Skins as a Mechanism for Making Languages Syntactically Extensible

Yuen-Chang Sun Chin-Laung Lei
Department of Electrical Engineering
National Taiwan University
E-Mail: sun@fractal.ee.ntu.edu.tw, lei@cc.ee.ntu.edu.tw

Abstract

This paper introduces the skin mechanism for extending programming language syntax. A skin associates an encapsulated collection of syntax rules to semantic handlers. These syntax rules are declared all at once and used exclusively so syntax conflicts and ambiguities can be avoided. The semantic handlers can thus be reused freely. Metaobject protocol is used as the semantic handling facility. It has access to meta-level information about the user program so advanced program transformations can be done. Extended examples are given, and a summary shows the methodology for designing and implementing a skin mechanism.

Keywords: *metaobject protocol, open implementation, programming language, skin, syntactic extensibility*

1 Introduction

Because the computing environment keeps evolving, and new application domains keeps emerging, user demands for programming language features are always changing. Although any general-purpose language is capable of any computation, with a language that has high-level features tailored specifically for an application domain, the user can write more readable and maintainable programs, and the compiler can have a better chance to generate efficient code. Thus it is desirable to make a language extensible in both syntax and semantics to express new notions and new ways of computation.

Open implementation [12] is a promising approach to language extensibility. In an open implementation, part of the meta-level information about the user program, which is traditionally hidden from the user, is exposed to the base-level program in the form of a collection of *metaobjects*. This *meta-level architecture* provides the base-level program with access to the implementation strategies of the interpreter or compiler. The user can change the behavior of the interpreter or compiler, thus altering the behavior of certain language constructs, by inspecting and manipulating the metaobjects. The organization of the metaobjects and the rules by which they are accessed are called the *metaobject protocol* (MOP [13]).

Various forms of MOPs have been implemented on several production or research languages, and have been applied on fields such as parallel or distributed environments and real-time systems. These applications show that MOPs are indeed an effective way to decrease the complexity and cost in system development. However, existing MOPs are deficient in syntactic extensibility. They either stress solely in semantic extensibility, or provide only a primitive or restricted way for extending syntax. The user is forced to work at a lower level and to use unnatural notations, reducing the benefits obtained from language extensibility.

Programming language syntax is a developed and well-understood area in computer science research. There are formal methods for describing the syntax of a programming language, and there are solid algorithms for parsing programs. Yet, surprisingly, it is a difficult problem to equip a language with syntax extension facilities. Research on this subject can be traced back to early 1960's, but today no major languages, whether used in the industry or in labs, provide satisfactory mechanisms for introducing new operators, new statement forms, new declaration constructs, and so on. We argue that this situation is due to the following two principles upon which traditional syntax extension mechanisms are based. First, syntax descriptions are tightly coupled with their corresponding semantics descriptions. It is not uncommon to encounter constructs that have the same functionality but have distinct appearances in different application domains, and the coupling of syntax and semantics makes it difficult to reuse such a functionality. Second, syntax descriptions are declared incrementally and can spread all over a program. The overall syntax at a given point in a program is the accumulation of all the syntax fragments declared prior to that point. This makes it difficult to avoid syntactic conflicts between constructs, detect and fix syntactic ambiguities, and perform efficient parsing. Together these two factors jeopardize the reusability and modularity of syntactic extensions, and the reliability and efficiency of the extension facilities.

In this paper we propose the skin mechanism for achieving syntactic extensibility. A skin is a language construct that encapsulate the syntax rules of all the variants in a syntax class. Our skin mechanism is built upon a MOP. Only syntax rules are defined in a skin; their corresponding semantic handlers are specified in the skin as references to metaobjects in the underlying MOP. Since metaobjects are implemented as ordinary objects, they can be organized in a highly modular way. One metaobject can be assigned to different syntax rules in multiple skins, enabling reusability and reducing the possibility of syntax conflicts. A skin must be declared as a whole at one spot, and must be used atomically and exclusively, so it can be processed monolithically by the compiler, making it easier to avoid or detect ambiguities and to construct efficient parsing facilities. Overall, our skin mechanism makes it easier and more effective to introduce new notations and new functionality into a language so that specific needs of an application domain can be satisfied.

