T-gen User's Guide

Justin O. Graver
University of Florida

Abstract

T-gen is a general-purpose object-oriented tool for the automatic generation of string-to-object translators. It is written in Smalltalk and lives in the Smalltalk programming environment. T-gen supports the generation of both top-down (LL) and bottom-up (LR) parsers, which will automatically generate derivation trees, abstract syntax trees, or arbitrary Smalltalk objects. The simple specification syntax and graphical user interface enhance the learning, comprehension, and usefulness of T-gen.

1 Introduction

In a computer-oriented environment, it is often necessary to translate a structured textual specification into either a different textual representation or into some internal data representation. A special case of this general translation process is compilation, where a source program is translated into some executable machine program. The formalisms behind such translations are well understood and it is straightforward to write programs that automatically build translators from an expression-based specification of the desired translation. T-gen (from "translator generator") follows a long heritage of translator generator tools. It was created out of the need for a general-purpose translator-generator tool within the Smalltalk programming environment. The following features distinguish T-gen from similar contemporary tools:

- support for building abstract syntax trees (or any structured objects)
- graphical integrated user-interface
- handles all common grammars: LL(1), SLR(1), LALR(1), and LR(1)
- object-oriented design and implementation
- almost unlimited extensibility

This guide describes how T-gen works, how to prepare specification files for translator generation, and provides numerous examples and hints on effective usage of T-gen. We assume that the reader is familiar with the translation process and with the related specification techniques. More information about these topics can be found in introductory compiler texts such as *Compilers: Principles, Techniques, and Tools* [ASU86] and *Crafting a Compiler* [FL88]. We also assume at least a rudimentary knowledge of the Smalltalk programming environment and of object-oriented programming. A good introduction to both of these can be found in *Inside Smalltalk* [LP90].

Author's address: Computer and Information Sciences, University of Florida, E301 CSE, Gainesville, FL 32611. *Telephone:* (904) 392-1507; *e-mail:* graver@ufl.edu.

2 Background

A translation scheme may involve both syntactic and semantic processing of the source text before the final (target) representation is obtained. The first step in the translation process is called *lexical analysis* and consists of grouping the source text characters into logical pieces called *tokens*. Tokens are usually specified by *regular expressions* (equivalent to *finite-state automata*) and are recognized by a *scanner*. The resulting sequence of tokens is then presented to a *parser* to determine the structure and validity of the input. The result of parsing may be the desired translation target (a structured object) or a *parse tree* that can be used to derive (either directly or indirectly) the target. The structure recognized by a parser is usually specified by a *context-free grammar* (CFG).

A CFG is defined by a finite set of *nonterminals*, a finite set of *terminals*, a *start symbol*, and a finite set of *productions*. Productions are translation rules of the form $A \rightarrow \omega$, where A is a nonterminal and ω is a possibly empty sequence of terminals and nonterminals. In T-gen, nonterminals (intermediate symbols) are denoted by identifiers. A T-gen identifier must begin with a letter followed by zero or more letters or digits. An underscore (_) is treated as a letter. Terminals (tokens) come in two varieties, literals and token classes. *Literal terminals* are strings (delimited by apostrophes) and may contain any printable characters (embedded apostrophes must be doubled). *Token class terminals* are identifiers surrounded by angled brackets, e.g. `<id_name>`. Since the set of terminals for a grammar must be finite (by definition), a token class terminal is used to represent an infinite class of related tokens.

A CFG is conventionally specified simply by a list of its productions. The left-hand-side of the first production is taken to be the start symbol. A T-gen grammar is specified in a similar fashion (the exact syntax rules are given in Section 4). For example,

context-free grammar	T-gen grammar specification
$S \rightarrow aS$	`S : 'a' S ;`
$S \rightarrow a$	`S : 'a' ;`

represents a grammar which defines a language consisting of all nonempty strings of `a`'s. Production with the same left-hand-side may be grouped together without repeating the left-hand-side. For example,

context-free grammar	T-gen grammar specification
$S \rightarrow aS\|bS\|\epsilon$	`S : 'a' S | 'b' S | ;`

defines a language consisting of all, possibly empty, strings of `a`'s and `b`'s.

While CFGs provide an excellent specification mechanism for machine-generated parsers, they are not always the most effective means for communicating grammatical structure to humans. A flexible and more human-readable extension of CFGs is provided by *regular right-part grammars* (RRPGs) [LaL77]. The right-hand-sides of a regular right-part grammar are nondeterministic finite state machines whose transition tokens are the terminals and nonterminals of the grammar. RRPG right-hand-sides can be equivalently specified using either Pascal-style syntax diagrams (a graphical representation) or regular expressions (a textual representation).

The extended language of regular expressions defined in [LaL77] for RRPGs is composed of

- the atomic base expressions:

$\emptyset$	the empty set
ϵ	the language consisting of the empty string
a	the language consisting of the terminal or nonterminal symbol a

- the traditional regular expressions operators, given the regular expressions p and q:

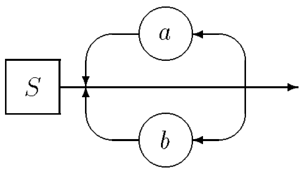
$p\|q$	alternation
pq	concatenation
p^*	closure

- and three additional shorthand forms:

$$\begin{aligned} p^? &= \epsilon|p \\ p^+ &= pp^* \\ p\ \mathit{list}\ q &= p(qp)^* \end{aligned}$$

Of the traditional regular expression operators, closure has the highest precedence, followed by concatenation, and then by alternation. The shorthand forms have the same precedence as closure.

Notice that all CFGs are RRPGs (i.e. the right-hand-sides of CFGs are simply alternations of concatenations of terminals and nonterminals). T-gen, like its contemporary parser generator ANTLR [PDC90], accepts RRPGs (the details are discussed in Section 4). For example, the language consisting of all, possibly empty, strings of `a`'s and `b`'s (see previous CFG example) can be specified by any of the following RRPGs

regular expression form	syntax diagram form	T-gen form
$S \rightarrow (a\|b)^*$		`S : ('a' | 'b')* ;`

3 Token Class Specifications

A T-gen token class specification is a sequence of token class definitions. A token class is defined by giving its token class terminal, followed by a colon (`:`), a regular expression (slightly different from those used to specify RRPGs), an optional scanner directive, and terminated by a semicolon (`;`). For example,

```
<identifier>  :  [A-Za-z][A-Za-z0-9]*     ;
<number>      :  [0-9]+                   ;
<whitespace>  :  [\s\t\r]+         {ignoreDelimiter} ;
```

defined three token classes. Tokens in the class `<identifier>` begin with a letter and are followed by zero or more letters or digits. Tokens in the class `<number>` are composed of one or more digits. Tokens in the class `<whitespace>` consist of one or more sequential space, tab, or carriage-return characters. The scanner directive {`ignoreDelimiter`} represents a message that will be sent to the scanner whenever it recognizes a token in that class (more on this later).

