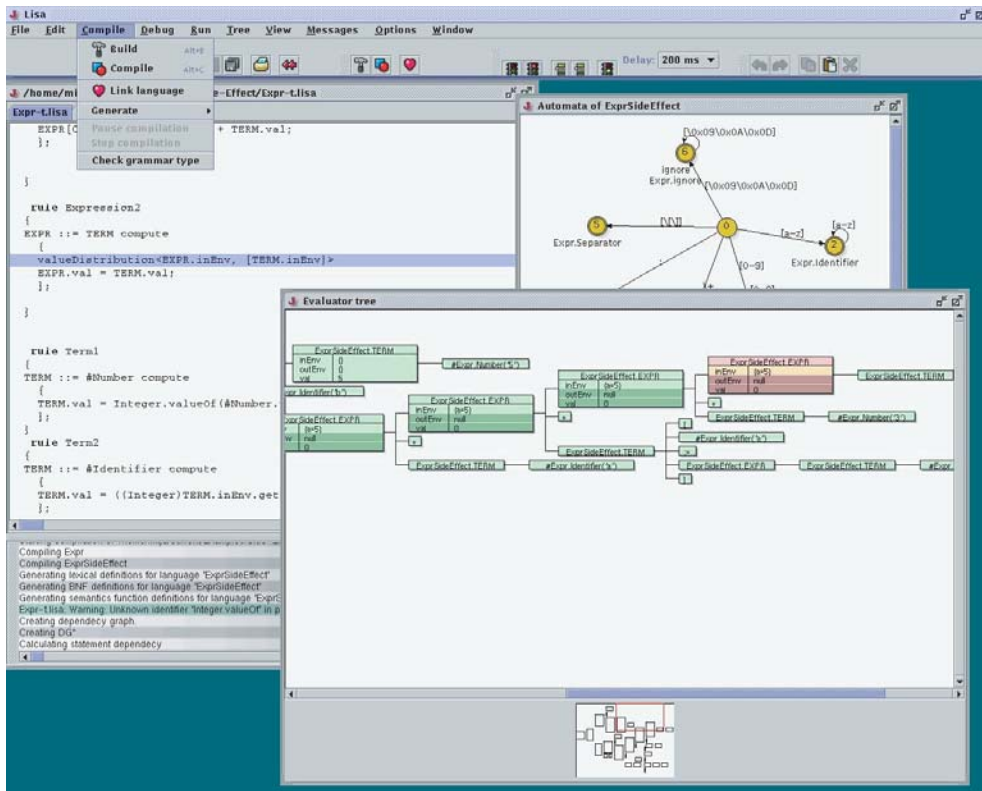


Fig. 1. LISA Integrated Development Environment

- It offers the possibility to work in a textual or visual environment.
- It offers an integrated development environment (Fig. 1) where users can specify - generate - compile-on-the-fly - execute programs in a newly specified language.
- Lexical, syntax and semantic analysers can be of different types and can operate standalone; the current version of LISA supports LL, SLR, LALR, and LR parsers, tree-walk, parallel, L-attribute and Katayama evaluators.
- Visual presentation of different structures, such as finite state automata, BNF, syntax tree, semantic tree, dependency graph.
- Animation of lexical, syntax and semantic analysers is provided.
- The specification language supports multiple attribute grammar inheritance.

3 An Example

A very simple language for moving a robot can illustrate our incremental language development approach. A robot can move in four directions from the initial position (0,0). After moving, it is stopped in an unknown location, which the user wants to compute (Fig. 2). An example of the program written in this language is `begin up right up right right down end` with the

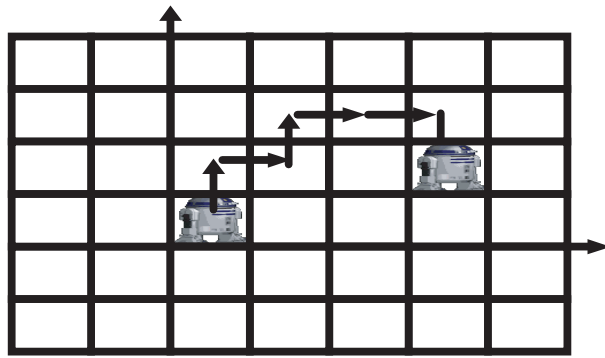


Fig. 2. Simple robot movement

meaning ($\text{outp}=(3,1)$). A robot, after executing the above program, stopped in the position (3,1). The language for robot movement is defined in Fig 3. The language specification is simple and does not need further explanations (the type of attributes is Point which is not presented in specifications).

