Recap: Typical Compiler Structure

This Lecture (1)

In this lecture, we will walk through a rudimentary but complete compiler implemented in Haskell for the *Trivial eXpression Language* (TXL):

- concrete illustration of different compiler phases
- “mental scaffolding”
- brush up your Haskell knowledge

This Lecture (2)

- TXL by example.
- Lexical aspects.
- Context Free Grammar.
- Static semantics.
**TXL into C compiler**

Scenario:
- We wish to develop a compiler for TXL.
- To save ourselves some effort, we are going to compile TXL into C, and then use an existing C compiler (GCC) to translate into executable machine code.

**Informal TXL Syntax and Semantics**

Some examples of TXL programs, *concrete syntax*, and their intended meaning, *semantics*:

- `1 + 3`
  
  Semantics: 4

- `1 + (3 * (2 + 2))`
  
  Semantics: 13

- `let x = 3 * 7 in x + 3`
  
  Semantics: 24

In TXL, all binary operators have the same precedence and are left associative. See the context-free grammar later.

**Informal TXL Syntax and Semantics**

- `let x = 3 * 7 in let x = x * 3 in x - 21`
  
  Semantics: ???

Some possibilities:
- Disallow re-definition of entities already in scope.
- Allow nested scopes, decide how to disambiguate; e.g., closest containing scope.
- Recursive definitions or not? I.e., is the defined entity in scope in its own definition?

We opt for nesting, closest containing scope, no recursion. Semantics: 42

**Informal TXL Syntax and Semantics**

- `let x = 3 in y + x`
  
  Semantics: ???

Some possibilities:
- Insist all variables be defined. The program can then be *statically* rejected as *meaningless*.
- Catch use of undefined variables *dynamically*, making the meaning of the program *undefined*.
- Assume some default value, like 0, for variables that are not explicitly defined.
TXL Really Is Trivial (1)

- TXL is indeed a trivial language.
- The meaning of a TXL “program” is just a number that could be computed *statically*, at compile time.
- Thus not *really* necessary to generate a program!
- Or we could generate a very simple program of the form
  
  ```
  print n;
  ```

Rough Concrete Compiler Design (1)

Each “box” of the compiler structure diagram corresponds to a function:

- `scanner :: [Char] -> [Token]`
- `parser :: [Token] -> AST`
- `checker :: AST -> ChkdAST`
- `codegen :: ChkdAST -> Code`

The compiler can thus be constructed by *composing* these functions in sequence. For example, scanning followed by parsing:

```
scanAndParse :: [Char] -> AST
scanAndParse = parser . scanner
```
What Happened to the Errors? (1)

Recall:

Any of the front-end phases could encounter errors. Thus, each function should return either the result of transformation or an error indication.

The TXL2C Compiler Design

However, we are going to cut some corners:

- For scanning and parsing errors, simply stop and print error message. I.e., these functions become partial, but not reflected in their type.
- TXL2C is so trivial, that there is no need to change or annotate the AST: we will use a data type Exp throughout.

What Happened to the Errors? (2)

data Either a b = Left a | Right b

scanner :: [Char] -> Either [Token] [ErrMsg]
parser :: [Token] -> Either AST [ErrMsg]
checker :: AST -> Either CheckedAST [ErrMsg]
codegen :: CheckedAST -> Code

Informal Lexical Aspects of TXL

If doing things properly, we would specify the lexical syntax of the basic language symbols, the tokens (like integers, identifiers, operators, keywords), using e.g. regular expressions.

