- Some data describing the location of this token in the input. This information may be used by the parser to help the programmer identify the location of syntactic errors.

In general, the internal structure of tokens is relevant only to the scanner and the parser, so we shall not describe it in any further detail.

Parsing

Parsing is the process of organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing or diagramming a sentence into clauses. The syntactic structure of a language is typically specified using a BNF definition, also called a *context-free grammar* (section 1.1.2).

The parser takes as input a sequence of tokens, and its output is an abstract syntax tree (section 2.2.2). The abstract syntax trees produced by an SLLGEN parser can be described by `define-datatype`. For a given grammar, there will be one data type for each nonterminal. For each nonterminal, there will be one variant for each production that has the nonterminal as its left-hand side. Each variant will have one field for each nonterminal, identifier, or number that appears in its right-hand side. A simple example appears in section 2.2.2. To see what happens when there is more than one nonterminal in the grammar, consider a grammar like the one in section 3.9:

$$\begin{array}{ll}
\langle\text{statement}\rangle & ::= \texttt{\{}\ \langle\text{statement}\rangle\ \texttt{;}\ \langle\text{statement}\rangle\ \texttt{\}} \\
& ::= \texttt{while}\ \langle\text{expression}\rangle\ \texttt{do}\ \langle\text{statement}\rangle \\
& ::= \langle\text{identifier}\rangle\ \texttt{:=}\ \langle\text{expression}\rangle \\
\langle\text{expression}\rangle & ::= \langle\text{identifier}\rangle \\
& ::= \texttt{(}\langle\text{expression}\rangle\ \texttt{+}\ \langle\text{expression}\rangle\texttt{)}
\end{array}$$

The trees produced by this grammar could be described by this data type

```
(define-datatype statement statement?
  (compound-statement
    (stmt1 statement?)
    (stmt2 statement?))
  (while-statement
    (test expression?)
    (body statement?))
  (assign-statement
    (lhs symbol?)
    (rhs expression?)))
```

```
(define-datatype expression expression?
  (var-exp
    (id symbol?))
  (sum-exp
    (exp1 expression?)
    (exp2 expression?)))
```

For each nonterminal in a right-hand side, the corresponding tree appears as a field; for each identifier, the corresponding symbol appears as a field. The names of the variants will be specified in the grammar when it is written in SLLGEN. The names of the fields will be automatically generated; here we have introduced some mnemonic names for the fields. For example, the input

```
{x := foo; while x do x := (x + bar)}
```

produces the output

```
(compound-statement
  (assign-statement x (var-exp foo))
  (while-statement (var-exp x)
    (assign-statement x
      (sum-expression (var-exp x) (var-exp bar)))))
```

Throughout this appendix, abstract syntax trees are displayed as lists.

Scanners and Parsers in SLLGEN

Specifying Scanners

In SLLGEN, scanners are specified by regular expressions. Our example would be written in SLLGEN as follows:

```
(define scanner-spec-a
  '((white-sp
      (whitespace)                        skip)
    (comment
      ("%" (arbno (not #\newline)))       skip)
    (identifier
      (letter (arbno (or letter digit))) symbol)
    (number
      (digit  (arbno digit))              number)))
```

- The symbol `string`. The characters in the buffer are converted into a Scheme string, and a token is emitted, with the class name as its class and with that string as its datum.

If there is a tie for longest match between two regular expressions, `string` takes precedence over `symbol`. This rule means that keywords that would otherwise be identifiers are treated as keywords.