In the rest of this paper, we summarize the previous efforts in syntactic extensibility in Section 2. In Section 3, we present the design principles and rationales of skin mechanisms. Then in Section 4 we exemplify our ideas with several skin mechanisms for various languages. Finally we conclude in Section 5.

2 Syntactic Extensibility Works

The most popular mechanism for doing syntactic extension is macros. Simple macro systems such as the C preprocessor allow the expansion of code fragments with a trivial pattern such as identifiers or parameterized identifiers. They operate on characters or tokens. Advanced macro systems operate on abstract syntax trees and have detailed knowledge about the syntactic structure of the expanded code fragment. They can recognize code patterns and do program transformations that are more sophisticated than simple macros can. Such systems are called syntactic macros. Even more advanced macro systems such as [14] have access to the static semantic information about the user program and are called semantic macros. Most early extensible languages are based on macros. Summaries on them can be found in [5, 15, 16]. More recent work include [6, 17].

The design of macro systems follows the syntax-semantics coupling and the accumulative definition principles, and thus suffer from the above-mentioned problems. It is pointed out [8] that it is impossible to avoid syntactic conflicts in practical use of such systems, especially when extensions for infix notations are allowed. Many syntactic macro systems have to use general context-free parsing algorithms, which can have time complexity quadratic to program size. Moreover, macro systems can only handle local code transformations, making a great deal of interesting extensions impossible.

Another approach to syntactic extensibility is compiler generator. Such type of systems can generate a whole compiler from the language specification given by the user. The language specification contains the description for both syntax and semantics of the target language and is written in formal notations such as attribute grammars. It is shown [1] that such specifications can be arranged in a modular fashion and can be composed easily. A new language feature can be implemented in the form of a language module and plug into the language specification to generate a new compiler. One problem with such systems is their inefficiency. It is not unusual to see a generated compiler run one or even two orders of magnitude slower than a hand-crafted one. In addition, writing or modifying a language specification or language module, though easier than crafting a compiler from scratch, are still challenging to language users.

The basic idea of metaobject protocols [13] and open implementations [12] is to expose part of the implementation details of a language to the user. These implementation details are represented in the form of objects that exist and operate at the metalevel. These metaobjects each corresponds to a language construct such as a class, a method, an expression, or a statement, or a component of the compiler such as the scanner, the parser, or the code generator. A program being compiled can obtain the information about itself or alter the behavior of certain language constructs by inspecting the status of the metaobjects or by sending messages to them. The functionality of a construct can be extended or overridden by constructing a new metaobject that inherits the metaobject corresponding to that construct. Since the metaobjects represent the logical structure rather than the syntactic structure of the user program, metaobject protocols are capable of non-local code transformations. For example, the user may declare a PersistentClass metaobject inheriting the Class metaobject that represents ordinary classes. This new metaobject can be arranged so that where Class generates the code for accessing ordinary data members, PersistentClass generates additional code that fetches the required data from secondary storage before accessing them. Such transformations can be performed for each access to every data member of the objects with PersistentClass as their metaclass. This is beyond the capability of macros.

Metaobject protocols can be implemented on both interpreted languages and compiled languages. This paper is focused on the latter. The primary goal of most such systems is to extend semantics, but some of them also address the problem of syntactic extensibility, among which are OpenC++ ([2, 3, 4]) and MPC++ ([9, 10]). In OpenC++, the user can register new keywords that can appear only in certain places, including the modifiers of type names, class names, and the "new" operator. For example, in order to make the instances of a class persistent, the user may declare that class as

```
metaclass Node: PersistentClass;
class Node {...};
```

Instead, the user may write

```
persistent class Node {...};
```

which is more natural and descriptive than the former declaration. To achieve this, the metaobject must register the "persistent" keyword to the parser when it is initialized.

