Chapter 2

Syntax

since feeling is first
who pays any attention
to the syntax of things
will never wholly kiss you;
...
for life's not a paragraph

And death i think is no parenthesis

— *e e cummings*

In the area of programming languages, syntax refers to the form of programs — how they are constructed from symbolic parts. A number of theoretical and practical tools — including grammars, lexical analyzers, and parsers — have been developed to aid in the study of syntax. By and large we will downplay syntactic issues and tools. Instead, we will emphasize the semantics of programs; we will study the meaning of language constructs rather than their form.

We are not claiming that syntactic issues and tools are unimportant in the analysis, design, and implementation of programming languages. In actual programming language implementations, syntactic issues are very important and a number of standard tools (like Lex and Yacc) are available for addressing them. But we do believe that syntax has traditionally garnered much more than its fair share of attention, largely because its problems were more amenable to solution with familiar tools. This state of affairs is reminiscent of the popular tale of the person who searches all night long under a street lamp for a lost item not because the item was lost there but because the light was better. Luckily, many investigators strayed away from the street lamp of parsing theory in order to explore the much dimmer area of semantics. Along the way, they developed many new tools for understanding semantics, some of which we will focus on in later chapters.

Despite our emphasis on semantics, however, we can't ignore syntax completely. Programs must be expressed in *some* form, preferably one that elucidates the fundamental structure of the program and is easy to read, write, and reason about. In this chapter, we introduce a set of syntactic conventions for describing our mini-languages.

2.1 Abstract Syntax

We will motivate various syntactic issues in the context of EL, a mini-language of expressions. EL describes functions that map any number of numerical inputs to a single numerical output. Such a language might be useful on a calculator, say, for automating the evaluation of commonly used mathematical formulae.

Figure 2.1 describes (in English) the abstract structure of a legal EL program. EL programs contain numerical expressions, where a numerical expression can be constructed out of various kinds of components. Some of the components, like numerals, references to input values, and various kinds of operators, are **primitive** — they cannot be broken down into subparts.[1] Other components are **compound** — they are constructed out of constituent components. The components have names; e.g., the subparts of an arithmetic operation are the **rator** (short for "operator") and two **rands**, (short for "operands") while the subparts of the conditional expression are the **test**, the **consequent**, and the **alternate**.

There are three major classes of phrases in an EL program: whole programs that designate calculations on a given number of inputs, numerical expressions that designate numbers, and boolean expressions that designate truth values (i.e., true or false). The structural description in Figure 2.1 constrains the ways in which these expressions may be "wired together". For instance, the test component of a conditional must be a boolean expression, while the consequent and alternate components must be numerical expressions.

A specification of the allowed wiring patterns for the syntactic entities of a language is called a **grammar**. Figure 2.1 is said to be an **abstract grammar** because it specifies the logical structure of the syntax but does not give any indication how individual expressions in the language are actually written.

The structure determined by an abstract grammar for an individual program phrase can be represented by an **abstract syntax tree (AST)**. Consider an EL

[1]Numerals can be broken down into digits, but we will ignore this detail.

A legal EL program is a pair of (1) a *numargs* numeral specifying the number of parameters and (2) a *body* that is a *numerical expression*, where a numerical expression is either:

- an *intlit* — an integer numeral *num*;
- an *input* — a reference to one of the program inputs specified by an *index* numeral.
- an *arithmetic operation* — an application of a *rator*, in this case a binary *arithmetic operator*, to two numerical *rand* expressions, where an arithmetic operator is either
  - addition,
  - subtraction,
  - multiplication,
  - division,
  - remainder;
- a *conditional expression* — a choice between numerical *consequent* and *alternate* expressions determined by a boolean *test* expression, where a *boolean expression* is either
  - a *boollit* — a boolean literal *bool*;
  - a *relational operation* — an application of *rator*, in this case a binary *relational operator*, to two numerical *rand* expressions, where a relational operator is one of
    - less-than,
    - equal-to,
    - greater-than;
  - a *logical operation* — an application of a *rator*, in this case a binary *logical operator*, to two boolean *rand* expressions, where a logical operator is one of
    - and,
    - or.

Figure 2.1: An abstract grammar for EL programs.

program that returns zero if its first input is between 1 and 10 (exclusive) and otherwise returns the product of the second and third inputs. The abstract syntax tree for this program appears in Figure 2.2. Each node of the tree corresponds to a numerical or boolean expression. The leaves of the tree stand for primitive phrases, while the intermediate nodes represent compound phrases. The labeled edges from a parent node to its children show the relationship between a compound phrase and its components. The AST is defined purely in terms of these relationships; the particular way that the nodes and edges of a tree are arranged on the page is immaterial.

