# Language Syntax

We take a layered approach to understanding L&O"s syntax. The lowest level is the character level; then there is the lexical or token level, the parse level and finally the well typed formulae level. Each of these levels identifies a different ‘level of abstraction’; each focusses on a different aspect of a well formed L&O program.

This chapter is concerned with the lower three of these abstractions: characters, tokens and parse trees. This progressively moves up from the physical character of files containing L&O programs to a tree-like representation of the contents of a L&O source file. In later chapters, we focus on the subset of parse trees that can be given a well defined meaning: i.e., are well typed and form coherent units of execution.

## Characters and lexical syntax

### Unicode characters

L&O uses the Unicode character encoding system . More specifically, the L&O run-time uses Unicode characters internally to represent symbols and the L&O compiler and run-time engine interpret streams of UTF-8 bytes and UTF-16 words as Unicode. Identifiers in L&O programs as well as data processed by L&O programs are assumed to be Unicode based.

A L&O language processor assumes that the input is encoded in UTF-8.

However, all of the the characters used within the language definition of L&O – such as the built-in operators and syntactic features – are contained within the ASCII subset of the Unicode character set. Thus, L&O can be used in an ASCII-based environment.

#### Character Categories

The Unicode consortium has identified a number of character catagories[[1]](#footnote-25). A character category distinguishes the typical role of a character – numeric digit, punctuation mark, regular text – in a language independent way.

L&O uses these character categories to classify characters for the the purposes of discriminating identifiers and number values occurring in the program source text. The purpose of this is to permit non-English programmers to use non-English identifiers in the text of the program.

The character categories used by L&O are:

## Tokens

It is required that the input of a L&O language processor is first partitioned into contiguous sequences of characters called *tokens*. There are several different kinds of tokens; corresponding to the identifiers, symbols, character literals, string literals and punctuation marks necessary to correctly parse a L&O program.

We use L&O’s own grammar notation to describe the tokenization and parsing rules for L&O. Apart from demonstrating the power of L&O’s grammar notation – which is modelled on Prolog’s DCG grammars – it also demonstrates that we can eat our own dog food!

In the productions for the tokenizer listed below we assume that, in an actual tokenizer, all the productions for a given non-terminal are collected together – as is required by L&O’s grammar. However, for explanatory purposes, the grammar rules for some of the non-terminals are distributed thoughout the text below. The tokenizer rules depend on the type defined below:

tokType ::= ID(symbol) -- Identifier  
 | IN(integer) -- Integer literal  
 | FT(float) -- Floating point literal  
 | ST(string) -- String literal  
 | SY(string) -- Symbol  
 | CH(char) -- Character literal  
 | LPAR | RPAR | LBRA | RBRA  
 | LBRCE | RBRCE -- Punctuation  
 | COMMA | CONS | TERM  
 | EOF. -- End of file

This type definition defines the legal kinds of tokens that can be expected; and also forms the basis of the stream processed at the next higher level in the L&O language specification: namely the syntactic grammar.

### Comments and whitespace

Tokens in a L&O source text may separated by zero or more *comments* and/or white space text – some pairs of tokens *require* some intervening space or comments for proper recognition. For example, a number following an identifier requires at least one white space character; otherwise the rules for identifier would ‘swallow’ the number token. White space characters and comments may be used for the purposes of recognizing tokens; but must otherwise be discarded by a language processor.

There are two styles of comment in L&O source texts – *line* comments and *block* comments.

#### Line comment

A line comment consists of the characters -- i.e., two hyphen characters and a whitespace character, followed by all the characters up to the next new-line character or end-of-file which ever is first.

#### Block comment

A block comment consists of the characters /\* followed by any characters and is terminated by the characters \*/

The L&O tokeniser rules for white space and comments are:

whiteSpace:[integer]{}.  
whiteSpace(X):-\_\_isZsChar(X).  
whiteSpace(X):-\_\_isZlChar(X).  
whiteSpace(X):-\_\_isZpChar(X).  
  
comments:(integer,integer)-->string.  
comments(Lno,Lx) --> comment(Lno,Ly)!, comments(Ly,Lx).  
comments(Lno,Lno) --> "".  
  