Specifying Grammars

SLLGEN also includes a language for specifying grammars. The simple grammar above would be written in SLLGEN as

```
(define grammar-a1
  '((statement
      ("{" statement ";" statement "}")
      compound-statement)
    (statement
      ("while" expression "do" statement)
      while-statement)
    (statement
      (identifier ":=" expression)
      assign-statement)
    (expression
      (identifier)
      var-exp)
    (expression
      ("(" expression "+" expression ")")
      sum-exp)))
```

A grammar in SLLGEN is a list described by the following grammar:

$$\begin{array}{lll}
\langle\text{grammar}\rangle & ::= & \texttt{(}\{\langle\text{production}\rangle\}^{*}\texttt{)} \\
\langle\text{production}\rangle & ::= & \texttt{(}\langle\text{lhs}\rangle\ \ \texttt{(}\{\langle\text{rhs-item}\rangle\}^{*}\texttt{)}\ \ \langle\text{prod-name}\rangle\texttt{)} \\
\langle\text{lhs}\rangle & ::= & \langle\text{symbol}\rangle \\
\langle\text{rhs-item}\rangle & ::= & \langle\text{symbol}\rangle \mid \langle\text{string}\rangle \\
 & ::= & \texttt{(arbno}\ \ \{\langle\text{rhs-item}\rangle\}^{*}\texttt{)} \\
 & ::= & \texttt{(separated-list}\ \ \{\langle\text{rhs-item}\rangle\}^{*}\ \ \langle\text{string}\rangle\texttt{)} \\
\langle\text{prod-name}\rangle & ::= & \langle\text{symbol}\rangle
\end{array}$$

A grammar is a list of productions. The left-hand side of the first production is the start symbol for the grammar. Each production consists of a left-hand side (a nonterminal symbol), a right-hand side (a list of ⟨rhs-item⟩'s) and a production name. The right-hand side of a production is a list of symbols or strings. The symbols are nonterminals; strings are literal strings. A

`arbno`s and `separated-list`s

An `arbno` is a Kleene star in the grammar: it matches an abitrary number of repetitions of its entry. For example, the production

⟨statement⟩ ::= `begin` {⟨statement⟩ `;`}* `end`

could be written in SLLGEN as

```
(define grammar-a2
  '((statement
      ("{" (arbno statement ";") "}")
      compound-statement)
    ...))
```

This makes a compound statement a sequence of an arbitrary number of semicolon-terminated statements.

This `arbno` generates a single field in the abstract syntax tree. This field will contain a *list* of the data for the nonterminal inside the `arbno`. Our example generates the following datatypes:

```
(define-datatype statement statement?
  (compound-statement
    (compound-statement32 (list-of statement?)))
  ...)
```

A simple interaction looks like:

```
> (define scan&parse2
    (sllgen:make-string-parser scanner-spec-a grammar-a2))

> (scan&parse2 "{x := foo; y := bar; z := uu;}")
(compound-statement
  ((assign-statement x (var-exp foo))
   (assign-statement y (var-exp bar))
   (assign-statement z (var-exp uu))))
```

We can put a sequence of nonterminals inside an `arbno`. In this case, we will get several fields in the node, one for each nonterminal; each field will contain a list of syntax trees. For example:

```
(define grammar-a3
  '((expression (identifier) var-exp)
    (expression
      ("let" (arbno identifier "=" expression) "in" expression)
      let-exp)))
```

```
(define scan&parse3
  (sllgen:make-string-parser scanner-spec-a grammar-a3))
```

This produces the datatype

```
(define-datatype expression expression?
  (var-exp (var-exp4 symbol?))
  (let-exp
    (let-exp9 (list-of symbol?))
    (let-exp7 (list-of expression?))
    (let-exp8 expression?)))
```

Here is an example of this grammar in action:

```
> (scan&parse3 "let x = y u = v in z")
(let-exp
  (x u)
  ((var-exp y) (var-exp v))
  (var-exp z))
```

The specification `(arbno identifier "=" expression)` generates exactly two lists: a list of identifiers and a list of expressions. This is convenient because it will let our interpreters get at the pieces of the expression directly.