The language of extended regular expressions recognized by T-gen for token class specifications is defined as follows.

c — the character c itself, can be most any printable character, but regular expression metacharacters must be escaped (see below).

$\backslash c$ — the (escaped) character c itself, used to specify characters that normally would be interpreted as regular expression operators, such as `[`, `*`, and `+`, also used to specify certain nonprintable characters. The following are defined:

specification	ASCII value	character
`\0`	0	null
`\b`	8	backspace
`\t`	9	horizontal tab
`\n`	10	linefeed (UNIX newline `\n`)
`\f`	12	form feed
`\r`	13	carriage return (Smalltalk newline `cr`)
`\e`	27	escape
`\s`	32	space
`\d`	127	delete

ddd the character whose decimal ASCII value is *ddd*, where *d* is a decimal digit (0-9).

oooo the character whose octal ASCII value is *ooo*, where *o* is an octal digit (0-7).

xhh the character whose hexadecimal ASCII value is *hh*, where *h* is a hexadecimal digit (0-9, A-F, or a-f).

$a|b$ either of the regular expressions a or b (alternation).

ab the regular expression a followed by the regular expression b (concatenation).

$a*$ zero or more repetitions of the regular expression a (Kleene Closure).

(a) the regular expression a as an indivisible unit (parentheses are used in the traditional fashion to group operators of lower precedence together — closure has the highest precedence, followed by concatenation, and then alternation).

$a+$ one or more repetitions of the regular expression a (shorthand for $aa*$).

$a?$ an optional occurrence of the regular expression a (shorthand for $(a|)$).

[*rangelist*] shorthand for the alternation of all characters specified by the *rangelist*. A *rangelist* is a list of range expressions of the form

c a simple (or escaped) character

c-d shorthand for a list of all the characters between *c* and *d* inclusive (based on ASCII character values).

For example, `[abd-g]` is equivalent to a|b|d|e|f|g.

~[*rangelist*] the alternation of all printable nonwhitespace characters in the complement of *rangelist* (i.e. all printable nonwhitespace characters *except* those specified).

Each token class definition may have an optional scanner directive, a Smalltalk unary message selector enclosed in braces ({}). The directive represents a message that will be sent to the scanner whenever a token of that class is recognized. Currently implemented scanner directives include

ignoreDelimiter discard the current token and scan the next (used to eliminate whitespace tokens, etc.)

ignoreComment discard the current token and scan the next (could be modified to save comments for later use)

compactDoubleApostrophies compact all two apostrophe sequences in the current (raw) token into a single apostrophe.

Users may add new scanner directives simply by adding new methods to the abstract class FS-ABasedScanner or to one of its concrete subclasses (see Section 5). Aa T-gen specification for token class specifications is given in Appendix D.

Contemporary lexical analyzer generators, like LEX [Les75] and DLG [PDC90], attach significance to the order in which token class definitions rules are given. Typically, rule ordering is used as an *ad hoc* technique for classifying tokens that belong to two or more overlapping token classes. Keywords are also difficult to detect and handle correctly. Currently, T-gen does not permit token classes to overlap and, hence, attaches *no* significance to token class definition ordering. Algorithms for automatically detecting relationships between overlapping token classes are being investigated. T-gen does provide automatic detection and recognition of keywords. It extracts literal tokens from the accompanying grammar specification and treats each literal token as if it belongs to its own singleton (i.e. one-element) token class.

4 Grammar Specifications

A T-gen grammar specification is a sequence of grammar rule specifications similar to those used by YACC [Joh75] and ANTLR [PDC90]. A grammar rule specification consists of a nonterminal, followed by a colon (:), one or more right-hand-side specifications separated by vertical bars (|), and is terminated with a semicolon (;). A right-hand-side specification is a regular expression (as defined in Section 2) optionally followed by a parse-tree-builder (PTB) directive. If the grammar being specified is a CFG then the regular expression is a (possibly empty) list of nonterminals and terminals. A PTB directive is a *translation symbol*, which is either a Smalltalk identifier or message selector, enclosed in braces. The translation symbol is used to construct a parse tree node for its associated grammar rule when building abstract syntax trees (more in Section 6).

T-gen is able to generate parsers for two major classes of CFGs: LL(1) and LR(1) variants. If a nontrivial RRPG is specified then it must first be transformed into an equivalent CFG before T-gen can attempt to build a parser. T-gen automatically performs any necessary RRPG-to-CFG transformations. If an LL(1) parser is desired, but the given (or transformed) grammar is not LL(1), T-gen will, at the user's direction, attempt to transform the grammar into one that is LL(1). The grammar transformations currently applied in this process are removal of left-recursion and left-factoring of common prefixes [ASU86, FL88]. If an LR(1) parser is desired, T-gen applies, in increasing order or complexity, three slightly different parser construction techniques: SLR(1), LALR(1), and LR(1). The construction ends when one of the techniques yields a parser or when all have failed (an extra user confirmation is required to initiate LR(1) analysis since it may be lengthy).

YACC uses several heuristics in an attempt to resolve certain conflicts that may arise during parser construction. T-gen uses no such heuristics; it either builds a parser or it doesn't. Consequently, there is no significance to the order in which right-hand-side specifications are given. YACC also allows ambiguous expression grammars using qualifiers to resolve operator associativity and precedence issues. T-gen requires associativity and precedence to be built into the grammar. A T-gen specification for grammar specifications is given in Appendix E.