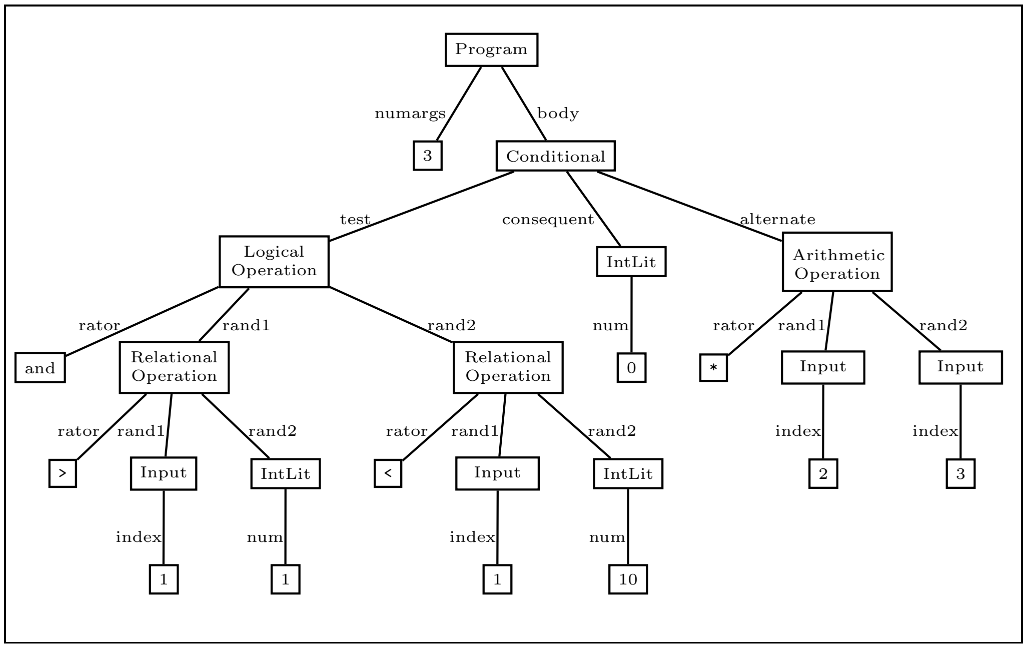


Figure 2.2: An abstract syntax tree for an EL program.

2.2 Concrete Syntax

Abstract grammars and ASTs aren't very helpful when it comes to representing programs in a textual fashion.[2] The same abstract structure can be expressed in

[2]It is also possible to represent programs more pictorially, and visual programming languages are an active area of research. But textual representations enjoy certain advantages over visual ones: they tend to be more compact than visual representations; the technology for processing them and communicating them is well-established; and, most importantly, they can effectively make use of our familiarity with natural language.

many different concrete forms. The sample EL conditional expression considered above, for instance, could be written down in some strikingly different textual forms. Here are three examples:

- `if $1 > 1 && $1 < 10 then 0 else $2 * $3 endif`

-
  ```
  (cond ((and (> (arg 1) 1) (< (arg 1) 10))
          0)
         (else (* (arg 2) (arg 3))))
  ```
- `1 input 1 gt 1 input 10 lt and {0} {2 input 3 input mul} choose`

The above forms differ along a variety of dimensions:

- *Keywords and operation names.* The keywords `if`, `cond`, and `choose` all indicate a conditional expression, while multiplication is represented by the names `*` and `mul`. Accessing the *i*th input to the program is written in three different ways: `$`*i*, `(arg` *i*`)`, and *i* `input`.
- *Operand order.* The example forms use infix, prefix, and postfix operations, respectively.
- *Means of grouping.* Grouping can be determined by precedence (`&&` has a lower precedence than `>` and `<` in the first example), keywords (`then`, `else`, and `endif` delimit the test, consequent, and alternate of the first conditional), or explicit matched delimiter pairs (such as the parentheses and braces in the last two examples).

These are only some of the possible dimensions; many more are imaginable. For instance, numbers could be written in many different numeral formats: e.g., decimal, binary, or octal numerals, scientific notation, or even roman numerals!

The above examples illustrate that the nature of concrete syntax necessitates making representational choices that are arbitrary with respect to the abstract syntactic structure. These choices are explicitly encoded in a **concrete grammar** that specifies how to **parse** a linear text string into a **concrete syntax tree (CST)**. A concrete syntax tree has the structural relationships of an abstract syntax tree embedded within it, but it is complicated by the handling of details needed to make the textual layout readable and unambiguous.

2.3 S-Expression Grammars Specify ASTs

While we will dispense with many of the complexities of concrete syntax, we still need *some* concrete notation for representing abstract syntax trees. Such a representation should be simple, yet permit us to precisely describe abstract syntax trees and operations on such trees. Throughout this book, we need to operate on abstract syntax trees to determine the meaning of a phrase, the type of a phrase, the translation of a phrase, and so on. To perform such operations, we need a far more compact representation for abstract syntax trees than the English description in Figure 2.1 or the graphical one in Figure 2.2.

We have chosen to represent abstract syntax trees using **s-expression grammars**. An s-expression grammar unites LISP's fully parenthesized prefix notation with traditional grammar notations to describe the structure of abstract syntax trees via parenthesized sequences of symbols and meta-variables. Not only are these grammars very flexible for definining unambiguous program language syntax, but it is easy to construct programs that process s-expression notation. This facilitates writing interpreters and compilers for the mini-languages we will study.

