

Grammars and Parsing, second week

Hayo Thielecke

17-18 October 2005

This is the material from the slides in a more printer-friendly layout.

Contents

1 Overview	1
2 Recursive methods from grammar rules	2
3 From grammars to methods	2
4 Predictive parser	2
5 The lookahead and match methods	2
6 Simplified view of parsing	2
7 Parsing with lookahead	3
8 Translation to code (only recognizing)	3
9 From grammars to Java classes	3
10 Classes for the parse tree	4
11 Parsing while building the parse tree	4
12 More general parsing	4
13 Parser generators	4
14 Parser generator notation	5
15 If you need to do parsing	5

1 Overview

So far:

Grammars, their language and parse trees.

Grammars \longrightarrow class hierarchy implementing parse trees.

Grammars \longrightarrow recursive methods for processing syntax.

This week: put together the pieces:

build simple parsers that generate parse trees,
translation to another syntax.

2 Recursive methods from grammar rules

From grammars to mutually recursive methods:

- For each non-terminal A there is a static method A . The method body is a switch statement that chooses a rule for A .
- For each rule $A \rightarrow X_1 \dots X_n$, there is a branch in the switch statement. There are method calls for all the non-terminals among X_1, \dots, X_n .

Each grammar gives us some recursive methods.

For each derivation in the language, we have a sequence of method calls.

3 From grammars to methods

Non-terminal \rightarrow method

Production \rightarrow branch of a switch

Occurrence of a non-terminal
on the right-hand side of a production \rightarrow method call

Occurrence of a terminal
on the right-hand side of a production \rightarrow I/O of the symbol

4 Predictive parser

A *predictive* parser can be constructed from grammar rules.

The parser is allowed to “look ahead” in the input; based on what it sees there, it then makes predictions.

Canonical example: matching brackets.

If the parser sees a `[` as the next input symbol, it “predicts” that the input contains something in brackets.

More technically: `switch` on the lookahead; `[` labels one case.

5 The lookahead and match methods

A predictive parser relies on two methods for accessing the input string:

`char lookahead()` returns the next symbol in the input, without removing it.

`void match(char c)` compares the next symbol in the output to `c`. If they are the same, the symbol is removed from the input. Otherwise, the parsing is stopped with an error.

6 Simplified view of parsing

In the real world, `lookahead` and `match` are calls to the lexical analyzer, and they return tokens, not characters.

There are efficiency issues of buffering the input file, etc.

We ignore all that to keep the parser as simple as possible.
(Only single-letter keywords.)

But this setting is sufficient to demonstrate the principles.

7 Parsing with lookahead

Parsing with lookahead is easy if every rule for a given non-terminal starts with a different terminal symbol:

$$\begin{aligned} S &\rightarrow [S] \\ S &\rightarrow + \end{aligned}$$

Idea: suppose you are trying to parse an S . Look at the first symbol in the input:
if it is a $[$, use the first rule;
if it is a $+$, use the second rule.

8 Translation to code (only recognizing)

```
void parseS()
{
    switch(lookahead()) { // what is in the input?
        case '[':         // I have seen a [
            match('[');   // remove the [
            parseS();     // now parse what is inside [...]
            match(']');   // make sure there is a ]
            break;        // done in this case
        case '+':
            match('+');
            break;
        default: error();
    }
}
```

9 From grammars to Java classes

We translate a grammar to some mutually recursive Java classes:

- For each non-terminal A there is an abstract class A
- For each rule $A \rightarrow X_1 \dots X_n$, there is a concrete subclass of A . It has fields for all non-terminals among X_1, \dots, X_n .
- The `toString` method of A concatenates the `toString` of all the X_i :
if X_i is a terminal symbol, it is already a string;
if X_i is a non-terminal, its `toString` method is called.

Thus each grammar gives us a class hierarchy.

10 Classes for the parse tree

```
abstract class S { }

class Bracket extends S
{
    private S inBrackets;

    Brackets(S s) { inBrackets = s; }
}

class Plus extends S { }
```

11 Parsing while building the parse tree

```
S parseS()      // return type is abstract class S
{
    switch(lookahead()) {
        case '[':
            {
                S treeS;
                match('[');
                treeS = parseS(); // remember tree inside [...]
                match(']');
                return new Bracket(treeS);
                // call the constructor of the class for this rule
            }
        ...
    }
}
```

12 More general parsing

Parsing with lookahead is easy if every rule for a given non-terminal starts with a different terminal symbol.

But what if not? The right-hand-side could instead start with a non-terminal, or be an empty string.

More general methods: FIRST and FOLLOW construction, covered in Compilers and Languages.

13 Parser generators

The lookahead is tedious to compute. Parser generators do it automatically.

Some parser generators, notably yacc and its descendants, use a more complicated parsing method that does not only use lookahead (LALR, a form of bottom-up parsing).

14 Parser generator notation

Parser generators use some ASCII syntax rather than symbols like \rightarrow .

ANTLR uses `:` instead of \rightarrow . There is one rule for each non-terminal, with alternatives indicated by `|`. The rule is terminated with a semicolon. Yacc/bison is similar.

ANTLR can construct the parse tree.

Typically, one attaches parsing actions to each production that tell the parser what to do.

15 If you need to do parsing

If you have a simple language to parse (rules start with different terminals), you can write a predictive parser by hand.

If the grammar is more complex, you could use a parser generator:

You give it the grammar;

you need to add parsing actions,

or walk the parse tree.

ANTLR uses tree grammars for that, SableCC a Visitor pattern.