comment:(integer,integer)-->string.  
comment(Lno,Lx) --> "-- ",lineComment(Lno,Lx).  
comment(Lno,Lx) --> "--\t",lineComment(Lno,Lx).  
comment(Lno,Lx) --> "/\*",bodyComment(Lno,Lx).  
comment(Lno,Lno+1) --> "\n".  
comment(Lno,Lno) --> [X],{\_\_isZsChar(X)}. -- ignore space  
  
lineComment:(integer,integer)-->string.  
lineComment(Lno,Lno+1) --> [X],{\_\_isZlChar(X)}. -- new line  
lineComment(Lno,Lx) --> [X],{\+\_\_isZlChar(X)},  
 lineComment(Lno,Lx).  
  
bodyComment:(integer,integer)-->string.  
bodyComment(Lno,Lno) --> "\*/".  
bodyComment(Lno,Lx) --> [X],{\_\_isZlChar(X)},  
 bodyComment(Lno+1,Lx).  
bodyComment(Lno,Lx) --> [X],bodyComment(Lno,Lx).

### Identifiers

Identifiers serve many purposes within a L&O program: to identify variables and parameters, to identify types, even to identify syntactic operators.

There are several classes of identifier, corresponding to how they are written: identifiers written using normal identifier rules, operators – which are often written with graphical characters – and quoted identifiers which are written like string literals but with single quotes.

L&O identifiers are modelled on the standard Unicode identifier syntax; which in turn is a generalization of the common notation for identifier syntax found in many programming languages.

The basic rule for identifiers is that they have an initial start letter – which is either a letter character, or a character which could be used in a word or an underscore character – followed by a sequence of zero or more characters from a somewhat extended set – including the digit characters.

What makes this style of identifier ‘different’ is that the Unicode standard defines many thousands of ‘letter’ character; this allows identifiers to be written in non roman scripts – such as Kanji for example.

The following L&O grammar rules capture the definition of an identifier:

tok:(tokType)-->string. -- a grammar that returns a token  
  
/\* An identifier is an idStart character,   
 followed by a sequence of idChar \*/  
tok(ID(I)) --> idStart(X), idChar(C)\*C^N, I=implode([X,..N]).  
\dots  
idStart:(integer)-->string.  
idStart(X) --> [X],{\_\_isLetterChar(X)}.  
idStart(0c\_) --> "\_".  
idStart(0c\+ff3f;) --> "\\+ff3f;". -- full width low line  
   
idChar:(integer)-->string.  
idChar(X) --> idStart(X).  
idChar(X) --> [X],{\_\_isNdChar(X)}.  
idChar(X) --> [X],{\_\_isMnChar(X)}.  
idChar(X) --> [X],{\_\_isMcChar(X)}.  
idChar(X) --> [X],{\_\_isPcChar(X)}.  
idChar(X) --> [X],{\_\_isCfChar(X)}.

The idStart rule defines those characters that may start an identifier, and the idChar rule identifies those characters that may continue an identifier. Together, these rules state that any ‘letter’ character may start an identifier, and letters and numbers may follow this initial letter.

The grammar condition

idChar(C)\*C^N

matches a sequence of zero or more idChars, and puts the resulting list in the variable N. See Section  for a more complete explanation.

Quoted identifiers are written as a sequence of characters – not including new-line or other control characters – surrounded by ' marks. More specifically, quoted identifiers are written as a sequence of character references (see Section  above) surrounded by ' characters.

For example,

'a symbol'

is the identifier with name a symbol.

The grammar rule for quoted identifier tokens is

tok(ID(L)) --> "'", strChar(C)\*C^L,"'".

Quoted identifiers are especially useful when integrating with other languages. For example, a record structure arising from parsing a CVS file may use field names that are constructed from the CVS input; in which case, being able to use quoted identifiers becomes crucial.

### Characters

There is no separate type for characters. This is partly due to the fact that there is no single mapping between strings and characters in Unicode. If it is necessary to process characters, as opposed to strings, then one must use the Unicode concept of *codepoint* – which is actually an integer.

However, there is a special notation for codepoint: a leading 0c in front of a *character reference* denotes the integer value of the codepoint corresponding to the character.

For example, the CodePoint corresponding to new-line character is written:

0c\n

The grammar rule for character literals is:

tok(IN(ch)) --> "0c", strChar(ch).

where strChar is the production to defines a legal character that may occur in a string value.

#### Character reference

Character references are used in several places in L&O’s syntax: within string literals, identifiers and codepoint literals. There are also several special forms of character reference. The common Unix names for characters such as \n for new-line are recognized, as is a special notation for entering arbitrary Unicode characters.

There are three categories of character references: characters which do not need escaping, characters that are represented using a backslash escape, and characters denoted by their hexadecimal character code.