The syntactic extensibility provided by OpenC++ is quite limited. MPC++, on the other hand, is much more ambitious in this respect. In MPC++, the user can introduce new forms of language constructs, including new modifiers and specifiers, new structures, new operators, new statements, and new storage classes. Also the decoupling of syntax and semantics is supported to some extent. The MPC++ syntactic extension mechanism, however, has deficiencies. First, only a few fixed patterns can be used in syntactic extensions. Operators are limited to unary, binary, and ternary, and the syntax of new statements must be chosen from six predefined patterns. Second, syntax rules are still declared accumulatively, allowing the possibility of syntax conflicts and ambiguities, especially when the size of extension libraries and the number of incorporated libraries get large.

3 Skins

A *skin* is an encapsulated collection of syntax rules plus the references to their corresponding *semantic handlers*, which are metaobjects in the underlying MOP. One skin defines a "mini-grammar" for exactly one syntax class. A *syntax class* is a sort of language constructs. The variations in a syntax class are called its *variants*. For example, all kinds of C++ statements form a syntax class, and the for statement, the while statement, the assignment statement, and so on, each is a variant of the statement syntax class. A skin is composed of a number of *skin elements*. Each of the skin elements represents a variant in the syntax class corresponding to that skin. There may be multiple skins defined for one syntax class. The collection of such skins is called a *skin type*.

When implementing a skin mechanism for a language, the language designer must first decide which syntax classes are to be made open to the user. When a syntax class is made open, a skin type is associated to it, and the user can then define one or more skins to specify the syntax of the variants in that syntax class. Not every syntax element needs to be made open. For example, the language designer may decide to open only the operators and make

the statement forms and declaration forms left closed. A syntax class may be associated to at most one skin type. The skin type decides the syntax of the skin declarations themselves.

Each skin is assigned a skin name. When multiple skins of the same skin type are defined, the skin name can be used to identify which skin is to be used. Though more than one skin of the same type can coexist in a user program, one and only one of them can be active at any point of the program. A special skin switching directive is used to change the active skin. If no skin switching directive is found prior to a point in the program, a default skin that specifies the original syntax rules of the language is used.

A skin element has two parts. The semantics part specifies which semantic handlers are used to handle the compilation of the variant. The syntax part specifies the syntax rule of the corresponding variant.

To make the above notions concrete, we consider a simple example. Extended examples can be found in the next section. The following skin mimics the keyword mechanism of OpenC++ shown in the previous section:

```
class_skin test {
  persistent: PersistentClass;
  counted: CountedClass;
};
```

A reserved word "class_skin" is used to mark the beginning of a skin for class modifiers. This skin is named "test" and has two skin elements. They each registers a keyword and specifies the name of the metaclass that is responsible for handling the class declared with the keyword. The usage is shown below:

```
use_class_skin test;
    // uses the test skin
persistent class Node {...};
    // declares a persistent class
use_class_skin;
    // uses the default skin
counted class Point {...};
    // THIS WILL CAUSE AN ERROR!
```

Here the reserved word "use_class_skin" is the skin switching directive. If it is followed by a skin name, the currently active skin is deactivated and the specified skin is activated. Otherwise the default skin is activated. Note that a skin switching directive is a compile-time directive, so the scope of a skin is statically determined.

Skins can be used to add, override, or remove variants to or from a syntax class. It can completely change the look and functionality of the constructs in a syntax class. This change is done locally, that is, restricted to a distinguished portion of a program. This makes it easier to implement embedded and domain-specific features in a language. In contrast, traditional language extension facilities do not allow the alteration or deletion of built-in features, increasing the possibility of syntax conflicts between user-defined features and built-in ones, and also giving the user a chance to misuse a feature in a wrong context. Furthermore, with a skin mechanism the user is allowed to pick all and only the needed extensions from extension libraries. These extensions are then assigned syntax rules most customized to the application domain, increasing program readability and maintainability. For example, the user may define the test skin as

```
class_skin testA {
  persistent: PersistentClass;
};
```

if counted classes are not needed, or

```
class_skin testB {
  preserved: PersistentClass;
  enumerated: CountedClass;
}
```

if the terms "preserved" and "enumerated" are preferred in the application domain, or

```
class_skin testC {
  persistent: PersistentClassEx;
  counted : CountedClassEx;
}
```

if enhanced functionality is needed. Moreover, these skins can coexist in a program and be used as appropriate. Since skin activations are exclusive, and since the user has full control in the design of the overall syntax, syntactic conflicts and ambiguities can be avoided.