5 User Interface

The T-gen user interface (Figure 1) can be created by evaluating the Smalltalk expression

```
TranslatorGeneratorView open.
```

The exact appearance of the user interface will vary slightly depending on the host windowing environment. Any token class definitions are typed into the upper-left pane and "accepted" by selecting **accept** from the <operate> (middle mouse button) menu. Accepting a token class specification

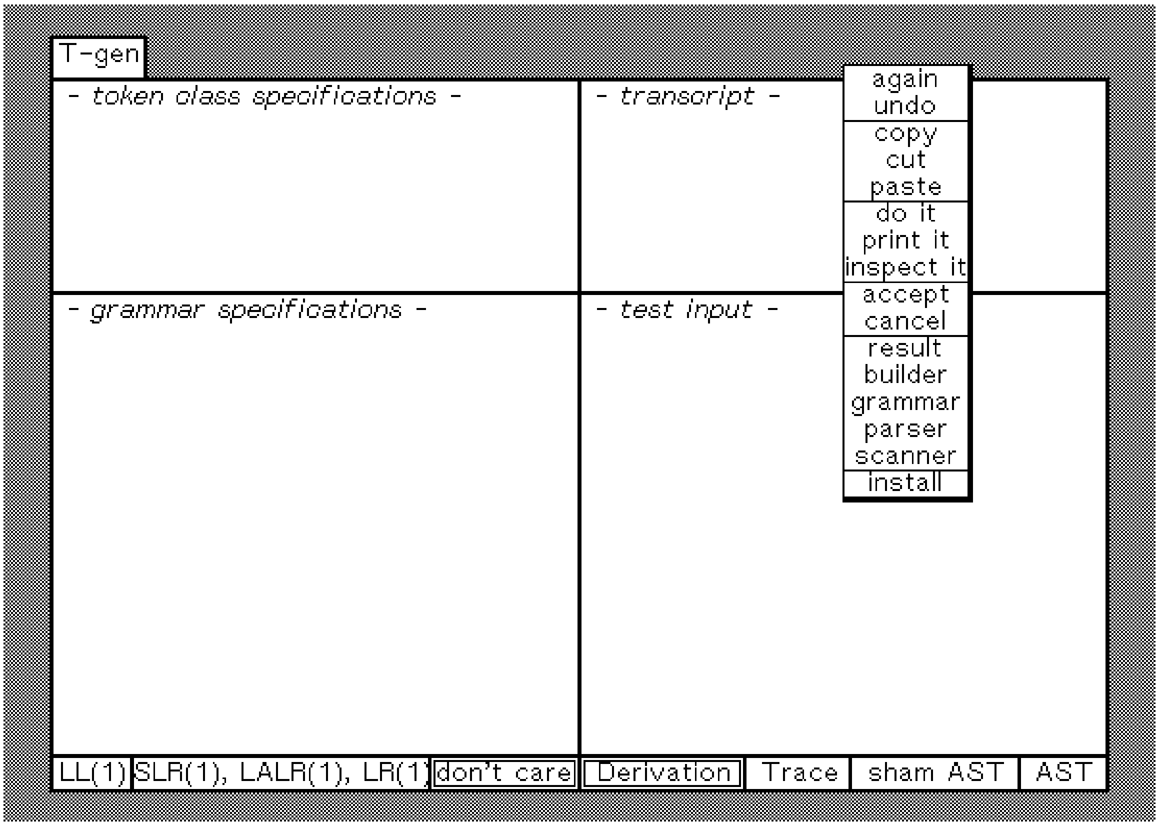


Figure 1: T-gen user interface with transcript-pane <operate> menu.

simply records it for later use during translator construction. Syntactic and semantic checking of the specification is done when the grammar is accepted (see below).

A grammar specification is typed into the lower-left pane of the interface. Translator construction is initiated when selecting **accept** from the <operate> menu in this pane. The grammar specification and any token class specification are processed and any errors are reported to the user. A log of the construction will be displayed in the upper-right transcript pane.

If a translator was successfully constructed then it can be tested by typing test input source into the lower-right pane and choosing **accept** from the <operate> menu in that pane. If the test input was successfully translated, the resulting object may be inspected by selecting **result** from the <operate> menu in the transcript pane.

Once a scanner and parser generated by T-gen have been tested, they can be added to the Smalltalk class hierarchy by means of the **install** item in the <operate> menu of the transcript pane. This amounts to creating a new subclass of FSABasedScanner and a new subclass of either LL1Parser or LR1Parser. The user is prompted for a base name, say "Foo", and T-gen then creates two new classes named FooScanner and FooParser. It then builds a class initialization method that contains the Smalltalk code to create the appropriate parse table object. The parser class is also linked to its corresponding scanner class (via an instance method) so that a scanner/parser pair may be created together. The new scanner and parser classes can be "filed-in" to other Smalltalk images and used independently of T-gen.

Two sets of "radio buttons" located at the bottom of the user interface control parser generation and the results of translating test input source. The buttons below the grammar specification pane allow the user to select what type of parser T-gen will attempt to build. The don't care button tries to build an LL(1) parser first. The result of translating test input source is controlled by the buttons beneath the test input pane. Derivation produces a derivation tree. Trace gives a step-by-step trace of the parse in the transcript pane, but produces no other result. The sham AST and AST buttons produce isomorphic abstract syntax trees as their results. If sham AST is chosen, generic derivation tree nodes will be used for the tree. If AST is chosen, instances of specific tree node classes will be used to build the tree based on the PTB directives. The specification and construction of parse trees will be discussed in detail in Section 6.

Scanner generation uses standard deterministic algorithms which should succeed on all syntactically valid token class specifications, so no special control over this process is required. However, potential problems may arise in disambiguating situations requiring multiple-token look-ahead. For example, if three different token classes contain the tokens `a`, `bd`, and `abc` and the input source is "abd" then scanners that try to match the longest possible token will commit to recognizing `abc` once the `b` has been seen. They will subsequently fail when seeing the `d` since they must have a `c` to complete the longer token. Some scanners (e.g. those generated by LEX) are able to backup and look for shorter tokens that may also match. Such scanners are able to backup and recognize the tokens `a` and `bd`. The power of a scanner to handle these kinds of situations is related to the number of valid tokens the scanner keeps buffered between the source and the parser (i.e. token lookahead).

The **scanner** item in the <operate> menu of the transcript pane prompts the user for the name of a scanner class. T-gen currently provides the following scanner classes with the indicated token lookahead.

scanner class	lookahead
FSABasedScanner	0
FSABasedScannerWithOneTokenLookahead	1
FSABasedScannerWithTwoTokenLookahead	2

The default is FSABasedScanner.

6 Building Parse Trees

Three different kinds of parse trees can be build by T-gen, derivation trees and two different varieties of abstract syntax trees. Derivation trees are built out of generic derivation tree nodes with fields for a name and an arbitrary number of ordered child (subtree) nodes. The derivation tree for a given input string is a tree, rooted at the start symbol, that represents each grammar rule that was applied during parsing. The derivation tree is a precise representation of the parse in that each step in the derivation is represented by some node in the tree. However, derivation trees often have long strings of nonterminals which add nothing but clutter to the structural interpretation of the input string. Abstract syntax trees (ASTs) are essentially derivation trees without such clutter. No PTB directives on grammar productions are required for building derivation trees. Thus, derivation trees are useful in the early stages of creating and debugging a grammar.

