SDTGEN

Syntax Directed Translator Generator

A unified scanner and LR(1) parser generator with automatic locally least-cost error repair

Version 1.0

8 Jul 2024

Roy J. Mongiovi

Table of Contents

1 Introduction....1
2 Acknowledgments....1
3 Sdtgen Command Line....2
3.1 Normal Parameters....2
3.1.1 -g – Display Standardized Grammar Productions....2
3.1.2 -h – Display Usage Message....2
3.1.3 -l – Display Input as it is Processed....2
3.1.4 -q – Perform Only a Syntax Scan of the Input....2
3.1.5 -r – Display the Token Regular Expressions....2
3.1.6 -t – Display the LR(1) Parsing Tables....2
3.1.7 -v – Display Verbose Conflict Repair Information....3
3.1.8 -x – Display a Cross-reference of Tokens....3
3.1.9 -w <Output File> – Select the Output File....3
3.2 Debug Parameters....3
3.2.1 -da – Display the Ancestors of Each LR(1) State....3
3.2.2 -dd – Display the Scanner DFA Before Minimization....3
3.2.3 -de – Display Error Repair Information....3
3.2.4 -df – Display Nonterminal First Sets....3
3.2.5 -dg – Display Input Grammar Productions....4
3.2.6 -di – Display LR(1) Item Sets....4
3.2.7 -dm – Display the Scanner DFA After Minimization....4
3.2.8 -dn – Display the Scanner NFA....4
3.2.9 -dp – Display the Parser Abstract Syntax Tree....4
3.2.10 -ds – Display the Scanner Abstract Syntax Tree....4
4 Input Format....4
4.1 Comments....4
4.2 Language Identifier....4
4.3 Language Title....5
4.4 Parser Options....5
4.4.1 AMBIGUOUS....5
4.4.2 ERRORREPAIR....5
4.4.3 SHIFTREDUCE....5
4.4.4 SPLITSTATES....6
4.5 Regular Expression Definition....6
4.5.1 Regular Expression....6
4.5.1.1 Elementary Regular Expressions....6
4.5.1.1.1 Strings....6
4.5.1.1.2 Character Class....6
4.5.1.1.3 Defined Regular Expressions....7
4.5.1.1.4 Parenthesized Regular Expressions....7
4.5.1.2 A Range of Characters....8
4.5.1.3 A Negated Character....8

4.5.1.4 Character Difference....8
4.5.1.5 Repetitions....8
4.5.1.5.1 A Range of Occurrences....8
4.5.1.5.2 A Fixed Number of Occurrences....8
4.5.1.5.3 An Arbitrary Number of Occurrences....9
4.5.1.5.4 One or More Occurrences....9
4.5.1.5.5 Optional Expressions....9
4.5.1.6 Concatenation....9
4.5.1.7 Alternation....9
4.5.2 Predefined Expressions....9
4.6 Scanner....10
4.6.1 Explicit Regular Expression....10
4.6.1.1 Token Lookahead....11
4.6.1.2 Attributes....11
4.6.1.2.1 Precedence....11
4.6.1.2.2 Associativity....11
4.6.1.2.3 Insertion Cost....11
4.6.1.2.4 Deletion Cost....12
4.6.1.2.5 Install....12
4.6.1.2.6 Ignore Case....12
4.6.2 Implicit Regular Expression....12
4.6.3 Token Alias....12
4.6.4 Ignored Regular Expression....13
4.6.5 Epsilon Token....13
4.6.6 End of File Token....13
4.6.7 install_token Callback....13
4.7 Parser Defaults....15
4.7.1 Distinguished Symbol....15
4.7.2 Error Repair Context....15
4.7.3 Error Repair Cost....15
4.8 Parser....16
4.9 Example Input File....16
5 Conflict Resolution....24
5.1 Shift-Reduce Conflicts....24
5.2 Reduce-Reduce Conflicts....27
6 Error Repair....30
6.1 Example Repair....33
7 Sdtgen Table Format....33
7.1 Header Line....33
7.2 Scanner Tables....34
7.2.1 End of Token Tables....34
7.2.1.1 End of Token Index....34
7.2.1.2 End of Token Values....34
7.2.2 Final State Table....35

7.2.3 Install Flag Table....35
7.2.4 Scanner Transitions Table....35
7.3 Sdtgen Parser Tables....35
7.3.1 Terminal Insertion Costs....35
7.3.2 Terminal Deletion Costs....35
7.3.3 Left Hand Side Token Numbers....35
7.3.4 Right Hand Side Lengths....35
7.3.5 Semantic Routine Numbers....36
7.3.6 Error Repair Values....36
7.3.7 String Tables....36
7.3.7.1 String Table Index....36
7.3.7.2 String Table Values....36
7.3.8 Parser Action Tables....36
8 Packtables....37
8.1 Packtables Command Line....37
8.2 Scanner Table Packing....37
8.3 Parser Table Packing....38
8.4 Packparse Table Format....39
8.4.1 Packed Scanner Table....39
8.4.1.1 Scanner Default State Table....39
8.4.1.2 Scanner Base Index Table....39
8.4.1.3 Scanner Check State Table....39
8.4.1.4 Scanner Next State Table....40
8.4.2 Packed Parser Table....40
8.4.2.1 Parser Base Index Table....40
8.4.2.2 Parser Check State Table....40
8.4.2.3 Parser Action Table....40
9 Tableformat....40
9.1 Tableformat Command Line....41
10 Using the Parser....41
10.1 perform_action Callback....42
10.2 Generating Error Messages....43
11 Example Driver....44

Index of Tables

Table 1: Character Escape Sequences....7
Table 2: Predefined Regular Expressions....9
Table 3: Tables Header Values....34
Table 4: Parsing Action Encoding....37

1 Introduction

Sdtgen is a table-driven scanner and LR(1) parser generator which provides automatic error repair so that parsing can continue in the event of a syntax error. Language tokens are defined by regular expressions in the scanner section of the input and include insertion and deletion costs among their attributes. If error repair has been enabled the generated parser will try inserting and deleting tokens searching for the least-cost repair which allows parsing to continue. Repairs are reported as a list of inserted tokens, a list of deleted tokens, or a list of replaced tokens.

Sdtgen is implemented in itself and serves as an example for how to use the tables produced by the generator. A minimal driver program is include as a starting point for using the generated scanner and parser.

2 Acknowledgments

The author would like to express his appreciation to Dr. Ted Baker of Florida State who first convinced him to try his hand at developing a parser generator and to Dr. Rich Leblanc of Georgia Tech who pointed him along the way to locally least-cost error repair. This software would surely not exist without the inspiration they provided many years ago.

3 Sdtgen Command Line

sdtgen { -[ghlqrtvx] | -d[adefgimnps] | -w tables.dat } [<input file>]

3.1 Normal Parameters

3.1.1 -g – Display Standardized Grammar Productions

Displays the augmented input grammar as it will be used when generating the parser. Whereas productions in the input may have alternative right hand sides, for parser generation each production has its own left hand side. Production semantic routine numbers are listed separately from the production left and right hand sides.

Additionally, if error repair has been requested with the ERRORREPAIR parser option each production is listed with its assigned derivation steps and insertion cost. See the Error Repair section for more details.

3.1.2 -h – Display Usage Message

Displays the command line summary usage message and exits.

3.1.3 -l – Display Input as it is Processed

Each line of the input file is displayed as it is parsed. If this option is not specified only those lines which are required for error repair messages will be displayed.

3.1.4 -q – Perform Only a Syntax Scan of the Input

The input file will be checked for syntax errors but no scanner or parser will be produced.

3.1.5 -r – Display the Token Regular Expressions

Displays all the regular expressions that will be used to produce the scanner along with their tokens and attributes. See the Scanner section for more details.