Sooner or later the language is extended with new features. For example, we would like to know when the robot will reach the final position. The new language (RobotTime) is specified (Fig. 4) as an extension to the Robot language. This is a good example of how different aspects can be modularized in our approach. In the Robot language just the semantic rules for robot movement have been described, while the RobotTime language contains just the semantic rules for time calculation. The syntax of both languages is the same, while the semantics is different. The meaning of the above program is now ($\text{outp}=(3,1)$, $\text{time}=6.0$). Therefore, the robot stopped in the final position after 6 time steps. The RobotTime language inherits regular definitions, syntax constructs and semantic rules from Robot language and adds new semantic rules for time calculation.

Another example of adding a new language feature is the possibility that the robot can move at a different speed. In that case new syntactic constructs have to be added to the language. The new language (RobotSpeed) is specified (Fig. 5) as an extension to the RobotTime language. Notice how regular definitions for `Commands` have been extended. Semantic rules for speed calculation have been specified incrementally. However, the new construct (`speed`) causes a particular movement to be performed faster. Hence, the semantic rule for the attribute `time` has to be redefined. In particular, the semantic rule `MOVE.time = 1.0/MOVE.inspeed` redefines the semantic rule `MOVE.time = 1`. This example shows that in multiple attribute grammar inheritance it is not enough just to simply glue together semantic rules for a particular syntax construct (for formal definition see [12]). All rules extend the inherited ones, except the rule `speed` which defines the syntax structure of the new construct `MOVE ::= speed #Number`. Therefore this rule has to define also the attributes `outp` and `time` which represent the outcome position and how much time the robot spent on executing this command.

```

language Robot {
  lexicon {
    Commands left | right | up | down
    ReservedWord begin | end
    ignore [\0x0D\0x0A\ ] // skip whitespaces
  }
  attributes Point *.inp, *.outp;
  rule start {
    START ::= begin MOVES end compute {
      START.outp = MOVES.outp;
      // robot position in the beginning
      MOVES.inp = new Point(0, 0); };
  }
  rule moves {
    MOVES ::= MOVE MOVES compute {
      MOVES[0].outp = MOVES[1].outp; // propagation of position
      MOVE.inp = MOVES[0].inp;      // to sub-commands
      MOVES[1].inp = MOVE.outp; }
    | epsilon compute {
      MOVES.outp = MOVES.inp; };
  }
  rule move {
    // each command changes one coordinate
    MOVE ::= left compute {
      MOVE.outp = new Point((MOVE.inp).x-1,(MOVE.inp).y); };
    MOVE ::= right compute {
      MOVE.outp = new Point((MOVE.inp).x+1,(MOVE.inp).y); };
    MOVE ::= up compute {
      MOVE.outp = new Point((MOVE.inp).x,(MOVE.inp).y+1); };
    MOVE ::= down compute {
      MOVE.outp = new Point((MOVE.inp).x,(MOVE.inp).y-1); };
  }
}

```

Fig. 3. Robot Language

The meaning of the following program `begin up speed 2 right up right right speed 1 down end` is (outp=(3,1), outspeed=1, time=4.0). As the result, the robot stopped in the final position sooner, after 4 time steps.

So far the language extensions were small. Let's try some radically different robot behavior. Consider a robot whose job is to clean an empty room. Now the robot movement is limited because of the presence of walls. The goal is to determine the space cleaned (Fig. 6). The new language (CleaningRobot) is specified (Fig. 7) as an extension to the RobotSpeed language. Notice how the rule `start` defines the new starting production (first production in the language specification is a starting production [12]). It contains a new production `CLEANTASK ::= DIM START` and an old production `START`

```

language RobotTime extends Robot {
  attributes double *.time;
  rule extends start {
    compute {
      // initial position is inherited
      START.time = MOVES.time; }
  }
  rule extends moves {
    MOVES ::= MOVE MOVES compute {
      // total time is sum of times spent in sub-commands
      MOVES[0].time = MOVE.time + MOVES[1].time; }
    | epsilon compute {
      MOVES.time = 0; };
  }
  rule extends move { // each command spent 1 time step
    MOVE ::= left compute {
      MOVE.time = 1; };
    MOVE ::= right compute {
      MOVE.time = 1; };
    MOVE ::= up compute {
      MOVE.time = 1; };
    MOVE ::= down compute {
      MOVE.time = 1; };
  }
}