For example the string "string" has the following code points in it:

0cs 0ct 0cr 0ci 0cn 0cg

A string containing just a new-line character is:

"\n"

and a string containing the Unicode sentinel character would be denoted:

"\+fffe;"

The rules for character references are:

/\* Special character sequence following a back-slash  
\* in a symbol or literal string  
\*/  
strChar:(integer)-->string.  
strChar(0ca) --> "\\a".  
strChar(0cb) --> "\\b".  
strChar(0cd) --> "\\d".  
strChar(0ce) --> "\\e".  
strChar(0cf) --> "\\t".  
strChar(0cn) --> "\\n".  
strChar(0cr) --> "\\r".  
strChar(0ct) --> "\\t".  
strChar(0cv) --> "\\v".  
strChar(0c') --> "\\'".  
strChar(0c") --> "\\"".  
strChar(0c`) --> "\\`".  
strChar(0c\\) --> "\\\\".  
strChar(Cr) --> "\\+",hexSeq(0,C),";",Cr=\_\_charOf(C).  
strChar(X) --> "\\", [X].  
strChar(X) --> [X],{\+\_\_isCcChar(X)}.  
...  
hexDig:(integer)-->string.  
hexDig(\_\_digitCode(X)) --> [X],{\_\_isNdChar(X)}.  
hexDig(10) --> "a". hexDig(10) --> "A".  
hexDig(11) --> "b". hexDig(11) --> "B".  
hexDig(12) --> "c". hexDig(12) --> "C".  
hexDig(13) --> "d". hexDig(13) --> "D".  
hexDig(14) --> "e". hexDig(14) --> "E".  
hexDig(15) --> "f". hexDig(15) --> "F".  
  
hexSeq:(integer,integer)-->string.  
hexSeq(I,N) --> hexDig(X)!,hexSeq(N\*16+X,N).  
hexSeq(N,N) --> "".

The rules for strChar above may be a little confusing, because of the rules for quoting characters in strings. The required sequence of characters to represent a new-line character is a back-slash followed by an n: \n". The string that denotes this in the grammar rule requires two backslashes – since a backslash in a string must itself be represented by two backslash characters. This is especially spectacular in the case of the strChar rule for the backslash character: this requires two backslashes in sequence and the string that denotes this sequence in the grammar rule consists of four backslash characters!

### String literals

String literals are written as a sequence of character references (see ) – not including new-line or paragraph separator characters – surrounded by " marks.

The reason for not permitting new-lines to occur in string literals is that that enables a particularly silly kind of syntax error to be picked up easily: a missing string quote will always generate a syntax error at the end of the line. The restriction does not affect the possible string literals, as it is always possible to use \n to indicate a new-line character, and the L&O compiler concatenates sequences of string literals into a single string literal.

One benefit of the automatic string merge feature is that when a program has a lengthy string embedded in it it can be formatted both for display and for program tidiness.

The grammar rule for string literals is very similar to the production for symbols. Note however, that string literals are interpreted as synonyms for lists of chars.

tok(ST(L)) --> "\"", strChar(C)\*C^L,"\"".

### Interpolated Strings

A particular variant of string literal is the *interpolated string*.

#### Numbers

L&O numbers are built from the \_\_isNdChar character class. This class of characters includes many digit characters; all of which share the semantic property that they can be interpreted as decimal digits.

L&O distinguishes integer literals (and values) from floating point literals and values; these are *not* generally substitutable for each other.

tok(Nm) --> [X],{\_\_isNdChar(X)},  
 numberSeq(\_\_digitCode(X),I),  
 ( fraction(F),exponent(E),  
 Nm = FT(n2float(I)+F)\*n2float(E)  
 | Nm = IN(I)).  
  
tok(IN(N)) --> "0x", hexSeq(0,N).  
tok(IN(C)) --> "0c", strChar(C).  
  
numberSeq:(integer,integer)-->string.  
numberSeq(I,N) --> [D],{\_\_isNdChar(D)!},  
 numberSeq(I\*10+\_\_digitCode(D),N).  
numberSeq(N,N) --> "".  
  
fraction:(float)-->string.  
fraction(F) --> ".", [C],{\_\_isNdChar(C)},  
 frSeq(10,1/\_\_digitCode(C),F).  
fraction(0) --> "".  
  
frSeq:(float,float,float)-->string.  
frSeq(X,F,R) --> [C],{\_\_isNdChar(C)},  
 frSeq(X\*10,F+\_\_digitCode(C)/X,R).  
frSeq(X,F,F) --> "".  
  
exponent:(float)-->string.  
exponent(Ex) --> "E-", numberSeq(0,X),Ex = 10 \*\* (-X).  
exponent(Ex) --> "e-", numberSeq(0,X),Ex=10 \*\* (-X).  
exponent(Ex) --> "E", numberSeq(0,X),Ex=10 \*\* X.  
exponent(Ex) --> "e", numberSeq(0,X),Ex=10 \*\* X.  
exponent(1) --> "".