2.3.1 S-Expressions

An **s-expression** (short for **symbolic expression**) is a LISP notation for representing trees by parenthesized linear text strings. The leaves of the trees are **symbolic tokens**, where (to first approximation) a symbolic token is any sequence of characters that does not contain a left parenthesis ('`(`'), a right parenthesis ('`)`'), or a whitespace character. Examples of symbolic tokens include `x`, `foo`, `this-is-a-token`, `17`, `6.821`, and `4/3*pi*r^2`.[3]

An intermediate node in a tree is represented by a pair of parentheses surrounding the s-expressions that represent the subtrees. Thus, the s-expression

```
((this is) an ((example) (s-expression tree)))
```

designates the structure depicted in Figure 2.3. Whitespace is necessary for separating tokens that appear next to each other, but can be used liberally to enhance the readability of the structure. Thus, the above s-expression could also be written as

```
((this is)
 an
 ((example)
  (s-expression
   tree)))
```

without changing the structure of the tree.

[3]We always write s-expressions in `teletype-font`.

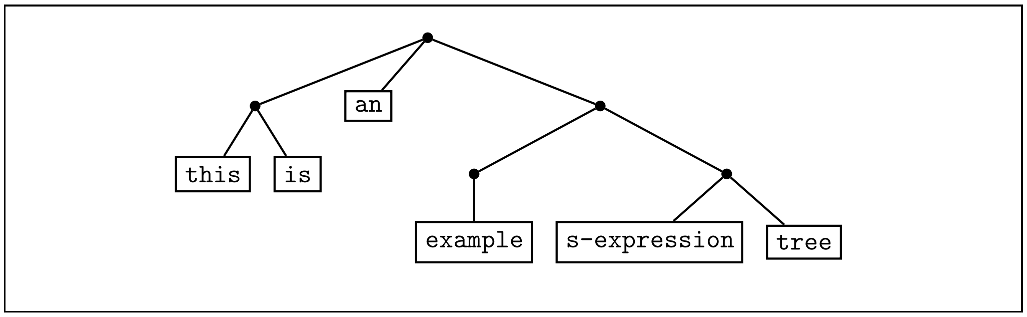


Figure 2.3: Viewing `((this is) an ((example) (s-expression tree)))` as a tree.

2.3.2 The Structure of S-Expression Grammars

An s-expression grammar combines the domain notation of Appendix A with s-expressions to specify the syntactic structure of a language. It has two parts:

1. A listing of **syntactic domains**, one for each kind of phrase.

2. A set of **production rules** that define the structure of compound phrases.

Figure 2.4 presents a sample s-expression grammar for EL.

A syntactic domain is a class of program phrases. **Primitive syntactic domains** are collections of phrases with no substructure. The primitive syntactic domains of EL are Intlit, BooleanLiteral, ArithmeticOperator, RelationalOperator, and LogicalOperator. Primitive syntactic domains are specified by an enumeration of their elements or by an informal description with examples. For instance, the details of what constitutes a numeral in EL are pretty much left to the reader's intuition.

Compound syntactic domains are collections of phrases built out of other phrases. Because compound syntactic domains are defined by a grammar's production rules, the syntactic domain listing does not explicitly indicate their structure. All syntactic domains are annotated with domain variables (such as *NE*, *BE*, and *N*) that range over their elements; these play an important role in the production rules.

The production rules specify the structure of compound domains. There is one rule for each compound domain. A production rule has the form

```
domain-variable ::= pattern [phrase-type]
                  | pattern [phrase-type]
                    . . .
                  | pattern [phrase-type]
```

where

Syntactic Domains:

$P \in \text{Program}$
$NE \in \text{NumExp}$
$BE \in \text{BoolExp}$
$N \in \text{Intlit} = \{\texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$
$B \in \text{BooleanLiteral} = \{\texttt{true}, \texttt{false}\}$
$A \in \text{ArithmeticOperator} = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$
$R \in \text{RelationalOperator} = \{\texttt{<}, \texttt{=}, \texttt{>}\}$
$L \in \text{LogicalOperator} = \{\texttt{and}, \texttt{or}\}$

Production Rules:

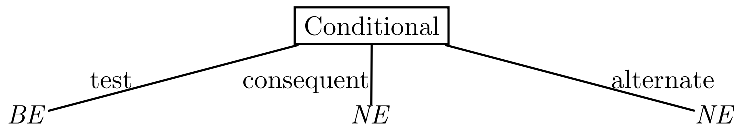
P	$::= \texttt{(el}\ N_{numargs}\ NE_{body}\texttt{)}$	[Program]
NE	$::= N_{num}$	[IntLit]
	$\mid \texttt{(arg}\ N_{index}\texttt{)}$	[Input]
	$\mid \texttt{(}A_{rator}\ NE_{rand1}\ NE_{rand2}\texttt{)}$	[Arithmetic Operation]
	$\mid \texttt{(if}\ BE_{test}\ NE_{consequent}\ NE_{alternate}\texttt{)}$	[Conditional]
BE_{bool}	$::= B$	[BoolLit]
	$\mid \texttt{(}R_{rator}\ NE_{rand1}\ NE_{rand2}\texttt{)}$	[Relational Operation]
	$\mid \texttt{(}L_{rator}\ BE_{rand1}\ BE_{rand2}\texttt{)}$	[Logical Operation]

Figure 2.4: An s-expression grammar for EL.

- *domain-variable* is the domain variable for the compound syntactic domain being defined,
- *pattern* is an s-expression pattern (defined below), and
- *phrase-type* is a mnemonic name for the subclass of phrases in the domain that match the pattern. It corresponds to the labels of intermediate nodes in an AST.

