Parser Specification

A parser can be specified by defining the production rules of the grammar. The left-hand side of a production rule can be used as a nonterminal in (other) production rules of the parser specification. The parser generator constructs a predictive parser, which is ELL(1), and recursive-descent. The attribute grammar is restricted to be L-attributed, meaning that information flow of attributes is strictly left-to-right.

ELL(1)-parsers are an extension of LL(1)-parsers; they are derived from Extended Context-Free Grammars (ECFGs). These grammars use syntactic operators to avoid left-recursive production rules. The syntactic operators specify the alternative and sequential composition of expressions from sub-expressions, and group sub-expressions.

Recursive-descent parsing is a top-down method of syntax analysis, and involves executing a set of recursive procedures to recognise input, with no backtracking. A possibly-recursive procedure is generated for every nonterminal of the designed language.

The parser generator performs a reachability and an availability test on the attributes of the input grammar. The reachability test checks whether the reference to every attribute occurrence, that acts as an argument of a semantic action, is legal. The availability test verifies for every production rule if all synthesised attribute occurrences of the left-hand side nonterminal, and all inherited attribute occurrences of the right-hand side nonterminals, are defined. Moreover, the availability of the input arguments in each semantic action is verified, if the input is parsed from left-to-right.

The generated parsers support syntactic and semantic error recovery. At any time during parsing, the parser maintains a look ahead set of legal tokens. If the next token is illegal, but is a member of the look ahead set, the parser assumes a token to be missing, inserts the missing token, and continues. If the illegal token is not in the look ahead set, the parser assumes the token to be superfluous, deletes it, and continues. The parser continues attribute evaluation after encountering a syntactic error, i.e. the parser keeps verifying the context-sensitive and semantic integrity constraints, expressed by the attributes, even after program text has been deleted or inserted. The algorithms used by the parser generator can be found in [AS90].

Syntactic operators

The following syntactic operators can be used in production rules:

`{ A }`	for grouping a regular expression `A`,
`A | B`	for separating alternative expressions `A` and `B`,
`A B`	for concatenating expressions `A` and `B`,
`A` **`CLOS`**	for zero or more instances of an expression `A`,
`A` **`OPTION`**	for zero or one instance of an expression `A`,

`A` **`SEQ`**	for one or more instances of an expression `A`,
`A` **`CHAIN`** `B`	for a sequence of instances of an expression `A` separated by the expression `B`,
`A` **`LIST`**	for a sequence of instances of an expression `A` separated by a *list_token* (`tListSep`), and
`A` **`PACK`**	for an expression `A` between *open_token* (`tPackLeft`) and *close_token* (`tPackRight`).

Precedence

All operators are left-associative and:

1. the unary operators `OPTION`, `SEQ`, `CLOS`, `LIST` and `PACK` have the highest precedence,
2. the binary operator `CHAIN` has the second highest precedence,
3. concatenation has the third highest precedence, and
4. the alternative separator `|` has the lowest precedence.

Alternative specifications:

`A` **`CLOS`**	can be rewritten as:	`{ A` **`SEQ`** `}` **`OPTION`**
`A` **`CHAIN`** `B`	can be rewritten as:	`A { B A }` **`CLOS`**
	or even:	`A { { B A }` **`SEQ`** `}` **`OPTION`**

Parser attributes

The parser generator allows four kinds of attributes to be used:

- inherited attributes associated with one or more nonterminals,
- synthesised attributes associated with one or more terminals or nonterminals,
- global attributes, and
- local attributes.

Attributes can be specified as arguments of semantic action calls. Inherited attributes should be assigned a value before the associated nonterminal is parsed. Synthesised attributes should be assigned a value after the associated (non)terminal is parsed. Synthesised and inherited attributes may be assigned a value only once. In the scanner specification, only synthesised attributes (of terminals) are defined and used. In the parser specification synthesised attributes of both terminals and nonterminals can be used.

A default action has to be specified for every synthesised parser attribute. The default action assigns a default value to the attribute and is performed only if the parser option *error_recovery* is specified and the associated (non)terminal needs to be inserted in the input of the generated parser.

Some attributes are used in almost every production rule. This can result in many copy actions of the attribute values. In order to prevent these copy actions and to save space, global attributes can be used. Global attributes can be assigned values repeatedly. Moreover, all nonterminals share global attributes. The global attributes

are associated with the total language, and form a shared storage, which can be referred to from any place in the input grammar. Global attributes are not checked for having a value before they are used. Inherited, synthesised, and local attributes are checked, though.
Local attributes are associated with a specific production rule. The local attributes can be used for intermediate calculations. Like global attributes, local attributes can be assigned values repeatedly.

Standard parser actions

See the Application Programming Interface.

Meta-grammar of production rules

The grammar of a language can be defined by one or more production rules. The right-hand side of a production rule is a regular expression of terminals, nonterminals and syntactic operators, extended with action calls. In Figure 8 the meta grammar of the production rule is shown.

