Parsing

Tasks of the Parser

The output of the tokenizer is the input of the parser.

The tokenizer has converted the input (which was a sequence of characters) into a sequence of tokens. (which are pairs, consisting of a tag and an attribute)

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

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 parser can output either

1. An abstract syntax tree (AST).
2. Executable code. (Only for very simple languages.)
3. A value. (e.g. for non-programmable calculators.)

Why are Parsers and Tokenizers Separated

- DFA's are very efficient, one should use them whenever possible.
- Irrelevance of comments 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 formalisms cannot decide.

Building a Parser

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

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

GCC used to use a parser based on Bison, nowadays it has a hand-written parser. I don't know why.

Grammars

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

- Σ is an alphabet. (Set of possible 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.

Terminal vs. Non-Terminal Symbols

Symbols that occur in Σ, but which are never constructed by the tokenizer, are called non-terminal symbols. They are important for defining the language, but they do not occur in the language.

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 both $w_1, w_2 \in \Sigma^*$. Membership in non context-free languages is undecidable.

The Rewrite Relation

Definition: Given a grammar $\mathcal{G} = (\Sigma, R, S)$, the one step rewrite relation $\Rightarrow$ is defined as follows:

- If $\alpha_1, \alpha_2 \in \Sigma^*$, and $(\sigma \to w) \in R$, then $\alpha_1 \sigma \alpha_2 \Rightarrow \alpha_1 w \alpha_2$.

Definition: The multi step rewrite relation $\Rightarrow^*$ is the smallest relation that has the following properties:

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

Accepted Words

Definition: Let $\mathcal{G} = (\Sigma, R, S)$ be a grammar. $\mathcal{G}$ is said to accept a word $w \in \Sigma^*$ if $S \Rightarrow^* w$.

If $S \Rightarrow^* w$, then a sequence of form

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w$$

is called a derivation of w. A derivation of w can be done in two directions:

1. Start with S and replace the left hand side of a rule by the corresponding right hand side, until w is reached.
2. Start with w and replace the right hand side of a rule by the corresponding left hand side, until S is reached.

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

Example

Let $\mathcal{G}$ be defined by $\mathcal{G} = (\{a, b, c, d\}, R, a)$ with R consisting of the rules

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

The following sequences are correct derivations:

$$a \Rightarrow abc \Rightarrow (bac)bc \Rightarrow b(d)cbc,$$

$$a \Rightarrow abc \Rightarrow (abc)bc \Rightarrow (abc)bcbc,$$

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

$$a \Rightarrow d.$$

A Realistic Grammar

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

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

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

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

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

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

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

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

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

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.

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

Tokens with Attributes

Like in the tokenizer, we want to attach attributes to the tokens. This means that tokens will have form (t, n), where t is a tag that identifies the token, and n is an attribute whose type depends on t.

Let Σ be the set of all possible tokens. Let $A_1, \ldots, A_n$ be the set of all possible attributes, e.g. $\mathcal{N}, \mathcal{R}$, strings, etc.

We could define the set of tokens with attributes as the elements of $\Sigma \times (A_1 \cup \cdots \cup A_n)$, but in this way, we would allow tokens (t, a), where a is of a meaningless type.

In order to avoid this problem, we will define the dependent product on the next slide.

The Dependent Product

Definition: Let S_1 be a set, and let S_2 be a function from S_1 to sets. (This means that for every $s \in S_1$, $S_2(s)$ is a set.) The dependent product of S_1 and S_2, written as $S_1 \otimes S_2$, is defined as the set

$$\{(s_1, s_2) \mid s_1 \in S_1, \text{ and } s_2 \in S_2(s_1)\}.$$

Dependent Product (2)

Using the dependent product, any type of tokens with attributes can be adequately typed.

One can take $\Sigma = \{\mathbf{int}, \mathbf{real}, \mathbf{ident}\}$, and
$A(\mathbf{int}) = \mathcal{N}$, $A(\mathbf{real}) = \mathcal{R}$, and $A(\mathbf{ident}) =$ (the set of strings).

Then the set of tokens can be defined as $\Sigma \otimes A$.

Dependent Product (3)

There is a formal difficulty when defining tokens without attribute, like for example reserved words.