Each line of the rule is called a **production**; it specifies a collection of phrases that are considered to belong to the compound syntactic domain being defined. The second production rule in Figure 2.4, for instance, has four productions specifying that a NumExp can be an integer literal, an indexed input, an arithmetic operation, or a conditional.

An s-expression pattern appearing in a production stands for the domain of all s-expressions that have the form of the pattern. S-expression patterns are like s-expressions except that domain variables may appear as tokens. For example, the pattern `(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)` contains the domain variables BE_{test}, $NE_{consequent}$, and $NE_{alternate}$. Such a pattern specifies the structure of a **compound phrase** — a phrase that is built from other phrases. Subscripts on the domain variables indicate their role in the phrase. This helps to distinguish positions within a phrase that have the same domain variable — e.g., the consequent and alternate of a conditional, which are both numerical expressions. This subscript appears as an edge label in the AST node corresponding to the pattern, while the phrase type of the production appears as the node label. So the `if` pattern denotes an AST node pattern of the form:



An s-expression pattern P is said to **match** an s-expression SX if P's domain variables $d_1, \ldots, d_n$ can be replaced by matching s-expressions $SX_1, \ldots, SX_n$ to yield SX. Each SX_i must be an element of the domain over which d_i ranges. A compound syntactic domain contains exactly those s-expressions that match the patterns of its productions in an s-expression grammar.

For example, Figure 2.5 shows the steps by which the NumExp production

`(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)`

matches the s-expression

```
(if (= (arg 1) 3) (arg 2) 4).
```

Matching is a recursive process: BE_{test} matches `(= (arg 1) 3)`, $NE_{consequent}$ matches `(arg 2)`, and $NE_{alternate}$ matches `4`. The recursion bottoms out at primitive syntactic domain elements (in this case, elements of the domain Intlit). Figure 2.5 shows how an AST for the sample `if` expression is constructed as the recursive matching process backs out of the recursion.

Note that the pattern `(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)` would not match any of the s-expressions `(if 1 2 3)`, `(if (arg 2) 2 3)`, or `(if (+ (arg 1) 1) 2 3)`, because none of the test expressions `1`, `(arg 2)`, or `(+ (arg 1) 1)` match any of the patterns in the productions for BoolExp.

More formally, the rules for matching an s-expression pattern to an s-expression are as follows:

- A symbolic token T in the pattern matches only T.
- A domain variable for a primitive syntactic domain D matches an s-expression SX only if SX is an element of D.
- A domain variable for a compound syntactic domain D matches an s-expression SX only if one of the patterns in the rule for D matches SX.
- A pattern `(`$P_1 \ldots P_n$`)` matches an s-expression `(`$SX_1 \ldots SX_n$`)` only if each subpattern P_i matches the corresponding subexpression SX_i.

We shall use the notation $s\text{-}exp_D$ to designate the domain element in D that an s-expression designates. When D is a compound domain, $s\text{-}exp_D$ corresponds to an abstract syntax tree that indicates *how* *s-exp* matches one of the rule patterns for the domain. For example,

`(if (and (> (arg 1) 1) (< (arg 1) 10)) 0 (* (arg 2) (arg 3)))`$_{\text{NumExp}}$

can be viewed as the abstract syntax tree depicted in Figure 2.2 on page 20. Each node of the AST indicates the production that successfully matches the corresponding s-expression, and each edge indicates a domain variable that appeared in the production pattern. The nodes are labeled by the phrase type of the production and the edges are labeled by the subscript names of the domain variables used in the production pattern.

In the notation $s\text{-}exp_D$, domain subscript D serves to disambiguate cases where *s-exp* belongs to more than one syntactic domain. For example, $\texttt{1}_{\text{Intlit}}$ is `1` as a primitive numeral, while $\texttt{1}_{\text{NumExp}}$ is `1` as a numerical expression. The subscript will be omitted when the domain is clear from context.

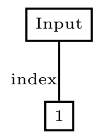
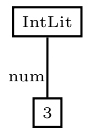
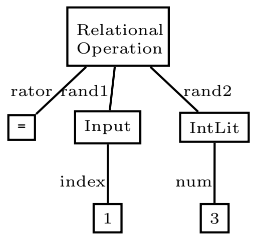
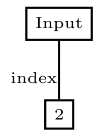
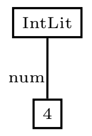
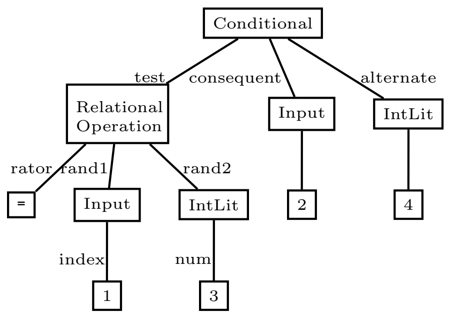
s-expression	domain	production	AST
`(arg 1)`	*NE*	`(arg` N_{index}`)`	
`3`	*NE*	N_{num}	
`(= (arg 1) 3)`	*BE*	`(`R_{rator} NE_{rand1} NE_{rand2}`)`	 Relational Operation rator rand1 rand2 = Input IntLit index num 1 3
`(arg 2)`	*NE*	`(arg` N_{index}`)`	
`4`	*NE*	N_{num}	
`(if (= arg 1) (arg 2) 4)`	*NE*	`(if` BE_{test} $NE_{consequent}$ $NE_{alternate}$`)`	 Conditional test consequent alternate Relational Operation Input IntLit rator rand1 rand2 = Input IntLit 2 4 index num 1 3

Figure 2.5: The steps by which `(if (= (arg 1) 3) (arg 2) 4)` is determined to be a member of the syntactic domain NumExp. In each row, an s-expression matches a domain by a production to yield an abstract syntax tree.

