Parsing

Tasks of the Parser

The parser works on the output of the tokenizer. The tokenizer has converted the input (a sequence of characters) into a sequence of tokens. (a sequence of tags with attributes)

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

Type checking and checking if every used variable has a declaration, are not task of the parser.

Why are Parsers and Tokenizers Separated

- Tokens have more ambiguity than the higher-level grammar, and different conventions for handling ambiguity. 1234567 could be a single integer, or 7 distinct integers. Tokenizer: Structure as long as possible. Parser: Structure as short as possible.
- Irrelevance of comment would be hard to express in a grammar.
- DFA's are very efficient, it is worth using them when possible.

Building a Parser

As with tokenizers, there are two ways of building a parser:

- Writing it by hand.
- Using a parser generator. (Yacc,Bison)

Grammars

Definition: An grammar is a structure of form $\mathcal{G} = (\Sigma, R, S)$, in which

- Σ is the alphabet. (a set of tokens)
- R is a set of rewrite rules. Each element of R has form $\sigma \to w$, where $\sigma \in \Sigma$ and $w \in \Sigma^*$.
- $S \in \Sigma$ is the start symbol.

Most definitions of a grammar separate Σ into terminal and non-terminal symbols.

Terminal symbols cannot occur on the lhs of a rule.

This distinction is not important for compiler construction, so we don't use it.

The definition on the previous slide actually defined context-free grammars. In a non context-free grammar, the rules in R can have form $w_1 \rightarrow w_2$, where $w_1, w_2 \in \Sigma^*$.

The Rewrite Relation

The one-step rewrite relation $\Rightarrow$ is the smallest relation having the following properties:

- If $(\sigma \rightarrow w) \in R$, then $\sigma \Rightarrow w$.
- If $w_1 \Rightarrow w_2$, and $a \in \Sigma$, then $w_1 a \Rightarrow w_2 a$ and $a w_1 \Rightarrow a w_2$.

The multi-step rewrite relation $\Rightarrow^*$ is the smallest relation with the following properties:

- For all words $w \in \Sigma^*$, $\quad w \Rightarrow^* w$,
- If $w_1 \Rightarrow^* w_2$ and $w_2 \Rightarrow w_3$, then $w_1 \Rightarrow^* w_3$.

Accepted Words

Let $\mathcal{G} = (\Sigma, R, S)$ be a grammar. $\mathcal{G}$ accepts a word $w \in \Sigma^*$ if $S \Rightarrow^* w$.

If $S \Rightarrow^* w$, then a sequence $S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = n$, is called a derivation.

A derivation of S can be interpreted in two ways:

1. Start with S and replace left hand sides of rules by right hand sides, until w is obtained.
2. Start with w and replace right hand sides of rules by left hand sides, until S is obtained.

(1) is called top down. (2) is called bottom up.

Example

Consider the grammar $(\{a, b, c, d\}, R, a)$ with R consisting of the rules

$$a \to abc, \quad a \to bac, \quad a \to d.$$

The following are correct derivations:

$$a \Rightarrow abc \Rightarrow bacbc \Rightarrow bdcbc,$$

$$a \Rightarrow abc \Rightarrow abcbc \Rightarrow abcbcbc,$$

$$a \Rightarrow bac \Rightarrow babcc,$$

$$a \Rightarrow d.$$

A Realistic Grammar

$$S \to \textbf{if } B \textbf{ then } S \quad S \to \textbf{if } B \textbf{ then } S \textbf{ else } S$$

$$S \to \textbf{while } B \textbf{ do } S$$

$$S \to \textbf{id} := N;$$

$$N \to N + N \quad N \to N - N \quad N \to N * N \quad N \to N/N$$

$$N \to -N \quad N \to (N)$$

$$N \to \textbf{id} \quad N \to \textbf{num}$$

$$B \to \textbf{true} \quad B \to \textbf{false}$$

$$B \to N = N \quad B \to N\ != N \quad B \to N < N \quad B \to N > N$$

$$B \to N <= N \quad B \to N >= N$$

$$B \to B \textbf{ and } B \quad B \to B \textbf{ or } B \quad B \to !B, \quad B \to (B)$$

Dependent Product

Definition: Let S_1 be some set, and let S_2 be a set of pairs. Then the dependent product of S_1 and S_2, written as $S_1 \otimes S_2$, is defined as the set

$$\{(s_1, s_2) \mid s_1 \in S_1 \text{ and there exists an } s_2', \text{ s.t. } s_2 \in s_2' \text{ and}$$

$$(s_1, s_2') \in S_2\}.$$

Tokens

Definition: A token type is a pair (Σ, T), in which Σ is a finite alphabet, and T is a function s.t. for each $\sigma \in \Sigma$, $T(\sigma)$ is a set.

A token (with attribute) is an element of $\Sigma \otimes T$.