As can be seen in the above, the semantics part of a skin element is simply the name of the metaobject that acts as the semantic handler. In some cases more than one metaobject can be listed. On the other hand, the exact form of the syntax part is determined by the language designer and can vary from one skin type to another and from one language to another. We deliberately leave this part of skin design open. It is tempting to use a general scheme such as BNF to describe the syntax, but we must reject this line of thought for the following reasons. First, different syntax classes may have different syntactic structures and thus may need different ways to describe the syntax. For example, for a class modifier a single keyword is enough, while for a statement a sequence of keywords, parameters, and code blocks must be given. A general scheme cannot make prominent the syntactic nature of a syntax class. Second, it is important to maintain the consistency of appearances between the variants in a syntax class. A statement should look like a statement, not an expression, and vice versa. With a general scheme the user will loose control over this consistency. Third, even with the same kind of language constructs, different languages may need grammars of different computation power, making it necessary to alter the form of syntax description. For example, while the syntax of Pascal expressions is pure context-free, type information is needed to parse a C expression, or ambiguity will be encountered.

A skin mechanism can be implemented on virtually any language, as long as a metaobject protocol for that language exists. This MOP must have access to the backstage implementation details of every constructs of the base language, and must have full control over the compilation process. The MOPs of OpenC++ and MPC++ serve this purpose well. The MOP can be implemented in a language other than the base language, for example a C++ MOP can be implemented in Scheme. A same-language solution will, of course, be more convenient to the user.

Skins are written as part of a program and must be processed when the program is compiled. The compiler must be modified to handle this properly. For recursive descent parsers, a skin for a certain syntax class can be compiled into a parser routine. Each time a non-terminal corresponding to that syntax class is to be processed, an appropriate parser routine is called. Skin switching is done by changing the currently active parser routine. For table-driven parsers, a skin can be compiled into a sub-parse table that is substituted into the main parse table as appropriate. Mixed approaches can also be taken.

One problem with the skin mechanism described here is that whenever a variant is to be introduced, the whole declaration of the original skin must be repeated. This

problem can be solved by a skin inheritance mechanism. A skin can be declared as a sub-skin of another skin, inheriting all the elements of the super-skin. Multiple inheritance can also be allowed. For example, with the following declarations:

```
class_skin skin1 {persistent:
PersistentClass;}
class_skin skin2 {counted:
CountedClass;}
class_skin skin3: skin1, skin2 {
  guarded: GuardedClass;
}
```

skin3 has three elements corresponding to PersistentClass, CountedClass and GuardedClass, respectively. Note that extending a skin or merging several skins may not be a trivial task. The language designer may have to deal with such tasks in a case-by-case basis. For example, operators have precedence and associativity, thus special care must be taken when an operator skin is to be inherited. The next section discusses this in detail.

Another problem with our skin mechanism is that the user may have to create a metaobject for just a simple extension such as a new operator for doing matrix inversion. Our solution is to allow run-time semantic handlers. A skin element can specify a base-level routine or object to work as the semantic handler. In contrast to metaobject semantic handlers, which work at compile time and at the meta-level, a run-time semantic handler work at run time and at the base-level. Because of this, a run-time semantic handler has no access to meta-level information and has no power beyond that the other base-level facilities have. Such restrictions are not a problem for extensions like the matrix inversion operator, of course.

4 Examples

This section gives two extended examples to illustrate the usage and power of skins. The first example is a statement skin for C, and the second example is an operator skin for Pascal. We apply our ideas to different syntax classes and different languages to show their wide applicability. The underlying MOPs are presented in C++ notations.