Apart from the normal decimal notation for numbers, L&O supports two additional notations: the hexadecimal notation and the character code notation. The hexadecimal number notation simply consists of a leading 0x followed by the hexadecimal digits of the number. All hexadecimal numbers are integral.

Notice that there is no definition of a negative numeric literal as a single token. Literal negative numbers are handled at a slightly higher level in the L&O language processing: the leading - character is interpreted as a prefix unary operator signifying arithmetic negation.

### Operators and punctuation marks

L&O uses a number of special punctuation marks to signify specific syntactic features – such as lists, theta expressions and so on.

tok(LPAR) --> "(".  
tok(RPAR) --> ")".  
tok(LBRA) --> "[".  
tok(RBRA) --> "]".  
tok(LBRCE) --> "{".  
tok(RBRCE) --> "}".  
tok(CONS) --> ",..".  
tok(COMMA) --> ",".  
tok(ID(". ")) --> ".", [X],{ \_\_isZsChar(X)  
 | \_\_isZlChar(X)  
 | \_\_isZsChar(X)  
 | \_\_isCcChar(X)}.

Note that the symbol consists of a period followed by any kind of white space character. This distinguishes it from other uses of the period; such as class body definition operator .. or within a floating point number.

In addition to these punctuation marks, L&O uses a number of symbols for the prefined operators:

tok(ID("||") --> "||".  
tok(ID("|") --> "|".  
tok(ID("::") --> "::".  
tok(ID("::=") --> "::=".  
tok(ID(":=") --> ":=".  
tok(ID(":-") --> ":-".  
tok(ID(":--") --> ":--".  
tok(ID(":") --> ":".  
tok(ID("-->") --> "-->".  
tok(ID("->") --> "->".  
tok(ID("~") --> "~".  
tok(ID("<~") --> "<~".  
tok(ID("\^") --> "\^".  
tok(ID("..") --> "..".  
tok(ID(".")) --> ".".  
tok(ID("?") --> "?".  
tok(ID("!") --> "!".  
tok(ID("==") --> "==".  
tok(ID("=") --> "=".  
tok(ID("\\=") --> "\\=".  
tok(ID("!=") --> "!=".  
tok(ID("\$") --> "\$".  
tok(ID("\\+") --> "\\+".  
tok(ID("+") --> "+".  
tok(ID("\*>") --> "\*>".  
tok(ID("\*\*") --> "\*\*".  
tok(ID("\*") --> "\*".  
tok(ID("/") --> "/".  
tok(ID("\\/") --> "\\/".  
tok(ID("/\\") --> "/\\".  
tok(ID("\\") --> "\\".  
tok(ID("<>") --> "<>".  
tok(ID("=>") --> "=>".  
tok(ID("<=") --> "<=".  
tok(ID(">=") --> ">=".  
tok(ID("=<") --> "=<".  
tok(ID(".=") --> ".=".  
tok(ID("%%") --> "%%".  
tok(ID("\#") --> "\#".  
tok(ID("@") --> "@".  
tok(ID("@@") --> "@@".  
tok(ID(">") --> ">".  
tok(ID("<") --> "<".  
tok(ID("-") --> "-".  
tok(ID(";") --> ";".

Note that all the graphic identifiers are ‘spelled out’ here. The reason that we can do so is that L&O completely specifies all the permitted operators. In turn, explicitly enumerating all the possible graphical tokens significantly improves the programmer’s experience of writing L&O programs – it is not necessary to separate sequences of characters such as:

A\*-B

as the L&O parser can correctly parse this as:

\*(A,-(B))

without requiring an explicit space between the \* and - characters.

Any character not referred to explicitly here, and not referred to in any of the token rules above is not considered a legal character in a L&O program.

### A Standard Tokenizer

Based on the productions for the tok introduces above, we can define a complete L&O tokenizer that consumes a string (i.e., a list of characters) and produces a list of tokens.

Note that (as used in a compiler) a realistic tokenizer would collect additional information as it tokenizes the stream; in particular, the line number where each token occurs.

Our tokenizer rules can be easily extended to count line numbers and associate a line number and source file with each token. We use an auxilliary constructor to wrap each token with this information:

TokenType ::= tk(tokType,integer).