Since only selected portions of the derivation will actually be represented in the AST, the parser needs to know which rules create nodes and how. This information is specified in the (optional) PTB directive of each right-hand-side specification. In general, whenever the parser recognizes a grammar rule with a PTB directive, it creates an AST node for that rule. The kind of node created depends on both the test input translation option selected (sham AST or AST) and on the PTB directive. Both sham and actual ASTs require at least some PTB directives in order to be constructed, and may also require additional classes to be defined. Since sham ASTs are built out of generic derivation tree nodes, they require much less external support to construct than real ASTs. This also makes sham ASTs a useful grammar debugging aid.

It should be noted that if a parser is built from a transformed grammar then PTB directives on the transformed grammar may no longer reflect the user's original intentions. A preferable technique for building ASTs from transformed grammars is to use the **grammar** item from the <operate> menu in the transcript pane to inspect the transformed context-free grammar. This CFG can then be copied from the inspector text pane into the grammar specification pane where PTB directives can be updated in accordance with the new grammar structure.

Conceptually, an AST is constructed by performing a postorder traversal of the derivation tree. At certain points in the traversal, nodes must be created to support the structure of the AST. At other points, previously created AST nodes simply "follow along" with the traversal, waiting to be incorporated as the children of future nodes. Suppose the following grammar rule is present

```
A   :  B C D   {symbol} ;
```

A postorder traversal of the derivation tree would compute values for the right-hand-side nonterminals and then process the production. Processing the production involves using the values associated with the right-hand-side nonterminals to compute a value to be associated with the left-hand-side. If $f(n)$ is the value function for nonterminals and the PTB directive *symbol* represents the name of a node value processing function, then processing the above production in the derivation tree can be characterized symbolically as

$$f(A) := symbol(f(B), f(C), f(D)).$$

There are two different kinds of processing functions that can be specified through translation symbols (PTB directives). If the symbol is a capitalized Smalltalk identifier then it is interpreted as a class name (typically a subclass of ParseTreeNode). This kind of directive signals the PTB to create an instance of the corresponding class (or, if the sham AST button is selected, a derivation tree node with that name) to be associated with the left-hand-side, and to give it the nodes associated with the right-hand-side nonterminals as its children. Consider, for example, the grammar rule

```
Expr   :   LeftTerm '+' RightTerm    {PlusNode} ;
```

Assume that this rule has just been recognized by the parser and that the ASTs currently associated with `LeftTerm` and `RightTerm`, respectively, are α and β. The PTB would create a new `PlusNode`, say γ, with children α and β, and associate γ with `Expr`.

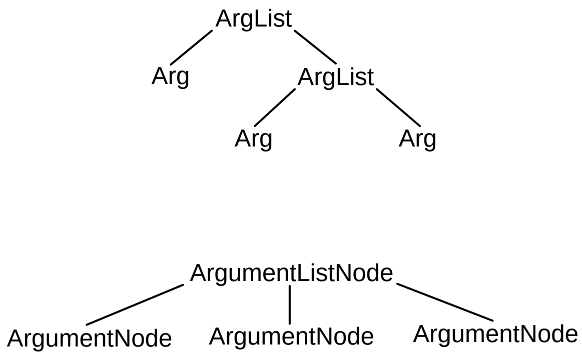


Figure 2: Derivation tree and abstract syntax tree for a three-element argument list.

The second variety of processing functions assumes the translation symbol represents a Smalltalk message selector. These message selectors may be one of several special predefined selectors or a user-defined selector. Currently, there are three special predefined selector-style PTB directives, `nil`, `liftRightChild`, and `liftLeftChild`.

The `nil` directive can be used with any production and simply associates nil (the empty AST) with the left-hand-side. Any ASTs associated with right-hand-side nonterminals are discarded. The `nil` directive is especially useful with epsilon productions (i.e. a grammar rule with an empty right-hand-side). For example,

```
Expr   :      {nil} ;
```

The `liftRightChild` and `liftLeftChild` directives are appropriate for grammar productions of the form

$$A \rightarrow B_1 B_2 \ldots B_n$$

for $n \geq 2$, where A and the B_i are nonterminals. (Terminals, though not represented explicitly in this template, may be freely interspersed with the right-hand-side nonterminals.) The `liftRightChild` directive instructs the PTB *not* to create a new node when the associated grammar rule is recognized. Instead, the root β_n of the AST currently associated with B_n is used as the "new" root node. The ASTs associated with B_1 through B_{n-1} are added as left-most children to β_n and it is associated with the left-hand-side. The `liftLeftChild` directive works analogously to the `liftRightChild` directive. It uses the root β_1 of the AST associated with B_1 as the "new" root node and the ASTs associated with B_2 through B_n are added as right-most children of β_1. The "lift" directives are useful for adding arbitrary numbers of children to parent nodes in an AST.

Consider the following grammar fragment

```
ArgList   :  Arg ArgList    {liftRightChild}
          |  Arg            {ArgumentListNode} ;
Arg       :  <argument>     {ArgumentNode} ;
```

Here, for each argument recognized, a corresponding ArgumentNode would be created. The second right-hand-part of the `ArgList` production, representing the base of an argument list, would be the first `ArgList` rule to be completely recognized by the parser. Accordingly, an ArgumentListNode would be created to hold all of the ArgumentNodes, its first child being the right-most argument. The other arguments are added to the list as occurrences of the first `ArgList` rule are unstacked inside the parser. Ultimately, an ArgumentListNode is associated with the `ArgList` nonterminal, which will be used elsewhere in the grammar. The resulting AST would be an ArgumentListNode with some number of ArgumentNodes as children (i.e. a "flattened" version of the corresponding derivation tree, as shown in Figure 2).

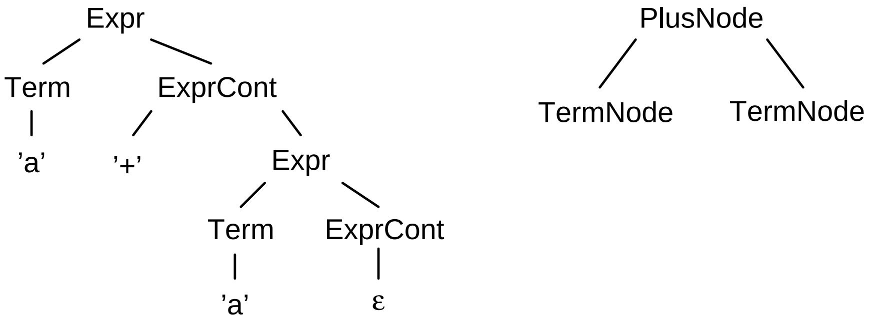


Figure 3: Derivation tree and abstract syntax tree for "a+a".