4.1. A Statement Skin for C

When designing a skin mechanism for a syntax class of a language, the first step is to figure out the general syntactic styles of the variants in that syntax class. These syntactic styles should reflect the essence of the base-language syntax so that the extensions will have the same "look and feel" of the base-language constructs. The grammar of the C statements [11] is shown in Figure 1.

In Figure 1 the syntax rules are grouped according to their functionality. Since we are finding syntactic styles, they are re-grouped into four categories: 1) special statements, including *expression-statement*, *compound-statement*, and labeled statements with an identifier as the label; 2) the other labeled statements; 3) trivial statements, those having zero or one simple parameter, including *jump-statement*; 4) non-trivial statements, including *selection-statement* and *iteration-statement*. We decide to leave the special statements as a special case for the parser because of their special forms. Observation shows that the non-trivial statements have a syntactic pattern that is a sequence of clauses that are of the form

keyword (...) *statement*

where either (...) or *statement* may be omitted. Thus the desired syntax for the syntax descriptors in the skin elements of a statement skin can be yielded, as shown in Figure 2.

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
labeled-statement:
    identifier : *statement*
    `case` *constant-expression* : *statement*
    `default` : *statement*
expression-statement:
    [*expression*] ;
compound-statement:
    { *declaration** *statement** }
selection-statement:
    `if` (*expression*) *statement* [`else` *statement*]
    `switch` (*expression*) *statement*
iteration-statement:
    `while` (*expression*) *statement*
    `do` *statement* `while` (*expression*) ;
    `for` ([*expression*] ; [*expression*] ;
            [*expression*]) *statement*
jump-statement:
    `goto` *identifier* ;
    `continue` ;
    `break` ;
    `return` [*expression*] ;

Figure 1: Statement Syntax of the C Language

syntax-descriptor:
    labeled-stmt
    trivial-stmt
    non-trivial-stmt
labeled-stmt:
    keyword [#] : @
trivial-stmt:
    keyword [$] ;
    keyword # ;
    keyword ⌊#⌋ ;
non-trivial-stmt:
    *statement-clause optional-clause** [;]
stmt-clause:
    keyword parameter-clause @
    keyword parameter-clause
    keyword @
optional-stmt-clause:
    stmt-clause
    ⌊ *stmt-clause*+ ⌋
    ⌊ *stmt-clause* ⌋*
    ⌊ *stmt-clause* ⌋+
parameter-clause:
    (# ; # ; ... ; #)
    (⌊#⌋ ; ⌊#⌋ ; ... ; ⌊#⌋)

Figure 2: The Syntax of the Statement Syntax Descriptor

In order to distinguish the option symbols [ ] in the syntax and the ones in the syntax descriptor, we mark the latter with underlines as [ ]. This way the skin for the base-language statements can be given in Figure 3. Here [ ] means optional, # means expression, @ means statement, $ means identifier, and * and + (not used in the built-in case) means repetition for at least zero or one time, respectively. Note how the whole family of C statements (except the special ones) can be described in a concise way.

```
statement_skin c_built_in {
  CaseLabelStmt: ( case # : @ );
  DefaultLabelStmt: ( default : @ );
  GotoStmt: ( goto $ );
  ContinueStmt: ( continue ; );
  BreakStmt: ( break ; );
  ReturnStmt: ( return [#] );
  IfStmt: ( if (#) @ [ else @ ] );
  SwitchStmt: ( switch (#) @ );
  WhileStmt: ( while (#) @ );
  DoStmt: ( do @ while (#) );
  ForStmt: ( for ([#];[#];[#]) @ );
}
```