```

Fig. 4. RobotTime Language

`::= begin MOVES end`. For the latter, only new semantic rules are added, all others (e.g. initial position) are inherited from the RobotSpeed language. The new meaning of programs requires new attributes. The attribute `maxpos` denotes room's length and width. The attributes `intable` and `outtable` are of type `Room` and represent a semantic domain. Semantic domains and operation on these domains are written in LISA in the method section of specifications (omitted from figure Fig. 7). Because of the presence of walls the position of the robot after each kind of movement has to be redefined. In the generalized rule `move` the semantic rules for attribute `outp` are redefined. Notice that in the generalized rule `speed` the semantic rule for attribute `outp` is inherited from the RobotSpeed language.

The meaning of the following program written in the CleaningRobot language `4 * 4 begin right right right right up up up left end` (see Fig. 6) is:

```

outtable: | 1 1 1 1 | 0 0 0 1 | 0 0 0 1 | 0 0 1 1 |
outp: (2, 3)
maxpos:(4, 4)

```

```

language RobotSpeed extends RobotTime {
  lexicon {
    extends Commands speed
    Number [0-9]+
  }
  attributes int *.inspeed, *.outspeed;
  rule extends start {
    compute {
      // initial position is inherited
      MOVES.inspeed = 1; // beginning speed
      START.outspeed = MOVES.outspeed; }
  }
  rule extends moves {
    MOVES ::= MOVE MOVES compute {
      MOVE.inspeed = MOVES[0].inspeed; // speed propagation
      MOVES[1].inspeed = MOVE.outspeed; // to sub-commands
      MOVES[0].outspeed = MOVES[1].outspeed; }
    | epsilon compute {
      MOVES.outspeed = MOVES.inspeed; };
  }
  rule extends move {
    // these commands do not change speed
    MOVE ::= left compute {
      MOVE.time = 1.0/MOVE.inspeed;
      MOVE.outspeed = MOVE.inspeed; };
    MOVE ::= right compute {
      MOVE.time = 1.0/MOVE.inspeed;
      MOVE.outspeed = MOVE.inspeed; };
    // MOVE ::= up, MOVE ::= down are omitted
  }
  rule speed {
    MOVE ::= speed #Number compute {
      MOVE.time = 0; // no time is spent for this command
      MOVE.outspeed = Integer.valueOf(#Number.value()).intValue();
      // this command does not change the position
      MOVE.outp = MOVE.inp; };
  }
}

```

Fig. 5. RobotSpeed Language

The table is read as:

```

0 0 1 1
0 0 0 1
0 0 0 1
1 1 1 1

```

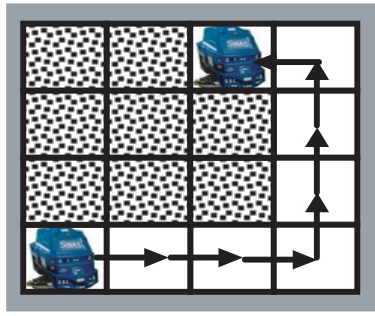


Fig. 6. Cleaning Robot

where 1 means a cleaned and 0 a dirty cell. Position (0,0) is in bottom-left corner. Last `right` command has no effect since the robot can not move outside the wall.

4 Successful implementation of real programming languages

The aim of this section is not to describe implementation of various programming languages in detail, but to show that proposed approach has been proven successful in various application domains.

An application domain for which our approach is very suitable is the development of domain-specific languages. In [14] the design and implementation of Simple Object Description Language SODL for automatic interface creation are presented. The application domain was network applications. Since the cross network method calls slow down performance of our applications, the solution was Tier to Tier Object Transport. However, with this approach the network application development time has been increased. To enhance our productivity a new domain-specific language SODL has been designed. The design and implementation of the SODL language took less than a week using the LISA tool, which proves that our methodology is successful.