One special case exists for grammar rules with exactly two right-hand-side nonterminals and a "lift" directive, e.g.

```
A   :   B C    {liftRightChild} ;
```

If the AST for C has a value of nil, then the AST associated with B is simply passed along to A. Consider the following grammar fragment

```
Expr        :   Term ExprCont    {liftRightChild} ;   (P1)
ExprCont    :   '+' Expr         {PlusNode}           (P2)
            |                    {nil} ;              (P3)
Term        :   'a'              {TermNode} ;         (P4)
```

and the input string "a+a". The first complete production encountered in the postorder traversal of the corresponding derivation tree (Figure 3) is $P4$. The PTB creates a TermNode (with no children) and associated it with the left child of the root of the derivation tree. The other occurrence of production $P4$ in the right subtree is handled similarly, and nil is associated with its ExprCont sibling according to the PTB directive for production $P3$. At this point in the traversal we are ready to process production $P1$. Since the AST associated with ExprCont is nil, the left-child value is passed up. The next production encountered is $P2$, for which a new PlusNode is created with the TermNode as its only child. Finally, the root production $P1$ is processed. This time the right child can be lifted and the left child (TermNode) added as a child to the PlusNode giving the resulting AST (Figure 3). Grammar rules with liftLeftChild directives are handled analogously.

User-defined PTB directives (message selectors) are required to accept the same number of arguments as there are nonterminals in the right-hand side of the corresponding production. Productions with user-defined directives are processed by sending the message with corresponding AST arguments to the PTB. The result of this message send is the AST that is associated with the left-hand-side. This means that any user-defined PTB directives must be defined in the AbstractSyntaxTreeBuilder class or in some user-defined subclass. If users desire to create their own PTB classes then the translator generator can be directed to use a different PTB (the default is AbstractSyntaxTreeBuilder) by selecting the **builder** item from the <operate> button menu of the transcript pane. The user is prompted for the class name of a alternative PTB class, which is subsequently used in the construction of both varieties of ASTs. Since sham ASTs are built using derivation tree nodes, user-defined PTB directives may not work properly when building a sham AST. However, once grammar development has progressed to the point of defining special PTB directives, the user will probably be more interested in building actual ASTs.

Terminals within productions are normally ignored, with one exception. Productions of the form

```
A   :   <tokenclass>    {symbol} ;
```

are allowed, where `symbol` is either `nil`, a class name, or a one-argument message selector. The `nil` directive does the expected thing, associates nil with the left-hand-side. If `symbol` is a class name then an instance of the specified class is created and associated with the left-hand-side. Additionally, the resulting object is sent the setAttribute: message with the token value (a String) of the token class as an argument. This allows the node to retain information about the specific element of the token class that was encountered for later processing. For example, it is usually not sufficient for a compiler to know that *any* variable was found, it also needs to know *which* variable was found. Similarly, if `symbol` is a one-argument message selectors then it is sent to the PTB with the token value as an argument. Note that these rules apply only to productions with a single token class terminal as their right-hand-side.

Finally, grammar rules of the form `A : B`, with no PTB directive, are special. When a rule of this form is recognized by the parser, the AST associated with the single right-hand-side nonterminal is simply passed to the left-hand-side. For example, no PTB directive is needed on the production

```
Term   :   '(' Expr ')'    ;
```

7 Smalltalk Support for Parse Tree Node Classes

In order to build ASTs from Smalltalk "node" classes, the PTB sends certain messages which must be understood by the nodes. The PTB sends the new message to a directive specified class to create a new AST node. Thus, nodes that requires special initialization should be sure to reimplement the new class message accordingly. If the production indicates that the newly created node will have children then the new node is sent the addChildrenInitial: with an OrderedCollection of ASTs as an argument. The order of the children will correspond to the order of nonterminals in the relevant grammar rule. Rules with a "lift" directive will send either the message addChildrenFirst: (for `liftRightChild`) or addChildrenLast: (for `liftLeftChild`) to the extant parent node with the same kind of argument as described above. Note that in the case of "lift" directives the number of children will be the number of right-hand-side nonterminals minus one, since the parent node is also taken from the right-hand side. Nodes that need to retain specific token value information must also implement the setAttribute: message, as described in Section 6.

It is common to have AST nodes that represent collections of related nodes or to have node collections and single nodes intermingled as logical children. The class OrderedChildren (a subclass of OrderedCollection), provided in the T-gen source code, is intended to aid in such situations. It can be used just like any other node class (for instance, in place of ArgumentListNode in one of the previous examples), but will be installed in its parent node as a single logical child.

In summary, all classes to be used as AST nodes should understand the messages:

addChildrenInitial: anOrderedCollection
addChildrenFirst: anOrderedCollection
addChildrenLast: anOrderedCollection
setAttribute: aString

The abstract class ParseTreeNode (a subclass of TreeNode) implements reasonable default versions of these messages, and is provided in the T-gen source code. It provides a good place to root AST node hierarchies.

8 A Comprehensive Example

This section describes how T-gen can be used to solve a simple, but practical, translation problem at varying levels of complexity. The problem is to construct a directed graph object from a sequence of graph building commands. Commands of the form `node <name1>` indicate that a new vertex with

INPUT:

node A
node B
node C
edge B C
edge A B
node D
edge C D
edge D C

OUTPUT:

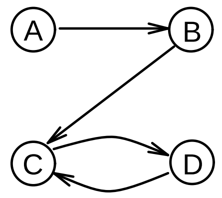


Figure 4: Sample input and output for the graph builder problem.

the given label should be created. Commands of the form edge <name1> <name2> mean to add an edge from the vertex labeled <name1> to the vertex labeled <name2>. A sample command sequence and the corresponding graph are shown in Figure 4.

The first step in using T-gen to solve this kind of problem is to define specifications for the token vocabulary and input structure. There are three different kinds of tokens in this problem: the keywords node and edge, vertex label identifiers, and the whitespace character used for separators. The keyword tokens will be handled in the grammar specification. The other tokens are handled with the following token class specification:

```
<name>         :  [A-Za-z][A-Za-z0-9]*    ;
<whitespace>   :  [\s\t\r]+               {ignoreDelimeter} ;
```

The input is structured as a list of graph building commands and the following grammar specification is fairly straightforward:

```
CommandList   :  CreateNode CommandList
              |  CreateEdge CommandList
              |  "nothing"                  ;
CreateNode    :  'node' <name>              ;
CreateEdge    :  'edge' <name> <name>       ;
```

The "nothing" is a comment; the corresponding *empty* grammar rule indicates that a command list may be empty, specifying a graph with no vertices.