Sometimes it is helpful for the syntax of a language to use lists with separators, not terminators. This is common enough that it is a built-in operation in SLLGEN. We can write

```
(define grammar-a4
  '((statement
      ("{" (separated-list statement ";") "}")
      compound-statement)
    ...))
```

This produces the datatype

```
(define-datatype statement statement?
  (compound-statement
    (compound-statement103 (list-of statement?)))
    ...)
```

Here is a sample interaction:

```
> (define scan&parse4
    (sllgen:make-string-parser scanner-spec-a grammar-a4))
> (scan&parse4 "{}")
(compound-statement ())
> (scan&parse4 "{x:= y; u := v ; z := t}")
(compound-statement
  ((assign-statement x (var-exp y))
   (assign-statement u (var-exp v))
   (assign-statement z (var-exp t))))
> (scan&parse4 "{x:= y; u := v ; z := t ;}")
Error in parsing: at line 1
Nonterminal <seplist3> can't begin with string "}"
```

In the last example, the input string had a terminating semicolon that did not match the grammar, so an error was reported.

As with `arbno`, we can place an arbitrary sequence of nonterminals within a `separated-list`. In this case, we will get several fields in the node, one for each nonterminal; each field will contain a list of syntax trees. This is exactly the same data as would be generated by `arbno`; only the concrete syntax differs.

We will occasionally use nested `arbno`s and `separated-list`s. A nonterminal inside an `arbno` generates a list, so a nonterminal inside an `arbno` inside an `arbno` generates a list of lists.

As an example, consider a `compound-statement` similar to the one in `grammar-a4`, except that we have parallel assignments:

```
(define grammar-a5
  '((statement
      ("{"
        (separated-list
          (separated-list identifier ",")
          ":="
          (separated-list expression ",")
          ";")
        "}")
      compound-statement)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    ))

> (define scan&parse5
    (sllgen:make-string-parser scanner-spec-a grammar-a5))
```

This generates the following datatype for `statement`:

```
(define-datatype statement statement?
  (compound-statement
    (compound-statement4 (list-of (list-of symbol?)))
    (compound-statement3 (list-of (list-of expression?)))))
```

A typical interaction looks like:

```
> (scan&parse5 "{x,y := u,v ; z := 4; t1, t2 := 5, 6}")
(compound-statement
  ((x y) (z) (t1 t2))
  (((var-exp u) (var-exp v))
   ((lit-exp 4))
   ((lit-exp 5) (lit-exp 6))))
```

Here the `compound-statement` has two fields: a list of lists of identifiers, and the matching list of lists of expressions. In this example we have used `separated-list` instead of `arbno`, but an `arbno` would generate the same data.

Exercise A.1 [⋆] The following grammar for ordinary arithmetic expressions builds in the usual precedence rules for arithmetic operators:

⟨arith-expr⟩	::= ⟨arith-term⟩ {⟨additive-op⟩ ⟨arith-term⟩}*
⟨arith-term⟩	::= ⟨arith-factor⟩ {⟨multiplicative-op⟩ ⟨arith-factor⟩}*
⟨arith-factor⟩	::= ⟨number⟩
	::= `(` ⟨arith-expr⟩ `)`
⟨additive-op⟩	::= `+` | `-`
⟨multiplicative-op⟩	::= `*` | `/`

This grammar says that every arithmetic expression is the sum of a non-empty sequence of terms; every term is the product of a non-empty sequence of factors; and every factor is either a constant or a parenthesized expression.

Write a lexical specification and a grammar in SLLGEN that will scan and parse strings according to this grammar. Verify that this grammar handles precedence correctly, so that, for example `3+2*66-5` gets grouped correctly, as $3 + (2 \times 66) - 5$.

Exercise A.2 [⋆⋆] Why can't the grammar above be written with `separated-list`?

Exercise A.3 [⋆⋆] Write an interpreter that takes the syntax tree produced by the parser of exercise A.1 and evaluates it as an arithmetic expression. The parser takes care of the usual arithmetic precedence operations, but the interpreter will have to take care of associativity, that is, making sure that operations at the same precedence level (*e.g.* additions and subtractions) are performed from left to right. Since there are no variables in these expressions, this interpreter need not take an environment parameter.