2.3.3 Phrase Tags

S-expression grammars for our mini-languages will generally follow the Lisp-style convention that compound phrases begin with a **phrase tag** that unambiguously indicates the phrase type. In EL, `if` is an example of a phrase tag. The fact that all compound phrases are delimited by explicit parentheses eliminates the need for syntactic keywords in the middle of or at the end of phrases (e.g., `then`, `else`, and `endif` in a conditional).

Because phrase tags can sometimes be cumbersome, we will often omit them when no ambiguity results. Figure 2.6 shows an alternative syntax for EL in which every production is marked with a distinct phrase tag. In this alternative syntax, the addition of 1 and 2 would be written `(arith + (num 1) (num 2))` — quite a bit more verbose than `(+ 1 2)`! But most of the phrase tags can be removed without introducing ambiguity. Because numerals are clearly distinguished from other s-expressions, there is no need for the `num` tag. Likewise, we can dispense with the `bool` tag. Since the arithmetic operators are disjoint from the other operators, the `arith` tag is superfluous; similarly for the `rel` and `log` tags. The result of these optimizations is the original EL syntax in Figure 2.4.

$$\begin{array}{rcll}
P & ::= & (\texttt{el}\ N_{numargs}\ \ NE_{body}) & \text{[Program]} \\
 & & & \\
NE & ::= & (\texttt{num}\ N_{num}) & \text{[IntLit]} \\
 & | & (\texttt{arg}\ N_{index}) & \text{[Input]} \\
 & | & (\texttt{arith}\ A_{rator}\ \ NE_{rand1}\ \ NE_{rand2}) & \text{[Arithmetic Operation]} \\
 & | & (\texttt{if}\ BE_{test}\ \ NE_{consequent}\ \ NE_{alternate}) & \text{[Conditional]} \\
 & & & \\
BE & ::= & (\texttt{bool}\ B) & \text{[Truth Value]} \\
 & | & (\texttt{rel}\ R_{rator}\ \ NE_{rand1}\ \ NE_{rand2}) & \text{[Relational Operation]} \\
 & | & (\texttt{log}\ L_{rator}\ \ BE_{rand1}\ \ BE_{rand2}) & \text{[Logical Operation]}
\end{array}$$

Figure 2.6: An alternative syntax for EL in which every production has a phrase tag.

2.3.4 Sequence Patterns

As defined above, each component of an s-expression pattern matches only s-expressions. But sometimes it is desirable for a pattern component to match *sequences* of s-expressions. For example, suppose we want to extend the `+` operator of EL to accept an arbitrary number of numeric operands (making `(+ 1 2 3 4)` and `(+ 2 (+ 3 4 5) (+ 6 7))` legal numerical expressions in EL). Using the simple patterns introduced above, this extension requires an infinite number of productions:

$$
\begin{array}{lll}
NE ::= \ldots & & \\
\quad \mid \texttt{(+)} & & \text{[Addition-0]} \\
\quad \mid \texttt{(+ } NE_{rand1}\texttt{)} & & \text{[Addition-1]} \\
\quad \mid \texttt{(+ } NE_{rand1}\ NE_{rand2}\texttt{)} & & \text{[Addition-2]} \\
\quad \mid \texttt{(+ } NE_{rand1}\ NE_{rand2}\ NE_{rand3}\texttt{)} & & \text{[Addition-3]} \\
\quad \mid \ldots & &
\end{array}
$$

Here we introduce a concise way of handling this kind of syntactic flexibility within s-expression grammars. We extend s-expression patterns so that any pattern can be annotated with a postfix '*' character. Such a pattern is called a **sequence pattern**. A sequence pattern P^* matches any consecutive sequence of zero or more s-expressions $SX_1 \ldots SX_n$ such that each SX_i matches the pattern P.

For instance, the extended addition expression can be specified concisely by the pattern `(+` $NE_{rand}{}^*$`)`. Here are some phrases that match this new pattern, along with the sequence matched by $NE_{rand}{}^*$ in each case:

`(+ 1 2 3 4)`	$NE_{rand}{}^* = [\texttt{1, 2, 3, 4}]_{\text{NumExp}}$
`(+ 2 (+ 3 4 5) (+ 6 7))`	$NE_{rand}{}^* = [\texttt{2, (+ 3 4 5), (+ 6 7)}]_{\text{NumExp}}$
`(+ 1)`	$NE_{rand}{}^* = [\texttt{1}]_{\text{NumExp}}$
`(+)`	$NE_{rand}{}^* = [\,]_{\text{NumExp}}$

Note that a sequence pattern can match any number of elements, including zero or one. To specify that an addition should have a minimum of two operands, we could use the following pattern:

$$\texttt{(+ } NE_{rand1}\ NE_{rand2}\ NE_{rest}{}^*\texttt{)}.$$

A postfix '+' is similar to '*,' except the pattern matches a sequence with at least one element. Thus, `(+` $NE_{rand}{}^+$`)` is equivalent to `(+` NE_{rand} $NE_{rest}{}^*$`)`.