3.1.6 -t – Display the LR(1) Parsing Tables

Displays the LR(1) parsing tables with “S” actions to shift to a new state, “SR” actions (if requested by the SHIFTREDUCE parser option) to shift a token and immediately reduce by a production number, “R” actions to reduce by a production number, and a single “A” action to indicate acceptance.

3.1.7 -v – Display Verbose Conflict Repair Information

Normally sdtgen will display shift-reduce or reduce-reduce conflicts and then resolve them if requested by the “AMBIGUOUS” and/or “SPLITSTATES” options respectively. If the “-v” parameter is specified sdtgen will also display some information about how the conflict was resolved.

3.1.8 -x – Display a Cross-reference of Tokens

Displays a list of all terminal and nonterminal tokens with the numbers of the grammar productions where they appear on the left hand side and/or the right hand side. Tokens are displayed sorted by name rather than by number.

3.1.9 -w <Output File> – Select the Output File

Sets the path for the scanner and parser tables file being generated. If specified as “-” the tables will be written to stdout. The default output file is “tables.dat”.

3.2 Debug Parameters

3.2.1 -da – Display the Ancestors of Each LR(1) State

Displays a list of all LR(1) states with the GOTO token that led to their creation and a list of the states that reference them.

3.2.2 -dd – Display the Scanner DFA Before Minimization

Displays the scanner deterministic finite state automaton as it was produced from the nondeterministic automaton. Each state lists the token numbers which end in this state, the token number which is recognized by this state, and the state character transitions.

3.2.3 -de – Display Error Repair Information

Displays the error repair action associated with each parser state if the ERRORREPAIR option is specified. Actions are either a token number which is appended to the continuation string generated for error repair or a production number triggering a reduce. See the Error Repair section for more details.

3.2.4 -df – Display Nonterminal First Sets

Displays the set of terminals which may begin the derivation of each nonterminal along with a “nullable” flag indicating that the nonterminal may derive epsilon.

3.2.5 -dg – Display Input Grammar Productions

Displays the parser grammar as produced from the input abstract syntax tree before it is converted into the standard format used by the parser generator.

3.2.6 -di – Display LR(1) Item Sets

Displays the LR(1) character finite state machine item sets including their lookahead sets.

3.2.7 -dm – Display the Scanner DFA After Minimization

Displays the scanner deterministic finite state automaton after redundant states have been removed.

3.2.8 -dn – Display the Scanner NFA

Displays the scanner nondeterministic finite state automaton as it was produced from the input abstract syntax tree. Each state lists the number of the token which ends in this state, the number of the token which is recognized by this state, the character set transitions for this state, and the list of next state numbers.

3.2.9 -dp – Display the Parser Abstract Syntax Tree

Displays the abstract syntax tree produced from the Parser section of the input file.

3.2.10 -ds – Display the Scanner Abstract Syntax Tree

Displays the abstract syntax tree produced from the Scanner section of the input file.

4 Input Format

4.1 Comments

```
% This is a full line comment
% This is a terminated comment %
```

Comments in the input file are introduced by the percent sign character “%”. They continue until end of line or another percent sign. Percent signs inside other tokens are not recognized as comments.

4.2 Language Identifier

```
IDENT identifier;
```

The sdtgen input file begins with an identifier for the language being defined. This identifier will be stored in the header line of the resulting Sdtgen Table Format output file and used as

the variable name for the C sdt_tables structure which is generated by Tableformat to contain all the scanner and parser data. A pointer to this structure is passed to the parser to control scanning and parsing the input file.

4.3 Language Title

TITLE “string”;

The sdtgen input file may contain an optional title string. This string may be used to provide information about the language defined by this file. It will be displayed on output produced by sdtgen display parameters but otherwise has no functional impact on the language being defined.

4.4 Parser Options

OPTIONS $option_1$, $option_2$, ...;

The sdtgen input file may contain an optional list of options which control some features of the generator. These options are case insignificant:

4.4.1 AMBIGUOUS

If the grammar in this input file generates shift-reduce conflicts and the AMBIGUOUS option has been specified, sdtgen will use the Precedence and Associativity values defined for the tokens in the Scanner section to choose whether to generate a shift or reduce action to resolve the conflict. See the Shift-Reduce Conflicts section for more details.

4.4.2 ERRORREPAIR

If this option is specified sdtgen will produce the table needed for automatic error repair. See the Error Repair section for more details.

4.4.3 SHIFTREDUCE

If this option is specified sdtgen will use default reduce actions to decrease the number of states required by the parser. If a parsing state contains only one item which would generate a shift on a particular token, and that token is the last token on the right hand side of a production, the parser would normally shift to another state and then immediately reduce by the recognized production. At the cost of delaying syntax error detection a bit we can replace that shift with a “shiftreduce” action which immediately reduces by the recognized production (effectively combining the shift with a default reduce) and avoid generating the state required for the shift. Any syntax error which would have been detected by the reduce in the state we are omitting will still be detected before shifting another input token.

4.4.4 SPLITSTATES

To generate a parser sdtgen creates the LR(0) characteristic finite state machine and then adds the LALR(1) lookahead sets to disambiguate reduce actions. If this results in reduce-reduce conflicts the SPLITSTATES option may be specified to identify the states whose merging by the LR(0) CFSM led to the conflicts. Provided the grammar is actually LR(1), sdtgen will create additional states to effectively "un-merge" the LR(0) states as necessary to resolve the conflict. See the Reduce-Reduce Conflicts section for more information.

4.5 Regular Expression Definition

```
DEFINE identifier1 = <expression1>;
       identifier2 = <expression2>;
       …
```

To improve readability, Regular Expressions may be given names for use in later regular expressions. This section of the input file is optional.

4.5.1 Regular Expression

Regular expressions are described by combining the following elements, listed from highest precedence to lowest.

4.5.1.1 Elementary Regular Expressions

At the highest precedence level are the elementary regular expressions for strings, character classes, regular expression definition identifiers, and parenthesized regular expressions.

4.5.1.1.1 Strings

```
"string1" or
'string2'
```

A regular expression which recognizes a string of characters is specified by surrounding the characters with quotes. Sdtgen allows strings to be quoted with either single or double quotes rather than providing an escaped quote character within the string.

4.5.1.1.2 Character Class

```
[character class] or
[^inverted class]
```

A single character which matches any of a set of characters is specified by a character class surrounded by square brackets "[" and "]".

If the first character of the class is a caret “^” the sense of the class is inverted: it will match any 8 bit character 0-255 except for those listed within the class. EOF is automatically excluded from an inverted character class.

A range of characters is specified as first character dash last character as in “a-z”. If the first character is not less than or equal to the last character no characters are added to the class. If the dash occurs at the beginning or end of the character class so that there is either no first or no last character then the dash is interpreted as an actual dash.

Within both strings and character classes the following escape sequences are interpreted:

Table 1: Character Escape Sequences

\a	Alert - \x07
\b	Backspace - \x08
\e	Escape - \x1B
\f	Formfeed - \x0C
\n	Newline - \x0A
\r	Carriage Return - \x0D
\t	Horizontal Tab - \x09
\v	Vertical Tab - \x0B
\\	Backslash - \x5C
\nnn	Character with 8 bit octal value “nnn”. Leading zeroes may be omitted.
\xhh	Character with 8 bit hexadecimal value “hh”. Leading zeroes may be omitted.

A backslash followed by a character not listed above is generally just interpreted as the following character. The exception to this is within a character class “\]” may be used to add a close bracket to the class. Note that “\-” does not escape the function of a dash specifying a range of characters.

4.5.1.1.3 Defined Regular Expressions

identifier

The name of a predefined regular expression inserts a copy of that regular expression.

4.5.1.1.4 Parenthesized Regular Expressions

(<expression>)

A parenthesized regular expression groups the contained expression as a unit.

