Top Down Parsing

Building a Parser

As with tokenizers, there are essentially two ways to build a parser:

- Write it by hand.
- Use a parser generator (CUP, Yacc, Bison).

If your grammar is big, likely to change, it is better to use a parser generator.

Tasks of the Parser

Determine the structure of the sequence of tokens that is produced by the tokenizer, and produce some representation that can be further processed.

Decide if the sequence of tokens is correct according to the grammar.

If yes, determine the value of the attribute. (It can be a tree representation, or a value.)

If not, give an error message that is as useful as possible.

Optional: Try to recover after the error.

Writing a Parser by Hand

For grammars that are not complicated, it is possible to write a parser by hand. In most cases, one needs to make changes in the grammar.

It turns out convenient to consider grammar rules where the right hand side is a regular expression.

The resulting parser is usually called recursive descent parser, because it recursively descends from the start symbol to the terminal symbols.

Example

We will construct a parser for the following context-free grammar

$$
\begin{array}{lcllll}
E & \to & E + E_1 & | & E - E_1 & | \quad E_1 \\
E_1 & \to & E_1 \times E_2 & | & E_1 \,/\, E_2 & | \quad E_2 \\
E_2 & \to & -E_2 & | & E_3 & \\
E_3 & \to & \textbf{int} & | & \textbf{double} & | \quad \textbf{string} \quad | \\
 & & \textbf{string}\,(\,A\,) & | & (\,E\,) & \\
A & \to & E & | & A\,,\ E &
\end{array}
$$

$$V = \{E, E_1, E_2, E_3, A\},$$

$$\Sigma = \{\ +,\ -,\ \times,\ /,\ \textbf{int},\ \textbf{double},\ \textbf{string},\ (,\ ),\ ',',\ \}.$$

Start symbol is E.

Example

Possible words are:

$$f(a, b, c)$$
$$f(a + b, f(a + b), a + b \times c),$$
$$(1 + 2) \times (2.0 + 3)$$

What is the purpose of the different non-terminals E, E_1, E_2, E_3?

What is the purpose of A?

We assume that we want to construct an AST, which will have type **tree**.

Implementing a Top Down Parser

In order to obtain a top down parser, we write a parsing function for every non-terminal symbol:

parseE, **parseE**$_1$, **parseE**$_2$, **parseE**$_3$, and **parseA**.

The functions must have access to the tokenizer. We come back to this later.

They must return the attribute of their non-terminal symbol.

parseE$_2$

Function parseE$_2$ is easy to implement:

if the current token is '-' **then**
    move to the next token
    $a =$ parseE$_2$(input)
    return $'-'(a)$
else
    return parseE$_2$(input)

There are two possible rules for E_2. If the current token is '-', we know it is the first rule. Otherwise, it must be the second.

$'-'(a)$ is a tree whose top node is $'-'$.

Problems with parseE

When writing parseE, we run into problems: There are three rules to choose from. Two start with an E, and one starts with E_1.

At the beginning, we don't know if we must call parseE or parseE$_1$.

We could try to read ahead and see if there is a '+' or '-' further in the input. This will not work. For example, in `a * -b`, the minus sign is unary, and in `a * (b-c)`, it's in parentheses. We still have to apply rule $E \to E_1$.

We could try to look at the first token with which a word generated from E or E_1 can start. Unfortunately both can start with $\{ -, \mathbf{int}, \mathbf{double}, \mathbf{string}, ' (' \}$, so that won't help.

Function parseE$_3$

Function parseE$_3$ has a similar problem, but not so bad. There are two rules that start with **string**: $E_3 \rightarrow$ **string** and
$E_3 \rightarrow$ **string**(A).

In this case, one can postpone the decision until after the **string** :

if the current token is **string then**
    $s =$ the attribute of the current token, move to the next token
    if the current token is '(' **then**
        move to the next token
        $a =$ parseA(input)
        if the current token is not ')' **then** syntax error: expected ')'
        move to the next token
        return $s(a)$
    else
        return s

Rewrite the Grammar

Symbol E rewrites to the following words

$$E_1, \quad E_1 + E_1, \quad E_1 - E_2, \quad E_1 + E_1 + E_1, \ldots, E_1 - E_1 - E_1.$$

One can write a regular expression for all words obtained by the rules $E \to E + E_1, \quad E \to E - E_1, \quad E \to E_1$.

$$E_1 \, (\, (\,'+' \mid '-'\,) E_1 \,)^*.$$

All problematic groups of rules can be replaced by regular expressions.

This is usually possible for realistic programming languages.

There is no general algorithm, so you have to use your insight and creativity.

Rewrite the Grammar (2)

The result is:

$$
\begin{array}{lcl}
E & \rightarrow & E_1\,(\ (\,'+'\ \mid\ '-'\,)E_1\ )^* \\
E_1 & \rightarrow & E_2\,(\ (\,'\times'\ \mid\ '/'\,)E_2\ )^* \\
E_2 & \rightarrow & (\,'-'\,)^*\,E_3 \\
E_3 & \rightarrow & \textbf{int} \mid \textbf{double} \mid\ '('\ E\ ')'\ \mid \\
 & & \textbf{string}\,(\,\epsilon \mid\ '('\ E\,(\ ','\ E\ )^*\ ')'\ )
\end{array}
$$

$$V = \{E, E_1, E_2, E_3\},$$

$$\Sigma = \{\ +,\ -,\ \times,\ /,\ \textbf{int},\ \textbf{double},\ \textbf{string},\ (,\ ),\ ',',\ \}.$$

Start symbol is E.

Choice '|' can be implemented with **if**. Repetition * can be implemented by using **while**.

Function parseE can be implemented as follows:

a = parseE$_1$(input)
while the current token is '-' or '+' **do**
    t = the current token.
    move to the next token.
    a_2 = parseE$_1$(input)
    if t = '+' **then**
        $a = \,'+'(a, a_2)$
    else
        $a = \,'-'(a, a_2)$
return a

The other functions can be written in similar way.

Interface to the Tokenizer

The tokenizer is used in two ways: Seeing the current token, and moving to the next token.

In order to implement this, a buffer between the tokenizer and the parse functions is needed:

```
class BufferedLexer
{
   Lexer lex;
   java_cup.runtime.Symbol lookahead;

   BufferedLexer( Lexer lex )
      { this. lex = lex; lookahead = lex. nextToken( ); }
         // lex. nextToken( ) reads a token
         // and moves input foward.
```

```
    Symbol peekToken( )
       { return lookahead; }

    void move( )
    {
       if( lookahead != EOF )
          lookahead = lex. nextToken( );
             // Never read beyond end of file.
    }
};
```

Errors

Errors must be taken seriously, and should be included in the design from the beginning.

Creating good error messages is difficult.

Giving up after an error, is acceptable only for very simple programs. Realistic programs need to continue as good as possible, in order to collect as much information as possible in one run. This is called error recovery.

Recovery

The ability to recover from errors is an important feature for the usefulness of a parser.

In most cases, one ignores the input, until a synchronization symbol is found. (For example `;` or a closing `)`)

One has to create a special attribute for the garbage read, and this attribute must not result in further errors.

Summary

For simple grammars, it is possible to implement a top down parser by hand. Such parsers are called recursive descent parsers.

In most cases, you have to modify the grammar rules in order to make the parse functions deterministic. Often, this can be achived by using regular expressions.

You have to write a parse function for every non-terminal, which returns the attribute of this non-terminal.

Errors must be taken seriously. Recovering from errors is difficult. It is easier with a parser generator.