It is not possible to take $A(\mathbf{while}) = \emptyset$, because then no tokens of form $(\mathbf{while}, e)$ are possible.

Instead, one has to put $A(\mathbf{while}) = \{\top\}$, where $\top$ is some object that is very easy to construct.

Then the token for the reserved word **while** can have form $(\mathbf{while}, \top)$.

Attribute Grammars

Attribute grammars are obtained when one adds attributes to Σ in the definition of a grammar.

To each rule $\sigma \to w_1 \cdot \ldots \cdot w_n$, we add a function f with arity n, that specifies how the attribute of σ will be obtained from the attributes of $w_1, \ldots, w_n$ when the rule is applied (in bottom up direction).

If one has a derivation $S \Rightarrow \cdots \Rightarrow w$, then it is possible to compute the attribute of S from the attributes in w using the functions that are attached to the rules.

The attributes of w are constructed by the tokenizer.

Attribute Grammars (2)

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

- Σ is an alphabet. (Set of tags of possible tokens.)
- A is a function from Σ to sets.
- R is a set of rules with attribute functions. Each $r \in R$ has form $(\sigma \to w) : f$, where $\sigma \in \Sigma$, $w \in \Sigma^*$, and f is a function from $T(w_1) \times \cdots \times T(w_n)$ to $T(\sigma)$.
- $S \in \Sigma$ is the start symbol.

Rewrite Relation for Attribute Grammars

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

- If $(\sigma \to w_1 \cdot \ldots \cdot w_n) : f \in R$, and $\alpha_1, \alpha_2 \in (\Sigma \otimes A)^*$, then

$$\alpha_1 \cdot (\sigma, f(a_1, \ldots, a_n)\ ) \cdot \alpha_2 \Rightarrow \alpha_1 \cdot (w_1, a_1) \cdot \ldots \cdot (w_n, a_n) \cdot \alpha_n.$$

Definition: For an attribute grammar $\mathcal{G} = (\Sigma, A, R, S)$, the multistep rewrite relation $\Rightarrow^*$ is defined in the same way as for usual grammars, i.e. as the smallest relation that has the following properties:

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

Attribute Grammar for a Pocket Calculator

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

f copies the attribute of T without change.

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

The attribute of S on the lhs is obtained by adding the attributes of S and T on the right hand side. The attribute of $+$ is ignored.

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

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

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

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

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

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

The attribute of U on the left hand side will be minus the attribute of U on the right hand side. The attribute of $-$ is ignored.

$$V \to (S), \quad f(x, y, z) = y.$$

The attribute of V is copied from the attribute of S. The attributes of (and) are ignored.

$$V \to \textbf{num}, \quad f(x) = x.$$

The attribute of V is copied from the attribute of **num**. The attribute of **num** orginates from the tokenizer.

$$V \to \textbf{ident}, \quad f(x),$$

where $f(x)$ is the result of looking up x in the symbol table. I assume that **ident** has a string attribute that is constructed by the tokenizer.

Translating Functional Expressions into a Stack Machine

The attribute of S is a a code fragment that pushes a single number on the stack.

$$S \to \textbf{ident}, \quad f(x) = (\textbf{push a}),$$

where a is the address of variable x, as found in the symbol table.

$$S \to \textbf{num}, \quad f(x) = (\textbf{push } \#x).$$

$$S \to \textbf{ident}(L), \quad f(x, y, z, t) =$$

the code fragment consisting of L.**code**, combined with a primitive instruction or subroutine for f. It has to take take L.**arity** numbers from the stack, and put back a single number.

Translating Functional Expressions into Stack Machine (2)

The attribute of L is a pair (n, C), where n is a natural number and C is a fragment of code that pushes n numbers on the stack. We treat the pair like a **struct**, we call the first element **arity**, and the second element **code**. Using this, we can describe the rest of the grammar:

$$L \to S, \quad f(x) = (1, x).$$

This is correct because the attribute of S is code that pushes one number on the stack.

$$L \to L, S, \quad f(x, y, z) = (x.\text{arity} + 1, \quad x.\mathbf{code}; z).$$

The arity of L on the left hand side is one more than the arity of L on the right hand side, and the code of L on the left hand side is obtained by concatenating the codes of L on the right hand side and the code of S.

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 that has 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**.