4.5.1.2 A Range of Characters

<expression> : <expression>

Two regular expressions which both represent single characters may be separated by a colon ":" to match any single character in that range. For example, "a":"z" is a single character which will match anything in the range "a" through "z". This is equivalent to the character class [a-z] except that with the colon "a" and "b" could be predefined regular expression identifiers, for example.

4.5.1.3 A Negated Character

~ <expression>

A regular expression which represents a single character may be prefixed with a tilde "~" to match any single character except for it. EOF is automatically excluded from the negated character value. For example, ~"a" matches any character in the range 0-255 except for "a". This is equivalent to the character class [^a] except that with the tilde the "a" could be a predefined regular expression identifier, for example.

4.5.1.4 Character Difference

<expression> - <expression>

Two regular expressions which represent single characters or character classes may be separated by a minus sign to match any character which is in the first expression but not in the second. For example, "a":"z"-"x" will match any character in the range "a" through "z" except for an "x".

4.5.1.5 Repetitions

The following expressions may be used to match the selected number of repetitions of a regular expression.

4.5.1.5.1 A Range of Occurrences

{<expression>} lower : upper

This matches a range of repetitions of the given regular expression. Lower and upper must be integers and lower must be less than or equal to upper.. For example, {[a-fA-F0-9]} 1:4 will match one through four hexadecimal digits in upper or lower case.

4.5.1.5.2 A Fixed Number of Occurrences

{<expression>} count

This matches exactly the specified number of repetitions of the given regular expression. For example, {"a":"z"} 5 matches exactly five characters each in the range of a through z.

4.5.1.5.3 An Arbitrary Number of Occurrences

{<expression>} *or* <expression> *

This matches zero or more repetitions of the given regular expression. The curly brace version of this result is convenient when precedence requires the expression to be surrounded by parenthesis in order to use the Kleene star.

4.5.1.5.4 One or More Occurrences

<expression> +

This matches one or more occurrences of the given regular expression.

4.5.1.5.5 Optional Expressions

<expression> ?

This matches zero or one occurrence of the given regular expression.

4.5.1.6 Concatenation

<expression> <expression>

Two regular expressions adjacent to each other match the first regular expression followed by the second.

4.5.1.7 Alternation

<expression> | <expression>

Two regular expressions separated by the pipe symbol "|" match the first expression or the second expression (but not both).

4.5.2 Predefined Expressions

Sdtgen automatically defines the following regular expressions:

Table 2: Predefined Regular Expressions

NUL	Null - 0x00	EOL	End of line - 0x0A	SYN	Synchronous idle - 0x16
SOH	Start of header - 0x01	VT	Vertical tab - 0x0B	ETB	End of trans. block - 0x17
STX	Start of text - 0x02	FF	Form feed - 0x0C	CAN	Cancel - 0x18
ETX	End of text - 0x03	CR	Carriage return - 0x0D	EM	End of media - 0x19
EOT	End of trans. - 0x04	SO	Shift out - 0x0E	SUB	Substitute - 0x1A
ENQ	Enquiry - 0x05	SI	Shift in - 0x0F	ESC	Escape - 0x1B

ACK	Acknowledge - 0x06	DLE	Data link escape - 0x10	FS	File separator - 0x1C
BEL	Bell - 0x07	DC1	Device control 1 - 0x11	GS	Group separator - 0x1D
BS	Backspace - 0x08	DC2	Device control 2 - 0x12	RS	Record separator - 0x1E
HT	Horizontal tab - 0x09	DC3	Device control 3 - 0x13	US	Unit separator - 0x1F
LF	Line feed - 0x0A	DC4	Device control 4 - 0x14	DEL	Delete - 0x7F
NL	Newline - 0x0A	NAK	Not acknowledge - 0x15	EOF	End of file

EOF is a special “character” with value 256 which is used internally by the scanner to represent end of file. The only way to reference it is via the EOF definition.

4.6 Scanner

```
SCANNER <token definition1>;
        <token definition2>;
        ...
```

This section of the input file defines the regular expressions which are recognized by the generated scanner. These regular expressions are given terminal symbol names for use in the Parser along with attributes which affect how the scanner is generated and how these terminals are handled by the parser generator. All terminal tokens must be defined in the scanner in order to be used in the Parser grammar later on.

If the scanner encounters a character which matches none of the defined regular expressions, it issues an error message, skips over the character, and continues searching for the next token.

There are four formats for scanner regular expression definitions: a token may be defined with an Explicit Regular Expression or by using the token name itself as an Implicit Regular Expression, a token may be defined as a Token Alias for another token, and finally an Ignored Regular Expression may be defined which is matched and discarded by the scanner.

4.6.1 Explicit Regular Expression

```
“token” = <token expression> <attributes>;
```

This defines “token” as the name of a terminal token to be used later in the grammar. The generated scanner will match the given token expression and return the number assigned to this token. If more than one token expression matches the same sequence of characters the one listed first will be recognized. The token attributes are optional and may be omitted if the default values are acceptable.

4.6.1.1 Token Lookahead

A token expression is a regular expression as described in the Regular Expressions section with the addition of an optional lookahead regular expression:

<expression> *or* <expression> / *or* <expression> / <expression>

The regular expression before the slash defines the token and indicates where it begins and ends in the scanned text. But in order for the scanner to recognize that this token has been found the regular expression after the slash must also match. After the token has been recognized the characters which matched the regular expression after the slash are returned to the input stream and will be reexamined when the scanner looks for the next token.

4.6.1.2 Attributes

A token regular expression is optionally followed by a comma and a list of one or more attributes to be associated with the token being defined.

4.6.1.2.1 Precedence

, precedence = integer

This specifies a precedence value for this token to be used when the grammar contains a shift-reduce conflict and the AMBIGUOUS option has been specified. Smaller integers indicate higher precedence. The default token precedence is 0. See the Shift-Reduce Conflicts section for more details.

4.6.1.2.2 Associativity

, associativity = left or , associativity = right or , associativity = none

This specifies an associativity value for this token to be used when the grammar contains a shift-reduce conflict, the AMBIGUOUS option has been specified, and the shift and reduce precedence values are equal. If the token associativity is “none”, which is the default value, the shift-reduce conflict cannot be resolved. See the Shift-Reduce Conflicts section for more details.

4.6.1.2.3 Insertion Cost

, insert = integer

This specifies the cost of inserting this token while repairing a syntax error. The default insert cost is 1. See the Error Repair section for more details.

4.6.1.2.4 Deletion Cost

, delete = integer

This specifies the cost of deleting this token while repairing a syntax error. The default delete cost is 1. See the Error Repair section for more details.

4.6.1.2.5 Install

, install

This specifies that the parser should push the actual string that matched this token onto the parse stack along with the token number. This allows the actual value of tokens which are variable, like [0-9]+ for an integer, to be available to the semantic routines for processing. In addition, the presence of this flag causes the scanner to call the install_token Callback function, passing it the new token stack entry. This allows a new input token to be examined, and potentially modified, before being passed on to the parser.

4.6.1.2.6 Ignore Case

, ignore case

This specifies that the scanner should treat every alphabetic character in this token's regular expression as though it is both upper and lower case. So, for example, a hexadecimal digit could be described as [a-fA-F0-9] or simply as [a-f0-9] (or [A-F0-9]) with the Ignore Case attribute.

4.6.2 Implicit Regular Expression

"token" <attributes>;

If the "= <token expression>" part of the token definition is omitted, the "token" string itself will be used as the defining token expression. This allows tokens like "+" to be defined without having to specify "+" = "+".

4.6.3 Token Alias

"token alias": "token definition" <attributes>;

This allows a new token alias name to be assigned the same token number as a previous token definition but with a new and normally different set of attributes. The scanner will return the token number of the original token definition when its regular expression is matched, but the attributes used by the parser generator will be those of the alias. For the purposes of GOTO and lookahead computation the alias appears to be the original token.