A postfix '*' or '+' can be attached to any s-expression pattern, not just a domain variable. For example, in the s-expression pattern

$$\texttt{(cond (}BE_{test}\ NE_{action}\texttt{)}^*\ \texttt{(else } NE_{default}\texttt{))},$$

the subpattern `(`BE_{test} NE_{action}`)`* matches any sequence of parenthesized clauses containing a boolean expression followed by a numerical expression.

To avoid ambiguity, s-expression grammars are not allowed to use s-expression patterns in which multiple sequence patterns enable a single s-expression to match a pattern in more than one way. As an example of a disallowed pattern, consider `(op` $NE_{rand1}{}^*$ $NE_{rand2}{}^*$`)`, which could match the s-expression `(op 1 2)` in three different ways:

- $NE_{rand1}{}^* = [\texttt{1, 2}]_{\text{NumExp}}$ and $NE_{rand2}{}^* = []_{\text{NumExp}}$

- $NE_{rand1}{*} = [\texttt{1}]_{\text{NumExp}}$ and $NE_{rand2}{*} = [\texttt{2}]_{\text{NumExp}}$
- $NE_{rand1}{*} = []_{\text{NumExp}}$ and $NE_{rand2}{*} = [\texttt{1, 2}]_{\text{NumExp}}$.

A disallowed pattern can always be transformed into a legal pattern by inserting explicit parentheses to demarcate components. For instance, the following are all unambiguous legal patterns:

$$\texttt{(op (}NE_{rand1}{*}\texttt{) (}NE_{rand2}{*}\texttt{))}$$

$$\texttt{(op (}NE_{rand1}{*}\texttt{) }NE_{rand2}{*}\texttt{)}$$

$$\texttt{(op }NE_{rand1}{*}\texttt{ (}NE_{rand2}{*}\texttt{)).}$$

2.3.5 Notational Conventions

In addition to the s-expression patterns described above, we will employ a few other notational conventions for syntax.

Domain Variables

In addition to being used in s-expression patterns, domain variables can appear inside s-expressions when they denote particular s-expression. For example, if NE_1 is the s-expression `(+ 1 2)` and NE_2 is the s-expression `(- 3 4)`, then `(*` NE_1 NE_2`)` is the same syntactic entity as `(* (+ 1 2) (- 3 4))`.

Sequence Notation

Sequence notation, including the infix notations for the *cons* (‘.’) and *append* (‘@’) sequence functions (see Section A.3.4), can be intermixed with s-expression notation to designate sequence elements of compound syntactic domains. For example, all of the following are alternative ways of writing the same extended EL addition expression:

```
(+ 1 2 3)
(+ [1, 2, 3])
(+ [1, 2] @ [3])
(+ 1 . [2, 3])
```

Similarly, if $NE_1 = \texttt{1}$, $NE_2{*} = [\texttt{2, (+ 3 4)}]$, and $NE_3{*} = [\texttt{(* 5 6), (- 7 8)}]$, then `(+` NE_1 `.` $NE_2{*}$`)` designates the same syntactic entity as

```
(+ 1 2 (+ 3 4)),
```

and `(+` $NE_2{*}$ @ $NE_3{*}$`)` designates the same syntactic entity as

```
(+ 2 (+ 3 4) (* 5 6) (- 7 8)).
```

The sequence notation is only legal in positions where a production for a compound syntactic domain contains a sequence pattern. For example, the following notations are illegal because `if` expressions do not contain any component sequences:

```
(if [(< (arg 1) 1), 2, 3])

(if [(< (arg 1) 1), 2] @ [3])

(if (< (arg 1) 1) . [2, 3]).
```

Sequence notation can be used in s-expression patterns as well. For example, the pattern

$$\texttt{(+ } NE_{\mathit{rand1}}\ \texttt{ . } NE_{\mathit{rest}}{}^{*}\texttt{)}$$

matches any addition expression with at least one operrand. The pattern

$$\texttt{(+ } NE_{\mathit{rands1}}{}^{*} \mathbin{@} NE_{\mathit{rands2}}{}^{*}\texttt{)}$$

can match an addition expression with any number of operands. If the expression has one or more arguments, the match is ambiguous (and therefore disallowed, see page 29) since there are multiple ways to bind $NE_{\mathit{rands1}}{}^{*}$ and $NE_{\mathit{rands2}}{}^{*}$ to sequences that append to the argument sequence.

Syntactic Functions

We will follow a convention (standard in the semantics literature) that functions on compound syntactic domains are defined by a series of clauses, one for each production. Figure 2.7 illustrates this style of definition for two functions on EL expressions: *nheight* specifies the height of a numerical expression, while *bheight* specifies the height of a boolean expression. Each clause consists of two parts: a *head* that specifies an s-expression pattern from a production; and a *body* that describes the meaning of the function for s-expressions that match the head pattern. The double brackets, $[\![\,]\!]$, are traditionally used in syntactic functions to demarcate a syntactic argument, and thus to clearly separate expressions in the language being defined (program code, for example) from the language of the semantics. These brackets may be viewed as part of the name of the syntactic function.