Some notes about white space:

- The only significance of white space is to separate tokens.
- Comments starts with ! and runs to the end of the line.
- Comments are treated as white space.
TXL Compilation: Lexical Analysis

A small TXL program (the source code):

\[
\text{let } x = 3 \times 7 \text{ in} \\
\text{let } x = x \times 3 \text{ in} \\
x - 21
\]

After lexical analysis (type: \text{[Token]}):

\[
[\text{T}_{\text{Let}}, \text{T}_{\text{Id}} \text{ "x"}, \text{T}_{\text{Equal}}, \text{T}_{\text{Int}} 3, \text{T}_{\text{Times}}, \text{T}_{\text{Int}} 7, \text{T}_{\text{In}}, \text{T}_{\text{Let}}, \text{T}_{\text{Id}} \text{ "x"}, \text{T}_{\text{Equal}}, \text{T}_{\text{Id}} \text{ "x"}, \text{T}_{\text{Times}}, \text{T}_{\text{Int}} 3, \text{T}_{\text{In}}, \text{T}_{\text{Id}} \text{ "x"}, \text{T}_{\text{Minus}}, \text{T}_{\text{Int}} 21]
\]

TXL Compilation: Syntactic Analysis

After parsing, AST drawn as a tree:

Exercise (home): Draw the derivation tree for

\[
\text{let } x = 3 \times 7 \text{ in let } x = x \times 3 \text{ in } x - 21
\]

Compare with the Abstract Syntax Tree (AST) on next slide!

Context Free Grammar for TXL

\[
\begin{align*}
\text{TXLProgram} & \rightarrow \text{Exp} \\
\text{Exp} & \rightarrow \text{Exp} + \text{PrimExp} \\
& \quad | \text{Exp} - \text{PrimExp} \\
& \quad | \text{Exp} \times \text{PrimExp} \\
& \quad | \text{Exp} / \text{PrimExp} \\
& \quad | \text{PrimExp} \\
\text{PrimExp} & \rightarrow \text{IntegerLiteral} \\
& \quad | \text{Identifier} \\
& \quad | ( \text{Exp} ) \\
& \quad | \text{let } \text{Identifier} = \text{Exp} \text{ in } \text{Exp}
\end{align*}
\]

After parsing, AST (type: \text{Exp}):

\[
\begin{align*}
\text{Let } \text{"x"} \\
& \quad (\text{BinOpApp } \text{Times} \\
& \quad \quad (\text{LitInt } 3) \\
& \quad \quad (\text{LitInt } 7)) \\
(\text{Let } \text{"x"} \\
& \quad (\text{BinOpApp } \text{Times} \\
& \quad \quad (\text{Var } \text{"x"}) \\
& \quad \quad (\text{LitInt } 3)) \\
& \quad (\text{BinOpApp } \text{Minus} \\
& \quad \quad (\text{Var } \text{"x"}) \\
& \quad \quad (\text{LitInt } 21)))
\end{align*}
\]
Notes on the TXL Parser

- Predictive, recursive-descent parser (see G52MAL).
- Parsing function for each non-terminal has type:
  \[ \text{[Token]} \rightarrow (\text{Exp}, \text{[Token]}) \]
- As recursive descent parsers cannot handle left-recursion, grammar transformed first to eliminate left-recursion, then systematically transformed into a parser.

Static Semantics

- TXL has only a single type, integer, so typing is trivial.
- Normal nested scope.
- Only defined variables may be used. For example, the following is invalid:
  \[ \text{let } x = 3 \text{ in } y + x \]

Compiling the TXL \texttt{let-expression} (1)

- The \texttt{let}-construct is an \textit{expression}, and standard rules for nested scope are supposed to be respected.
- This is really the only complication when compiling TXL into C, since C does not have \texttt{let}-expressions or anything similar.
- In particular, it is \textit{not} possible to directly replace \texttt{let} by assignment!

Compiling the TXL \texttt{let-expression} (2)

To see the last point, consider
\[
\text{let } x = 23 \text{ in } (\text{let } x = x * 3 \text{ in } x - 50) + x
\]

The meaning should again be 42.
However, assuming \texttt{x} is an \texttt{int} variable, the value of the following C expression is 88:
\[
(x = 23, (x = x * 3, x - 50)) + x
\]

(Final value of \texttt{x} is 69, 69 - 50 = 19, 19 + 69 = 88.)