Example: If one sets $\Sigma = \{\mathbf{int}, \mathbf{real}\}$, with
$T(\mathbf{int}) = \mathcal{Z}$, $T(\mathbf{real}) = \mathcal{R}$, then
$(\mathbf{real}, 3)$, $(\mathbf{real}, 3.141526535)$, $(\mathbf{int}, 2)$, $(\mathbf{int}, -1)$ are tokens.

$(\mathbf{int}, 2.718271828)$ is not a token.

Tokens (2)

In the slides about tokenizers, we used Λ instead of Σ, because Σ was reserved for the input alphabet.

The alphabet of the parser are tokens constructed by the tokenizer.

Remember that T (a function) is just a set of pairs.

Attribute Grammars

Definition: An attribute grammar is a structure of form $\mathcal{G} = (\Sigma, T, R, S)$, in which

- (Σ, T) are a token type.
- R is a set of annotated rewrite rules. Each element of R is a pair of form $\sigma \to w : f$, where $\sigma \in \Sigma$, $w \in \Sigma^*$, f is a function from $T(w_1) \times \cdots \times T(w_n)$ to $T(\sigma)$.
- S is the starting symbol.

Rewrite Relation for Attribute Grammars

For an attribute grammar, the one step rewrite relation $\Rightarrow$ is defined as the smallest relation of type $(\Sigma \otimes T)^* \times (\Sigma \otimes T)^*$, which has the following two properties:

- If $(\sigma \rightarrow w_1 \cdot \ldots \cdot w_n)\colon f \in R$, then

$$(\sigma, f(x_1, \ldots, x_n)) \Rightarrow (w_1, x_1) \cdot \ldots \cdot (w_n, x_n).$$

- If $(v_1, x_1) \cdot \ldots \cdot (v_m, x_m) \Rightarrow (w_1, y_1) \cdot \ldots \cdot (w_n, y_n)$, and $(\sigma, z) \in \Sigma \otimes T$, then

$$(v_1, x_1) \cdot \ldots \cdot (v_m, x_m) \cdot (\sigma, z) \Rightarrow (w_1, y_1) \cdot \ldots \cdot (w_n, y_n) \cdot (\sigma, z),$$

and

$$(\sigma, z) \cdot (v_1, x_1) \cdot \ldots \cdot (v_m, x_m) \Rightarrow (\sigma, z) \cdot (w_1, y_1) \cdot \ldots \cdot (w_n, y_n).$$

Accepted Words

Let $\mathcal{G} = (\Sigma, T, R, S)$ be an attribute grammar. $\mathcal{G}$ accepts a word $w \in (\Sigma \otimes T)^*$ with attribute x if $(S, x) \Rightarrow^* w$.

Grammar for a Pocket Calculator

We want to build a calculator that has floating point numbers, identifiers, and operators.

$$S \to T, \qquad f(x) = x.$$

$$S \to S + T, \qquad f(x, y, z) = x + z.$$

$$S \to S - T, \qquad f(x, y, z) = x - z.$$

$$T \to U, \qquad f(x) = x.$$

$$T = T \times U, \qquad f(x, y, z) = x \times z.$$

$$T = T/U, \qquad f(x, y, z) = x/z.$$

$$U \to V, \qquad f(x) = x.$$

$$U \to -U, \qquad f(x, y) = -y.$$

$$U \to U!, \qquad f(x, y) = x!.$$

$$U \to \mathbf{num}, \qquad f(x) = x,$$

$$U \to (S), \qquad f(x, y, z) = y.$$

Operators

Operators are convenient way of representing binary or unary functions.

An operator is either

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

$$A + B,\ A - B.$$

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

$$-A,\ ++A.$$

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

Operators

Operators can have conflicts

- Between two infix operators:

$$A + B \times C.$$

- Between prefix and infix:

$$-A + B.$$

- Between infix and postfix:

$$A + B!.$$

- Between prefix and postfix:

$$-A!.$$

Non-Recursive Nature of Realistic Attribute Grammars

Consider a rule $\sigma \rightarrow w : f$. The attribute function f specifies how to compute the attribute of σ from the attributes of f.

Not all computations can be expressed by attributes, because it can happen that information in one subtree is needed in another subtree. (Told differently: In must languages, there is not only an information flow from the leaves to the root of the parse tree, but also from left to right.)

Grammar for Calculator with Variables

$$S \to SC, \quad S \to C.$$

(The input is a sequence of commands, C is a single command.)

$$C \to id := E., \quad C \to E.,$$

(A single command is either an assignment, or command to evaluate some expression E)

$$E \to E + F, \quad E \to E - F, \text{ etc.}$$

$$H \to \text{ id }, \quad H \to \text{ num }.$$

Variables that are assigned by the rule $C \to id := E.$, need to be remembered and used when processing the rule $H \to$ id .

The attribute function for $H \rightarrow$ id needs to know the assignment of id that was made in rule $C \rightarrow \text{id} := E$.

