Domain Specific Meta Languages

Eric Van Wyk *
Oxford University Computing Laboratory
Eric.Van.Wyk@comlab.ox.ac.uk

ABSTRACT

There are several different problem domains in the implementation of language processing tools. The manipulation of textual data when generating code, creation and inspection of environments during type checking, and analysis of dependency graphs during program optimization and parallelization are but a few. The use of domain specific languages to solve these sub problems can reduce the complexity of a tools specification. We argue this point in the realm of attribute grammars and use domain specific meta languages to write attribute definitions.

1. INTRODUCTION

Attribute grammars [2; 14] provide a convenient specification mechanism for defining language processing tools. One attaches attribute definitions to the productions of a context free grammar which define attribute values of the language constructs in the production. Although the specification is declarative and nicely decomposed by the production of the grammar, attribute grammars can be complex, repetitive, and difficult to write, read and debug [10; 8; 6]. One technique for addressing these problems comes from the realization that in the specification of language processing tools, there are several different problem domains. We claim that using domain specific languages to express solutions to these sub problems can reduce the complexity of the tools specification. In the framework of attribute grammars, this means that the attribute definitions should be written in domain specific languages appropriate to the problems addressed by the attributes. We call these languages domain specific meta languages. This approach is different from other attribute grammar systems which use a single meta language for all attribute definitions.

A common task in language translators is to generate target language text or a text based set of error messages. Macro processors provide a convenient mechanism for generating textual output and we adopt a macro language as a domain specific meta language for text attributes and show its use in an example translator for a simple language. Translators often generate complex internal data structures which are queried during language translation. In type checking, an internal representation of program defined types is generated and an environment structure is created for associating type and variable names with their types. This is a very specific domain and the definition of attributes used in solving this sub problem can benefit from being written in a domain specific language. We describe a simplified version of such a language in our example. Another area in which domain specific languages are useful is in program optimization and parallelization. The process often represents the source program in the form of flow graphs, data dependency graphs, and program dependency graphs [7]. In this domain a language with built in graph construction and analysis operations can simply solution specifications [21].

Since we have many distinct problem sub domains in language processing tool specification, it is only natural to specify their solutions in domain specific languages. These languages raise the level of abstraction by providing data types and control structures for the specific items in a domain. This frees the specification writer from dealing with implementation details and leaves him or her free to concentrate on the problem solution, not on its encoding in a general purpose language. The use of domain specific meta languages in the definitions of attributes can thus significantly simplify attribute grammar specifications.

The aim of this research is to explore possible meta languages for defining various semantic aspects of programming language extensions, called intentions, for the Intentional Programming (IP) system [19] under development at Microsoft Research. An intention can be seen as a production and definitions of the appropriate attributes which allow it to be added into an existing language framework. The aim of IP is to allow applications programmers to extend their programming languages to meet their particular needs in a given problem domain. When a programmer defines a new intention, the syntactic representation and its semantics must be specified. Thus, we are interested in exploring different meta languages which the programmer might find useful in defining intentions. The system described here provides a convenient mechanism for designing and implementing such meta languages.

We describe some related work in simplifying the specifications of language processors in Section 2. Section 3 provides a motivating language translation problem. In Section 4 we give its solution in the form of an attribute grammar.
which uses three meta languages for defining attributes: a
general purpose language similar to those in most attribute
grammar systems, a macro language for defining the textual
target code, and a simple language designed to specify an en-
vironment. Section 5 briefly describes how domain specific
meta languages can easily be implemented and integrated
into our system. Section 6 provides a discussion.

2. RELATED WORK

There are several other techniques designed to address the
problems of complexity in attribute grammars. The use of
domain specific meta languages should be described in a modular
manner [13]. Attribute coupled grammars of Ganzinger and
Geirech [8] increase the modularity of attribute grammars
by recognizing that the translation process is often broken
into distinct phases mapping the source program through
a series of intermediate representations until its final tar-
target language representation is generated. Attribute coupled
grammars specify each translation phase as an attribute
grammar which generates the intermediate form expected
by the following phase.