Note that the install and ignore case attributes have no effect when specified by an alias. The values which were specified on the original token which defined the token expression are the ones which have an effect.

4.6.4 Ignored Regular Expression

```
ignore <token expression>;
```

This specifies a token expression which is to be discarded by the scanner when recognized. This is generally used to skip over whitespace between tokens as well as to remove comments from the input stream. Be aware that the scanner has to collect all the matching text into its input buffers before it can ignore it so if things like comments are effectively unbounded they can drastically increase memory usage just by being ignored.

4.6.5 Epsilon Token

```
“token” = “” <attributes>;
```

When resolving shift-reduce errors with the AMBIGUOUS option, the precedence of the reduce production is that of the last terminal token on the right hand side. This is generally the correct thing to do but the possibility exists that the right hand side is composed entirely of nonterminals or the precedence of the rightmost terminal doesn’t give the correct result. To give the production a context dependent precedence value an epsilon token can be defined with that precedence and added to the end of the right hand side in question. The scanner will never actually return this token since it does not match epsilon regular expressions but the terminal can be placed in the grammar without any effect. As the dot moves through productions by the GOTO operation it will automatically skip over these epsilon tokens. They are present for the benefit of resolving shift-reduce conflicts but otherwise it is as though they do not exist.

4.6.6 End of File Token

The scanner generator automatically appends a special token to the end of the token list. This token ‘”’$’””, which matches end of file, is used by the parser generator when it augments the input grammar. It has an insertion cost of 49999 and a deletion cost of 99999 to strongly suggest that error repair not affect it. The name is chosen so that it cannot possibly conflict with any token defined in the input file.

4.6.7 install_token Callback

After a terminal with the Install attribute has been identified by the scanner but before it is passed to the parser, the scanner calls install_token to allow it to be inspected:

```
void install_token(sdt_tables *tables, tokenentry *token);
```

The first parameter is a pointer to the sdt_tables structure created by sdtgen to control the parse. The second parameter is a pointer to the newly recognized token:

```
struct tokenentry               /* One entry on the token stack */
```

```
{
  int                token;       /* Token number for parser */
  unsigned char *symbol;     /* Token string (if installed) */
  location           locus;       /* Start of containing line */
  location           where;      /* Token start position */
};
```

The token member and the symbol member will be the actual token number and the matching string identified by the scanner. The members locus and where are pointers into the I/O buffers which are used when generating error messages and aren't important here.

The install_token function can be used to examine the symbol string and change the token number for tokens which cannot be identified by their character pattern alone. An example from the C language will hopefully make this clear.

In C, there are three tokens which all match the same pattern of alphanumeric characters and underscore: an identifier, a typedef name, and an enum constant. The C grammar will contain productions using each of those tokens. But when we define them for the scanner with:

```
"identifier"                = [a-zA-Z_] [a-zA-Z_0-9]*, install;
"typedef name"              = [a-zA-Z_] [a-zA-Z_0-9]*, install;
"enumeration constant" = [a-zA-Z_] [a-zA-Z_0-9]*, install;
```

only the identifier token will ever be recognized because all three tokens match the same regular expression and the first one declared takes precedence.

This problem can be solved by having the semantic routines which define typedef names and enumeration constants place the matching identifiers (which aren't typedefs or enums yet) into a dictionary. Subsequently, when the scanner sees an identifier and calls install_token it can consult that dictionary and determine the actual token which has been recognized.

To map the identifier token to typedef name or enumeration constant we need the token numbers that were assigned to those tokens. We can look them up with:

```
lookup_token(tables, “typedef name”, TERMINAL, LOOKUP)
lookup_token(tables, “enumeration constant”, TERMINAL, LOOKUP)
```

This function returns a pointer to a name structure:

```
struct name                    /* Name to token number mapping */
{
  unsigned char *name;      /* Terminal/nonterminal name without quotes */
  int                type;       /* TERMINAL or NONTERMINAL */
  int                token;      /* Corresponding token number */
```

```
  nameentry     *next;          /* Next entry in hash table bucket */
};
```

or NULL if there is no matching table entry. Just replace the value of the token member in the tokenentry which was passed to install_token with the appropriate token value from the struct name to correctly identify the token.

Of course, if the language scanner is no longer being modified you can use the -r – Display the Token Regular Expressions sdtgen parameter to find the typedef name and enumeration expression token numbers and hard-code them.

4.7 Parser Defaults

```
DEFAULT identifier1 = value1;
        identifier2 = value2;
        …
```

The sdtgen input file may contain an optional list of default values to be used by the generated parser.

4.7.1 Distinguished Symbol

```
START = <nonterminal>;
```

The parser generator normally selects the first left hand side symbol encountered in the input grammar as the distinguished symbol – the goal the parser will recognize. If that isn't the correct production the Distinguished Symbol default can be used to specify the correct nonterminal.

4.7.2 Error Repair Context

```
CONTEXT = integer;
```

When searching for the least cost error repair the parser will check a potential correction to see if it allows a specified number of future input tokens to be successfully parsed. If the repair doesn't make the next Error Repair Context input tokens valid its cost is increased by an amount proportional to the number of unparsed context tokens. The default context is 0 so if error repair is being used this default should be specified to allow for improved repair quality.

4.7.3 Error Repair Cost

```
COST = integer;
```

If the parser encounters another syntax error while checking the Error Repair Context of a potential repair it will add a percentage of this additional default cost to the calculated cost of the tokens inserted and deleted by the repair. If the repair makes the next “context” input tokens valid then the additional cost is 0. The fewer of the context tokens that are made valid the higher the additional cost. If none of the context tokens are made valid the entire additional cost will be added. The default Error Repair Cost is 0 so if error repair is being used this default should be specified to allow for improved repair quality.

4.8 Parser

```
PARSER <left hand side1> --> <right hand side1> | <right hand side2> | …;
          <left hand side2> --> <right hand side1> | <right hand side2> | …;
```

The final section of the sdtgen input file is the specification of the language grammar. The syntax is fairly standard BNF. The left hand side of each production is a nonterminal enclosed in angle brackets “<>”. The right hand side is a sequence of terminals enclosed in quotes and/or nonterminals enclosed in angle brackets. Terminals must have been defined in the Scanner section described above. Nonterminals may be used before they are defined later in the Parser section. Of course, if the nonterminal is not eventually defined the grammar will not recognize the language. Each right hand side may end in an optional semantic routine number which is represented as @integer. The parser will call the function perform_action with that semantic routine number when it reduces the right hand side to the left hand side symbol.

The parser generator automatically prepends the following production to the grammar:

```
<<Goal>> --> <distinguished symbol> ‘”’$’”’;
```

The <Goal> nonterminal contains angle brackets in its name which cannot be entered legally in the input file to ensure that this nonterminal cannot occur elsewhere in the input. Likewise, the terminal ‘”’$’”’ which represents end of file contains both single and double quotes and cannot occur elsewhere in the input file.