Two solutions:

1. Cheating: The attribute functions are implemented in an imperative language which allows global variables. Store the variable assignments in a global variable.
2. Replace the attribute type of C by $(\Sigma \rightarrow \Sigma) \times \mathcal{R}$, where $\mathcal{R}$ is the type of reals and Σ is the type of partial functions from identifiers to $\mathcal{R}$. (Type of binding stores)

It is good to know that (2) exists, but (1) is practically useful.

Making Side-Effects Compositional

E, F, G, H receive attribute type $\Sigma \to \mathcal{R}$.

S and C receive the attribute type $(\Sigma \to \Sigma)$.

$$H \to \mathbf{id}, \quad F = \lambda n{:}I\ \lambda\sigma{:}\Sigma\ (\sigma\ n),$$

$$H \to \mathbf{num}, \quad F = \lambda x{:}\mathcal{R}\ \lambda\sigma{:}\Sigma\ x$$

$$E \to E + F, \quad \lambda x_1{:}\Sigma \to \mathcal{R}\ x_2{:}\top\ x_3{:}\Sigma \to \mathcal{R}\ \lambda\sigma{:}\Sigma\ (x_1\ \sigma) + (x_2\ \sigma),$$

$$E \to E - F, \quad \lambda x_1{:}\Sigma \to \mathcal{R}\ x_2{:}\top\ x_3{:}\Sigma \to \mathcal{R}\ \lambda\sigma{:}\Sigma\ (x_1\ \sigma) - (x_2\ \sigma),$$

etc.

$$C \to \mathrm{id} := E, \quad \lambda n{:}I\ x_2{:}\top\ x_3{:}\Sigma \to \mathcal{R}\ \lambda\sigma{:}\Sigma\ \sigma^{n}_{(x_3\ \sigma)}.$$

$$C \to E, \quad \lambda x_1{:}\Sigma \to \mathcal{R}\ \lambda\sigma{:}\Sigma\ \sigma.$$

(And $(x_1\ \sigma)$ is printed)

Directions of Parsing

Parsing can be either top down or bottom up. Bottom up is more natural, because one can compute the attributes, and it has the advantage that decisions can be somewhat postponed.

I will not discuss top down parsing.

Shift/Reduce Parsing

As far as I know, shift/reduce parsing is the approach to parsing that is used most often.

Let $\mathcal{G} = (\Sigma, T, R, S)$ be an attribute grammer.

A state of a shift/reduce parser consists of a triple $\Sigma^* \times \Sigma^* \times \Sigma^*$. The first component of the pair is called stack, the second component is called lookahead, and the third component is the unread input.

Shift/Reduce Parsing

The transition relation $\vdash$ of a shift/reduce parser is defined by the following three cases:

read: For all $\phi, \psi, \rho \in \Sigma^*,\ w \in \Sigma,\ x \in T(w)$,

$$(\phi, \psi, (w, x) \cdot \rho) \vdash (\phi, \psi \cdot (w, x), \rho).$$

shift: For all $\phi, \psi, \rho \in \Sigma^*,\ w \in \Sigma,\ x \in T(w)$,

$$(\phi, (w, x) \cdot \psi, \rho) \vdash (\phi \cdot (w, x), \psi, \rho).$$

reduce: If $(\ \sigma \to w_1 \cdot \ldots \cdot w_n\ ) : f \in R$, then for all $\phi, \psi, \rho \in \Sigma^*$,

$$(\phi \cdot (w_1, x_1) \cdot \ldots \cdot (w_n, x_n), \psi, \rho) \vdash (\phi \cdot (\sigma, f(x_1, \ldots, x_n)), \psi, \rho).$$

Shift/Reduce Parsing

Define $\vdash^*$ in the usual way from $\vdash$.

Let $\mathcal{G} = (\Sigma, T, R, S)$ be an attribute grammar. A shift/reduce parser based on Γ accepts a word w if

$$(\epsilon, \epsilon, w) \vdash^* (S, \epsilon, \epsilon).$$

In practice, the following definition is used: Let $\mathcal{G} = (\Sigma, T, R, S)$ be an attribute grammar. A shift/reduce parser based on $\mathcal{G}$ accepts a word w, if

$$(\epsilon, \epsilon, w \cdot \#) \vdash^* (S, \#, \epsilon).$$

Here $\# \notin \Sigma$ is a special symbol to mark the end of the input. (Think of # as the EOF symbol.)

Shift/Reduce Parsing

We want parsing to be a deterministic process: In other words, we have to find a subrelation $\models \subseteq \vdash^*$ that

- $\models$ is deterministic: forall
  $S, S_1, S_2, \quad S \models S_1, \ S \models S_2 \Rightarrow S_1 = S_2$.
- has as small as possible lookahead,
- still finds all solutions.

Shift/Reduce Parsing

I hope you agree that shift/reduce parsing is easy in principle.

Only one question remaining: How to find $\models$?