These specifications are typed into the left-hand panes of the T-gen interface and accepted (the grammar is both LL(1) and SLR(1)). The sample input is then tested. At this point in the problem solving process, the only appropriate testing options are Derivation and Trace (see Figure 5). These options are useful for debugging the initial grammar specification. In order to obtain a more succinct structural representation of the input (i.e. an AST), it is necessary to augment the grammar productions with PTB directives. But it is first necessary to determine what kind of AST will be helpful in solving the problem.

Recall that the goal is to build a directed graph from the command sequence. If we can obtain a list of command objects with arguments it would be a simple matter to iterate over this list and to build the corresponding graph. Towards this end, the AST shown in Figure 5 is appropriate and

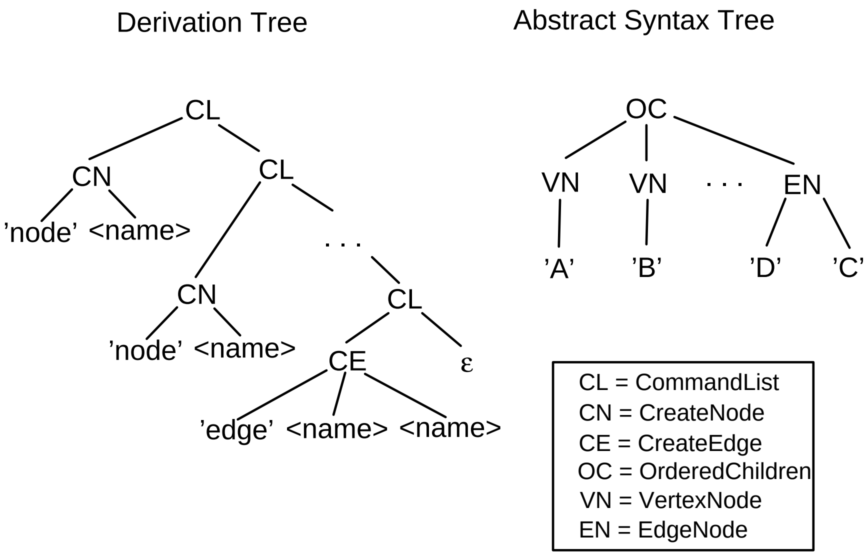


Figure 5: Results of different testing options for graph builder problem.

can be obtained with the following grammar specification:

```
CommandList   :  CreateNode CommandList   {liftRightChild}
              |  CreateEdge CommandList   {liftRightChild}
              |  "nothing"                {OrderedChildren} ;
CreateNode    :  'node' Name              {VertexNode} ;
CreateEdge    :  'edge' Name Name         {EdgeNode} ;
Name          :  <name>                   {NameNode} ;
```

When the end of the command list is reached, an empty OrderedChildren node is created. As the parser backups each command object is added to this collection using the liftRightChild PTB directive. The command objects are either VertexNodes or EdgeNodes with NameNode arguments. The `Name` production allows the actual text of the command argument to be saved in the NameNode. At this point the sham AST testing option can be used to debug the PTB directives and obtain the AST structure shown in Figure 5.

In order to build an actual AST with instances of specific AST node classes, the classes mentioned in PTB directives must be properly defined (as described in Section 7). An implementation of the necessary classes is shown in Figure 6. Once an AST is built from these node classes, methods can be added which traverse the structure and create a digraph during the traversal. Hence, translators generated by T-gen can be used in a two-pass process: the first pass verifies the structure of the input and builds an AST, the second pass traverses the AST the produces the desired translation target.

Using a user-defined parse tree builder it is possible for T-gen to construct a single-pass translator that goes directly from the input to the target. In this case, PTB directives are given as message selectors which are implemented by the new PTB class:

```
CommandList   :  CreateNode CommandList   {addVertex:toGraph:}
              |  CreateEdge CommandList   {addEdge:toGraph:}
              |  "nothing"                {createGraph} ;
CreateNode    :  'node' Name              {createVertexLabeled:} ;
CreateEdge    :  'edge' Name Name         {edgeFrom:to:} ;
Name          :  <name>                   {answerArgument:} ;
```

```
ParseTreeNode subclass: #NameNode
	instanceVariableNames: 'name'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'T-gen-Examples'


NameNode methodsFor: building parse trees

setAttribute: aString
	name := aString



ParseTreeNode subclass: #VertexNode
	instanceVariableNames: 'node'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'T-gen-Examples'


VertexNode methodsFor: building parse trees

addChildrenInitial: anOrderedCollection
	anOrderedCollection size = 1
		ifTrue: [node := anOrderedCollection first]
		ifFalse: [self error: 'wrong number of children']



ParseTreeNode subclass: #EdgeNode
	instanceVariableNames: 'fromNode toNode'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'T-gen-Examples'


EdgeNode methodsFor: building parse trees

addChildrenInitial: anOrderedCollection
	anOrderedCollection size = 2
		ifTrue:
			[fromNode := anOrderedCollection removeFirst.
			toNode := anOrderedCollection first]
		ifFalse: [self error: 'wrong number of children']
```

Figure 6: AST node classes for graph builder.

The corresponding PTB class definition is shown in Figure 7.

When testing the GraphBuilder PTB class it was discovered that, due to the structure of the CommandList grammar production, the command list was processed backwards. This caused an error when an edge was introduced between two (as yet) nonexistent vertices. (This is not a problem in the two-pass translation.) The best solution is to restructure the grammar as follows, also making minor changes to the GraphBuilder class methods:

```
CommandList   :  CommandList CreateNode   {toGraph:addVertex:}
              |  CommandList CreateEdge   {toGraph:addEdge:}
              |  "nothing"                {createGraph} ;
CreateNode    :  'node' Name              {createVertexLabeled:} ;
CreateEdge    :  'edge' Name Name         {edgeFrom:to:} ;
Name          :  <name>                   {answerArgument:} ;
```

This grammar is SLR(1) but not LL(1).

Using the modified grammar and GraphBuilder class, the result of the test input is a digraph object. T-gen-independent scanner and parser classes for the graph builder can be created using the **install** menu item. These classes can then be filed-out and distributed to other programmers.

9 Grammar Analysis and Error Handling

Debugging a translator specification can be as challenging as debugging a computer program, especially if no debugging aids are provided. T-gen attempts to provide at least minimal support for this process.

