Parsing

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Tasks of the Parser

The main task of the parser is to analyze the input and to determine its structure.

The input of the parser is created by the tokenizer.

The tokenizer has converted the input (which was a sequence of characters) into a sequence of tokens (which are pairs, consisting of a label and an attribute whose type is dictated by the value of the label).

Type checking and checking whether variables are declared, 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.)

Why are Parsers and Tokenizers Separated

- DFAs are very efficient, one should use them whenever possible.
- Irrelevance of comments (or whitespace) would be very hard to express using a grammar.
- Some decisions can be only made at the end of a token (double vs. int), which would lead to decision conflicts that standard parsing approaches cannot solve.

Building a Parser

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

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

GCC used to use a parser using GNU bison, present versions use a hand-written parser. The reasons seem to be that C^{++} has a hard grammar (declarations look similar to statements), and hand-written parsers can give better error messages.

Grammars

From Sipser, Introduction to the Theory of Computation:

Definition A context-free grammar is a 4-tuple $\mathcal{G} = (V, \Sigma, R, S)$, in which

- V is the set of non-terminal (or helper) symbols.
- Σ is the set of terminal symbols.
- R is a set of rewrite rules. Elements of R have form $A \to w$, with $A \in V$, and $w \in (V \cup \Sigma)^*$.
- $S \in V$ is the start symbol.

Terminal vs. Non-Terminal Symbols

In compiler construction, it is not necessary that $\Sigma \cap V = \emptyset$.

Σ is the set of symbols that can be constructed by the tokenizer.

V is the set of symbols that occur to the left of a rewrite rule.

Attributes

Context-free grammars can only give yes/no answers. (Does some word belong to $\mathcal{L}(\mathcal{G})$?)

In order to compute meaning, we need to add attributes to the symbols and to the rewrite rules.

As we did with tokens, we will represent symbols by pairs of form (σ, a), where σ is the type of the symbol, and a is the attribute. We write Σ for the set of possible symbol types, and use A for the attribute function, that assigns to each $\sigma \in \Sigma$ the type of its attribute. In a symbol (σ, a), the type of a must be $A(\sigma)$.

Attribute Grammars

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. Each rule has form $v \to w/f$, where $v \in \Sigma$ and $w \in \Sigma^*$, and f is a function, whose exact form we will describe later.
- $S \in \Sigma$ 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 $\Sigma \otimes A = \{ \ (\sigma, a) \ \mid \ \sigma \in \Sigma \text{ and } a \in A(\sigma) \ \}$.

Rewrite Relation for Attribute Grammars

Definition: For an attribute grammar $\mathcal{G} = (\Sigma, A, R, S, T, V)$, the one step rewrite relation $\Rightarrow$ is defined as follows:

- If $v \rightarrow w \ / \ f \in R$, and
- $x, y \in (\Sigma \otimes A)^*$.
- $a_1 \in T(w_1), \ldots, a_n \in T(w_n)$,

then

$$x\,(\,v,\ f(a_1, \ldots, a_n)\,)\,y \Rightarrow x\,(w_1, a_1) \cdots (w_n, a_n)\,y.$$

Language Generated by Attribute Grammar

Definition: Define

- $w_1 \Rightarrow^0 w_2$ if $w_1 = w_2$,
- $w_1 \Rightarrow^{i+1} w_2$ if there exists a word (with attributes) $w' \in (\Sigma \times A)^*$, s.t. $w_1 \Rightarrow^i w'$, and $w' \Rightarrow w_2$.
- $w_1 \Rightarrow^* w_2$ if there exists an $i \geq 0$, s.t. $w_1 \Rightarrow^i w_2$.

Given an attribute grammar $\mathcal{G}$, and a word $w \in (\Sigma \otimes A)^*$, the parser needs to answer the following question:

For which attribute $a \in A(S)$, do we have $(S, a) \Rightarrow^* w$?

Example (A Calculator)

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

```
1 + 1;      // replies with 2.
a = 3 + 4; // assigns 7 to a.
b = a + a; // assigns 14 to b.
```

Expressions end with a (;).

