Generating Software from Specifications WS 2013/14 - Assignment 4

Published: Nov 6, 2013 -- Turn in until Nov 13 at 12h

What to turn in: see Assignment 1

Exercise 9 (Parser generation)

This task exercises to generate parsers using Eli. Keep Eli's manual on "Syntactic Analysis" at hand to lookup open questions. The directory `blatt4/calendar` contains three specification files named `parsing1.fw`, `parsing2.fw`, and `parsing3.fw`. Download them into a fresh working directory. Then perform the following steps for each of them:

1. Read and understand the specification of the concrete syntax. Write a correct input file into the provided macro.
2. Execute Eli's command to check whether the concrete syntax belongs to the class of context-free grammars, which the parser generator can handle, i.e. LALR(1):

   ```
   parsing1.fw:parsable>
   ```

   If the Parser generator fails, it complains "not LALR(1)". Furthermore, it presents a set of derivations from the start symbol down to the point where a decision conflict occurs, (shift-reduce conflict, or reduce-reduce conflict).
3. That information may help you to find out why the parser generator can not process the grammar. Find that reason and describe it in terms of properties of the grammar.
   The third specification file contains EBNF constructs. They are expanded before the parser generator processes it. It will be helpful to look at the concrete syntax given to the parser generator, using the command

   ```
   parsing3.fw:consyntax > .
   ```

4. Change the grammar, such that a parser can be generated. You may need to specify a superset of the language of the original concrete syntax. Generate an executable processor and let it process your input file.

Exercise 10 (Language design and tree construction)

In this excercise you will design a concrete syntax for a little language and specify a complete structuring phase. Elements of the language are sequences of typed declarations. Use the specification file `Signatures.fw` of the directory `blatt4/calendar`. It contains further descriptions of the task and empty macros to fill the specification in.

Apply the design strategy of the course PLaC (slides 3.4aa, ...), and use the patterns of the course GPS (slides 3.9, ...) to specify precedences of operators.

Derive the abstract syntax using the Eli command

```
Signatures.fw:absyntax > .
```

Eliminate unnecessary chain productions from it by specifying mapping rules.

Exercise 11 (Simple computations and checks in tree contexts)

In this exercise you will specify simple computations in tree contexts in order to check semantic conditions.

Download the file `Calendar4.fw` from the directory `blatt4/calendar`; it corresponds to the solution of Exercise 8 of Assignment 3. Some specifications fragments are added, to help you solving this exercise. Perform the following steps:

1. Before you begin to augment the abstract syntax by computations, check whether you can simplify it. Derive the abstract syntax. You will find at least two targets for simplification: (1) The two notations of days of the year can be unified by rule mapping. (2) A distinction between `GeneralPattern` and `SimplePattern` is redundant for the abstract syntax; it can be unified by symbol mapping. (We withdrew the modification of

Exercise 8.4 to enable this simplification.) Insert your mapping specifications into the prepared macro. Additionally, for the rule mapping you need to write the resulting unifying abstract rule into a `.lido` macro, e.g.

```
RULE: Date ::= DayNum MonNum END;
```

Derive the abstract syntax again and check which productions have vanished.

2. Up to step (6) develop a computation which checks that in every time span the first `Time` token denotes a time that is earlier than the one denoted by the second `Time` token. First insert in your test cases both, a correct and an erroneous time span.
3. Derive the abstract syntax into your working directory. Find the rule which specifies the context of a time span, and copy it into the macro `CalendarChecks2.lido` of your specification file. Eliminate the rule name, such that the rule begins `RULE:`, and insert the keyword `COMPUTE` before `END;`.
4. Find out, what the meaning of the values is which the coding function of `Time` tokens provide to be accessed in the rule context, and how it can be used to check for correct time spans.
5. In the specification language LIDO, such a check with an error message has the following form:

   ```
   IF (condition,
       message (ERROR, "message text", 0, COORDREF));
   ```

   Lookup in Eil's manual "LIDO Reference Manual" section "Predefined Entities" which predefined C macros you may use to formulate the condition for that message. The codes of the `Time` tokens are distinguished by `Time[1]` and `Time[2]`.
6. Insert the computation into the rule context and test it.
7. Implement a check whether days in a year are correct with respect to the number of days in every month. The C module in the specification file provides suitable functions for that purpose. Be aware that the the values of the `Integer` token can not be accessed in the context where the nonterminals `DayNum` and `MonNum` form a day in a year. Hence, you have to introduce an attribute, which propagates the value of the terminal one context up.
8. Implement checks for spans of days in a week, and for spans of days in a year. Use the provided functions for that purpose.
9. Test all your checks with correct and erroneous test cases.

Exercise 12 (Homework, repeat: Computations in tree contexts)

Computations in tree contexts are the means to perform the semantic analysis tasks and the transforamtion tasks. They are specified in terms of attribute grammars. Repeat

- the fundamental notions of attribute grammars: PLaC-4.1 to 4.9;
- attribute grammar specification in Eli: PLaC-4.19 to 4.22, the Eli documentation "Computations in Trees" (up to section "Symbol Computation").

Do not forget to take notes on what you did, what you learned, and which problems you encountered, and turn them in.