Parsing In Practice Using Attribute Grammars

Tasks of the Parser

The parser is an important part of a compiler.

Its task is to analyze the input according to the grammar of the language, and to construct a parse tree.

The problem of checking types, and checking if all variables are declared and used in accordance with their type, does not belong to the tasks of the parser.

Output of the Parser

Dependent on the complexity of the language being compiled, the output of the parser can either be

1. A cleaned up parse tree (Abstract Syntax Tree).
2. Executable code. (Possible only for simple languages. Most languages need more translation stages.)
3. A value. (e.g. for simple calculator program.)

Tokenizer vs. Parser

The input program is a sequence (a file) of characters, usually ASCII.

In the first stage the input is grouped into words of the programming language. This is done by a separate procedure called tokenizer.

A tokenizer recognizes numbers, operators, identifiers, strings and keywords. It removes comments.

The reason for this a separate stage are:

- Words can be recognized with DFAs, which is very efficient.
- It is difficult to define a complete programming language on the level of characters. (For example the fact that comments can be inserted everywhere in code and should be ignored by the compiler.)

Grammars (How it should be done in Real Life)

Definition: An attribute grammar has form $\mathcal{G} = (\Sigma, A, R, S, T, V)$, in which

- Σ is the set of symbols. We will not distinguish between terminal symbols and variable symbols.
- A is a function that attaches to each $\sigma \in \Sigma$ a non-empty attribute set $A(\sigma)$.
- R is a set of rewrite rules with associated actions.
- $S \in \Sigma$ is the start symbol, $T \subseteq \Sigma$ is the set of terminator symbols. These are symbols that follow after a correct input (e.g. EOF or ;).
- V is a finite set of variables that can be used by actions. (These are variables, as used in a programming language)

We define $\Sigma \otimes A = \{ \ (\sigma, a) \ \mid \ \sigma \in \Sigma \text{ and } a \in A(\sigma) \ \}$.

Example (A Calculator)

We want to build a simple calculator, where the user can type expressions, like for example

```
1 + 1;     // compute 1 + 1
a = 3 + 4; // assign 7 to a.
b = a + a; // Must assign 14 to b.
```

The expressions are ended with a (;).

Grammar

The rewrite rules are as follows:

$$\begin{array}{rcl} S & \to & E \mid \text{ident} = E \\ E & \to & E + F \mid E - F \mid F \\ F & \to & F \times G \mid F \,/\, G \mid G \\ G & \to & -G \mid \text{num} \mid \text{ident} \mid (\, E \,) \end{array}$$

S is the start symbol. We put $T = \{ \; ; \; \}$. This means that input must always be terminated with a $(;)$.

We put $A(S) = A(=) = A(+) = A(-) = A(\times) = A(\,/\,) = A(\,'(' \,) = A(\,')' \,) = \{\top\}$. (A placeholder denoting the empty attribute.)

We put $A(E) = A(F) = A(G) = A(\text{num}) = \mathcal{R}$. (The set of real numbers).

$A(\text{ident}) =$ (the set of strings).

Attributes

Symbols have an attribute attached to them. The set of possible attributes depends on the type of the symbol. For example, **num** has **double** attribute, while **ident** has a string.

The attributes are initially read from the input by the tokenizer.

If the input is `+`, the tokenizer constructs $(+, \top)$.

If the input is `abc`, the tokenizer constructs $(\text{ident}, 'abc')$.

If the input is `2.71828`, the tokenizer constructs $(\text{num}, 2.71827)$.

Terminals vs. Non-Terminals

The standard definition of a grammar is $\mathcal{G} = (V, \Sigma, R, S)$, where V are the non-terminal symbols (variables) and Σ are the terminal symbols.

For practical use, this distinction is not important. One could say that non-terminals are the symbols that occur to the left of a rewrite rule, while terminals are the symbols that can be constructed by the tokenizer.

But there is no reason why a token cannot be both at the same time. We simply use Σ for the set of all tokens.

Priorities

The aim of the different variables E, F, G in the grammar is to ensure that operator priorities are applied correctly.

Supppose that one removes F and G from the grammar:

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid E \times E \mid E \,/\, E \mid \; - E \\ E & \rightarrow & \mathrm{num} \mid \mathrm{ident} \mid (\, E \,) \end{array}$$

In this grammar, the expressions $\mathrm{num} - \mathrm{num} + \mathrm{num}$ and $\mathrm{num} + \mathrm{num} \times \mathrm{num}$ can be parsed in different ways, resulting in different outcomes.

If one replaces the rules for E by

$$E \rightarrow F + E \mid F - E \mid F$$

then $+$ and $-$ will be applied from right to left, so that the outcome of $4 - 1 - 2$ will be 5 instead of 1.

