Object-oriented Compiler Construction

Axel-Tobias Schreiner, Bernd Kühl
University of Osnabrück, Germany
{axel,bekuehl}@uos.de, http://www.inf.uos.de/talks/hc2

Extended Abstract

A compiler takes a program in a source language, creates some internal representation while checking the syntax of the program, performs semantic checks, and finally generates something that can be executed to produce the intended effect of the program.

The obvious candidate for object technology in a compiler is the symbol table: a mapping from user-defined names to their properties as expressed in the program. It turns out however, that compiler implementation can benefit from object technology in many more areas.

If the internal representation is a tree of objects, semantic checking and generation can be accomplished by sending a message to these objects or by visiting each object. If the result of generation is a set of persistent objects, program execution can consist of sending a message to a distinguished object in this set.

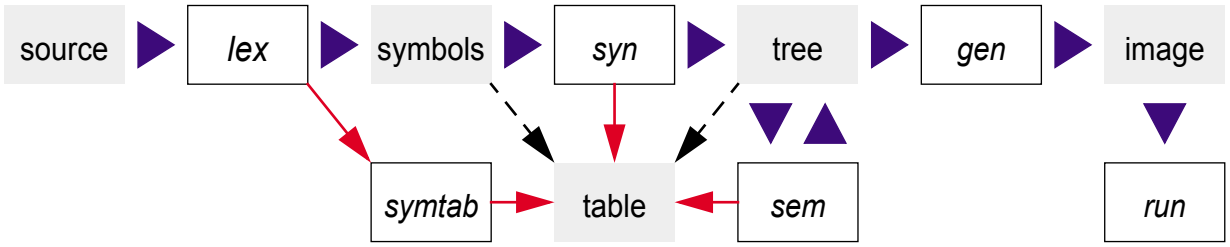
Compilers are usually made with tools such as parser and lexical analysis generators. A parser-generator takes a grammar, specified in a language such as BNF or EBNF, checks it, and constructs a representation (the parser) which will execute semantic actions as phrases over the grammar are recognized. If the parser consists of a set of persistent objects, checking the grammar is accomplished by sending a message to the objects. Similarly, recognizing a program amounts to a message to the start symbol of the grammar. Goal objects are then created to represent the phrases to be recognized and the user actions are defined as methods for the goals which are called from the parsing objects.

The presentation will discuss existing Java implementations of these ideas: *oops*, a parser-generator based on EBNF and objects; the compiler kit, a class library for implementing parse trees and interpreters for typical programming language constructs; and *jag*, an object-based tool for tree traversal, helpful in debugging and classical code generation.

Introduction

This paper summarizes our experiences how compiler construction benefits from object-oriented programming techniques. We have gained them in several projects and used them to great advantage in two courses on compiler construction with Objective C and Java [1].

It turns out that OOP can be applied productively in every phase of a compiler implementation and it delivers the expected benefits: *objects* enforce information hiding and state encapsulation, *methods* help to develop by *divide and conquer,* all work is carried out by *messages* which can be debugged by instrumenting their methods. Most importantly, *classes* encourage code reuse between projects and *inheritance* allows code reuse within a project and modifications from one project to another. As an added benefit, modern *class libraries* contain many pre-fabricated algorithms and data structures.



The diagram shows the major components of a typical compiler: *lex* collates source characters into words. *symtab* manages a table mapping these words to symbol descriptions which are passed around during the rest of the compilation process. *syn* checks the symbol sequence against a grammar and usually constructs a tree representing the source. *sem* checks the tree for semantic correctness and might modify it to account for implicit operations. Finally, *gen* produces an image of the source that can be executed in some *run* environment.

The rest of this paper tours these components and discusses how object-orientation can be applied in their implementation and what benefits we found.

Symbol Table

symtab manages a container for descriptions of mostly user-defined symbols: *lex* assembles a word form the source and hands it as a key to *symtab* to locate a description or create a new one. The lookup happens once per word of the source: from this point on the description is passed along by reference. Other parts of the compiler query the description or contribute more information.

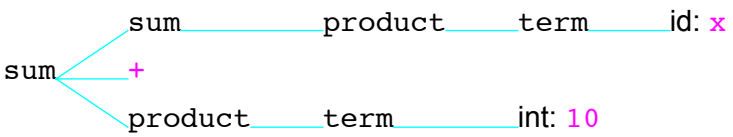
Symbol table and descriptions are obvious candidates for OOP within a compiler: the table is a container object where each description object is held. The descriptions share at least the ability to be located by key. If a base class is used to hold the key, inheritance helps to encapsulate and share the lookup mechanism while keeping it separate from the information constituting the actual description. Depending on the implementation language, it is highly likely that an existing class library provides a suitable container, an efficient lookup mechanism, and perhaps even value representations for some of the description information.