Figure 3: The C Statement Skin Declaration

To make efficient parsing possible, we must put some constraints on the skin. In this case the rules are simple. First, no two elements can have identical leading keyword, nor can any keyword coincide any other C keyword such as `int`. Second, if a semicolon ends a syntax descriptor, the last stmt-clause must be ended by a parameter-clause. The former rule is used to distinguish statements at the first place, making the syntax top-down parsable. The latter rule is used to guarantee that a statement ends with a single semicolon, as every C statement does. Note that although the whole syntax is context-free, each single syntax descriptor itself is regular if we treat expressions, identifiers, and embedded statements as terminals. Thus a finite state machine can be built for each skin element, and the parse routine for statements can be generated as in Figure 4. Note how the metaobject is generated and invoked to compile a statement. In addition to the `Compile` method, a statement metaobject must have another method `CheckSyntaxDescriptor`. It is a class method. When compiling a skin element, its syntax descriptor is properly encoded and passed to this method to check if the syntax descriptor provides exactly the parameters the semantics handler expects. This way the compiler can make sure, for example, the descriptor `( do @ after @ )` will not be associated to `IfStmt`.

The Compile method of a semantics handler performs transformation on abstract syntax trees. For the built-in statements, no transformation is needed, and the original AST is returned. For user-defined statements, new ASTs may be created and returned. For example, with the following skin:

```
statement_skin extended: c_built_in {
  ForeverStmt: ( forever @ );
}
```

the `ForeverStmt::Compile` method works by constructing an `AstWhileStmt` metaobject which represents the AST for a while statement, passing the constructor with the ASTs for a true value and the given statement, and returning the constructed metaobject.

Skin inheritance is simple with the statement case. The compiler only has to make sure that no two syntax descriptors start with identical keyword after the skin is extended or merged with another skin.

The repetition operators * and + in Figure 2 is used to implement statements that have a repeating part. For example, a C++ `try` statement can be introduced as:

```
statement_skin exception: c_built_in {
  TryStmt: ( try @ [catch (#) @]+ );
}
```

```
Ast ParseStatement (Environment E)
{
  if (matching a leading keyword of statements?) {
    find the corresponding finite state machine;
    parse accordingly, getting all the arguments;
    pack the arguments into an abstract syntax tree A;
    generate the corresponding metaobject M;
    return M.Compile(A, E);
  } else if (matching a '{'?) {
    parse and return a compound statement;
  } else if (matching an identifier and a ':'?) {
    parse and return a labeled statement;
  } else {
    attempt to parse and return an expression statement;
    report error if the attempt failed;
  }
}
```

Figure 4: The Parse Routine for Statements

4.2. An Operator Skin for Pascal

The grammar for the Pascal operators [7] is shown in Figure 5. Unlike C, postfix things like field designator, index designator and pointer dereference are not treated as operators. It may be a good idea to make them operators, but for simplicity it is not done here. This can save as the trouble of dealing with identifier operands (as in the case of field designator) and deciding the priority between prefix operators and postfix ones (what does *p++ mean in C?). We do, however, add a new type of operators, enclosing operators, to the Pascal operator family, making things that have both opening and closing symbols, like the set constructor, real operators. Thus we have three groups of operators, namely enclosing operators, prefix operators, and binary operators.

Each operator has three attributes, namely the symbol or symbols it uses, its precedence, and its associativity. Apparently, enclosing operators should take the highest precedence, prefix operators the next highest, and binary operators below them. Binary operators themselves may have different precedence levels. Associativity is not an issue for enclosing operators. Prefix operators always associate from right to left. Binary operators may have left-to-right or right-to-left associativity, but operators with the same precedence must have the same associativity. The analysis above leads to the grammar shown in Figure 6. Note that this grammar is for the whole body part of an operator skin, not for a single skin element only. This is because we have to group the skin elements according to their precedence. The declaration of the skin for the original Pascal operators is given in Figure 7.