The production below for goTokens also show how the treatment of white space and comments meshes with the treatment of the individual types of token:

goTokens:(list[(tokType,integer)])-->string.  
goTokens([tk(Tok,Ln),..L],Lno) -->  
 comment(Lno,Ln), tok(Tok), goTokens(L,Ln).  
goTokens([],Lno) --> comment(Lno,\_).

This requires a slight modification to the rules for comment handling, and furthermore assumes (correctly) that a L&O token cannot span across multiple lines of input.

## Operator Grammar

The grammar of L&O specifies the rules for sequencing tokens into syntactically reasonable structures. However, a grammar parser is not itself capable of determining *semantically* meaningful program structures – that analysis requires the type inferencing system and other phases of a language processor.

The grammar of L&O is based on an *operator precedence grammar*. Although they might not be aware of it, most programmers are quite familiar with operator precedence grammars – it is typically the grammar used for arithmetic expressions in regular programming languages. In L&O – as in Prolog – we extend the use of operator-style grammars to cover the whole language.

An operator grammar allows us to write expressions like:

X \* Y + X / Y

and to know that this means the equivalent of:

(X \* Y) + (X / Y)

or more specifically:

+(\*(X, Y), /(X, Y))

I.e., an operator grammar allows us to write operators between arguments instead of before them, and an operator precedence grammar allows us to avoid using parentheses in many common situations.

It is not necessary to restrict the scope of an operator grammar to arithmetic expressions – a fact used by the early originators of the and Prolog programming languages. We can also use an operator grammar for the whole of a programming language. For example, in L&O, an equation such as:

app([E,..X],Y) => [E,..app(X,Y)]

can be interpreted – by treating => as an operator – as:

=>(app([E,..X],Y),[E,..app(X,Y)])

A somewhat more complicated example allows us to interpret a conditional equation in terms of operators:

sort([E,..X])::split(E,X,A,B) => sort(A)<>[E]<>sort(B)

as

=>(::(sort([E,..X]),split(E,X,A,B)),:-(<>(sort(A),<>([E],sort(B)))))

Here, we have relied on the fact that =>, <> and :: are *infix* operators, in addition to the ‘normal’ infix operators: "+" and - operators.

The major benefit of this sleight of hand is simplicity – it allows us to focus on the logical structure of a program fragment rather than the inessential details.

The major demerit is that, occasionally, syntax errors can be somewhat harder to interpret. Furthermore, a syntax error at this level tends to create a cascade of spurious error messages as the parser attempts to recover from the incorrect input.

The output of parsing a L&O text using the operator precedence grammar is a *parse tree*. This parse tree represents the syntactic structure of the program text in an abstract way.

### Standard operators

The standard operators in L&O are listed in order of priority below in Section . Each operator has a priority, associativity and a role.

The priority of an operator is the indication of the ‘importance’ of the operator: the higher the priority the nearer the top of the abstract syntax tree the corresponding structure will be. Priorities are numbers in the range 1..2000; by convention, priorities in the range 1..899 refer to entities that normally take the role of expressions, priorities in the range 900..1000 refer to predicates and predicate-level connectives and priorities in the range 1001..2000 refer to entries that have a statement or program level interpretation. The comma and the semi-colon operators are the only one with a priority of exactly 1000.

##### Representing operators

In our standard definition of L&O grammar, we represent the different operators as clauses in the preOp, infOp and postOp relations. Each clause in the infOp predicate takes the form:

infOp:(string,integer,integer,integer){}.  
...  
infOp("\*",720,720,719).

where "\*" is a left associative infix operator of priority 720. This is encoded in the three numbers: the first number represents the priority of the term expected on the left of the \* expression, the second represents the priority of the \* expression itself and the third number represents the expected priority of expressions to the right of the \* operator. Since \* is left associative, this implies that a \* expression can appear to the left, i.e., the expected priority on the left is the same as the actual priority of the \* operator. For example, an expression such as

A\*B\*C

is parsed as though it were

(A\*B)\*C

A non-associative operator, such as !=, is represented by a clause such as:

infOp("!=",899,900,899). -- not equal predicate

The Left and Right expected priorities are both less than the priority of != itself; which implies that multiple occurrences of them must be correctly parenthesized. An expression such as

a != b != c

occuring in the program text would not be valid.

Prefix operators are represented in a similar way. The prefix operator - which is not associative and has priority 300 is represented by a clause in the preOp relation:

preOp:(string,integer,integer){}.  
...  
preOp("-",300,299).

