Types & Programming Languages
Notes on the SEL (Simple Expression Language) Typechecker

Location

The SEL typechecker can be obtained from `http://www.dcs.gla.ac.uk/~simon/teaching/tpl`

Grammar Specification

The file `sel.grm` specifies the grammar of SEL in a form suitable for SableCC.

- The declaration `Package sel` is necessary so that the generated Java code is part of the right package.
- The `Helpers` section defines names for characters and sets of characters.
- The `Tokens` section defines the syntax of tokens. All tokens must be defined in this section and given names; it is not possible for tokens to be specified directly as strings when the productions are defined. We define Java-style single-line comments, introduced by `//`.
- The `Ignored Tokens` section specifies that blanks and comments are discarded by the lexer; they do not appear in syntax trees.
- The `Productions` section defines the phrase-level grammar. The first non-terminal to be defined is the start symbol. Implicitly a non-terminal `start` is also defined — in this case its definition is `start = expression` — and a class `Start` is generated.
- Each production must be given a label (actually this is only necessary when there are two or more productions for a given non-terminal) and these labels are used to generate the names of classes. The initial letters are capitalized and underscores are replaced by capitalization of the following letter. From the definition of `expression` we will get classes `PExpression`, `ATermExpression`, `APlusExpression` and so on.
- If a production contains two or more occurrences of a particular non-terminal then each occurrence must be labelled, for example `[left]:` and `[right]:` in production `term`. These labels, with underscores before and after, form the names of instance variables in the class definitions (look in the file `node/APlusExpression.java` to see this). If there is only one reference to a non-terminal then the label can be omitted and the name of the corresponding instance variable is derived from the name of the non-terminal instead (look in the file `node/ATermExpression.java` to see this).
- In order to avoid ambiguity of the grammar we have defined both `expression` and `term`. An operator must be applied to `term`s and an `expression` is converted into a `term` by enclosing it in brackets. This means that instead of `1+2+3` in an input file we must write either `1+(2+3)` or `(1+2)+3`. This introduces some complexity into the grammar which is reflected in the generated classes and which we must also take account of when defining the tree traversal methods.

Syntax Tree Classes

Everything in the directory `sel/node` is generated by SableCC.

- There is a large amount of infrastructure associated with the syntax tree nodes, allowing the structure of a syntax tree to be modified after the tree has been constructed by the parser. Some of this infrastructure is inherited from the `Node` class. We are not concerned with this aspect of the syntax tree classes; we can focus on the classes corresponding to non-terminals (`PExpression` etc.) and the classes corresonding to productions (`ATermExpression` etc.).

- The classes generated by SableCC represent complete parse trees rather than abstract syntax trees, because they store all the tokens which were present in the original input. For example, class `APlusExpression` contains an instance variable `TPlus _plus_`. The only use we need to make of token classes such as `TPlus` is that they contain file positions, which we need when producing error messages. If we wanted to know the line number (in the input file) of this `+` operator, we would use `_plus_.getLine()`. Every token has an instance variable `String text` which holds the characters from the input file which caused the token to be generated. In the case of integer literals this is where the actual value is stored, but for typechecking we are never interested in the value itself and therefore there is no need to inspect this variable.

Lexer and Parser

Everything in the directories `sel/lexer` and `sel/parser` is generated by SableCC. The only thing we need to know is how to invoke the parser on a desired input file, and the code to do this is included in `Main.java` (see below).

Representation of Types

The directory `sel/types` contains definitions of classes representing the types `int` and `bool`. These are not generated by SableCC. For other languages it might be possible to use classes from the syntax tree representation to represent the types calculated during typechecking — it depends on whether or not the types which the typechecker manipulates have corresponding syntax in the language. In the case of SEL, the syntax does not contain any representation of types.

The abstract class `Type` specifies a method which returns the name of a type (used to display the type of an expression after typechecking) and a method which tests for equivalence with another type. Type equivalence in SEL is simple — just equality of the names.

Tree Traversal

Everything in the directory `sel/analysis` is generated by SableCC. We are only concerned with `DepthFirstAdapter`. This class defines, for each kind of syntax tree node, three methods: `in`, `out`, and `case`. The `case` method is the one which is called (via an `apply` method in a class such as `APlusExpression`) as part of the Visitor pattern. The `case` method calls `in`, then visits the child nodes by calling the `apply` methods of the instance variables, then calls `out`.