Functions on syntactic domains are formally maps from s-expressions to a result domain. However, for all intents and purposes, they can also be viewed as maps from abstract syntax trees to the result domain. Each clause of a syntactic function definition specifies how the function at the node of an AST is defined in terms of the result of applying this function to the components of the AST.

$nheight : \text{NumExp} \rightarrow Nat$
$nheight[\![NE]\!] = 0$
$nheight[\![\texttt{(arg}\ NE\texttt{)}]\!] = 0$
$nheight[\![\texttt{(}A\ \ NE_1\ \ NE_2\texttt{)}]\!] = (1 + (max\ nheight[\![NE_1]\!]\ nheight[\![NE_2]\!]))$
$nheight[\![\texttt{(if}\ BE_{test}\ \ NE_{con}\ \ NE_{alt}\texttt{)}]\!]$
$\quad = (1 + (max\ bheight[\![BE_{test}]\!]\ (max\ nheight[\![NE_{con}]\!]\ nheight[\![NE_{alt}]\!])))$

$bheight : \text{BoolExp} \rightarrow Nat$
$bheight[\![B]\!] = 0$
$bheight[\![\texttt{(}R\ \ NE_1\ \ NE_2\texttt{)}]\!] = (1 + (max\ nheight[\![NE_1]\!]\ nheight[\![NE_2]\!]))$
$bheight[\![\texttt{(}L\ \ BE_1\ \ BE_2\texttt{)}]\!] = (1 + (max\ bheight[\![BE_1]\!]\ bheight[\![BE_2]\!]))$

Figure 2.7: Two examples illustrating the form of function definitions on syntactic domains.

2.4 The Syntax of PostFix

Equipped with our syntactic tools, we are now ready to formally specify the syntactic structure of PostFix, the stack language introduced in Section 1.4, and to explore some variations on this structure. Figure 2.8 presents an s-expression grammar for PostFix. Top-level programs are represented as s-expressions of the form `(postfix` $N_{numargs}$ Q_{body}`)`, where $N_{numargs}$ is a numeral specifying the number of arguments and Q_{body} is the command sequence executed by the program. The sequence pattern C^* in the production for Commands (Q) indicates that it is a sequence domain over elements from the Command domain. Most of the elements of Command (C) are single tokens (e.g., `add` and `sel`), except for executable sequences, which are parenthesized elements of the Commands domain. The mutually recursive structure of Command and Commands permits arbitrary nesting of executable sequences.

The concrete details specified by Figure 2.8 are only one way of capturing the underlying abstract syntactic structure of the language. Figure 2.9 presents an alternative s-expression grammar for PostFix. In order to avoid confusion, we will refer to the language defined in Figure 2.9 as PostFix2.

There are two main differences between the grammars of PostFix and PostFix2.

1. The PostFix2 grammar strictly adheres to the phrase tag convention introduced in Section 2.3.3. That is, every element of a compound syntactic domain appears as a parenthesized structure introduced by a unique tag. For example, `1` becomes `(int 1)`, `pop` becomes `(pop)`, and `add` becomes

$P \in$ Program
$Q \in$ Commands
$C \in$ Command
$A \in$ ArithmeticOperator $= \{$`add`, `sub`, `mul`, `div`, `rem`$\}$
$R \in$ RelationalOperator $= \{$`lt`, `eq`, `gt`$\}$
$N \in$ Intlit $= \{\ldots,$ `-2`, `-1`, `0`, `1`, `2`, $\ldots\}$

P ::= `(postfix` $N_{numargs}$ Q_{body}`)`	[Program]
Q ::= C^*	[Command Sequence]
C ::= N	[IntLit]
| `pop`	[Pop]
| `swap`	[Swap]
| A	[Arithmetic Operator]
| R	[Relational Operator]
| `nget`	[NumGet]
| `sel`	[Select]
| `exec`	[Execute]
| `(`Q`)`	[Executable Sequence]

Figure 2.8: An s-expression grammar for PostFix.

`(arithop add)`.[4]

2. Rather than representing command sequences as a sequence domain, PostFix2 uses the `:` and `(skip)` commands to encode such sequences. `(skip)` is intended to be a "no op" command that leaves the stack unchanged, while `(:` C_1 C_2`)` is intended first to perform C_1 on the current stack and then to perform C_2 on the stack resulting from C_1. The `:` and `(skip)` commands in PostFix2 serve the roles of $cons_{\text{Command}}$ and $[\,]_{\text{Command}}$ in PostFix. For example, the PostFix command sequence

$$[\texttt{1}, \texttt{2}, \texttt{add}]_{\text{Command}} = (cons\ \texttt{1}\ (cons\ \texttt{2}\ (cons\ \texttt{add}\ [\,]_{\text{Command}})))$$

can be encoded in PostFix2 as a single command:

```
(: (int 1) (: (int 2) (: (arithop add) (skip)))).
```

The difference in phrase tags is a surface variation in concrete syntax that does not affect the structure of abstract syntax trees. Whether sequences are explicit (the original grammar) or implicit (the alternative grammar) is a deeper

[4]the `arithop` keyword underscores that the arithmetic operators are related; similarly for `relop`.