The first number is the priority of the - operator as a prefix operator, and the second number is the priority of the expected term that follows the operator. Note that the *prefix* priority of an operator can be different to its *infix* priority; however, if an operator’s postfix priority is different to its infix priority then this may cause ambiguities. All of L&O’s operators have consistent infix and postfix priorities.

Postfix operators are represented as instances of the postOp relation. The postfix operator ! which is not associative and has priority 905 is represented by a clause in the postOp relation:

postOp:(string,integer,integer){}.  
...  
postOp("!",904,905);

In this case, the first number is the priority of the expected term to the left of the ! operator as a postfix operator, and the second number is the priority of the ! expression itself.

### Parsing analysis

It is most convenient to consider parsing analysis as generating *parse trees* from streams of tokens. A parse tree is a tree-like structure that reflects operators such as "+"; as well as syntactic constructions such as lists, tuples and applicative expressions.

##### Abstract syntax type definition

Parse trees are expressed in terms of an *abstract syntax*. An abstract syntax is a notation which has operators for core elements – such as characters, strings, symbols and numbers – and a way of combining these into applicative forms – such as lists, tuples and function application. The type definition of the L&O abstract syntax is:

absTree ::=  
 I(string) -- identifier  
 | INT(integer) -- integer literal  
 | FLT(float) -- floating point literal  
 | STR(list[char]) -- string literal  
 | APP(absTree,absTree) -- applicative  
 | TPL(string,list[absTree]) -- tuple  
 .

where the I, INT, FLT and ST forms correspond to identifiers, integers, floating point and literal strings respectively. Note that the type definition used here is simplified compared to that used in a compiler: a ‘real’ version would include line number information that would help to identify the location of terms for reporting errors and for support for run-time debugging. We omit this for the sake of clarity.

The TPL form is used for syntactic forms such as:

(A,B,C)

which is represented as:

TPL("()",[I("A"),I("B"),I("C")])

and

[1,2,3]

which becomes:

TPL("[]",[INT(1),INT(2),INT(3)])

I.e., both are tuples, but with a different bracketing.

TPL is also used to represent the contents of a {} term:

{a(). b().}

which is represented as:

TPL("{}",[APP(I("a"),TPL("()",[])),APP(I("b"),TPL("()",[]))])

##### Parsing over streams of tokens

Note that the grammar rules for parsing L&O are expressed in terms of streams of tokType expressions rather than streams of characters. This is not a problem for the grammar formalism itself as the grammar notation is inherently polymorphic. For example, the grammar rule for a number in the token stream is:

term0:(absTree)-->list[tokType].  
term0(INT(N)) --> [IN(N)].

The basic rules for parsing L&O programs revolve around the notion of a *primitive* expression and an *operator* expression. These distinctions have nothing to do with the semantics of L&O programs; they only relate to the syntactic relationships of elements of the language. In Chapter  we provide an analysis of the elements of L&O that more closely relates to the meaning of a program.

As with the standard rules for L&O tokens, we give the rules for parsing sequences of tokens into parse trees as L&O grammar rules. For readability, we assume the input is parsed as a sequence of tokType entries (see section ), rather than the more realistic tokenType entries which carry line number information.

#### Primitive parse tree

A primitive parse tree can be a literal (such as a number, symbol, character, string or regular identifier), an applicative expression (such as a function application, or a rule head) or a bracketted expression (such as a list, or parenthesised expression.

##### Simple literals

The main grammar production that corresponds to the primitive expression is term0. The first few cases of this are straightforward, and correspond to occurrences of simple literal values in the token stream:

term0:(absTree)-->list[tokType].  
term0(STR(S)) --> [ST(s)]. -- A string literal  
term0(INT(N)) --> [IN(N)]. -- An integer literal  
term0(FLT(N)) --> [FT(N)]. -- A floating point literal

Mention interpolation

##### Identifiers and applicative expressions

An identifier may occur by itself, as in an occurrence of a variable, or it may signal an applicative expression, as in a function application. The various rules that capture these cases are amongst the most complicated in the entire L&O grammar. In part, this is to allow the grammar to parse expressions such as:

f(X){  
 X.Ok() -> stdout.outLine("Ok")  
}

The core grammar rule for applicative expressions uses term00 to express the rules for the *function* part of an applicative expression:

term0(T) --> term00(L), termArgs(L,T).

and termArgs to capture the possible forms of arguments – including none.