Parse Tree

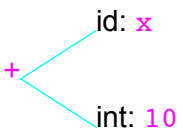
Given an excerpt from a typical grammar:

```
sum:     product | sum '+' product
product: term | product '*' term
term:    identifier | literal
```

Then `x + 10` is a sentence for which the following trees can be built:



abstract syntax tree



parse tree

The leaves of both trees are input symbols, other nodes represent grammar phrases. An abstract syntax tree contains a node for each recognized nonterminal symbol and the children correspond to the symbols in a phrase for the nonterminal symbol; parser generator tools such as *JavaCC* [2] often produce an abstract syntax tree automatically. A parse tree is a

pruned version of the abstract syntax tree; while it must be built more or less by hand during syntax analysis, it can be designed to be much more suitable for the tree traversals constituting the remaining phases of the compilation process.

Once the tree nodes are objects of classes specific to the grammar phrases, the rest of the compilation is carried out by methods in these classes, i.e., there is an automatic *divide and conquer* imposed for semantic analysis, etc. Inheritance further simplifies the implementation, e.g., there is little difference in checking all six relational operators or in generating code for some commutative operators and different value types.

For our courses we have implemented an extensible *compiler kit* (class library) as a reusable universal back end. Syntax analyzers for different languages build parse trees using these classes and the compiler kit does everything else: it performs semantic analysis and generates a persistent, executable tree as an image. The Java version of the kit was carefully designed so that semantic analysis can be modified by inheriting from and overwriting classes in the kit. Data type operations can be restricted or extended, new data types can be added, and type mixing rules can be adapted. The Objective C version of the kit included code generation into Holub's C-code [3], an assembler-like coding style for C emulating a fictitious machine architecture.

OOP opens the possibility to implement the expression part of a language by selecting from the compiler kit's data types (or adding one's own) and building a parse tree using these classes. The drawback is that there are many classes and many, mostly small methods, but the advantage is a very significant gain in reusability. The following sections discuss some details.

Execution

Using OOP a compiler can transform a program source into a tree of persistent objects as an image. Execution is accomplished by sending a message to the root node of that tree which results in partial traversal as the message is passed along the tree.

Specifically in Java it is very simple to make objects persistent. This results in a cheap, platform-independent image format and images can be executed wherever a Java machine is available. Moreover, the execution message can be reused in the compiler, e.g., when constant expressions need to be evaluated or expressions should be partially folded. This is a significant advantage as it ensures that identical mechanisms are used for evaluation during compilation and execution.

Semantic Analysis

Semantic analysis decides if a syntactically acceptable program is meaningful. For a large part it is concerned with the interaction of various data types in expressions: during a post-order traversal of the parse tree, result types are computed for each part of an expression and stored in the nodes for the benefit of code generation.

Parse tree nodes are objects, their classes must implement a method to perform semantic checking of the node. If necessary, the method can augment the parse tree with conversion nodes. Types are modeled as unique objects. They have methods informing the semantic analysis about available operations for the type and about permissible interaction with other types. Other type methods generate a simplified, persistent runtime tree which can be used as an interpreter or traversed for code generation.

Implementing semantic analysis as a method for the node classes automates a *divide and conquer* approach which immediately carries over to new classes. A lot of effort in a compiler

implementation is spent on code to check expressions. OOP and a careful design of type modeling permits reusing this code in new projects.

Types

It is very important to design the type mechanism so that it can be modified, extended, or restricted in different projects. The key is to let operator nodes ask type objects during semantic analysis whether the intended operations are in fact available:

```
sem:
  for children: child.sem()   // sets children's types
  for children:
    if child.type.supports('+', otherType):
      type = child.type.result('+', otherType)
      break
  if type not set: error      // impossible operation
```

This basic approach is extended to let each type decide if it is willing to convert from the other; if so, the type is asked to add a conversion node to the tree.

Types are represented by type objects, i.e., a type class from the compiler kit can be extended or restricted by subclassing and using a subclass object to represent a modified type. Thusly, the compiler kit can be reused to implement a language with a different set of types and mixing rules.

Syntax Analysis

A compiler converts a sentence written in some language, i.e., a program, into an executable image. A compiler-compiler converts a sentence written in a language like EBNF, i.e., a grammar, into an image which is itself a compiler.

As discussed above, an image is a tree of objects which understand a message for semantic analysis and another message for execution. Semantic analysis for a grammar means to check if the grammar is suitable for parsing, e.g., because it fulfills a condition such as LL(1). Using the image resulting from a grammar means at least to perform recognition, i.e., the execution message must implement at least a parsing algorithm.

When the parser recognizes a phrase, it must either build an abstract syntax tree or it should execute some user action. A popular generator, *yacc*, augments phrases with C statements. The phrases are specified in BNF, i.e., without an iteration syntax, to simplify how the C statements get access to the symbols accepted by the phrase.

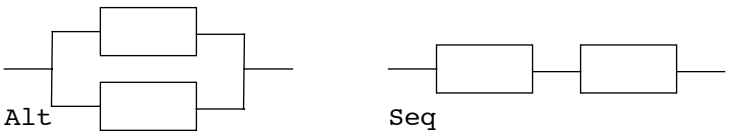
We have employed OOP to implement a parser generator *oops* [4] which accepts a grammar written in EBNF, checks that it is LL(1), and creates a recursive descent parser for it. *oops* compiles itself; it was bootstrapped with *jay*, a version of *yacc* which we retargeted to Java [5]. Both versions of *oops* share the class library for the execution trees.

