ECE251: Programming Languages & Translators Fall 2010

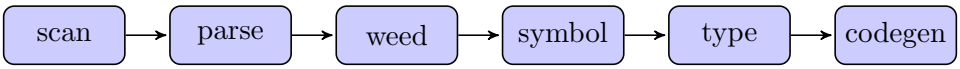
Lecture 33 — December 2, 2010

Patrick Lam *version 1*

Course Summary

We will have a review session next Monday where you can work on questions and ask me questions. This time, I'll just briefly summarize the different topics that we've seen in this course (although I've been doing that all week, in some sense, by talking about domain-specific languages).

Recall this diagram from the beginning of the term.



We have now seen how each of these boxes work. Let me summarize what you have learned about/implemented for each of these boxes. The abstract questions list will express "what you have learned" more directly, in terms of "what you should be able to do".

Bootstrapping. Compiling source-to-source came up repeatedly: key for compiling compilers.

Scanning. You built three lexers, using regular expressions to specify the tokens. Usually, you gave these expressions to ANTLR, but in Lab 2, you specified regular expressions to a custom lexing engine that I wrote. We also saw how regular expressions work underneath the hood: they get compiled into finite automata, either nondeterministic automata or deterministic automata.

Parsing. Each of the labs and part 1 of the project have also included a parsing component. Recall that parsers transform a stream of tokens, from the lexer, into a parse tree. A context-free grammar specifies legal ways of forming parse trees from the grammar.

Specifying a grammar is easy, but we saw several problems that occured with naïvely-specified grammars, mainly related to ambiguity (dangling-else, expressions, left-recursion, and common prefixes). Some of these problems show up as shift/reduce conflicts in some parser generators.

You also manually implemented a recursive-descent parser in Lab 2. There are many other technologies for implementing parsers. Usually you use a parser generator, e.g. ANTLR or others.

Parse Trees vs. ASTs. Recall that parse trees have lots of cruft, which makes it difficult to understand the structure of the program. We therefore convert the parse tree into an Abstract Syntax Tree. (ANTLR actually provides facilities for automatically generating ASTs, but I thought that having you manually code them in the project would be a more educational experience).

The AST encodes just what you need to further process the program (or, in our case, a bit less: we forgot to include line numbers, hindering error reporting.) We have been using Visitors over the Abstract Syntax Tree to carry out symbol table manipulations, type checking, and code generation.

Symbol tables. Symbol tables link names to types (and possibly addresses). You implemented a really simple symbol table in Lab 1, where you had a global scope only. The project requires you to implement a more sophisticated symbol table with static scoping of names. You used a Java `Map` for each scope, and link them with a `Stack`. There's also a `Map` which lets you look up the symbol table for a scope, given its AST node.

Type checking. The project also includes a type checker, which verifies that operators get appropriate types. This type checker is fairly standard; it's missing type coercion. The other big idea here is type inference, where the compiler automatically computes the types of variables.

Code generation. Compilers are most useful when they produce some output, and code generators do that. Source-to-source translators are the most common form of code generation, and that's on the project. It's also possible to generate machine code, or bytecode, from a code generator.

Interpretation. Alternatively, you can go through all of the trouble of implementing a compiler, but instead execute the code right away; then you have an interpreter. You implemented an interpreter for the Abstract Datatype Language in Lab 1. This interpreter executed the program when it encountered parser actions. Interpretation can also use a Visitor in an AST traversal.

One can also emit bytecode and then execute the bytecode—the most common way of interpreting programs. Executing the bytecode could be direct execution, or through a just-in-time compiler.

Interpretation requires a store, which associates variables with values. Stores and symbol tables are similar: the symbol table keeps a type for each variable, while the store keeps a value for each variable. The store also has dynamically-allocated heap objects, which aren't in the symbol table.

Domain-Specific Languages. The labs and the project have all investigated programming languages suitable to some particular domain (not general-purpose programming). Lab 1 was for teaching students how to program; Lab 2 was an SQL parser; SQL is a domain-specific language for interacting with databases; and the project is a domain-specific language for writing CGI scripts.

We also saw a number of DSLs throughout the course, including today. If you have a programming problem to solve which requires a lot of boilerplate, consider writing a domain-specific language.

Other Topics: Functional and logic programming. I briefly introduced two other programming paradigms, functional and logic programming, and talked about scripting languages. Functional programming allows you to pass functions as parameters to other functions, as well as receive functions as return values. These higher-order functions allow you to express certain code idioms much more concisely than in other paradigms. Logic programming allows programmers to specify clauses and query for deductions which follow from these clauses. Scripting languages coordinate actions, which are typically written in some other language.