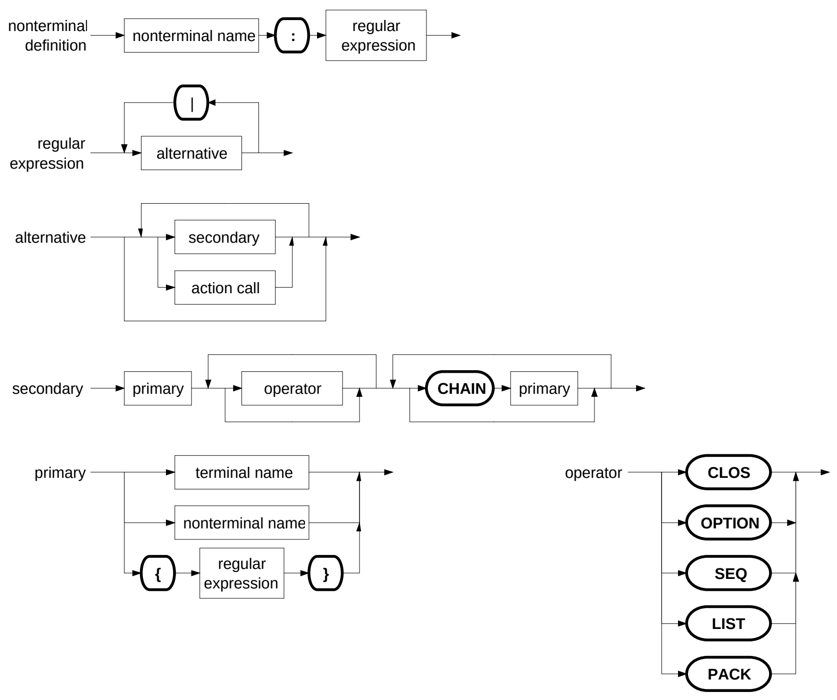


Figure 8: *Meta grammar of the production rule.*

In order to associate the actions with specific positions in the production rules, action calls can be inserted in the production rules. An action call consists of an action name and arguments (i.e. actual parameters), which conform to the formal parameters of the action definition. In Figure 9 the meta grammar of an action call is shown. The position of the action call in the production rule determines when the action is performed. Therefore, action calls are allowed in production rules at every position where (non)terminals are allowed. From the syntactic viewpoint, the action call is considered to be an extra nonterminal having a production rule with an empty right-hand side. A syntactic operator, however, may never follow an action call, as this can give rise to LL(1) conflicts.

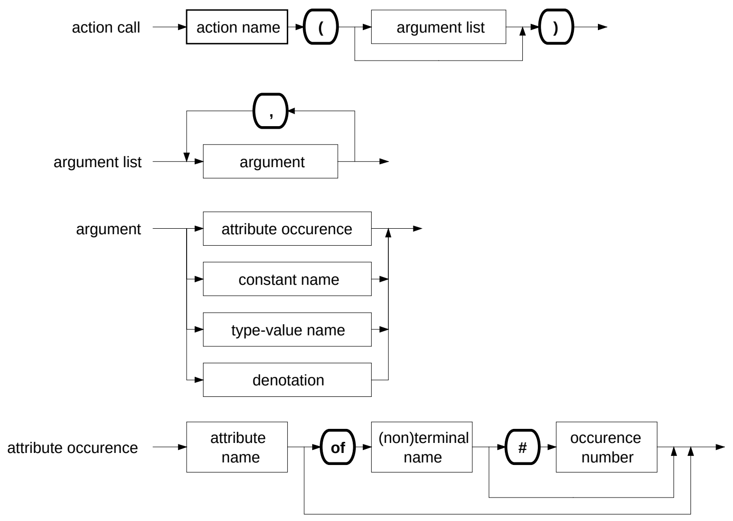


Figure 9: *Meta grammar of the action call.*

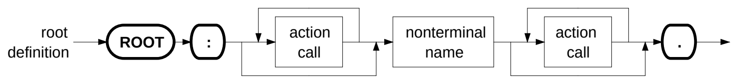


Figure 10: *Meta grammar of the ROOT production rule.*

The start symbol of a grammar can be specified in the ROOT production rule. In Figure 10 the meta grammar of the ROOT production rule is shown. Actions to be performed before and after parsing the start symbol (i.e. the entire grammar) can be specified.
In order to indicate that an action call belongs to a specific (non)terminal, braces are used. For example, in the production rule:

```
A:
    B CHAIN C
        action1()
        action2().
```

`action1()` and `action2()` are performed once after `B CHAIN C` is parsed. If `action1()` has to be performed every time after *C* is parsed, the production rule should be:

```
A:
   B CHAIN
   {
      C
         action1()
   }
      action2().
```

In summary, if the generated parser encounters a terminal, the existence of that terminal is checked. If the generated parser encounters a nonterminal, the right-hand side of the production rule corresponding to that nonterminal is parsed. Finally, if an action call is encountered, it is performed.
An attribute occurrence is a combination of a (non)terminal occurrence and its attribute. In order to distinguish different occurrences of the same (non)terminal in a production rule, the occurrences are assigned unique numbers. An attribute occurrence in a production rule is specified as follows:
`a` **`of`** `S#i`, where:

- `a` is the attribute name,
- `S` is the (non)terminal name, and
- `i` is the occurrence number of (non)terminal `S` in the production rule.