T-gen signals lexical and syntactic specification errors by inserting a highlighted error message into the specification text at the approximate location where the error was detected. Where possible, the message includes information about what characters or tokens were expected.

Token class specifications are relatively simple. Hence, most token class specification errors will be of either a syntactic nature (detected and signaled by T-gen) or of a logical nature beyond the capabilities of T-gen to detect (i.e. the wrong regular expression for the desired token).

On the other hand, grammar specifications can be quite complex and may suffer from several kinds of maladies. Two easily detected grammar errors are caused by useless nonterminals. A *useless nonterminal* is one which is either

- unreachable in any derivation beginning with the start symbol, or
- not be capable of deriving a terminal string.

Each of these errors is illustrated in the following grammar by the nonterminals C and B, respectively

$$\begin{array}{lcl} S & \rightarrow & A|B \\ A & \rightarrow & a \\ B & \rightarrow & Bb \\ C & \rightarrow & c \end{array}$$

Productions involving useless nonterminals may be removed, resulting in a *reduced grammar*. However, these errors are typically caused by misspellings or grammar maintenance activities. While T-gen could automatically reduce grammars, the appropriate solution is usually to correct the offending productions rather than to remove them. Useless nonterminals are detected by T-gen and the user is informed accordingly with a message in the transcript pane.

Grammars can also be ambiguous, may not generate the correct language, or may not be LL(1) or LR(1). Each of these are more serious errors that must be solved by restructuring the grammar and are beyond the capabilities of T-gen to correct.

```
AbstractSyntaxTreeBuilder subclass: #GraphBuilder
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'T-gen-Examples'


GraphBuilder methodsFor: production processing

addEdge: edgeArray toGraph: aGraph
	aGraph addEdgeFromNodeLabeled: (edgeArray at: 1)
		toNodeLabeled: (edgeArray at: 2).
	^aGraph

addToGraph: aGraph edge: edgeArray
	aGraph addEdgeFromNodeLabeled: (edgeArray at: 1)
		toNodeLabeled: (edgeArray at: 2).
	^aGraph

addToGraph: aGraph vertex: vertex
	aGraph add: vertex.
	^aGraph

addVertex: vertex toGraph: aGraph
	aGraph add: vertex.
	^aGraph

answerArgument: arg
	^arg

createGraph
	^LabeledDigraph new

createVertexLabeled: aString
	^NodeLabeledDigraphNode label: aString

edgeFrom: label1 to: label2
	^Array with: label1 with: label2
```

Figure 7: GraphBuilder class definition (LabeledDigraph and NodeLabeledDigraphNode are T-gen support classes and are provided in the T-gen source code).

10 Conclusion

T-gen has been used by a number of individuals and groups for a wide variety of applications from specialize language parsers to command processors to constructing structured objects from high-level specifications. Advanced features of T-gen include:

- user-defined parse tree builders
- automatic transformation of non-LL(1) and RRPG grammars
- the ability to inspect transformed grammars and parsers (useful for debugging and manual conflict resolution)
- automatic keyword detection and zero, one, or two token scanner lookahead
- creation of scanner and parser classes for use independently of T-gen

Possible future enhancements to T-gen include:

- semantic preservation of translation symbols in transformed grammars
- graphical editor/debugger for scanners and parsers
- support specification of translation symbol semantics, which would permit automatic generation of parse tree builder classes
- automatic detection and handling of overlapping token classes

We would also like to consider more sophisticated grammar analysis and error handling techniques.

The Appendices provide several examples of T-gen specifications. The Smalltalk source code for T-gen and the PostScript source code for this document are available via anonymous ftp from `bikini.cis.ufl.edu` in the directory `/pub/smalltalk/T-gen`. The current version of T-gen runs under Release 4 of ParcPlace's *Objectworks/Smalltalk*. Questions, comments, problems, and suggestions should be sent to the author (author contact information is given at the bottom of the first page).

References

[ASU86] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin/Cummings, 1988.

[Joh75] Stephen C. Johnson. YACC— yet another compiler compiler. CSTR 32, Bell Laboratories, Murray Hill, NJ, 1975.

[LaL77] Wilf R. LaLonde. Regular right part grammars and their parsers. *Communications of the ACM*, 20(10):731–741, October 1977.

[Les75] M. E. Lesk. LEX — a lexical analyzer generator. CSTR 39, Bell Laboratories, Murray Hill, NJ, 1975.

[LP90] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*, volume 1. Prentice Hall, 1990.

[PDC90] Terence Parr, Hank Dietz, and Will Cohen. Purdue compiler-construction tool set. Technical Report TR-EE 90-14, Purdue University, February 1990.

Appendices

A Simple Expression Grammar

In the following simple expression grammar, multiplication has higher precedence than addition and both operators are right-associative.

Token class specification:

```
<space>  :  [\s\t\r]+   {ignoreDelimeter} ;
```

Grammar specification:

```
E    :  T Ec     {liftRightChild} ;
Ec   :  '+' E    {Plus}
     |           {nil} ;
T    :  P Tc     {liftRightChild} ;
Tc   :  '*' T    {Times}
     |           {nil} ;
P    :  'a'      {A}
     |  'b'      {B}
     |  'c'      {C} ;
```

Grammar status: LL(1) and SLR(1)

Test input: `a + b * c * c`

B List Grammar

The following grammar provides for the creation of one-level (flattened) lists with an arbitrary number of elements.

Token class specification: (none)

Grammar specification:

```
E   :  E P   {liftLeftChild}
    |        {OrderedChildren} ;
P   :  'a'   {A}
    |  'b'   {B}
    |  'c'   {C} ;
```

Grammar status: SLR(1)

Test input: `abcaa`

C Small Programming Language

Presented here are T-gen specifications for a programming language with simple arithmetic, assignment, and nested statement lists.

Token class specification:

```
<id>       :  [a-z]+          ;
<number>   :  [0-9]+          ;
<space>    :  [\s\t\r]+   {ignoreDelimeter} ;
```

Grammar specification:

```
Z         :  'program' Decls Stmts         {Program} ;
Decls     :  'var' IdList ':'  'integer'   {Decls} ;
IdList    :  Name IdList                   {liftRightChild}
          |  Name                          {IdList} ;
Stmts     :  'begin' SL 'end'              {Stmts} ;
SL        :  S SL                          {liftRightChild}
          |  Stmts SL                      {liftRightChild}
          |  S                             {StmtList}
          |  Stmts                         {StmtList} ;
S         :  Name ':=' E ';'               {Assign} ;
E         :  E '+' T                       {Plus}
          |  T                              ;
T         :  P '*' T                       {Times}
          |  P                              ;
P         :  '(' E ')'
          |  Name
          |  <number>                      {Number} ;
Name      :  <id>                          {Id} ;
```

Grammar status: SLR(1)

Test input:

```
program
      var
            a b c :  integer
begin
      a := 3;
      b := a * 4;
      begin
            c := a + b;
            a := a + 1;
      end
end
```

D T-gen Token Class Specifications

This appendix presents the T-gen specifications for T-gen token class specifications.

