A compiler for compiler-compilers

Leveraging language support to improve syntax-directed translation

The problem

According to the authors of the “dragon book”, the compiler is a dragon, a huge and complicated monster. Compiler researchers have been working on slaying that monster and replace it with the white knight of syntax-directed translation who is supposed to be just as powerful.
But in the industry, the dragon is still the ruler. No matter what language – the wide-spread (and, usually, most powerful) compilers are still recursive-decent multi-pass compilers, barely making any use of advances in the field of compiler research since the 60s.
In my opinion, the problem is two-fold:

1. Compiler generators are too restrictive
2. Compiler generators cannot be sufficiently debugged
   - Bison, for example, **only adds tracing** when debugging is enabled. It does not allow for **breakpoints in a meaningful context**. When an exception or unrecoverable interrupt arises, we will look at an **unreadable stacktrace**.

Both these problems have a common root: Insufficient language support caused by the non-transparent (opaque) two-step compilation:
Bison does not know the exact meaning of it's input and thus cannot allow that code to be debugged properly. Instead, it generates a new file that contains **meaningful but unreadable source code** of some language.

Proposed solution

The solution is: Add a new language construct “grammar” for syntax-directed translation schemes to the compiler of the language that is used to write a compiler-compiler. The result is not an actual compiler-compiler but a construct that can be used to easily write a compiler.
The grammar definition contains: An attributed grammar definition with code to be executed upon successful parsing of a non-terminal. The code execution can be delayed until after the entire file has been parsed, so a code generator would not generate code before the parser verified correctness of the code.
The programmer can then write her own universal scanner and parser, and use the grammar definition to execute code for each symbol in the language. The code in the grammar definition can access the scanner, the parser and everything else, depending on one's own implementation.
This way, we strike a balance between the terrible dragon and the noble knight:
We still have to write the universal parser and scanner. But we only have to do so once and can then re-use it, given it is universal enough. Also, the compiler can generate debugging information to allow for breakpointing and meaningful stacktraces.
When compared to the average recursive-decent parser, we do not have to describe how the grammar should be parsed in exact detail. It will use a uniform definition to parse all symbols and give us the resulting AST to work with. We can also make calls to the scanner and parser to add exceptions to the parsing process to accommodate for more complex constructs, such as the C array initializer, or if-statements with dangling else.

Goal

I will change Javac to read Grammar statements and generate classes that I can use in my universal parser.

Implementation in detail

From grammar to class

The compiler will generate a class for each grammar statement. It will also generate classes for each Non-terminal and each rule within the grammar. Each Non-terminal contains a list of the symbols that have been parsed, according to it's rules. The code, attached to each rule inside the grammar expression is added to the class of the corresponding Non-terminal. That gives it access to the list of parsed symbols at that stage.

Inlined non-terminals

Since I am not planning on implementing Grammar-rewriting algorithms (yet), the existing Grammar definition will be changed to accommodate for “inline non-terminals”. Those non-terminals were added to allow for correct parsing and might not actually convey any meaning. Classes for those symbols will not be generated, and all the symbols that it is made of, will be directly appended to the class of the symbol that uses them, or rather, the first non-inlined ascendant.

Accumulators

Consider the following grammar definition for `X` that defines a list of `V`s:

```
X : V X | lambda;
```

Our parser would most certainly want to generate a list, whenever this construct appears. For that to happen, one can use the new **accumulator operator (*)** that will not produce a single non-terminal but instead create a list of symbols to which all `V`s will be added.

```
A : B *X C;
D : B X C;
X : V *X | lambda;
```

In the above grammar, X represents a list of symbols. During parsing of A, the list is then used to parse X itself where another accumulating X is found causing the grammar parser to use the same list to continue accumulating X. We could also add an asterisk in front of V which will have the effect that the list will not contain a set of V's but instead a set of all symbols that V is made of. When looking at D, we will find the 3 symbols as-is, but when looking at X, inside D, we will find a single V and a list of more V's. To make these lists less constrained, we could also give names to accumulators, so that multiple accumulators can be used in a single symbol.

Operator precedence

If our grammar definition contains several different operators that interact with the same grammar symbols, there is usually the problem of precedence: The linearly derived parse tree might not correspond to the correct expression tree. To solve that problem, we can employ *Accumulators* to first read in a list of symbols that define an expression and then use a precedence lookup table (or simply the Token name information) to convert the expression into **Reverse Polish notation** which is easy to evaluate.
Consider the following grammar definition:

```
1: VarDecl :   “int” id “=” *Expr    { System.out.format(“int %s = %d”, symbol(1), calcExpr(symbol(3)))); }
2:             ;
3: Expr : UnaryExpr *OpExpr;
4: UnaryExpr :  “(“ *Expr “)”        { trace(“Expr in parentheses: %s“, symbol(1)); }
5:             | num                 { trace(“Number: %d”, symbol(0).getAttributeInt()); }
6:             ;
7: OpExpr : “+” *Expr | “*” *Expr;
```

After parsing, the last element of the symbol list of the *VarDecl* object (Called `*Expr` inside the rule and `symbol(3)` inside the code) is a list of all symbols that the expression is made of. That list will contain a mixture of *UnaryExpr* and operator terminal objects, which can be easily converted to Reverse Polish notation using the Shunting-yard algorithm. Similarly, in line 4, the enclosed expression (Called `*Expr` inside the rule and `symbol(1)` inside the code) is also such a list. Note that expression evaluation is not actually a feature of my changes to the compiler, but is made easier due to it's implementation.

Current status

I already wrote a parser to read in the standard grammar, as well as a universal LL(1) parsing algorithm. The next step will be to adjust the grammar definition and implement it in javac's *Parser* and use *Lower* to convert the statement into a class. Finally, I will use this new construct to write the CMM compiler for lab4 to demonstrate it's usefulness and correctness.