4.9 Example Input File

```
IDENT sdtgen;
TITLE "Syntax Directed Translator Generator Version 1.0";
```

This is the input file which generates the scanner and parser for sdtgen.

```
OPTIONS
  errorrepair, shiftreduce;
```

The parser being generated supports Error Repair, and SHIFTREDUCE actions will be used to reduce the number of states in the parser.

```
DEFINE
  alpha = "A" : "Z" | "a" : "z";
  digit   = [0-9];
```

For convenience, the Regular Expression Definitionsection is used to define two regular expressions to recognize alphabetic characters and decimal digits. For variety the first definition uses Regular Expression syntax and the second uses a Character Class. Either syntax could have been used for both.

```
SCANNER
  "IDENT",                    insert = 2, delete = 6, ignore case;
  "TITLE",                    insert = 2, delete = 6, ignore case;
  "OPTIONS",                  insert = 2, delete = 6, ignore case;
  "DEFINE",                   insert = 2, delete = 6, ignore case;
  "SCANNER",                  insert = 4, delete = 6, ignore case;
  "IGNORE",                   insert = 2, delete = 6, ignore case;
  "PRECEDENCE",               insert = 3, delete = 6, ignore case;
  "ASSOCIATIVITY",            insert = 3, delete = 6, ignore case;
  "LEFT",                     insert = 3, delete = 6, ignore case;
  "RIGHT",                    insert = 3, delete = 6, ignore case;
  "NONE",                     insert = 2, delete = 6, ignore case;
  "INSTALL",                  insert = 1, delete = 6, ignore case;
  "INSERT",                   insert = 2, delete = 6, ignore case;
  "DELETE",                   insert = 2, delete = 6, ignore case;
  "CASE",                     insert = 1, delete = 6, ignore case;
  "PARSER",                   insert = 4, delete = 6, ignore case;
  "DEFAULT",                  insert = 2, delete = 6, ignore case;
  "START",                    insert = 2, delete = 6, ignore case;
  "COST",                     insert = 2, delete = 6, ignore case;
  "CONTEXT",                  insert = 2, delete = 6, ignore case;
```

The reserved tokens in the language are defined using the Implicit Regular Expression format. These tokens are specified before the identifier token so that they take precedence over it. More specific regular expressions have to be declare before more general or they will not be recognized.

```
  "identifier" = (alpha | "_") {alpha | digit | "_"},  insert = 2, delete = 4, install;
```

The identifier token is defined using the Explicit Regular Expression format. An identifier must begin with an alphabetic character or an underscore which may be followed by zero or more alphanumerics or underscores. The install flag is specified so that the actual identifier name is stored on the parse stack along with the token number.

```
"symbol"    = "<" (~">" - EOL)+ ">"?,          insert = 2, delete = 4, install;
```

The symbol token is an open angle bracket followed by one or more characters which are not close angle brackets or an end of line followed by an optional close angle bracket. This regular expression allows a nonterminal symbol ostensibly bracketed by angle brackets to be recognized even if the close angle bracket has been mistakenly omitted. The install flag causes the actual symbol string including angle brackets to be stored on the parse stack along with the token. Semantic routines which interpret symbols check the final character of the symbol string and issue an error message if it is not a close angle bracket.

```
"string"    = "'" {~"'" - EOL} "'"?
            |  '"' {~'"' - EOL} '"'?,          insert = 1, delete = 2, install;
```

A string token may be surrounded with single or double quotes and must be contained on a single line. A pair of quotes with nothing between them is allowed and represents epsilon. As with the symbol token, the install flag causes the actual matching string to be stored on the parse stack and semantic routines which interpret strings check that the final character matches the first character and issue an error message if it does not.

```
"class"     = "[" (~"]" - "\\" - EOL | "\\" ~EOL)+ "]"?,    insert = 4, delete = 4, install;
```

A character class token consists of an open square bracket followed by one or more characters which are either something other than a close square bracket/backslash/end of line or a backslash followed by something other than an end of line. This allows a backslash escaped close bracket to be a member of the class. As with the previous two tokens the close square bracket is optional and the character class semantic routine checks the final character and issues an error message if it is not a close square bracket.

```
"integer"   = digit+,                          insert = 1, delete = 4, install;
```

The integer token is a string of one or more digit characters.

```
"semantic" = "@" digit+,                       insert = 3, delete = 6, install;
```

The semantic routine number token is an at sign followed by one or more digit characters.

```
"/",                                           insert = 4, delete = 8;
"|",                                           insert = 2, delete = 4;
"-",                                           insert = 3, delete = 4;
"{",                                           insert = 4, delete = 4;
"}",                                           insert = 3, delete = 2;
"?",                                           insert = 3, delete = 4;
"*",                                           insert = 2, delete = 4;
"+",                                           insert = 3, delete = 4;
":",                                           insert = 2, delete = 4;
```

```
  "~",                                  insert = 2, delete = 4;
  "(",                                  insert = 2, delete = 4;
  ")",                                  insert = 1, delete = 2;
  "-->",                                insert = 3, delete = 4;
  "=",                                  insert = 2, delete = 4;
  ",",                                  insert = 1, delete = 4;
  ";",                                  insert = 2, delete = 4;
```

As with the reserved tokens, the special character tokens are defined using the Implicit Regular Expression syntax.

```
  ignore        "%" {~"%" - EOL} "%"?;      %Comments
  ignore        (" " | FF | HT | EOL)+;     %Whitespace
```

Two regular expressions are defined with the Ignored Regular Expression syntax: comments and whitespace characters.

```
DEFAULT
  START      = <Input File>;                %Distinguished symbol
  CONTEXT = 5;                              %Number of symbols lookahead
  COST       = 20;                          %Default error correction cost
```

This input file specifies all the Parser Defaults. The distinguished symbol of the grammar is <Input File>. This happens to be the first production in the grammar but there's no harm in defining it. When validating an error repair the parser will check the next five input tokens. If a second syntax error occurs immediately after parsing the repair the repair cost will be incremented by 20. If the second syntax error occurs after successfully parsing one token of the context the repair cost will be incremented by ((5 - 1) / 5) * 20 = 16. If two tokens of context are successfully parsed the additional cost will be ((5 - 2) / 5) * 20 = 12, etc. If the entire context is successfully parsed there will be no additional cost.

```
PARSER
  <Input File> -->
        <Ident> <Title> <Options> <Definitions> <Scanner> <Defaults> <Parser>
  ;
  <Ident> -->
        "IDENT" "identifier" ";" @1                 %Set translator name
  ;
  <Title> -->
        "TITLE" "string" ";" @2                     %Set listing title string
      | ""
  ;
  <Options> -->
```

```
      "OPTIONS" <Identifier List> ";"
    | ""
;
<Definitions> -->
      "DEFINE" <Definition List>
    | ""
;
<Scanner> -->
      "SCANNER" <Token List> @3     %Finish creation of scanner regular expressions
;
<Defaults> -->
      "DEFAULT" <Default List>
    | ""
;
<Parser> -->
      "PARSER" <Production List> @4                %Finish creation of parser productions
;
```

The sdtgen Parser section describes an input file as an Ident followed by an optional Title, an optional Options list, an optional Definitions list, a list of Scanner tokens, optional Default values, and finally the Parser.

```
<Identifier List> -->
      <Identifier List> "," "identifier" @5        %Process option
    | "identifier" @5                              %Process option
;
```

The Parser Options identifier list is just a comma-separated list of identifiers.

```
<Definition List> -->
      <Definition List> <Definition>
    | <Definition>
;
<Definition> -->
      "identifier" "=" <Expression> ";" @6         %Define regular expression
;
```

The Regular Expression Definition list is a sequence of definitions which are terminated by semi-colons.

```
<Token List> -->
      <Token List> <Token> @7                      %Alternate regular expressions
    | <Token>
```

```
;
<Token> -->
      "string" "=" <Regular Expr> <Attributes> ";" @8 %Create token from  expression
    | "string" ":" "string" <Attributes> ";" @9          %Create alias for token
    | "string" <Attributes> ";" @10                      %Create token from name string
    | "IGNORE" <Regular Expr> ";" @11                    %Set expression should be ignored
;
```

The Scanner token list is a sequence of the various token definition types each of which is terminated with a semi-colon.

```
<Default List> -->
      <Default List> <Default>
    | <Default>
;
<Default> -->
      "START" "=" "symbol" ";" @12                       %Set start symbol for parser
    | "COST" "=" "integer" ";" @13                       %Set default repair cost
    | "CONTEXT" "=" "integer" ";" @14                    %Set error correction context
;
```