Unlike *yacc*, *oops* can deal with EBNF and has no syntax for user actions. When the generated parser starts on a phrase it creates a goal object from a phrase-specific class. The goal object receives *shift* and *reduce* messages as the phrase is recognized and completed. User actions can be implemented in the required methods for the goal classes. We provide trivial goal classes for checking a grammar and tracing recognition.

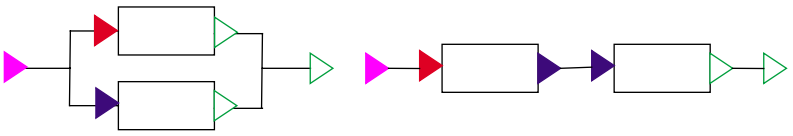
Objects play a major role in *oops*: parser objects encapsulate lookahead and follow sets, which are used for LL(1) checking and for steering the recursive descent during execution, goal objects encapsulate the state of phrase recognition. We have implemented an automatic error recovery based on the lookahead and follow sets, but the goal objects could be allowed to participate as well.

Divide and Conquer

The OOP approach taken in *oops* automates a design by *divide and conquer* for LL(1) checking and parsing. *oops* is simple enough for use in a compiler construction class to lead to discovery of the algorithm based on syntax graph building blocks like the following:



These blocks can be represented by `Alt` and `Seq` nodes which accomplish recognition by deferring to their subtrees to process alternatives or a sequence. Specifically, `Alt` needs to know which subtree to invoke. This can be decided by considering lookahead sets:

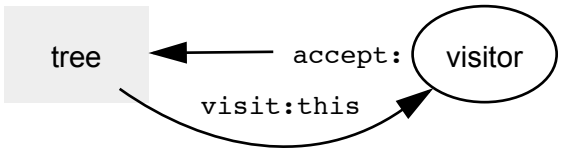


`Alt` requires the lookahead sets of the subtrees to be distinct, and to be distinct from the follow set if there is an empty alternative. The lookahead sets can be computed by inspection; `Alt` adds the alternatives, `Seq` adds terms as long as there is a possibly empty term. The follow sets result from back propagation of lookahead sets; this has to be iterated as long as a nonterminal acquires a bigger follow set within a graph.

The point is that all these considerations are local within each building block for a syntax graph and lead directly to methods for the class representing the block. They are simple enough to be developed in class and they yield a working LL(1) parser generator.

Tree Traversal Techniques

Many algorithms in a compiler employ tree traversal. For dealing with object trees, OOP has several techniques to offer:




```
interface Visitor {
  visit (SomeClass node);
     // for each kind of node class
}
```

In the visitor design pattern, a `Visitor` object is sent to the root of the tree. A node always calls the visitor back sending itself as an argument. The class of the argument, i.e., the class of the visited node, is used to divide node processing among different methods in the visitor. In Java, overloading can be employed: `Visitor` objects must implement a `visit` method for each possible node class and the `accept` method is implemented in each node class so that overloading selects the appropriate method at the visitor. To allow for tree traversal, the node class gives a visitor access to a node's children.

`Visitor` support can be generated by a tree-building parser such as JavaCC but is fairly difficult to extend or inherit later, i.e., reuse of visitors is hard once the node class library is modified. The visitor pattern certainly encourages design by *divide and conquer* but it is awkward to share the same action for different node classes.

In particular for semantic analysis we found it more convenient to implement a tree traversal by requiring each node class to implement a suitable method directly. While this precludes different implementations for the same traversal job, e.g., transparently selecting code generation for different architectures, it simplifies inheritance and code reuse significantly over the visitor pattern implemented by JavaCC. In Objective C, categories can be added to existing classes to add new methods — this is a very useful mechanism to add new traversals to an entire class hierarchy.

A third, more powerful technique is method selection based on a pattern of node and children classes:

```
node-class child-class... { Java statements with access to node and children }
...
```

We implemented a simple tool, *jag,* to convert these pattern/action statements into Java methods which are conceptually attached to the node classes and inherited by subclasses, much like the effect of Objective C categories. Inheritance combined with overloading permits refinement of initially very coarse traversal rules and significant reuse between projects.

Conclusion

The popularity of Java has made it the language of choice for many university courses if not industrial projects. While in our opinion Java is not yet quite robust, efficient, and above all portable enough for mission-critical applications, it is likely to get there soon and it does make sense to investigate old programming techniques using new paradigms.

Java supports and (much more than C++) encourages OOP which makes significant promises to improve critical aspects of the software development process. We tried to show in this paper that these promises do apply to compiler construction: for projects, OOP in compilers permits significant code reuse, for instruction, OOP in compilers simplifies the design effort and makes some important algorithms accessible and transparent.

References

[1] `http://www.vorlesungen.uos.de/informatik/compilerbau98`

[2] `http://www.metamata.com`

[3] A. Holub, *Compiler Design in C*, ISBN 0-130-255-252-5.

[4] `http://www.inf.uos.de/oops`

[5] `http://www.inf.uos.de/jay`