Attribute grammars are usually specified with all attribute
definitions for a production's non-terminals written together
with the production. Instead of decomposing attribute gram-
grams by production, they can also be decomposed by at-
tribute, that is, writing all definitions for one or more related
attributes together [1; 6; 5]. Most attribute grammar sys-
tems [9; 11; 12; 18] provide methods for grouping attribute
definitions by attribute by textually regrouping definitions
by production before analysis of the attribute specifications.
Grouping definitions by attribute allows one to concentrate
on a specific sub problem solved by an attribute's definitions.
In a method proposed by Dueck and Cormack [6] attribute
definition templates are used to automatically generate sev-
several attribute definitions. Definition templates are associ-
ated with a production pattern that is matched against the
productions in a context free grammar. Those which match
are given attribute definitions generated from the templates
by the matching information. Ebl [9] provides similar facili-
ties but also allows different types of reference mechanisms
to attribute values in remote nodes in the tree so that a
definition is not restricted to referencing attributes on the
parent and child nodes only.

Macro processors have often been used to implement lan-
guage translators [23; 4; 22; 17; 21]. Essentially, one defines
a macro for each construct in the source language. The body
of the macro expands into the translation of the construct in
the target language. Some macro processors provide pow-
erful extensions to perform rudimentary semantic analysis
tasks on the source program. Tanenbaum [29], for instance,
used the symbol table facilities of the ML/I macro processor
to hold the type and run-time address of the program vari-
ables in a source program. Macro processors are however
rarely used as a single solution in writing language trans-
slators since the extensions provided for handling non-textual
data are inadequate for the complex semantics of modern
languages. Macro processors are an appropriate mecha-
nism for generating textual attribute values, but processes
in other domains should be written in a more appropriate

3. MOTIVATING EXAMPLE

As a motivating example, we shall implement a translator
for a toy language with the scope rules of Algol60. The
scope rules state that an identifier $x$ is visible in the smallest
enclosing scope except for any inner blocks which also define
an identifier $x$. A concrete syntax for this language is given
below:

```plaintext
program: <Program> = "program" <Block> "."
lst: <Block> = <StmtList>
lst1: <StmtList> = <Stmt>
lst2: <StmtList> = <StmtList> <Stmt>
local: <Stmt> = "[" <Block> "]"
dec: <Stmt> = "dec" "id"
use: <Stmt> = "use" "id"
```

An example program may be:

```plaintext
program use a use b dec b
 [ dec a use a use b ] dec a use a .
```

The use of $a$ in the inner block refers to the inner declara-
tion of $a$ while the use of $b$ in the inner block refers to the
declaration in the outer block.

We will map this language into a simple stack machine lan-
guage whose instructions are identifier references or state-
ments marking the entrance or exit to a block. A block entrance
has an entrance statement which is labelled by the
lexical level of the block and the number of local variables
defined in the block. The exit of a block has an exit state-
ment labelled by only the lexical level. A variable reference
is indicated by a reference statement which is labelled by the
lexical level and offset of the variable's declaration. The offset
of a variable declaration is the order in which it appears
in the block. Our example program above would be mapped
into the following target program:

```plaintext
Enter 0 2, Ref 0 0, Ref 0 1,
Enter 1 1, Ref 1 0, Ref 0 1, Exit 1,
Ref 0 0, Exit 0
```

4. META LANGUAGES

In this section, we provide the specification of an attribute
grammar which maps code in our toy language into the stack
machine code and uses domain specific meta languages in
the definitions of its attributes. The attribute grammar has
three attributes: code – a synthesized text attribute defining
the target code, level – an inherited integer attribute defin-
ing the lexical or block nesting level, and env – an inherited
attribute defining the environment that can be queried by
operations id_level and id_offset to give, respectively,
the level and offset of a variable's declaration. A third operation
number_locals returns the number of local variable decla-

```plaintext
Enter 0 2, Ref 0 0, Ref 0 1,
Enter 1 1, Ref 1 0, Ref 0 1, Exit 1,
Ref 0 0, Exit 0
```

```plaintext
number_locals returns the number of local variable decla-
```
We first present the definitions of the level attribute written in a general purpose meta language to introduce our basic framework and notations. This language is essentially Haskell [3] with embellished naming conventions for referencing attributes associated with terminal and nonterminal symbols. It is similar to the attribute definition languages used in other systems. We then present the definitions of the code attributes using a domain specific macro language. These definitions compare favorably to the equivalent definitions written in the general purpose meta language. Finally we present the definitions of the environment attribute env in its domain specific language.