The occurrence number of the left-hand side nonterminal is 0. The occurrence numbers of (non)terminals on the right-hand side start with 1. If a (non)terminal occurs only once in the production rule, no occurrence number has to be specified. In that case, the default occurrence number of the left-hand side is 0, and the default occurrence number of right-hand side (non)terminals is 1.

Parser options

The following options can be selected:

List names	print reserved and defined names,
List symbols	print terminal and nonterminal symbols,
List actions	print standard (API) and user-defined parser actions,
*List attributes**	print inherited, synthesised and global parser attribute definitions,
*List locals**	print local attributes definitions,
List numbers	print attribute occurrence numbers,
*List action calls**	print action calls in production rules,
*Error recovery**	add error recovery statements to the generated syntactic procedures,
*LL(1) test**	perform LL(1)-test on input grammar before parser generation,
Empty rules	print EMPTY value(s) of production rules (implies *LL(1) test*),

Reduced rules	print whether production rules are reduced (implies *LL(1) test*),
First sets	print FIRST sets of production rules (implies *LL(1) test*),
Follow sets	print FOLLOW sets of production rules (implies *LL(1) test*),
Last sets	print LAST sets of production rules (implies *LL(1) test*),
Dirsets	print DIRSETS of production rules (implies *LL(1) test*),
*Dependencies**	perform reachability/availability test on input grammar before parser generation,
*Parser**	generate the parser if no errors occur,
*Program**	generate a main program,
Module	generate a function module.

The options marked with a * are default active. All output is printed to the *.lst* file.

Combining syntactic operators and attributes

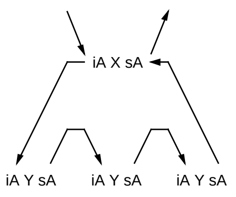
The syntactic operators **`OPTION`**, **`SEQ`**, **`CLOS`**, **`CHAIN`**, and **`LIST`** allow a compact and clear input grammar. However, these operators can cause problems in attribute evaluations. Using local attributes can circumvent these problems. Below, these problems, and solutions to these problems, for each of the syntactic operators, are outlined.

Problem 1: `X: Y` **`OPTION`**`.`
Parsing of the (non)terminal *Y* is optional. The inherited attributes of *Y* are only assigned values when *Y* occurs in the input. The synthesised attributes of *Y* may not be referred to if *Y* does not occur. In that case dummy values should be assigned to the synthesised attributes of *Y*. Solution:

```
X ( LOCAL h: a_type ):
        assign(h, iA of X)
    {
            assign(iA of Y, h)
        Y
            assign(h, sA of Y)
    } OPTION
        assign(sA of X, h).
```

*Problem 2***:** `X: Y` **`SEQ`**`.`
The attributes of an occurrence of *Y* should be copied to the next occurrence of *Y*. Further, the inherited attributes of the first occurrence of *Y* are copied from the inherited attributes of the nonterminal *X*, and the last occurrence of *Y* copies its synthesised attribute values to the synthesised attributes of *X*. Solution:

```
X ( LOCAL h: a_type ):
        assign(h, iA of X)
    {
            assign(iA of Y, h)
        Y
```



```
            assign(h, sA of Y)
    } SEQ
        assign(sA of X, h).
```

Problem 3: `X: Y CLOS.`
This problem is a combination of problems 1 and 2: zero or more occurrences of *Y* are allowed. Solution:

```
X ( LOCAL h: a_type ):
        assign(h, iA of X)
    {
            assign(iA of Y, h)
        Y
            assign(h, sA of Y)
    } CLOS
        assign(sA of X, h).
```

Problem 4: `X: Y LIST.`
In this case, *Y* is covered by problem 2, and the terminal *tListSep* is covered by problem 3. Generally, the terminal *tListSep* will not have any attributes. Solution:

```
X ( LOCAL h: a_type ):
        assign(h, iA of X)
    {
            assign(iA of Y, h)
        Y
            assign(h, sA of Y)
    } LIST
        assign(sA of X, h).
```

Problem 5: `X: Y CHAIN Z.`
Problem 2 applies to (non)terminal *Y*, and problem 3 applies to (non) terminal *Z* Solution:

```
X ( LOCAL h1: a_type1; h2: a_type2 ):
        assign(h1, iA1 of X)
        assign(h2, iA2 of X)
    {
            assign(iA of Y, h1)
        Y
            assign(h1, sA of Y)
    } CHAIN
    {
            assign(iA of Z, h2)
        Z
```

```
            assign(h2, sA of Z)
    }
          assign(sA1 of X, h1)
          assign(sA2 of X, h2).
```

Note that local attributes can also be used to copy attribute values from occurrences of `Y` to occurrences of `Z`, and vice versa, keeping left-to-right one-pass in mind. Instead of using local attributes, global attributes and attributes of the left-hand side nonterminal could be used. However, global attributes cannot be used in recursive production rules, and attributes of left-hand side nonterminals would have to be assigned values more than once, which conflicts with the definition of synthesised and inherited attributes.