In [15] the design and implementation of COOL and AspectCOOL languages are described. Here the application domain was aspect-oriented programming (AOP). AOP is a programming technique for modularizing concerns that crosscut the basic functionality of programs. In AOP, aspect languages are used to describe properties, which crosscut basic functionality in a clean and a modular way. AspectCOOL is an extension of the class-based object-oriented language COOL (Classroom Object-Oriented Language), which has been designed and implemented simultaneously with AspectCOOL. Both languages were formally specified with multiple attribute grammar inheritance. We were able to reuse much of the COOL specifications while developing AspectCOOL language (Fig. 8).

```

language CleaningRobot extends RobotSpeed {
  lexicon { operator \* }
  attributes
    Room *.intable, *.outtable;
    Point *.maxpos; int *.x, *.y;
  rule extends start {
    CLEANTASK ::= DIM START compute {
      START.intable = new Room(DIM.x, DIM.y);
      START.maxpos = new Point(DIM.x, DIM.y);
      CLEANTASK.outtable = START.outtable;
      CLEANTASK.outp = START.outp; };
    START ::= begin MOVES end compute {
      // initial position is inherited
      START.outtable = MOVES.outtable;
      MOVES.maxpos = START.maxpos;
      MOVES.intable = START.intable; };
  }
  rule dimensions {
    DIM ::= #Number * #Number compute {
      DIM.x = Integer.valueOf(#Number[0].value()).intValue();
      DIM.y = Integer.valueOf(#Number[1].value()).intValue(); };
  }
  rule extends moves {
    MOVES ::= MOVE MOVES compute {
      MOVES[1].intable = MOVE.outtable; // cleaning table
      MOVE.intable = MOVES[0].intable; // propagation
      MOVES[0].outtable = MOVES[1].outtable;
      MOVES[1].maxpos = MOVES[0].maxpos; // max positions
      MOVE.maxpos = MOVES[0].maxpos; } // distribution
    | epsilon compute {
      MOVES.outtable = MOVES.intable; };
  }
  rule extends move {
    MOVE ::= left compute {
      MOVE.outtable = update(MOVE.intable, MOVE.inp,
                             MOVE.maxpos, LEFT);
      MOVE.outp=calculatePosition(MOVE.inp, MOVE.maxpos, LEFT); };
    // MOVE ::= right, MOVE ::= up, MOVE ::= down are omitted
  }
  rule extends speed {
    compute { // this command does not clean the position
      MOVE.outtable = MOVE.intable; }
  }
} // language CleaningRobot

```

Fig. 7. CleaningRobot Language

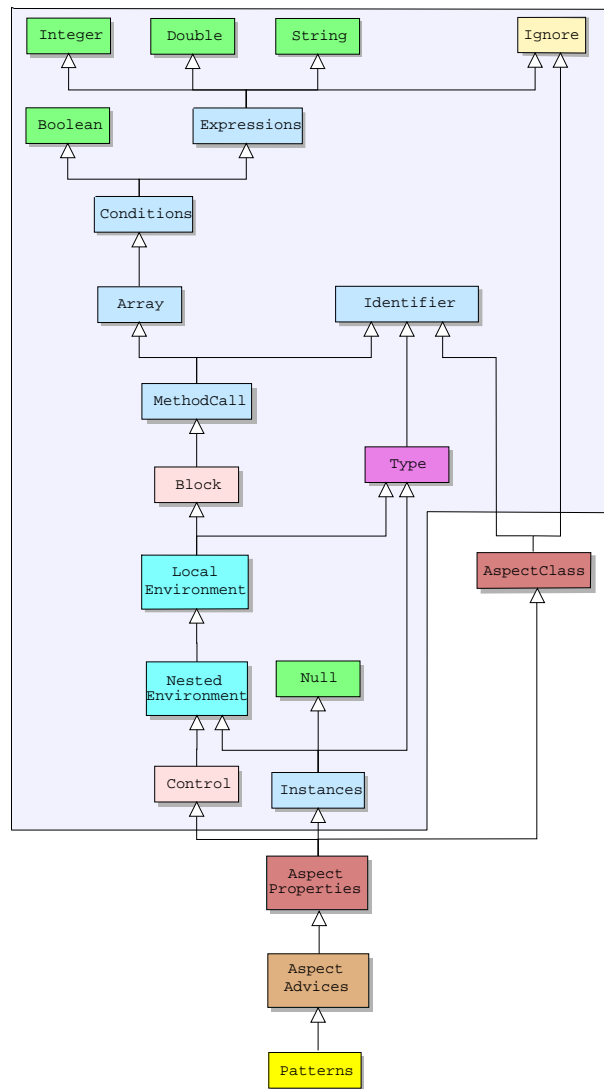


Fig. 8. COOL specifications in the gray frame were reused while developing Aspect-COOL language

We have incrementally developed various small but real programming languages, such as PLM [16]. PLM language was incrementally developed in a following manner [17]. First, the language Constants was specified. This language can be reused whenever constant definitions are part of the language. Next, the language PLM1 was specified where declarations and assignment statements can be written only in the main program. The next increment was the language PLM2 where if and while statements were added (introducing labels into the object code). Also constant definitions were added to the language PLM2 by multiple inheritance. Finally, procedures and call statements were added to the language PLM.