Token class specification:

```
<tokenclass>  :  \<[a-zA-Z_][a-zA-Z_0-9]*\>    ;
<directive>   :  \{[a-zA-Z_][a-zA-Z_0-9]*\}    ;
<char>        :  [!-\~]           "all printable ASCII characters"
<dchar>       :  \\[0-9][0-9][0-9]             ;
<ochar>       :  \\o[0-7][0-7][0-7]            ;
<hchar>       :  \\x[0-9A-F][0-9A-F]           ;
<eschar>      :  \\[!-\~]                      ;
<comment>     :  \"~[\"]*\"                    {ignoreComment} ;
<space>       :  [\s\t\r]+                     {ignoreDelimiter} ;
```

Grammar specification:

```
spec         :  rule spec
             |  ;
rule         :  tokenclass ':'  regexpr directive ';' ;
tokenclass   :  <tokenclass>
directive    :  <directive>
             |  ;
regexpr      :  catexpr '|' regexpr
             |  catexpr ;
catexpr      :  expr catexpr
             |  expr ;
expr         :  baseexpr '*'
             |  baseexpr '+'
             |  baseexpr '?'
             |  baseexpr ;
baseexpr     :  atom
             |  '(' regexpr ')'
             |  '[' atomlist ']'
             |  '~' '[' atomlist ']' ;
atomlist     :  listelmt atomlist
             |  listelmt ;
listelmt     :  atom
             |  atom '-' atom ;
atom         :  <char>
             |  <dchar>
             |  <ochar>
             |  <hchar>
             |  <eschar> ;
```

Grammar status: SLR(1)

E T-gen Grammar Specifications

This appendix presents the T-gen specifications for T-gen grammar specifications.

Token class specification:

```
<id>           :  [a-zA-Z_][a-zA-Z_0-9]*          ;
<tokenclass>   :  \<[a-zA-Z_][a-zA-Z_0-9]*\>      ;
<keyword>      :  ([a-zA-Z_][a-zA-Z_0-9]*\:)+     ;
<literal>      :  '(~[']|'')+'                   {compactDoubleApostrophes} ;
<comment>      :  \"~[\"]*\"                     {ignoreComment} ;
<space>        :  [\s\t\r]+                      {ignoreDelimiter} ;
```

Grammar specification:

```
gram          :  rule gram
              |  ;
rule          :  nonterm ':'  rightparts ';' ;
rightparts    :  rightpart '|' rightparts
              |  rightpart ;
rightpart     :  regexpr directive ;
regexpr       :  expr regexpr
              |  ;
expr          :  baseexpr 'list' baseexpr
              |  baseexpr '*'
              |  baseexpr '+'
              |  baseexpr '?'
              |  baseexpr ;
baseexpr      :  nonterm
              |  term
              |  '(' catexpr ')' ;
              |  ;
catexpr       :  regexpr '|' catexpr
              |  regexpr
nonterm       :  <id> ;
term          :  <literal>
              |  <tokenclass>
tcname        :  <id> ;
directive     :  '{' dirname '}'
              |  ;
dirname       :  <id>
              |  <keyword> ;
```

Grammar status: SLR(1)

F ASCII Character Set

0 NUL 0 0	20 DLE 10 16	40 SP 20 32	60 0 30 48	100 @ 40 64	120 P 50 80	140 ‘ 60 96	160 p 70 112
1 SOH 1 1	21 DC1 11 17	41 ! 21 33	61 1 31 49	101 A 41 65	121 Q 51 81	141 a 61 97	161 q 71 113
2 STX 2 2	22 DC2 12 18	42 " 22 34	62 2 32 50	102 B 42 66	122 R 52 82	142 b 62 98	162 r 72 114
3 ETX 3 3	23 DC3 13 19	43 # 23 35	63 3 33 51	103 C 43 67	123 S 53 83	143 c 63 99	163 s 73 115
4 EOT 4 4	24 DC4 14 20	44 $ 24 36	64 4 34 52	104 D 44 68	124 T 54 84	144 d 64 100	164 t 74 116
5 ENQ 5 5	25 NAK 15 21	45 % 25 37	65 5 35 53	105 E 45 69	125 U 55 85	145 e 65 101	165 u 75 117
6 ACK 6 6	26 SYN 16 22	46 & 26 38	66 6 36 54	106 F 46 70	126 V 56 86	146 f 66 102	166 v 76 118
7 BEL 7 7	27 ETB 17 23	47 ’ 27 39	67 7 37 55	107 G 47 71	127 W 57 87	147 g 67 103	167 w 77 119
10 BS 8 8	30 CAN 18 24	50 (28 40	70 8 38 56	110 H 48 72	130 X 58 88	150 h 68 104	170 x 78 120
11 HT 9 9	31 EM 19 25	51) 29 41	71 9 39 57	111 I 49 73	131 Y 59 89	151 i 69 105	171 y 79 121
12 LF A 10	32 SUB 1A 26	52 * 2A 42	72 : 3A 58	112 J 4A 74	132 Z 5A 90	152 j 6A 106	172 z 7A 122
13 VT B 11	33 ESC 1B 27	53 + 2B 43	73 ; 3B 59	113 K 4B 75	133 [5B 91	153 k 6B 107	173 { 7B 123
14 FF C 12	34 FS 1C 28	54 , 2C 44	74 < 3C 60	114 L 4C 76	134 \ 5C 92	154 l 6C 108	174 | 7C 124
15 CR D 13	35 GS 1D 29	55 - 2D 45	75 = 3D 61	115 M 4D 77	135] 5D 93	155 m 6D 109	175 } 7D 125
16 SO E 14	P36 RS 1E 30	56 . 2E 46	76 > 3E 62	116 N 4E 78	136 ^ 5E 94	156 n 6E 110	176 ~ 7E 126
17 S1 F 15	37 US 1F 31	57 / 2F 47	77 ? 3F 63	117 O 4F 79	137 _ 5F 95	157 o 6F 111	177 DEL 7F 127

KEY

octal	107		ASCII character
		G	
hex	47	71	decimal