To program a tree traversal, we define a class `Visitor` which extends `DepthFirstAdapter`, and create an object `v` of class `Visitor`. If `n` is an object of some class extending `Node` (usually we would work with an object `n` of class `Start`, representing the whole syntax tree) then calling `n.apply(v)` launches a tree traversal.

If `Visitor` does not over-ride any of the methods of `DepthFirstAdapter` then the tree traversal will pass through every node, but not do anything. Usually we do want to do something, so we over-ride the `in` or `out` methods. When typechecking we really want each `out` method to return the type of the node being checked, but this is not possible directly because the result types are `void`.

Instead we make use of the `Hashtables` `in` and `out` which are defined by the class `AnalysisAdapter`. The details are described below.

The standard tree traversal visits every node, including all of the tokens (because we have a complete syntax tree). If we don't over-ride `in` or `out` for tokens (and usually we won't) then visiting the tokens has no effect because the `defaultIn` and `defaultOut` methods are empty. If we cared about efficiency, we could over-ride the `case` methods so that they don't cause visits to uninteresting tokens.

Typechecker

The file `checker/Checker.java` defines the typechecker. It is not generated automatically — I wish it could be! I have a final year project proposal to develop a tool which will generate this file from a suitable specification of typing rules.

We define a class `Checker` which extends `DepthFirstAdapter`. We over-ride the `out` methods, and use the `out` hashtable (via the methods `setOut` and `getOut`) to store the type of each expression.

The best place to start is the method `outAPlusExpression`.

1. A `APlusExpression` contains two `Terms`, called `_left_` and `_right_`. When `outAPlusExpression` is called, the tree traversal has already visited `_left_` and `_right_` which means that these terms have been typechecked and their types have been stored in the hashtable `out`. We extract these types and assign them to `leftType` and `rightType`.

2. The first `if` statement checks that the left argument of the addition has the correct type, i.e. `int`. First, however, we check that the left argument has a type at all: if there is a type error within that term then the typechecker is unable to return its type and the value stored in the hashtable will be `null`. In that case there is no need to generate an error message because an error message has already been generated while typechecking the left argument. But if the left argument has a type, then we generate an error message (see below for details of the error table and error classes) if that type is not `int`.

3. The second `if` statement goes through the same process for the right argument. Note that even if an error is found in the left argument, we still check the right argument. The aim is to find as many type errors as possible.

4. The third `if` statement deals with the case in which both arguments have type `int`, so that we can record the type of the whole expression, again `int`, in the hashtable.

5. There are other ways of organising the code in this method, and the specific implementation here has been chosen for clarity. If you can find a reformulation which you prefer for reasons of efficiency or aesthetics, then by all means adopt a different coding style.

The method `outACondExpression` is a little different because of the need to check that both branches of the conditional have the same type. If this is true, then either one of them can be used as the type of the expression.

The methods `outABoolLitTerm` and `outAIntLitTerm` deal with the cases in which we immediately know the type of a literal value.

The methods `outAExpTerm`, `outATermExpression` and `outStart` simply pass calculated types upwards without any further checking or processing. They correspond to nodes which only exist in the syntax tree because of the way that the grammar is defined; they have no counterpart in the "pure" ASTs which we looked at in lectures.

Error Reporting

The directory `sel/errors` contains definitions of `ErrorTable` (a wrapper around a `Vector` which is used during typechecking to accumulate a list of errors), `ErrorTableEntry` (a line number and a specific error), `Error` (an abstract class of errors) and a class for each kind of error, for example `EqLeftError` for when the left argument of an equality test is not an integer.

Top-level Driver

The file `sel/Main.java` defines a class `Main` containing the top-level method `main` which calls `check`. The `check` method creates a lexer object, creates a parser object from the lexer object, calls the parser to construct a syntax tree from the input file, creates a typechecker object, calls the typechecker on the syntax tree, extracts the final error table from the typechecker, displays any error messages, and if there are no errors, retrieves the type of the expression in the input file and displays it.