Example (Calculator)

Consider the following context free grammar:

$$
\begin{array}{lcll}
E & \to & E + F & f(x,y,z) = x + z \\
 & & F & f(x) = x \\
F & \to & F \times G & f(x,y,z) = xz \\
 & & G & f(x) = x \\
G & \to & \text{num} & f(x) = x \\
 & & (E) & f(x,y,z) = f(y)
\end{array}
$$

$\Sigma = \{E, F, G\} \cup \{\ '+',\ '\times',\ \text{num},\ '(',\ ')'\ \}, \quad S = E.$

$$T(E) = T(F) = T(G) = T(\text{num}) = \textbf{double}.$$

$$T(\sigma) = \textbf{unit}, \text{ for the other labels.}$$

Example

Suppose that we wish to parse the word $3 + 4 \times 5$. The tokenizer constructs $(\text{num}, 3) + (\text{num}, 4) \times (\text{num}, 5)$. (**unit** attributes are omitted) Applying $\Rightarrow$ from right to left, one obtains:

$$
\begin{aligned}
&\underline{(\text{num}, 3)} + (\text{num}, 4) \times (\text{num}, 5) \\
&\underline{(G, 3)} + (\text{num}, 4) \times (\text{num}, 5) \\
&\underline{(F, 3)} + (\text{num}, 4) \times (\text{num}, 5) \\
&(E, 3) + \underline{(\text{num}, 4)} \times (\text{num}, 5) \\
&(E, 3) + \underline{(G, 4)} \times (\text{num}, 5) \\
&(E, 3) + (F, 4) \times \underline{(\text{num}, 5)} \\
&(E, 3) + \underline{(F, 4) \times (G, 5)} \\
&\underline{(E, 3) + (F, 20)} \\
&(E, 23)
\end{aligned}
$$

A Bigger Grammar (without Attributes)

$$S \to \textbf{ if } E \textbf{ then } S, \quad S \to \textbf{ if } E \textbf{ then } S \textbf{ else } S.$$

$$S \to \textbf{ while } E \textbf{ do } S, \quad S \to \textbf{ ident } := \textbf{ E}.$$

$$S \to \textbf{begin } L \textbf{ end}, \quad L \to S, \quad L \to L; S.$$

$$E \to E + E, \quad E \to E - E, \quad E \to E * E, \quad E \to E/E.$$

$$E \to -E, \quad E \to +E, \quad E \to (E).$$

$$E \to \text{ident}, \quad E \to \text{num}, \quad E \to \text{true}, \quad E \to \text{false}.$$

$$E \to E = E, \quad E \to E \neq E.$$

$$E \to E < E, \quad E \to E > E, \quad E \to E \leq E, \quad E \to E \geq E.$$

$$E \to E \wedge E, \quad E \to E \vee E, \quad E \to \neg E.$$

The grammar on the previous slide does not handle operators in a realistic way, because it does not take priorities into account.

For example, **ident** $-$ **ident** $+$ **ident** can be parsed as (**ident** $-$ **ident**) $+$ **ident**, or **ident** $-$ (**ident** $+$ **ident**).

Priorities can be introduced either by separating E into different symbols for different levels of priority, or by handling the operator priorities in another way.

I hope that you see how easy it is to define grammars for realistic programming languages.

In order to obtain a full compiler, one has to create attribute functions that check types of expressions, and emit intermediate code.

In a parser generator one can create attribute functions using Java or C code. The parser generator constructs a parser that automatically calls the attribute functions.

Dealing with Operators

Operators are a convenient way of writing binary or unary functions. Operators can be classified into three types, dependent on their arity, and the place where they are written:

infix: An infix operator is a binary operator that is written between its operands. Examples are

```
a + b > 4 && ( c <= d ) || ( d > d ).
```

prefix: A prefix operator is a unary operator that is written in front of its operand. Examples are

```
! ( ++ b ), - 4, & p.
```

postfix: A postfix operator is a unary operator that is written behind its operand.

```
a ++, b -- .
```

Usage of Operators (2)

If one wants to parse a language with operators, the main question is: Is it required to add new operators while the program is running? For example, in Prolog, it is possible to define new operators interactively. In most programming languages, the set of operators is fixed.