There are two main rules for term00 – the identifier rule and the parenthesised expression rule. The first rule says that an identifier is a term00 expression, provided that it is not also a standard operator. I.e., it isn’t one of the symbols referred to in Table .

term00:(absTree)-->list[tokType].  
term00(I(S)) --> [ID(S)], {\+isOperator(S)}.  
term00(T) --> parenTerm(T).

The termArgs grammar production either encounters an opening parenthesis – in which case the arguments of an applicative expression are parsed – or not – in which case the leading ‘function symbol’ is interpreted as is.

The rules for termArgs are:

termArgs:(absTree,absTree)-->list[tokType].  
termArgs(Pref,Term) --> parenTerm(Args),  
 termArgs(APP(Pref,Args),Term).  
termArgs(Pref,Term) --> [ID("."),ID(Fld)],  
 termArgs(APP(I("."),TPL("()",[Pref,I(Fld)])),Term).  
termArgs(T,T)-->[].

where parenTerm decribes the various forms of parenthesized terms.

In effect, an applicative term consists of a ‘function’ applied to a sequence of expressions. This rule is iterative, allowing expressions of the form:

f[A](B,C)

whose parse tree representation is:

APP(APP(I("f"),TPL("[]",[I("A")])),TPL("()",[I("B"),I("C")]))

There are other kinds of APP terms, arising from operator expressions, in addition to expressions using termArgs production.

Note that we have a special rule for dealing with the dot operator which enforces the requirement that the right hand side must be an identifier. This reflects the fact that method access in an object is always tightly bound: an expression such as

O.f(A,B)

is parsed as though it were:

(O.f)(A,B)

##### Parenthesised expressions

Parenthesised expressions are enclosed by bracket characters. There are three such groups of characters – parentheses () which indicate tuples as well as operator overriding, square brackets [] which indicate list expressions and braces {} which typically indicate program structure such as theta expressions and classes.

parenTerm:[absTree-]-->list[tokType];  
parenTerm(TPL("()",[])) --> [LPAR,RPAR].  
parenTerm(TPL("()",deComma(Els)) --> [LPAR],term(Els,2000),[RPAR].  
parenTerm(TPL("[]",[])) --> [LBRA,RBRA].  
parenTerm(TPL("[]",deCons(Els))) --> [LBRA], term(Els,2000), [RBRA].  
parenTerm(TPL("{}",[])) --> [LBRCE,RBRCE].  
parenTerm(TPL("{}",Els)) --> [LBRCE],terms(Els),[RBRCE].  
  
terms([T|Els]) --> term(T,2000), [TERM], terms(Els).  
terms([]) --> [].

##### List expressions

The two functions deComma and deCons implement the conversion from comma-terms to lists of elements: the only difference between them is whether the list ,.. notation is legal or not:

deComma:(absTree)=>list[absTree].  
deComma(APP(I(","),TPL("()",[L,R]))) => [L,..deComma(R)].  
deComma(T) => [T].

The rules for deCons implements the list notation. A L&O list expression is a sequence of terms separated by COMMAs, and enclosed in square brackets. If the last element of a list expression is separated by a ,.. (CONS) token then it denotes the remainder of the list rather than the last element.

deCons:(absTree)=>list[absTree].  
deCons(APP(I(","),TPL("()",[L,R]))) => [L,..deCons(R)].  
deCons(APP(I(",.."),TPL("()",[L,R]))) => [L,..R].  
deCons(T) => [T].

The abstract syntax of a list literal is based on the ,.. applicative expression. Thus the parse tree corresponding to the list literal [A] is:

APP(I(",.."),TPL("()",[I("A"),I("[]")]))

Note that this refers only to the abstract parse tree representation of list pairs; the run-time representation of a list such as this may be considerably more succinct.

#### Operator expression

An operator expression can be an infix, prefix or a postfix operator expression. Most operators are infix; however it is possible for an operator to be simultaneously an infix and/or a prefix and/or a postfix operator. The grammar requires some disambiguation in the case that an operator is both infix and postfix.

Our productions for operator expressions are split into two fundamental cases: prefix expressions – handled by the termLeft grammar rules – and infix and postfix expressions – handled by the termRight grammar rules.

##### Prefix operator expressions

A prefix operator expression consists of a prefix operator, followed by a term. The priority of the prefix operator should not be greater than the ‘allowed’ priority; and that the expected priority of the term that follows the prefix operator is based on the priority of the prefix operator itself.