4.1 Lexical level

The level attribute is an inherited attribute which defines the lexical level of the blocks in the source program. Since it is an inherited attribute, we can use a default copy rule which copies inherited attribute values from parent to child nodes if no other rule is specified. Thus, we need only write attribute definitions for the program and local productions:

```haskell
/\ level Haskell
program: <Program> = "program" <Block> "."
<Block>.level is 0

local: <Stmt> = "{" <Block> "}"
<Block>.level is <Stmt>.level + 1
```

The first line gives the name of the attribute being defined (level) and the domain specific language used to write the definitions (a Haskell-like meta language). Attributes of a nonterminal are referenced by following the name of the nonterminal in angle brackets by a dot (.) and the attribute name. If more than one nonterminal in a production have the same name, the name of the non-terminal is followed by an integer indicating its order in the production.

4.2 Target code

In this section we use macros as a domain specific meta language to define the code attribute. When using macros, one does not write the commands to build up the textual value of an attribute, but instead writes a macro which is expanded by plugging attribute values of terminals and nonterminals into the holes in the macro body. These holes are the formal parameters of the macro and are just the references in the macro to attribute values of terminals and nonterminals written in the same style as above. When using macros to define a synthesized attribute of the nonterminal on the left hand side of the production we can drop the attribute reference to the left of the is keyword seen above so that all of the text between the production and the following production is seen as the body of the macro defining the attribute's value. Since we group attribute definitions by attribute, all definitions in a single file define the same attribute, thus there is no confusion. The specification for the unary productions program, slist1, and local are not shown since they simply copy the code attribute of the single child to the parent.

```
/\ code Text
list: <Block> = <StmtList>
Enter <Block>.level "number_locals <StmtList>.env <StmtList>.level"
  <StmtList>.code
Exit <Block>.level
```

The definition of the code attribute for the <Block> nonterminal of the list rule is the three line macro shown above beginning with Enter and ending on the last non-blank line above the following rule. At compile time, this macro is expanded by filling the formal parameters (the attribute references) with their values. The generated text begins with an Enter statement, followed by the lines of code of the statement list (each indented two spaces as shown in the macro) and concludes with an Exit statement. This text becomes the value of the code attribute of <Block>. (Note that the backslash (\) is a line continuation symbol.) The Enter statement, which needs the number of local variables in the block, and the Ref statement, which needs the level and offset of the referenced variable, both take parameters which are not stored directly as attributes. These values are extracted from data structures stored in attributes and thus we need some common interface language which a macro can use to extract the required information. We naturally choose as this interface the general purpose meta language used in Section 4.1. Inside a macro, phrases in this language are written between tildes (\~) and are expression which return the desired value to be copied into the expanded text of the macro. In the list macro above, the phrase

```
  "number_locals <StmtList>.env <StmtList>.level"
```

is written in the general purpose attribute definition language and makes use of the number_locals function to compute the number of local variables defined in the block with the level of the statement list. Similarly, the use macro uses the common interface language to extract from the environment <Stmt>.env the level and offset of the identifier whose lexeme is given by "id".lex.