The Parser Defaults list is a list of default specifications terminated by semi-colons.

```
<Production List> -->
      <Production List> <Production> @15                 %Add production to grammar
    | <Production>
;
<Production> -->
      "symbol" "-->" <Alternate List> ";" @16            %Join LHS and RHS into production
;
```

The Parser production list is a sequence of productions with alternate right hand sides which is terminated with a semi-colon.

```
<Expression> -->
      <Expression> "|" <Term> @7                         %Alternate regular expressions
    | <Term>
;
<Term> -->
      <Term> <Factor> @17                                %Concatenate regular expressions
    | <Factor>
;
<Factor> -->
```

```
      "{" <Expression> "}" "integer" ":" "integer" @18 %Create range of occurrences
    | "{" <Expression> "}" "integer" @19               %Create multiple occurrences
    | "{" <Expression> "}" @20                        %Create 0 or more occurrences
    | <Transition> "*" @20                            %Create 0 or more occurrences
    | <Transition> "+" @21                            %Create 1 or more occurrences
    | <Transition> "?" @22                            %Create optional expression
    | <Transition>
;
<Transition> -->
      <Transition> "-" <Element> @23                  %Create character difference
    | <Element>
;
<Element> -->
      "~" <Secondary> @24                             %Create complement of character
    | <Secondary>
;
<Secondary> -->
      <Primary> ":" <Primary> @25                     %Create a range of characters
    | <Primary>
;
<Primary> -->
      "(" <Expression> ")"
    | "identifier" @26                                %Copy a predefined expression
    | "string" @27                                    %Create transitions for string
    | "class" @28                                     %Create transition for character class
;
```

Regular Expressions are define with multiple productions to enforce operator precedence.

```
<Regular Expr> -->
      <Expression Head> <Expression> @17              %Concatenate regular expressions
    | <Expression Head>
    | <Expression>
;
<Expression Head> -->
      <Expression> "/" @29                            %Create lookahead expression
;
```

Token expressions have an extra level of precedence to support the Token Lookahead operator.

```
<Attributes> -->
```

```
        "," <Attribute List>
      | ""
;
<Attribute List> -->
        <Attribute List> "," <Attribute>
      | <Attribute>
;
<Attribute> -->
        "PRECEDENCE" "=" "integer" @30          %Set token precedence
      | "ASSOCIATIVITY" "=" "LEFT" @31          %Set token is left associative
      | "ASSOCIATIVITY" "=" "RIGHT" @32         %Set token is right associative
      | "ASSOCIATIVITY" "=" "NONE" @33          %Set token has no associativity
      | "INSERT" "=" "integer" @34              %Set token insertion cost
      | "DELETE" "=" "integer" @35              %Set token deletion cost
      | "INSTALL" @36                           %Set token should be installed
      | "IGNORE" "CASE" @37                     %Set token is case insignificant
;
```

The optional token Attributes are a leading comma followed by a comma-separated sequence of the attributes.

```
<Alternate List> -->
        <Alternate List> "|" <Alternate> @7     %Add alternate to list
      | <Alternate>
;
<Alternate> -->
        <Symbol List> "semantic" @38            %Set semantic routine number
      | <Symbol List>
;
```

The right hand side of a production is a list of one or more alternatives separated by the pipe symbol. Each alternative may end in an optional semantic routine number.

```
<Symbol List> -->
        <Symbol List> <Symbol> @17              %Concatenate symbol list
      | <Symbol>
;
<Symbol> -->
        "symbol" @39                            %Create nonterminal
      | "string" @40                            %Create terminal
;
```

The actual right hand side is a list of one or more nonterminal symbols in angle brackets or quoted strings which are terminals.

5 Conflict Resolution

A parse conflict arises when a state in the parser indicates two different actions on the same token. If an item in the state has a dot in front of a terminal indicating a shift but that same terminal is also in the lookahead set of an item whose dot is all the way to the right indicating a reduce that is a shift-reduce conflict. If two reduce items with the dot all the way to the right have lookahead sets which contain one or more of the same terminals that is a reduce-reduce conflict.

Of course, shift-shift conflicts cannot arise because the GOTO operation moves the dot across all items with the same next token when creating the GOTO state.

5.1 Shift-Reduce Conflicts

The standard example of a shift-reduce conflict is a grammar which describes arithmetic expressions. We normally write this grammar with multiple levels of productions to reflect the precedence of the arithmetic operators: i.e. multiplication happens before addition.

```
<expr> --> <expr> “+” <term> | <expr> “-” <term> | <term>;
<term> --> <term> “*” <factor> | <term> “/” <factor> | <factor>;
<factor> --> “(“ <expr> “)” | “-” “int’ | “int”;
```

By introducing multiple layers of recursive productions this grammar ensures that unary minus happens before the multiplicative operators which happen before the additive operators. However, this same language can be described with a single recursive production:

```
<expr> --> <expr> “+” <expr> | <expr> “-” <expr> | <expr> “*” <expr> | <expr> “/” <expr>
          | “(“ <expr> “)” | “-” “int” | “int”;
```

This grammar generates four states with shift-reduce conflicts:

```
Shift-Reduce conflict in state 11 on ["+" "-" "*" "/"]
  Reduce  by <expr> --> <expr> "+" <expr> ., ["+" "-" "*" "/" ")" ""$""]
  Shift       by <expr> --> <expr> . "+" <expr>
  Shift       by <expr> --> <expr> . "-" <expr>
  Shift       by <expr> --> <expr> . "*" <expr>
  Shift       by <expr> --> <expr> . "/" <expr>

Shift-Reduce conflict in state 12 on ["+" "-" "*" "/"]
  Reduce  by <expr> --> <expr> "-" <expr> ., ["+" "-" "*" "/" ")" ""$""]
  Shift       by <expr> --> <expr> . "+" <expr>
```

```
  Shift      by <expr> --> <expr> . "-" <expr>
  Shift      by <expr> --> <expr> . "*" <expr>
  Shift      by <expr> --> <expr> . "/" <expr>

Shift-Reduce conflict in state 13 on ["+" "-" "*" "/"]
  Reduce  by <expr> --> <expr> "*" <expr> ., ["+" "-" "*" "/" ")" ""$""]
  Shift      by <expr> --> <expr> . "+" <expr>
  Shift      by <expr> --> <expr> . "-" <expr>
  Shift      by <expr> --> <expr> . "*" <expr>
  Shift      by <expr> --> <expr> . "/" <expr>

Shift-Reduce conflict in state 14 on ["+" "-" "*" "/"]
  Reduce  by <expr> --> <expr> "/" <expr> ., ["+" "-" "*" "/" ")" ""$""]
  Shift      by <expr> --> <expr> . "+" <expr>
  Shift      by <expr> --> <expr> . "-" <expr>
  Shift      by <expr> --> <expr> . "*" <expr>
  Shift      by <expr> --> <expr> . "/" <expr>
```

In each of these states we have an item representing an arithmetic expression on the top of the stack which may be reduced as well as items with a dot in front of a new arithmetic operator about to be scanned. The choice of whether to reduce, which will make the operator at the top of the stack come first, or to shift, which will make the operator about to be recognized come first, depends on the relative precedence of the reduction and the shift and on the associativity of the token about to be scanned if reduce and shift have the same precedence.

To configure sdtgen to resolve the shift-reduce conflicts above, we enable shift-reduce conflict resolution by specifying the AMBIGUOUS parser option and define the operator tokens with precedence and associativity Attributes:

```
  "+", precedence=2, associativity=left;
  "-", precedence=2, associativity=left;
  "*", precedence=1, associativity=left;
  "/", precedence=1, associativity=left;
```

Smaller numeric values represent higher precedence. To resolve a conflict sdtgen first determines the precedence of the reduce. This is the precedence of the rightmost terminal in the production being reduced. If there is no terminal this conflict cannot be resolved. To remedy this, define an Epsilon Token with the desired precedence value and append it to the right hand side of this production.