With the described approach, we were able to reuse parts of specifications of one programming language to define another. We have designed a simple

functional language FUN which has been further extended to the imperative language IMP. From this example some indicators on reuse were obtained. Using monolithic specifications the language FUN was written in 99 lines and the language IMP in 142 lines. The latter was rewritten in 78 lines using multiple attribute grammar inheritance. Therefore, about 45% of specifications were reused.

5 Related work

Many tools have been built in the past years, based on the different formal methods and implementing different parts of language, such as scanner generators, parser generators and compiler generators. Several tools for language implementation have been built based on attribute grammars, (e.g. GAG [18], FNC-2 [9], Eli [8]), denotational semantics (e.g. MESS [19]), algebraic specifications (e.g. ASF+SDF [20]), abstract state machines (e.g. Gem-Mex [21]), or action semantics (e.g. Actress [22]). However, none of the currently available tools support incremental language development to the extent that our approach does. In the technical report [23] the author proposed the following grammatical operators: rename, export, abstract, remove, extend, and integrate. These grammatical operators transform one context-free grammar into another. In our approach operator extend already works on attribute grammars, while extension of other grammatical operators to attribute grammars is under investigation.

In this section our approach is compared in more detail only to two very recent attempts to achieve better modularization and reusability of language definition. Among different formal methods, action semantics poses very good inherent modularity better than, for example, denotational semantics, where each module typically defines a semantic function on an entire syntactic sort. One of the recent achievements regarding better reusability and modularity of action semantics is reported in [24]. The authors propose a finer modular structure where a new semantic equation module is constructed for each production. The final language definition module is obtained simply by importing them together, assuming that the symbols they share correspond to common features. It is our belief that a fine modular structure is not feasible for real programming languages, just as monolithic structure is infeasible, since optimal granularity is somewhere between two extreme options. However, it might be feasible for domain-specific languages. Only experience gained in the development of real general-purpose and domain-specific languages will show the feasibility of their approach. No such experience report exists yet.

Some language implementation systems use object-oriented specifications [7] [25] where context-free grammars define the class hierarchy. Nonterminals act

as abstract superclasses and productions act as specialized concrete subclasses that specify the syntactic structure, attributes and semantic rules. All these elements can be inherited, specialized, and overridden in subclasses. One of the shortcomings of this approach is that right-hand nonterminals can not have inherited attributes (in our approach no such limitation exists) and the other is that only small features can be added to the language. In other words, language can not evolve dramatically as in our case. Another problem is that the class hierarchy defines the modularization based on language syntax constructs, whereas the language developer also wants to have modules based on different aspects (e.g. name analysis, type checking, code generation, etc). To overcome this problem authors in [26] proposed special modules for describing aspects which are then woven into context-free grammar classes using aspect-oriented techniques.

6 Conclusion

The challenge in programming language definition is to support modularity and abstraction in a manner that allows incremental changes to be performed as easily as possible. We have achieved this goal only partially, but to a much greater extent than any other approach, with multiple attribute grammar inheritance where the language designer/implementer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications. Multiple attribute grammar inheritance was successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0 and was used in the design and implementation of real general-purpose programming languages (e.g. PLM, COOL) and domain-specific languages (e.g. SODL). Our experience with these non-trivial examples shows that multiple attribute grammars inheritance is useful in managing the complexity, reusability and extensibility of language definitions. Specifications become much easier to read, maintain and to modify.

7 Acknowledgements

We would like to thank Barrett Bryant and Philipp Kutter for useful comments on earlier versions.