$P \in$ Program
$C \in$ Command
$A \in$ ArithmeticOperator $=$ {`add`, `sub`, `mul`, `div`, `rem`}
$R \in$ RelationalOperator $=$ {`lt`, `eq`, `gt`}
$N \in$ Intlit $=$ {…, `-2`, `-1`, `0`, `1`, `2`, …}

P ::= `(postfix` $N_{numargs}$ C_{body}`)`	[Program]
C ::= `(int` N`)`	[IntLit]
| `(pop)`	[Pop]
| `(swap)`	[Swap]
| `(arithop` A`)`	[Arithmetic Operator]
| `(relop` R`)`	[Relational Operator]
| `(nget)`	[NumGet]
| `(sel)`	[Select]
| `(exec)`	[Execute]
| `(seq` C`)`	[Executable Sequence]
| `(:` C_1 C_2`)`	[Compose]
| `(skip)`	[Skip]

Figure 2.9: An s-expression grammar for PostFix2, an alternative syntax for PostFix.

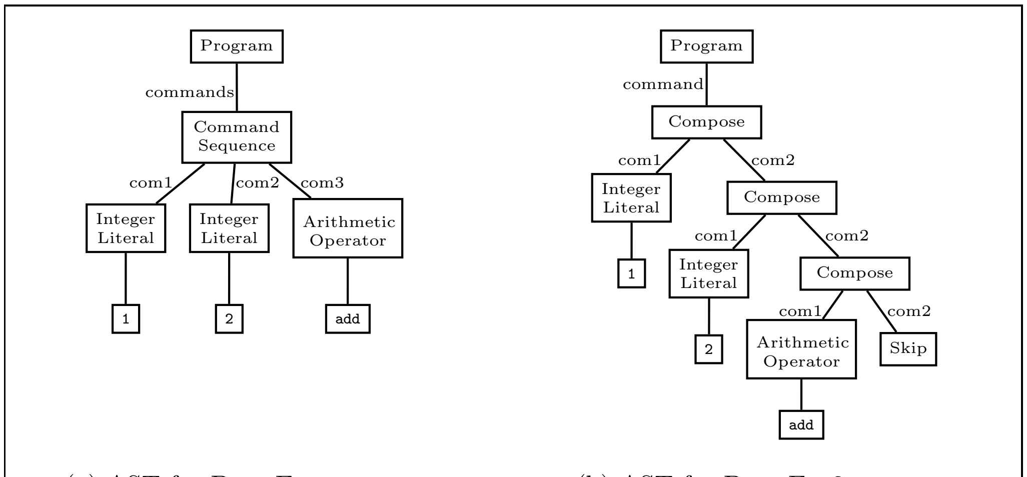
variation because the abstract syntax trees differ in these two cases (see Figure 2.10).

Although the tree structures are similar, it is not *a priori* possible to determine that the second tree encodes a sequence without knowing more about the semantics of compositions and skips. In particular, `:` and `(skip)` must satisfy two behavioral properties in order for them to encode sequences:

- `(skip)` must be an **identity** for `:`. I.e., `(:` C `(skip))` and `(: (skip)` C`)` must behave like C.
- `:` must be **associative**. I.e., `(:` C_1 `(:` C_2 C_3`))` must behave the same as `(: (:` C_1 C_2`)` C_3`)`.

These two properties amount to saying that (1) skips can be ignored and (2) in a tree of compositions, only the order of the leaves matters. With these properties, any tree of compositions is isomorphic to a sequence of the non-skip leaves. The informal semantics of `:` and `(skip)` given above satisfies these two properties.

Is one of the two grammars presented above "better" than the other? It depends on the context in which they are used. As the following example indicates, the PostFix grammar certainly leads to programs that are more concise than those generated by the PostFix2 grammar:



(a) AST for PostFix program (b) AST for PostFix2 program

Figure 2.10: A comparison of the abstract syntax trees for two encodings of a PostFix program.

```
; PostFix
(postfix 1 (1 2 add) (3 4 mul) sel exec)

; PostFix2
(postfix2 1
  (: (seq (: (int 1) (: (int 2) (: (arithop add)
                                    (skip)))))
     (: (seq (: (int 3) (: (int 4) (: (arithop mul)
                                       (skip)))))
        (: (sel)
           (: (exec) (skip))))))
```

Additionally, we shall see that the explicit sequences of PostFix make it more amenable to certain kinds of semantic analysis. On the other hand, other semantic and pragmatic tools are easier to apply to PostFix2 programs. Though we will focus on the PostFix grammar, we will consider PostFix2 when it is instructive to do so. In any event, the reader should be aware that even the fairly constrained boundaries of s-expression grammars leave some room for design decisions.

Reading

The notion of abstract syntax is due to McCarthy [McC62]. This notion is commonly used in operational and denotational semantics to ignore unimportant syntactic details (see the references in Chapters 3–4). Interpreters and compilers often have a "front-end" stage that converts concrete syntax into explicit data structures representing abstract syntax trees.

Our s-expression grammars are based on McCarthy's Lisp s-expression notation [McC60], which is a trivially parsable generic and extensible concrete syntax for programming languages. Many tools — most notably Lex [Les75] and Yacc [Joh75] — are available for converting more complex concrete syntax into abstract syntax trees. A discussion of these tools, and the scanning and parsing theory behind them, can be found in almost any compiler textbook. For a particularly concise account, consult one of Appel's textbooks [App98b, App98a, AP02].