The macro language provides a domain specific meta language which is clearly a better choice than a general purpose specification language. If we had written the definition of the code attribute for the slist2 production using the general purpose meta language, it would have looked like the following where ++ is string concatenation:

```
slist2: <StmtList> = <StmtList> <Stmt>
<StmtList>.code is <StmtList>_1.code ++ "\n"
  ++ <Stmt>.code
```

The definition for list is even more daunting:

```
list: <Block> = <StmtList>
<Block>.code is "Enter " ++ show <Block>.level ++ 
  " " ++ show (number_locals <StmtList>.env <StmtList>.level) ++ "\n" ++ <StmtList>.code
 ++ "\nExit " ++ show <Block>.level
```

The definition for list is even more daunting:
These are much more difficult to comprehend than the domain specific macro versions in which we see exactly the structure of the target code text we want to generate. In the general language definition, we have to study the code to understand what the target code will be. (The show function converts a value to its textual representation.) Another significant advantage of the macro specification language is that the white space (spaces, tabs, new lines) between text and attribute values is inserted automatically into the expanded text. Also, it only inserts white space between parameters if they are both not the empty string. For example, since the dec rule generates the empty string as the code attribute’s value, if the $<\text{Stmt}>$ in the slist2 rule is a declaration we would not want the new line between the formal parameters in the macro body to be inserted into the generated code. This would lead to spurious blank lines in the final target code. The macro language ensures that this does not happen. Adding this feature to the general purpose specifications would make them even more complicated. The macro meta language lets us write specifications which say exactly what we want the target code to look like and shields us from the many complexities one must deal with in a general purpose specification language.

4.3 Environment

In this section we define the environment attribute $\text{env}$ using a domain specific language tailored to our example. This language is used to define the environment of a block and allows remote contributions of declared local variables to an environment. Since $\text{env}$ is an inherited attribute, we can again use the default copy rule to copy the attribute value from parents to children. Thus we need only specify attribute definitions for $\text{env}$ for the rules $\text{program}$ and $\text{list}$.

```
\ /
\ /
\ /
env EnvironLang
program: $<$Program$>$ = "$\text{program}$ $<$Block$>$ .$.$
$<$Block$>$.$\text{env}$ = $<$empty$>$ environment

list: $<$Block$>$ = $<$StmlList$>$
$<$StmlList$>$.env$ = add locals at $<$StmlList$>$.level
to $<$Block$>$.env

dec: $<$Stmt$>$ = "$\text{dec}$ "$\text{id}$
contribute "$\text{id}\"".lex to locals
```

These rules define the environment of the outer most block to be empty, and the environment of inner blocks by adding an ordered list of local variables at a specific lexical level to the enclosing environment. The $\text{contribute}$ statement in the $\text{dec}$ rule adds the lexeme of the identifier ($"\text{id}\".\text{lex}$) to a list of local variables which will be added to the environment at the enclosing block. When the $\text{id}\_\text{level}$ and $\text{id}\_\text{offset}$ operations query an environment with an identifier name, they search each set of local variables in the environment by decreasing lexical level until the identifier is found. The $\text{contribute}$ statement is similar to the remote attribute access methods in EH [9] and allows us to avoid creating another attribute to pass local variable names up to the nearest enclosing block up the tree. Behind the scenes the meta language may implement $\text{contribute}$ by defining a hidden synthesized attribute to pass local variables up to the definition of the enclosing environment, but it is not the concern of the person writing attribute definitions. We are simply seeking, in this admittedly simple example, to raise the level of abstraction when dealing with the environment.

5. THE TOOLS

The set of tools we use to evaluate attribute grammars using domain specific meta languages map our attribute grammar specifications into a lazy functional program written in Haskell[3]. This generated Haskell program is composed of a set of functions which embody the structure and functionality of the attribute grammar. These functions are composed to generate a single function which translates a given input program to the desired attribute values. In fact, attribute grammars can be seen as a style of writing lazy functional programs [10; 16]. Of special importance is the fact that when attribute grammars are written in this style, the Haskell type system verifies that all attributes are defined exactly once and the lazy evaluation strategy of Haskell schedules the evaluation of the attributes.

The generated Haskell program is written in an aspect oriented style [5] in which all the code defining a particular attribute is held in a single extensible record structure called an\textit{aspect}. Each aspect can be type checked, parameterized and separately compiled. All the aspect definitions, one for each attribute, are then combined to generate the Haskell function mapping input programs to attribute values. Since our attribute grammar specifications are decomposed by attribute and thus attribute definitions of an individual attribute are kept in a separate file, it is straightforward to write translators which map attribute definitions into the desired Haskell\textit{aspect} code which is then merged into the final target program.