There are three forms of enclosing operators. The syntax descriptor of an enclosing operator may have two strings. In this case the first string denotes the opening symbol of the operator, the second string denotes the closing symbol, and there must be exactly one operand between them. If a third string is given, it is treated as the delimiter of the operands, and there may be zero or more operands. The fourth string, if any, is used to separate pairs

of operands. If it is preceded by a + sign, pairing is enforced, otherwise singletons may appear, as in the set constructor case. As an example, the skin in Figure 8 declares an operator that constructs association lists.

expression:
    simple-expression [*relational-operator simple-expression*]
simple-expression:
    term [*adding-operator term*]*
term:
    factor [*multiplying-operator factor*]*
factor:
    constant
    var
    (*expression*)
    prefix-operator factor
    identifier (*expression* [, *expression*]*)
    ⊥ [*set-member* [, *set-member*]*] ⊥
var:
    identifier
    var . *identifier*
    var ⊥ *expression* [, *expression*]* ⊥
    var ^
member:
    expression [. . *expression*]
relational-operator: one of
    = <> < > <= >= in
adding-operator: one of
    + - or
multiplying-operator: one of
    * / div mod and
prefix-operator: one of
    + - not

Figure 5: Operator Syntax for the Pascal Language

skin-body:
    [*enclosing-section*] [*prefix-section*] [*binary-section*]*
enclosing-section:
    `enclosing` [*enclosing-element*]+
prefix-section:
    `prefix` [*ordinary-element*]+
binary-section:
    `ltor` (*identifier*) [*ordinary-element*]*
    `rtol` (*identifier*) [*ordinary-element*]*
enclosing-element:
    handler-list : *string* , *string* [, *string* [, [+] *string*]*] ;
ordinary-element:
    handler-list : *string* ;
handler-list:
    identifier [, *identifier*]+

Figure 6: The Syntax of the Body Part of Operator Skins

The position designator in a binary section is used to determine the precedence levels of the operator groups when skin inheritance takes place. This can be best explained by an example. In Figure 9, `Skin3` inherits `Skin1` and `Skin2` and contains three operator groups. The first group has the same identifier as the one in `Skin2`, so `GroupB` will take the highest precedence in `Skin3`. It is empty in `Skin3` so no new operator is introduced. On the other hand, `GroupA` in `Skin3` is placed the last so it will take the lowest precedence. It introduces a new operator so now `GroupA` has two operators. Between `GroupB` and `GroupA` a new group containing an operator is introduced. In summary, the precedence relationship between the four operators is `Op2 > Op3 > Op1 = Op4`. Generally speaking, when operator skins are inherited, groups with the same identifier are merged, and the precedence relationship must be clear. If there is any ambiguity, an error will be issued.

```
opskin PascalOriginal;
begin
  enclosing
    SetConstructOp: '[', ']', ',', '..';
  prefix
    PositiveOp: '+';
    NegativeOp: '-';
    NotOp: 'not';
  ltor(Multiplying)
    RealMulOp, IntMulOp: '*';
    RealDivOp: '/';  IntDivOp: 'div';
    ModOp: 'mod';  AndOp: 'and';
  ltor(Adding)
    RealAddOp, IntAddOp: '+';
    RealSubOp, IntSubOp: '-';  OrOp: 'or';
  ltor(Relational)
    EqOp: '=';  NeqOp: '<>';
    LTOp: '<';  GTOp: '>';
    LEqOp: '<=';  GEqOp: '>=';  InOp: 'in';
end;
```

// set constructor
// field designator
// index designator
// pointer dereference

Figure 7: The Pascal Operator Skin Declaration

```
opskin AssocList;
begin
  enclosing
    AssocListOp: '[*', '*]', ',', +':';
end;
```

Figure 8: A Skin Declaring an Association List Constructor

```
opskin Skin1;
begin
  ltor(GroupA)
    Op1: ...;
end;

opskin Skin2;
begin
  rtol(GroupB)
    Op2: ...;
end;

opskin Skin3(Skin1, Skin2);
begin
  rtol(GroupB)
  ltor(GroupC)
    Op3: ...;
  ltor(GroupA)
    Op4: ...;
end;
```

Figure 9: Uses of the Position Designator

Like a C statement skin, a Pascal operator skin can be compiled into a parsing routine provided an appropriate MOP has been established. Alternatively, a LL(1) grammar can be generated out of a skin declaration. Furthermore, rules must be given so that syntactic conflicts will not occur between operators. These details are not

discussed here. Another issue is run-time semantic handling. It is more convenient to define an operator at the base level instead of the meta level. This can be done by allowing ordinary functions to act as semantic handlers. Since Pascal operators are free of side-effects, such functions may not have variable parameters. When the number of operands is indefinite, as in the association list case, the operands are passed in data structures, which are provided by the MOP and must be properly memory-managed.