Once given the reduce precedence sdtgen examines each potential shift. If the precedence of the terminal after the dot is higher than the precedence of the reduce then the terminal about to be scanned should come first and the parse table action will be the shift.

If the precedence of the terminal after the dot is lower than the precedence of the reduce then the production on the stack should come first and the parse table action will be the reduce.

If the two precedence values are the same then the associativity of the terminal after the dot is used to resolve the conflict. If that terminal has associativity NONE, which is the default, the conflict cannot be resolved. If the terminal after the dot has left associativity then the operator which is already on the stack should happen first and the parse table action will be the reduce. If the terminal after the dot has right associativity then the operator which is about to be scanned should happen first and the parse table action will be the shift.

In the first example above:

```
Shift-Reduce conflict in state 10 on ["+" "-" "*" "/"]
  Reduce  by <expr> --> <expr> "+" <expr> ., ["+" "-" "*" "/" ")" ""$""]
  Shift       by <expr> --> <expr> . "+" <expr>
  Shift       by <expr> --> <expr> . "-" <expr>
  Shift       by <expr> --> <expr> . "*" <expr>
  Shift       by <expr> --> <expr> . "/" <expr>
The reduce by production <expr> --> <expr> "+" <expr> . has precedence 2
The shift by production <expr> --> <expr> . "+" <expr> has precedence 2 and associativity = LEFT
Shift precedence 2 equals reduce precedence 2 and associativity = LEFT; action will be reduce
The shift by production <expr> --> <expr> . "-" <expr> has precedence 2 and associativity = LEFT
Shift precedence 2 equals reduce precedence 2 and associativity = LEFT; action will be reduce
The shift by production <expr> --> <expr> . "*" <expr> has precedence 1 and associativity = LEFT
Shift precedence 1 is higher than reduce precedence 2; action will be shift
The shift by production <expr> --> <expr> . "/" <expr> has precedence 1 and associativity = LEFT
Shift precedence 1 is higher than reduce precedence 2; action will be shift
Shift-Reduce conflict has been resolved
```

The expression to be reduced is “+”, which has the same precedence value as the shifts on “+” and “-”, so in both those cases the fact that they are left associative resolves the conflict in favor of the reduce action. The precedence of the “+” expression is less than the precedence of “*” and “/” so in their cases precedence alone resolves the conflict in favor of shift.

Sometime a terminal has two different precedence values depending on where it is used; for example unary operators which are the same symbol as binary operators. If required to resolve a shift-reduce conflict, a Token Alias can be defined to give multiple precedence and associativity values to the same token. A token and its alias are identical from the point of view of the grammar. GOTOs and lookahead sets will only ever contain the originally defined

token. Only shift-reduce conflict resolution will distinguish between the token and its alias and use the appropriate precedence and associativity values to resolve conflicts.

5.2 Reduce-Reduce Conflicts

Sdtgen normally produces LALR(1) parsing tables. LALR(1) parser generation begins with the LR(0) characteristic finite state machine. If the input grammar is LR(1) but not LALR(1) the merging of states caused by the fact that LALR(1) uses the LR(0) CFSM can introduce reduce-reduce conflicts which are not present in canonical LR(1) parsing tables.

If the grammar is in fact LR(1) but not LALR(1), the SPLITSTATES parser option can be specified to have sdtgen identify and duplicate the states leading to the reduce-reduce conflict effectively undoing the state merge in order to resolve the conflict. If this process succeeds the result is a parser with slightly more states than LALR(1) but with the power of canonical LR(1).

When sdtgen calculates the LALR(1) lookahead sets it does it in two steps. First it calculates the spontaneous follow sets that are produced entirely within each state. Then it propagates these spontaneous sets to GOTO states in order to produce the actual LALR(1) lookahead sets which are used to produce the parsing tables. During this process, the spontaneous follow sets are preserved unchanged separately from the full lookahead sets.

When a reduce-reduce conflict occurs and SPLITSTATES has been specified sdtgen follows each item in the conflict back through the sequence of items which propagate lookahead sets to them. To do this, sdtgen creates a “lane” for each conflict item. A lane consists of a follow set which has been propagated to the conflict item by the states in the lane and a sequence of lane entries. Each lane entry consists of a state number and a set of items which propagate lookaheads to the previous entry in the sequence.

Each lane in the conflict is initialized to the conflict state and one of the conflict items and its follow set is initialized to the spontaneous follow set of the conflict item. Since spontaneous follow sets have already been propagated throughout the item set these follow sets are the entirety of the lookahead which has been propagated to the conflict items by the state in which the conflict occurred.

If the reduce-reduce conflict exists in the accumulated lane follow sets then it arises within a state and cannot be resolved by splitting states. The attempt to repair the conflict has failed.

Now sdtgen has to search back to the states which have a GOTO to this state at the end of each lane. To do this, each lane in the conflict must first end in a kernel item. If a lane ends in a closure item, a new entry is pushed onto the end of the lane consisting of the state of the closure item and a set of all the kernel items in that state which propagate lookaheads to the closure item. If no kernel item propagates lookaheads to the closure item then this lane is

completely explored and the accumulated follow set associated with the lane is the entirety of the lookahead contributed to the conflict by this lane+.

If there are still lanes that end in kernel items a new copy of these lanes is created for every ancestor state whose GOTO produced this state. Sdtgen appends one of the ancestor states to each of the copies along with the items in the ancestor state which are the predecessors of the items at the end of the lane. The follow sets of the predecessor items are merged with the accumulated lane follow sets to mimic the propagation of the GOTO follow set to the conflict item lookaheads.

If the new state at the end of a lane also exists earlier in the lane we have discovered a loop in the characteristic finite state machine. This lane has been completely explored and the accumulated follow set for this lane is the entire follow set that it can contribute to the conflict.

If the lookahead sets for the predecessor items do not exhibit the reduce-reduce conflict, this lane resolves the conflict and has been completely explored. However, if the reduce-reduce conflict still exists in the lookahead this process repeats again starting with the check for a conflict in the accumulated lane follow sets. Sdtgen continues working backwards through the ancestor states until it either discovers that the conflict exists in the spontaneous lookahead sets or it discovers a set of ancestor states that are free of conflict.

Each copy of the conflict lanes will contain the same sequence of states and items except for the entry at the end of each lane which resolves the conflict. It is possible that some copies can be merged without reintroducing the conflict so sdtgen first groups together all the conflict lane copies which have compatible lookahead sets. Then it creates copies of the common lane states so that each grouped copy of the conflict lanes has its own set of states. The states at the end of the lane are modified so that they point to the independent sequence of states created for them from the states earlier in the lane. The results in a characteristic finite state machine where the LR(1) states whose merge created the conflict are no longer merged.

After the new states have been created sdtgen recomputes the propagated lookahead sets of the now conflict free characteristic finite state machine.