In addition to the Haskell code for each attribute, there is a main set of enclosing functions generated by a translator from the productions which combine the attribute definition functions into the main function implementing the attribute grammar. These functions are also generated by a translator which maps the set of productions into this set of functions. The heart of our system is collection of loosely integrated translators: one which maps the source language syntax to the main structural code of the attribute grammar and one for each domain specific meta language which maps attributes defined by that language into their Haskell code implementation. All the generated functions are combined into a single Haskell module which implements the the language processing tool defined by the attribute grammar. Since each meta language requires a translator as its implementation, it is only natural to define and implement all of these translators using this same set of tools. Thus, for each meta language we define an attribute grammar specifying the translation from attribute definitions written in that meta language into their Haskell function implementation. All meta language processors have the same enclosing syntactic structure, sketched below:

```
<Aspect> = <SpecList>
<SpecList> = <SpecList> <Spec>
<SpecList> = <Spec>
<Spec> = "$\text{rule name}\" <BNF_Rule>
<MetaLanguageFragment>
```

The\textit{MetaLanguageFragment} construct contains the entire attribute specification associated with a production. Thus the meta language has complete control over the syntax and semantics of its components. Because we can automatically generate meta language processors from their specifications it is not difficult to extend a meta language with new features or to build a completely
new one, as we did for the environment attribute. Also, it is
very easy to add new meta language translators to the sys-
tem and thus integrate new domain specific meta languages
into our attribute grammar system.

6. DISCUSSION
When attribute definitions are presented in an aspect ori-
eted approach, these specifications are easier to read be-
cause all visible attribute computations are written in the
same language. When grouped by production, one may
see several attribute computations written in different lan-
guages. This may be confusing, but reflects the fact that
reading attribute grammars grouped by production can be
confusing itself because one is not allowed to separate one's
concerns. However, we also note that one can put defini-
tions for more than one attribute written in different meta
languages into our attribute grammar system.

Thus, there is usually an appropriate balance between
grouping all attribute definitions in one file or spreading
them all out into individual files.

7. REFERENCES

Language Prototyping. PhD thesis, University of Southam-


ware implementation. Computer Journal, 12(4):327–331,
November 1969.

oriented compilers. In First International Symposium on Generative and Component-Based Software Engi-
neering, 1999.

170, 1990.

dependence graph and its use in optimization. ACM
Transactions on Programming Languages and Systems,


and W. M. Waite. ELI: A complete, flexible compiler
construction system. Communications of the ACM,

[10] T. Johnsson. Attribute grammars as a functional pro-
gramming paradigm. In G. Kahn, editor, Functional
Programming Languages and Computer Architecture,
volume 274 of Lecture Notes in Computer Science,

lec. Design, implementation and evaluation of the
FNC-2 attribute grammar system. In Conference on
Programming Languages Design and Implementation,
pages 209–222, 1990. Published as ACM SIGPLAN No-
tices, 25(6).

Practical Compiler Generator, volume 141 of Lecture
Notes in Computer Science. Springer Verlag, 1982.


construction using attribute grammars. SIGPLAN No-

[16] M. Kuiper and S. D. Swierstra. Using attribute gram-
mars to derive efficient functional programs. In Com-

Master's thesis, The University of Iowa, Department of
Computer Science, Iowa City, IA 52242, 1990.

[18] T. W. Reps and T. Teitelbaum. The Synthesizer Gener-
ator: A system for constructing language-based editors.
Texts and Monographs in Computer Science. Springer-
Verlag, 1989.

in the legacy age. Presented at IFIP Working
microsoft.com/ip/, 1996.

[20] A. Tanenbaum. A general purpose macroprocessor as a
poor man's compiler compiler. IEEE Transactions Soft-

PhD thesis, The University of Iowa, Iowa City, Iowa,
52240 USA, July 1998.


compiler for a simple list processing language. Annual Re-