References

- [1] C. Hankin, H. R. Nielson, J. Palsberg, Position statements on strategic directions for research on programming languages, *ACM SIGPLAN Notices* 32 (1) (1997) 59–65.
- [2] A. Taivalsaari, On the notion of inheritance, *ACM Computing Surveys* 28 (3) (1996) 438–479.
- [3] M. Mernik, J. Heering, A. Sloane, When and how to develop domain-specific languages, Tech. rep., University of Maribor, CWI Amsterdam, and Macquarie University (2003).
- [4] J. Heering, P. Klint, Semantics of Programming Languages: A Tool-Oriented Approach, *ACM Sigplan Notices* 35 (3) (2000) 39–48.
- [5] P. Henriques, M. V. Pereira, M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, Automatic generation of language-based tools, in: M. van den Brand, R. Laemmel (Eds.), *Electronic Notes in Theoretical Computer Science*, Vol. 65, Elsevier Science Publishers, 2002.
- [6] D. A. Schmidt, On the need for a popular formal semantics, *ACM SIGPLAN Notices* 32 (1) (1997) 115–116.
- [7] J. Paakki, Attribute grammar paradigms - a high-level methodology in language implementation, *ACM Computing Surveys* 27 (2) (1995) 196–255.
- [8] R. Gray, V. Heuring, S. Levi, A. Sloane, W. Waite, Eli: A complete, flexible compiler construction system, *Communications of the ACM* 35 (2) (1992) 121–131.
- [9] M. Jourdan, D. Parigot, C. Julie, O. Durin, C. L. Bellec, Design, implementation and evaluation of fnc-2 attribute grammar system, in: *Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation*, 1990, pp. 209–222.
- [10] R. Farrow, T. J. Marlowe, D. M. Yellin, Composable attribute grammars: Support for modularity in translator design and implementation, in: *19th Annual ACM Sigplan - Sigact Symposium on Principles of Programming Languages*, 1992, pp. 223–234.
- [11] D. Wile, Supporting the DSL Spectrum, *Journal of Computing and Information Technology*, Special Issue on Domain-Specific Languages, R. Laemmel and M. Mernik, eds. 9 (4) (2001) 263–287.
- [12] M. Mernik, M. Lenič, Enis Avdičaušević, V. Žumer, Multiple Attribute Grammar Inheritance, *Informatica* 24 (3) (2000) 319–328.
- [13] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, LISA: An Interactive Environment for Programming Language Development, in: N. Horspool (Ed.), *11th International Conference on Compiler Construction*, Vol. 2304, Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 1–4.

- [14] M. Mernik, U. Novak, E. Avdičaušević, M. Lenič, V. Žumer, Design and implementation of simple object description language, in: Proceedings of 16th ACM Symposium on applied computing, 2001, pp. 203–210.
- [15] E. Avdičaušević, M. Lenič, M. Mernik, V. Žumer, AspectCOOL: An experiment in design and implementation of aspect-oriented language, ACM SIGPLAN Notices 36 (12) (2001) 84–94.
- [16] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, 1976.
- [17] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer, A reusable object-oriented approach to formal specifications of programming languages, L'Objet 4 (3) (1998) 273–306.
- [18] U. Kastens, The gag-system – a tool for compiler construction, in: B. Lorho (Ed.), Methods and Tools for Compiler Construction, Cambridge University Press, 1984, pp. 165–181.
- [19] P. Lee, Realistic Compiler Generation, MIT Press, 1989.
- [20] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Oliver, J. Scheerder, J. Vinju, E. Visser, J. Visser, The ASF+SDF Meta-environment: A component-based language development environment, in: R. Wilhelm (Ed.), 10th International Conference on Compiler Construction, Vol. 2027, Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 365–370.
- [21] P. W. Kutter, Montages - engineering of computer languages, Ph.D. thesis, ETH Swiss Federal Institute of Technology Zurich (to appear).
- [22] D. Brown, H. Moura, D. Watt, Actress: Action semantics directed compiler generator, in: U. Kastens, P. Pfahler (Eds.), Compiler Construction (CC'92), Vol. 641, Lecture Notes in Computer Science, Springer-Verlag, 1992, pp. 95–109.
- [23] D. Wile, Integrating syntaxes and their associated semantics, Tech. rep., USC/Information Science Institute, Marina del Rey (1999).
- [24] K.-G. Doh, P. D. Mosses, Composing programming languages by combining action-semantics modules, in: M. van den Brand, D. Parigot (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 44, Elsevier Science Publishers, 2001.
- [25] G. Hedin, Reference attributed grammars, Informatica 24 (3) (2000) 301–318.
- [26] G. Hedin, E. Magnusson, Jastadd—a java-based system for implementing front ends, in: M. van den Brand, D. Parigot (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 44, Elsevier Science Publishers, 2001.