Action Code

In bottom-up parsing, rules are applied from right to left. This means that the parser recognizes the right side of a rule in the tokenized input, and replaces it by the corresponding left side.

When it makes such a replacement, it executes action code that is associated to the rule. The action should compute the attribute for the left hand side.

When computing the attribute for the left hand side, actions can use and modify the global variables in V.

In the calculator example, we use one global variable v which stores the values that were assigned to variables.

For example, to the rule $E \to E + F$, one can attach action code as follows:

- Attach variable names to the right hand side, so that the attributes have names that can be used in the code: $E{:}\,e + F{:}\,f$.
- Provide code that uses e and f. In this case, we simply write **return** $e + f$.

  The returned value will become the attribute of the E in the left hand side.

Action Code

Rule $S \to \text{ident} = E$ has the following action code.

- Write it as: $S \to \text{ident}{:}\,\text{i} = E{:}\,e$.
- The action code has form $v.\text{assign}(i, e);\ \ \textbf{return}\ \top;$

  Store e (the value of expression E) in i (the variable attached to ident). After that, return the empty attribute. (Because S has no attribute.)

Example

Suppose that the user types the input $3 + 4 \times a$; and assume that a has value 6 in v (the variable store).

The tokenizer returns

$$(\mathbf{num}, 3)\,(+, \top)\,(\mathbf{num}, 4)\,(\times, \top)\,(\mathbf{ident}, 'a')\,(\,;\,, \top).$$

Applying rules from right to left, as soon as possible, results in:

$$\underline{(\mathbf{num}, 3)},\ (+, \top)\,(\mathbf{num}, 4)\,(\times, \top)\,(\mathbf{ident},'a')\,(\,;\,, \top)$$
$$\underline{(G, 3)},\ (+, \top)\,(\mathbf{num}, 4)\,(\times, \top)\,(\mathbf{ident},'a')\,(\,;\,, \top)$$
$$\underline{(F, 3)},\ (+, \top)\,(\mathbf{num}, 4)\,(\times, \top)\,(\mathbf{ident},'a')\,(\,;\,, \top)$$
$$(E, 3),\ (+, \top)\,\underline{(\mathbf{num}, 4)}\,(\times, \top)\,(\mathbf{ident},'a')\,(\,;\,, \top)$$
$$(E, 3),\ (+, \top)\,(F, 4)\,(\times, \top)\,\underline{(\mathbf{ident},'a')}\,(\,;\,, \top)$$
$$(E, 3),\ (+, \top)\,\underline{(F, 4)\,(\times, \top)\,(G, 6)}\,(\,;\,, \top)$$
$$\underline{(E, 3)\,(+, \top)\,(F, 24)}\,(\,;\,, \top)$$
$$\underline{(E, 27)}\,(\,;\,, \top\ )$$
$$(S, 27)\,(\,;\,, \top\ )$$

This is the start symbol followed by an end symbol, so it's the final state.

Obtaining a Parser

There are two ways to obtaining a parser:

1. Write it by hand.
2. Use a parser generator.

(1) seems easy, because you don't have to download and install anything, and don't have to read a boring manual, but you will regret later, unless the language is very simple and will never change.

Writing a Parser by Hand

If you write a parser by hand, it will be a top down parser.

Proceed as follows:

For each non-terminal symbol E in the grammar, write a function parse$E$ that recognizes the words that can be obtained from E.

The function parse$E$ returns the attribute that was obtained from E.

The parse function for E looks at the next symbol in the input, decides which grammar rule for E applies, and processes the input. If necessary, it calls parse$V$ for another symbol V.

Recursive Descent

This way of parsing is also called 'recursive descent', because the parser descends from S to the terminal symbols, and the functions for the non-terminal symbols recursively call each other.

Unfortunately, is usually necessary to change the grammar. For example, with rules $E \rightarrow E + F \mid F$, the function parse$E$ has to start by recursively calling parse$E$ or parse$F$.

Unfortuntately, this can be decided only when $+$ encountered.

In order to solve this problem, the rules have to be merged into a single rule with a regular expression to the right: $E \rightarrow F\,(+F)^*$.

Now one can write

$$\text{parseF}();\, \textbf{while}(\, \text{nextsymbol} =' +'\,)\{\, \text{moveforward}(); \text{parseF}();\, \}.$$

Using Parser Generation Tools

Once one has learnt how to use the tool, using a parser generator is much easier.

It will be easy to change the grammar, and the parser generator will automatically check for ambiguity, and other possible problems (for example unrewritable symbols.)

We will follow this approach in the remaining slides. If one uses a parser generator, the resulting parser will be bottom up. This means that one starts with the input, and applies the grammar rules backwards, i.e. from right to left.