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 $(\sigma \rightarrow w) \in R$, 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

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

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

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

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

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

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

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

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

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

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

For example, $\mathbf{ident} - \mathbf{ident} + \mathbf{ident}$ can be parsed as $(\mathbf{ident} - \mathbf{ident}) + \mathbf{ident}$, or $\mathbf{ident} - (\mathbf{ident} + \mathbf{ident})$.

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
$A(\mathbf{int}) = \mathcal{N}, \quad A(\mathbf{real}) = \mathcal{R},$ 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 $\sigma \rightarrow w_1 \cdot \ldots \cdot w_n$, 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 $(\sigma \rightarrow w) : f$, 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 $(\sigma \rightarrow w_1 \cdot \ldots \cdot w_n) : f \in R$, 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 multi step 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.

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

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

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

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.

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

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

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

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

the code fragment consisting of $L.\textbf{code}$, combined with a primitive instruction or subroutine for f. It has to take take $L.\textbf{arity}$ 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 the stack.

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

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 combining the codes of L on the right hand side and z.

Usage of Operators

Operators are a convenient way of writing binary or unary funcitons. There are three types of operators:

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)

Consider the expression

```
++ p -- = a && b < 4 + * aa && b.
```

How should it be parsed?

If one wants to parse a language with operators, the main question is: Is it possible to add new operators while the programme is running?

If not, then the priorities can be encoded into the grammar.

If yes, one has to use an ambiguous grammar, and use other methods to decide priorities. (I will come back to this topic later.)

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

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

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

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

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

Shift/Reduce Parsing

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

The shift/reduce parser operates on triples
$(s, v, u) \in (\Sigma \otimes S)^* \times (\Sigma \otimes S)^* \times (\Sigma \otimes S)^*$, where

- $s \in (\Sigma \otimes A)^*$ is the stack.
- $v \in (\Sigma \otimes A)^*$ is the lookahead,
- $u \in (\Sigma \otimes A)^*$ is the input that is not yet read.

Shift/Reduce Parsing

We write $\vdash$ for the transition relation of the parser.

The parser starts in a state of form (ϵ, ϵ, u).

(Empty stack, empty lookahead, no input read.)

Read

A read means that the parser moves one unread token to the lookahead:

$$(s, v, (\sigma, \alpha) \cdot u) \vdash (s, v \cdot (\sigma, \alpha), u).$$

Shift

A shift means that the parser shifts one token from lookahead to the stack:

$$(s, (\sigma, \alpha) \cdot v, u) \vdash (s \cdot (\sigma, \alpha), v, u).$$

Reduction

A reduction means that the parser replaces the right hand side of a grammar rule by the left hand side. It uses the attribute function of the grammar rule to compute the new attribute.

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

$$(s \cdot (w_1, \alpha_1) \cdot \ldots \cdot (w_n, \alpha_n),\ v,\ u) \vdash (s \cdot (\sigma, f(\alpha_1, \ldots, \alpha_n)),\ v,\ u).$$

Reductions can only be made at the top of the stack!

Accept

The shift/reduce parser accepts its input if it is in a state

$$(\ (S, \alpha),\ \epsilon,\ \epsilon).$$

This means that it has read all the input, has empty lookahead, and it managed to rewrite the input to S.

Note that in practice, an EOF symbol is used. Let $\# \notin \Sigma$ be a special EOF symbol.

The shift/reduce parser accepts its input if it is in a state

$$(\ (S, \alpha), \#, \epsilon).$$

Making the Decisions

At each state, the parser has the following choices:

- If the top of the stack contains the right hand side of a rule, it can reduce.
- It it didn't reach end of file, it can shift.

It is possible that more than one reduction is possible. If a reduction is possible, it is still possible to shift. In order to decide, the parser uses the **lookahead**.

A good parser makes its decisions as early as possible, that means with the smallest possible lookahead.

LALR parsing uses a lookahead of at most 1.

Parser Generation Tools/Practical Aspects

There exist many parser generation tools that support attribute grammars. (Yacc, Bison, Maphoon). The attribute functions are usually represent by general C/C^{++} -statements. In the code, `$1,$2,$3, ...` refer to the attributes of the first,second, etc. token on the right hand side.

The notation `$$` refers to the attribute of the token on the left hand side.

A rule of form $A \rightarrow A + B: \ f(x, y, z) = x + z$ is represented by:

```
A -> A + B // $$ = $1 + $3;
```

Parser Generation Tools/Practical Aspects