Note that an operator skin element may have multiple semantic handlers. In this case, which handler to use is determined by type resolution. If the resolution is successful for exactly one handler, it is taken, otherwise an error is issued. This way the ability of operator overloading can be introduced to Pascal.

4.3. Summary

The examples given in this section show how a skin mechanism can be added to a language. The methodology is summarized as the following:

- If there is not an appropriate MOP, implement one. Guidelines and examples can be found in [2, 3, 4, 9, 10, 12, 13] and so on.
- Decide which syntax class is to be extended.
- Observe the original syntax rules of the target syntax class. Figure out a general pattern. Advanced features may be added to this general pattern, but care must be taken not to go too far.
- Write down the grammar for the skin and the skin switching directive. Skin inheritance must be taken into consideration.
- Write down the declaration of the skin for the original language. It is the default skin. Also it can make sure the skin grammar is correct.
- If feasible, design a mechanism for run-time semantic handling.
- Build a compiler for the skin or extend an existing one. A third choice is a preprocessor.

5 Conclusion

The skin mechanism proposed in this paper can bring syntactic extensibility into a language without suffering from the possibility of syntax conflicts and ambiguities. Large collections of language extensions can be created in the form of MOP libraries, from which the user can pick all and only the desired extensions and compose them into solid language features that can be used exactly the same way as the original features, without any discrimination in semantics or syntax. Since syntax conflicts are avoided, those extension libraries can be reused freely. Designing and implementing a skin mechanism may be not easy, but such a facility allows the use of notations close to the target problem domain, improving the readability and maintainability of programs, and enabling the compiler to generate more efficient code.

References

[1] S. R. Adams, *Modular Grammars for Programming Language Prototyping*, Ph.D. thesis, University of Southampton, March 1991.

[2] S. Chiba, "A Metaobject Protocol for C++," *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299, Austin, Texas, October 15–19, 1995.

[3] S. Chiba, "Macro Processing in Object-Oriented Languages," *Proceedings of Technology of Object-Oriented Languages and Systems*, Melbourne, Australia, November 1998.

[4] S. Chiba, *OpenC++ 2.5 Reference Manual*, Institute of Information Science and Electronics, University of Tsukuba, 1999.

[5] C. Christensen and C. J. Shaw, ed., *Proceedings of the Extensible Languages Symposium*, Boston, Massachusetts, May 13, 1969.

[6] R. Hieb, R. K. Dybvig, and C. Bruggerman, "Syntactic Abstraction in Scheme," University of Indiana Computer Science Technical Report 355, 1982.

[7] *An American National Standard: IEEE Standard Pascal Computer Programming Language*, IEEE, 1983.

[8] E. T. Irons, "Experience with an Extensible Language," *Communications of the ACM*, vol. 13, no. 1, pp. 31–40, 1970.

[9] Y. Ishikawa, *Meta-level Architecture for Extendable C++*, TR-94024, Real World Computing Partnership, 1995.

[10] Y. Ishikawa et al., "Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach," *Proceedings of Reflection'96*, San Francisco, California, April 1996.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall International, Inc., 1988.

[12] G. Kiczales, J. Lamping, and G. Murphy, "Open Implementation Design Guidelines," *Proceedings of the 19th International Conference on Software Engineering*, 1997.

[13] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1995.

[14] W. Maddox, "Semantically-Sensitive Macroprocessing," Report No. UCB/CSD 89-545, University of California, Berkeley, 1989.

[15] S. A. Schuman, ed., *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, September 6–8, 1971.

[16] N. Solntseff and A. Yezerski, "A Survey of Extensible Programming Languages," *Annual Review in Automatic Programming*, vol. 7, pp. 267–307, Pergammon Press, 1974.

[17] D. Weise and R. Crew, "Programmable Syntactic Macros," *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pp. 156–165, June 1993.