The termLeft grammar does not report an error if it encounters an out-of-range prefix operator as the termLeft production may have been invoked recursively; and the prefix operator expression *may* have significance at an outer level. We rely on backtracking within the parser to resolve this particular conflict.

termLeft:(absTree,integer,integer)-->list[tokType].  
  
termLeft(APP(I(Op),TPL("()",[Right])),P,Oprior) -->  
 [ID(Op)],  
 {isPreOp(Op,Oprior,OrPrior), P>=Oprior},  
 term(Right,OrPrior).

The second rule for termLeft defaults to the primitive term expression in the case that the lead token is not a prefix operator:

termLeft(Term,P,0) --> term0(Term).

##### Infix and postfix operator expressions

The termRight grammar production defines how infix and postfix operators are handled.

The ‘input’ to termRight includes the term encountered to the left of the infix or postfix operator. If the next token is an infix operator, then the right hand side term is parsed – with appropriate expected priorities – and we recursively look again for further infix and/or postfix operators:

termRight:(absTree,integer,integer,integer,absTree)-->  
 list[tokType].  
  
termRight(Left,prior,lprior,aprior,Term) -->  
 [ID(Op)],  
 { isInfOp(Op,L,O,R),O=<prior,L>=lprior },  
 term(Right,R),  
 termRight(APP(I(Op),TPL("()",[Left,Right])),  
 prior,O,aprior,Term).

The input to the recursive call to termRight includes the infix term just discovered.

Postfix operators are treated in a similar way to infix expressions, except that postfix operators do not have a ‘right hand’ expression:

termRight(Left,prior,lprior,aprior,Term) -->  
 [ID(Op)],  
 { isPstOp(Op,L,O),O=<prior,L>=lprior },  
 termRight(APP(I(Op),TPL("()",[Left])),prior,O,aprior,Term).

Note that we make use of L&O’s backtracking to help disambiguate the case where an operator is simultaneously an infix and a postfix operator. There are several such operators, for example: ". ", "+" and \uphat. When encountering such dual-mode operators, the first interpretation as an infix operator is tried first; and if that fails then the postfix interpretation is used.

##### Special infix operators

The ". "(TERM) operator is a special operator in that it can serve both as punctuation and as an operator. As punctuation, it serves as an expression terminator and it serves as a separator operator when encountered in a class body or package.

These punctuation symbols’ roles as operators are captured via additional rules in the termRight grammar:

termRight(Left,\_,lprior,lprior,Left),[TERM] --> [TERM].  
termRight(Left,prior,lprior,aprior,Term,true) -->   
 [TERM],  
 { 2000=<prior,1999>=lprior },  
 term(Right,2000),  
 termRight(APP(I(". "),TPL("()",[Left,Right])),  
 prior,2000,aprior,Term).  
termRight(Left,prior,lprior,aprior,Term) -->  
 [TERM],  
 { 2000=<prior,2000>=lprior },  
 termRight(APP(I(". "),TPL("()",[Left])),  
 prior,2000,aprior,Term).

We also need a special rule to deal with tupling operator , (COMMA):

termRight(Left,prior,lprior,aprior,Term) -->  
 [COMMA],  
 { isInfOp(",",L,O,R),O=<prior,L>=lprior },  
 term(Right,R),  
 termRight(APP(I(","),TPL("()",[Left,Right])), prior,O,aprior,Term).

The final rule for termRight simply returns the left-hand expression and does not consume further tokens:

termRight(Left,\_,prior,prior,Left) --> [].

##### Top-level term parse tree

Our final production, for term itself, invokes the grammar for prefix/simple expressions and the grammar for infix and postfix expressions:

term:(absTree,integer)-->list[tokType].  
term(T,prior) --> termLeft(Left,prior,Lprior),  
 termRight(Left,prior,Lprior,\_,T).

### Example parse tree

The abstract parse trees of even simple expressions can be quite large. For example, the abstract tree corresponding to:

app([E,..X],Y) => [E,..app(X,Y)]

is

APP(I("=>"),  
 TPL("()",  
 [APP(I("app"),  
 TPL("()",  
 [APP(I(",.."),  
 TPL("()",[I("E"),I("X")])),  
 I("Y")])),  
 APP(I(",.."),  
 TPL("()",[I("E"),  
 APP(I("app"),  
 TPL("()",[I("X"),I("Y")]))]))]))

Moreover, abstract parse trees only represent one intermediate kind of structure in a typical L&O compiler. However, by the time that the above equation is compiled, it will have been reduced to just a few instructions.

1. A classification of characters introduced by the Unicode consortium to abstract common roles of character glyphs. [↑](#footnote-ref-25)