When the set of operators is fixed, it is possible to encode the priorities into the definition of the context free grammar.

If the set of operators is extendable, then one has to use an ambiguous grammar, and use other methods to decide operator priorities.

Possible Conflicts between Operators

There are four types of conflicts possible:

- Between infix and infix:

```
A + B * C.
```

- Between prefix and infix:

```
- A + B.
```

- Between infix and postfix:

```
A + B ++ .
```

- Between prefix and postfix:

```
-- A ++ .
```

Using Priority and Associativity

In order to avoid ambiguity, one can assign a priority and an associativity to each operator.

In case of a conflict

$$\cdots \text{ op1 } E \text{ op2 } \cdots,$$

the operator with highest priority wins.

If both operators have the same priority (or are the same) and both are left associative, then parse as

$$(\cdots \text{ op1 } E) \text{ op2 } \cdots.$$

If both are right associative, then parse as

$$\cdots \text{ op1 } (E \text{ op2} \cdots).$$

If the operators have different associativities, or no associativity, then the expression is syntactically incorrect.

Expressing Priorities in the Grammar

Assume that the possible priorities are $1, \ldots, n$, where n is the highest priority (strongest attraction).

Create a sequence of symbols $E_1, \ldots, E_{n+1}$.

For each i, $1 \leq i \leq n$, add a rule $E_i \rightarrow E_{i+1}$.

Add rules $E_{n+1} \rightarrow$ (E_1), $E_{n+1} \rightarrow$ num, $E_{n+1} \rightarrow$ ident.

Expressing Priorities in the Grammar (2)

For an infix operator **op** with priority i,

- if **op** is left associative, then add a rule $E_i \to E_i$ **op** E_{i+1},
- if **op** is right associative, then add a rule $E_i \to E_{i+1}$ **op** E_i,
- if **op** has no associativity, then add a rule $E_i \to E_{i+1}$ **op** E_{i+1}.

For a prefix operator **op** with priority i,

- if **op** is left associative, then add a rule $E_i \to$ **op** E_{i+1},
- if **op** is right associative, or not associative, then add a rule $E_i \to$ **op** E_i.

For a postfix operator **op** with priority i,

- if **op** is right associative, then add a rule $E_i \to E_{i+1}$ **op**,
- if **op** is left associative, or not associative, then add a rule $E_i \to E_i$ **op**.

Dangling Else Problem

The dangling else problem is related to the operator priorities: We have the following rules:

$$S \rightarrow \textbf{ if } E \textbf{ then } S, \quad S \rightarrow \textbf{ if } E \textbf{ then } S \textbf{ else } S.$$

The sequence **if** E_1 **then if** E_2 **then** S_1 **else** S_2 can be parsed in two ways.

It's called the dangling else problem. It can be solved with ad hoc solutions.

A systematic solution separates S into two symbols S and S_e. Symbol S_e is like S, but every **if** must have an **else**.

$$S \rightarrow \textbf{ if } E \textbf{ then } S, \quad S \rightarrow \textbf{ if } E \textbf{ then } S_e \textbf{ else } S.$$

$$S_e \rightarrow \textbf{ if } E \textbf{ then } S_e \textbf{ else } S.$$

Top-Down Parsing

Top Down Parsing starts with the start symbol S and tries to rewrite it into the input word w. Although this is natural from the mathematical point of view, there are several problems with it:

- Attribute computation is easier with bottom up parsing.
- Top down parsing cannot deal with left recursion (rules of form $A \to A\,w$, or rules with shared prefixes (rules of form $A \to w\,w_1,\ A \to w\,w_2$) Because of this, one nearly always has to change the grammar. This makes the computation of the meaning harder.

The big advantage of top down parsing is that it is easy to understand, and that often it can be implemented by hand.