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.
- Bottom up parsing cannot deal with left recursion (rules of form $A \to Aw$, or rules with shared prefixes (rules of form $A \to ww_1,\ A \to ww_2$) 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 it can be implemented by hand.

Grammars based on Deterministic Finite Automata

In order to solve the problems with left recursion and shared prefixes, I will assume that the right hand sides of the grammar rules are DFA's. (Deterministic Finite Automata.)

This is a very natural representation, because syntactic rules are often drawn in diagrams anyway. In addition, automata are usually easier to code by hand than grammar rules.

Most standard books (e.g. Aho/Ullman) do not use automata. Instead, they apply transformations on the grammar to avoid the problematic cases. It amounts to the same.

Grammars based on Deterministic Finite Automata

Definition: A DFA-based grammar is a triple (Σ, G, S), where Σ is an alphabet, G is a partial function G from Σ to deterministic finite automata over Σ, and S is the start symbol.

If $G(A)$ is defined, then we write $G(A) = (Q^A, \Sigma, Q_s^A, Q_a^A, \delta^A)$, and call A a non-terminal symbol, otherwise a terminal symbol.

We assume that the automata have no useless states: Every state in Q^A is reachable from the starting state in Q_s^A, and from every state in Q^A, it is possible to reach an accepting state in Q_a^A.

We define $w_1.A.w_2 \Rightarrow w_1.w.w_2$ as: $w_1, w_2 \in \Sigma^*$, $G(A)$ is defined, and $G(A)$ accepts the word w.

$\Rightarrow^*$ is defined from $\Rightarrow$ as usual.

Top-Down Parsing

Let (Σ, G, S) be a DFA-based grammar. We assume that there exists a special symbol $\# \in S$ denoting the end of the input. It occurs only (and always) at the end of an input word.

A state of the parser (S, w) consists of two components:

- A stack $S \in (\Sigma, q)^*$, where for each pair (A, q) occurring on the stack, A_i is a non-terminal, and $q \in Q^A$. (Using $\otimes$, we can write: $S \in (\Sigma \otimes Q)^*$).
- a word $w \in \Sigma^*$. The word w has exactly one occurrence of $\#$ at the end. The word w denotes the input that is not yet read.

The initial state of the parser is $(\ (S, Q_s^S),\ w.\#)$, where S is the start symbol of (Σ, G, S), and $w \in (\Sigma \backslash \{\#\})^*$ is the word that we want to parse.

Transition Relation

We define the transition relation $\vdash$ between states of the parser:

Read: If $s \in \Sigma \backslash \{\#\}$ is a terminal, and $(q, s, q') \in \delta^A$, then $(S.(A, q),\ s.w) \vdash (S.(A, q'),\ w)$. (The parser reads the symbol s.)

Return: If $q \in Q_a^A$, then $(S.(A, q),\ w) \vdash (S,\ w)$. (If the automaton on the top of the stack is in an accepting state, it can be removed.)

Descend: If $G(B)$ is defined, and $(q, B, q') \in \delta^A$, then $(S.(A, q),\ w) \vdash (S.(A, q').(B, Q_s^B),\ w)$. (If the automaton on top allows a transition for a non-terminal symbol B, we can start an automaton for this symbol B, in the hope that it will be able to reach an accepting state.)

Acceptance Conditions

The parser accepts the input $w \in (\Sigma \backslash \{\#\})^*$ if

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

(This implies that it reached an accepting state of $(Q^S, \Sigma, Q_s^S, Q_a^S, \delta^S\ ).\quad)$

Selecting the Transitions

Although the automata are deterministic, there is non-determinism in the parser:

There may be conflicts between descend and read, between different descends, between read and return, and between read and descend.

We need to find ways of making the decisions.

In order to do this, one needs to look at the first symbol of w. This symbol is called the lookahead symbol.

The Nullable Symbols

Definition: Let (Σ, G, S) be a DFA-based grammar. We call $A \in \Sigma$ nullable if $A \Rightarrow^* \epsilon$.

The set of nullable symbols can be easily computed from the following condition:

The set of nullable symbols is the smallest set $N \subseteq \Sigma$, s.t. if $G(A)$ is defined, and $G(A)$ accepts a word $w_1.\cdots.w_n$ with each $w_i \in N$, then $A \in N$.

(Checking reachability in a DFA is easy.)

