Object-oriented Compiler Construction

Axel-Tobias Schreiner, Bernd Kühl
University of Osnabrück, Germany
{axel, bekuehl}@uos.de, http://www.inf.uos.de/talks/hc2
http://www.vorlesungen.uni-osnabrueck.de/informatik/compilerbau98

Object Orientation
- objects: information hiding, state encapsulation
- methods: develop by divide and conquer
- messages: debug by instrumentation
- classes: reuse between projects
- inheritance: reuse within and between projects
- class libraries: many pre-fabricated algorithms and data structures

Compiler Construction

source ➔ lex ➔ symbols ➔ syn ➔ tree ➔ gen ➔ image
d ➔ symtab ➔ table ➔ sem ➔ run
Symbol Table

- container for descriptions of (mostly user-defined) symbols
- *lex* locates a description by key (identifier), elsewhere it is passed by reference,
  *syn* and *sem* contribute information
- symbol table and symbols can be objects

**object advantage**

- inheritance: encapsulate and share lookup mechanism, separate it from actual description
- objects: encapsulate and hide description information
- class libraries: provide containers, lookups, and value representations
Parse Tree

given a rudimentary grammar:

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

\( x + 10 \) is a sentence for which the following trees can be built:

```
  sum   product   term   id: x
     +
  product   term   int: 10
```

abstract syntax tree

```
  id: x
     +
  int: 10
```

parse tree

leaves are input symbols, other nodes represent grammar phrases,
tree is built during syntax analysis, traversed and modified later

tree nodes can be objects

**object advantage**

- methods automate *divide and conquer* to implement *sem*, *gen*, and *run*
- inheritance simplifies implementation, e.g., for related operators
- class libraries extensible *compiler kit* as a reusable universal back end,
  includes *sem* that can be modified by inheritance and overwriting,
  *sem* generates a persistent execution tree for *run*
Execution

A tree of persistent objects, `run` is a method that partially traverses the tree

**object advantage**
- Objects persistence provides cheap, platform-independent image store
- Methods reuse between compilation (e.g., constant expressions) and runtime
Semantic Analysis

- during a traversal of the parse tree, result types are computed and checked for the nodes

- $sem$ is a method for each class of nodes and augments parse tree with conversion nodes, types are modeled as unique objects, type’s methods generate simplified, persistent runtime tree

**object advantage**

- methods automate *divide and conquer* to implement $sem$
- class libraries allow reuse of types and semantic analysis of expressions
- inheritance permits restricting and extending a library type’s capabilities
Types

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

id: x
+ int: 10
Syntax Analysis

- compiler converts sentence in a language, i.e., a program, into image to be executed
- our compiler-compiler `oops` converts sentence in EBNF, i.e., a grammar, into image
- image is a tree of objects,
  - `sem` checks if the grammar is suitable for parsing,
  - `run` implements a recursive descent parser based on the grammar
- for each phrase, `run` creates goal object of phrase-specific class,
  - goal object receives `shift` and `reduce` messages as phrase is recognized

**object advantage**

- **objects**  parser objects encapsulate lookahead and follow sets,
  - goal objects encapsulate state of phrase recognition,
  - both can be used for automatic error recovery
- **methods**  automate *divide and conquer* for LL(1) checking and parsing,
  - goals avoid the need for parser action syntax within grammar
- **messages**  provide precise semantics for `shift` and `reduce` even in EBNF
Divide and Conquer

- EBNF can be viewed as syntax graphs and represented using classes like `Alt` and `Seq`

![Diagram of syntax graphs](image)

- Successful parsing hinges on determining which route to take through the graph, i.e., on certain lookahead and follow sets being mutually distinct

![Diagram of lookahead and follow sets](image)

- Discover algorithm to compute the sets by inspecting syntax graph building blocks, implement it as methods for classes like `Alt` and `Seq` representing the blocks

- Object approach leads from visual discovery to functioning implementation
Tree Traversal Techniques

- A Visitor object is sent to the root of the tree, which calls it back; Visitor objects must implement a visit method for each node class, the node class must provide access to the node’s children.

- Visitor support can be generated by a tree-building parser such as jjtree fairly difficult to extend or inherit, but permits adding new traversals later.

- Alternatively, each node class implements traversal method; easy to inherit, can be added as a category in Objective C to existing classes.

- Another idea is method selection based on a pattern of node and children classes.

- Jag is our tool to convert these pattern/action statements into Java methods which are conceptually attached to the node classes.

**Object advantage**

- **Methods** Visitor enforces divide and conquer.

- **Inheritance** combined with overloading permits refinement of initially very coarse traversal rules and reuse between projects.
Conclusion

- objects: information hiding, state encapsulation
- methods: develop by *divide and conquer*
- messages: debug by instrumentation
- classes: reuse between projects
- inheritance: reuse within and between projects
- class libraries: many pre-fabricated algorithms and data structures