The FIRST Sets

Let (Σ, G, S) be a DFA-based grammar. If $G(A)$ is defined, then FIRST(A) is the set $\{\sigma \mid A \Rightarrow^* \sigma.w\}$.

(The set of possible first symbols of a word w, that can be obtained from A.)

The sets FIRST(A) must be computed simultaneously for all symbols $A \in \Sigma$ by the following closure condition:

If $G(A)$ exists, and $s_1.s_2.\cdots.s_n.w$ with $n \geq 0,\ s_1, \ldots, s_n \in \Sigma$ are nullable, $w \in \Sigma^*$ is a word accepted by $G(A)$, then

- if w is not ϵ, and w_1 is a terminal, then $w_i \in$ FIRST(A).
- if w is not ϵ, and w_1 is a non-terminal, then FIRST$(w_i) \subseteq$ FIRST(A).

The FOLLOW Sets

Let (Σ, G, S) be a DFA-based grammar. If $G(A)$ is defined, then FOLLOW(A) is the set

$$\{\sigma \in \Sigma \mid \exists w_1, w_2 \in \Sigma^*, \text{ s.t. } S \Rightarrow^* w_1.A.\sigma.w_2\}.$$

(The set of possible symbols that come after something derived from A in words of the language.)

Computation of the FOLLOW Sets

The sets FOLLOW(A) can be simultaneously computed from the following condition:

If A is a non-terminal, and there is some B, for which $G(B)$ is defined and accepts a word of form $v.A.s_1.\cdots.s_n.w$, with $v, w \in \Sigma^*, \quad n \geq 0, \quad s_1, \ldots, s_n \in \Sigma$, and all of $s_1, \ldots, s_n$ are non-terminals that are nullable, then

- If w is nonempty, and the first symbol w_1 of w is a terminal, then $w_1 \in$ FOLLOW(A).
- If w is nonempty, and the first symbol w_1 of w is a non-terminal, then FIRST(w_1) $\subseteq$ FOLLOW(A).
- If $w = \epsilon$, then FOLLOW(B) $\subseteq$ FOLLOW(A).

Making the Decisions

Now we have the tools to make the decisions:

In order to apply Return, it must be the case that
$w_1 \in \text{FOLLOW}(A)$, for the lookahead symbol w_1.

In order to apply Descend, it must be the case that
$w_1 \in \text{FIRST}(B)$, or B is nullable and $w_1 \in \text{FOLLOW}(B)$.

(The condition that $w_1 \in \text{FOLLOW}(B)$ can be replaced by a more precise analysis, taking into account that B will be followed by the first letter in the continuation of (A, q'). It would be complicated.)

Recursive Descent Parsing

A parser that is obtained by direct implementation of the top-down parser is called recursive descent parser.

For every non-terminal A, one writes a procedure of form `parse_A( inputsource& lookahead, std::istream& input )`, that is based on the automaton $G(A)$. For most languages, this is doable in practice. *GCC* uses recursive descent for *C* and C^{++}. Earlier versions used Bison.

The procedures `parse_A( )` can either return a value, (the computed attribute), or produce output in some other way.

Errors

Errors should be taken very seriously, and included in the design from the beginning. Designing good error messages is difficult, because they are often automatically composed from different parts. (Line numbers, subtask, general task.)

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.

Unfortunately, the requirements of a good recovery strategy are not formalizable.

(This is a situation that happens more often than theoreticians like to admit. Most logicians believe that if all programs will meet their specifications, then all problems of computer science will be solved. In reality, many tasks don't have clear specifications, or the specification would be harder to understand than a program.)

A proposed solution:

Introduce a new kind of transitions to the automata, called error-transitions. Error transitions must be marked by a $\sigma \in \Sigma \cup \{\epsilon\}$. After an error has occurred:

1. Look for possible error transitions on the stack. If an error transition is possible, then make the transition.
2. If no error transition is possible, then throw away the lookahead symbol, read a new token, and try again until either EOF is reached, or an error transition is possible.
3. If a new error occurs within 3 tokens after a recovery, then assume that this is not a new error, but a wrong recovery. In that case, don't report the second error.

Finding a good error recovery strategy is a challenging task. Use your creativity and experiment a lot. One could try to give preferences to error transitions.