Here is an example grammar from Dr. DeRemer's paper "Practical Translators for LR(k) Languages."

```
IDENT deremer2;
TITLE "Grammar G4 from Practical Translators for LR(k) Languages";
OPTIONS
  shiftreduce, splitstates;
SCANNER
  "a"; "b"; "c"; "d"; "e";
PARSER
```

```
  <E> -->
        "a" <A> "d"
      | "a" <B> "c"
      | "b" <A> "c"
      | "b" <B> "d"
  ;
  <A> -->
        "e" <A>
      | "e"
  ;
  <B> -->
        "e" <B>
      | "e"
  ;
```

This grammar is LR(1) but not LALR(1) so the SPLITSTATES parser option has been specified to correct the conflict.

```
Reduce-Reduce conflict in state 5 on ["c" "d"]
  <A> --> "e" ., ["c" "d"]
  <B> --> "e" ., ["c" "d"]
Conflict Resolution 1:
  Lane 1: follow ["d"]
    5.  1.  <A> --> "e" ., ["c" "d"]
    2.  3.  <A> --> . "e", ["d"]
  Lane 2: follow ["c"]
    5.  3.  <B> --> "e" ., ["c" "d"]
    2.  5.  <B> --> . "e", ["c"]
Conflict Resolution 2:
  Lane 1: follow ["c"]
    5.  1.  <A> --> "e" ., ["c" "d"]
    3.  3.  <A> --> . "e", ["c"]
  Lane 2: follow ["d"]
    5.  3.  <B> --> "e" ., ["c" "d"]
    3.  5.  <B> --> . "e", ["d"]
Conflict Resolution 3:
  Lane 1: follow []
    5.  1.  <A> --> "e" ., ["c" "d"]
    5.  5.  <A> --> . "e", ["c" "d"]
  Lane 2: follow []
    5.  3.  <B> --> "e" ., ["c" "d"]
```

```
    5.  7.  <B> --> . "e", ["c" "d"]
The lookaheads for conflict resolutions 1 and 3 are compatible
Reduce-Reduce conflict as been resolved
```

The reduce-reduce conflict occurs in state 5 between the productions “<A> --> “e” .” and “<B> --> “e” .” both of which have the lookahead set [“c” “d”]. Since there are two reduce items in conflict we have two lanes. One starts with state 5 item 1, the reduce by “<A> --> “e” .” and the other starts with state 5 item 3, the reduce by <B> --> “e” .”. Both of these items are in the state kernel so we can immediately proceed to their ancestor states.

Three states have a GOTO on “e” to state 5 – state 2, state 3, and state 5 – so sdtgen creates a copy of these two lanes for each. The first copy gets the items in state 2 which are the ancestors of the lane items, the second copy gets the items in state 3, and the third copy gets the items in state 5.

The items in state 2 have the lookahead sets [“d”] and [“c”] respectively so they do not conflict. The items in state 3 have the lookahead sets [“c”] and [“d”] respectively so they also do not conflict. The items in state 5 are the top of a loop. Since their spontaneous follow sets (reported on the “Lane” line above) have no conflict (and in fact contribute nothing to the lookahead sets) these three copies of the lanes are conflict free.

Sdtgen groups Conflict Resolution 1 and Conflict Resolution 3 since doing so does not reintroduce the conflict. It was actually the merge of Conflict Resolution 1 and Conflict Resolution 2 in the original LR(0) CFSM which introduced the error.

Sdtgen therefore creates a single additional copy of state 5 to act as the GOTO on “e” for state 3 as well as itself. The original state 5 remains the GOTO on “e” for both states 2 and 5. State 5 now contains “<A> --> “e” ., [“d”]” and “<B> --> “e” ., [“c”]” and the state 5 copy contains “<A> --> “e” ., [“c”]” and “<B> --> “e”., [“d”]”, successfully resolving the original conflict.

6 Error Repair

The error repair method implemented in sdtgen is based on the paper “Methods for the Automatic Construction of Error Correcting Parsers” by Dr. Johannes Röhrich with the addition of insertion and deletion costs inspired by locally least-cost error repair. This method uses the idea of a “continuation grammar” which is a subset of the grammar for the language being generated. From this continuation grammar, a “continuation automaton” can be selected whose purpose is to generate the shortest sequence of terminals which consumes the parse stack and results in acceptance. This sequence of terminals is called the continuation string.

When the ERRORREPAIR parser option has been selected, sdtgen assigns two values to each production in the grammar: the number of steps required to derive a string of terminals

from the production right hand side and an insertion cost for that derived right hand side string.

The number of steps for a terminal is, of course, 0. The number of steps for a production is the sum of the number of steps for each nonterminal on its right hand side plus 1 for the production itself. The number of steps for a nonterminal on the right hand side is the minimum number of steps required for any production with that nonterminal on the left hand side.

Similarly, the insertion cost for a terminal is the cost assigned to that terminal in the scanner token definition. The insertion cost of a production is the sum of the insertion costs of each terminal and nonterminal on its right hand side. The insertion cost for a nonterminal on the right hand side is the minimum insertion cost of any production with that nonterminal on the left hand side.

Once each production's number of steps and insertion cost have been calculated the right hand sides with the same left hand side are independently sorted using the computed steps as the primary key and computed cost as the secondary key. This produces a grammar where the first right hand side for each left hand side is the continuation grammar production with the lowest insertion cost.

In order to maintain this ordering in the LR(0) characteristic finite state machine the closure operation is modified to insert additional items to the state in a depth-first rather than breadth-first fashion. This ensures that the first item in each state is a member of the continuation automaton and the first items in the closure were added by that item in order of steps and cost. If the dot is all the way on the right of the first item, the error repair action for this state is a reduce by this production. If the dot is in front of a terminal, that terminal is the error repair value for this state and will be appended to the continuation string. If the dot is in front of a nonterminal, the depth-first closure order ensures that the first item in this state's closure with the dot all the way on the right or with the dot in front of a terminal specifies the error repair value for this state.

This allows the continuation automaton to be encoded by storing a single additional value per parser state: the continuation string terminal or the reduce production number. Note that it is possible to write a pathological grammar where a nonterminal does not actually derive a terminal. In this case a state can arise which has no error repair value because no item has a reduce or a dot in front of a terminal. In this case sdtgen will issue a warning that the state has no valid error repair value. The tables generated for this language will, of course, be unusable.

When the parser encounters a syntax error it stops parsing input tokens and saves the current parse and token stack. It then continues parsing until acceptance using the continuation automaton values selected when the parser was generated in place of input tokens. Each shift appends the terminal token to the continuation string and finds all the

terminal tokens which are valid in the new parse state. These are the terminals which will become valid after the insertion of the prefix of the continuation string that has been generated thus far.

When the parse is complete the continuation string is the shortest string of terminals such that the input which has been parsed thus far followed by the continuation string is a valid input string in the language. As a side effect we have a table of the tokens which become valid after the insertion of each prefix of the continuation string.

Unfortunately, because the continuation grammar (and therefore automaton) is only a small subset of the entire language grammar the choice of insertion strings that it provides is quite limited. Since the only tokens which can be inserted are those which are a prefix of the continuation string this greatly limits the quality of the repairs that are possible with only the continuation. To help alleviate this near-sightedness sdtgen considers two potential insertions at each step of the repair.

After the continuation string has been generated sdtgen examines the current parse state for the terminal with the lowest insertion cost which selects a valid parsing action. Inserting this token is one potential repair. It then checks to see if inserting this token makes the next Error Repair Context input tokens valid. If it does then the cost of this repair is just the Insertion Cost of the token. If a second syntax error is encountered before the entire context has been parsed the cost of the repair is increased by the Error Repair Cost times the number of unparsed tokens divided by the number of context tokens. This rewards repairs that make more unparsed input tokens valid.

If there is a prefix of the continuation string which makes the current input token valid, inserting this prefix is a second potential repair. As with the valid token insertion described above, sdtgen checks the next Error Repair Context input tokens. If they are all valid the cost of this repair is the sum of the Insertion Costs of each terminal in the continuation string prefix. Encountering a second syntax error increases the repair cost proportionally as described above.

Sdtgen selects the cheaper of these two insertions as the best current repair. Then it tries deleting the current input token and repeating this process. The repair costs this time will be the Deletion Cost of the deleted input token plus the repair cost of the cheapest insertion alternative based on the new current input token. If this new deletion plus insertion repair is cheaper than the previous best repair then it become the new best repair and the deleted token is permanently deleted.

This process continues until the total deletion cost up to this point exceeds the cost of the best previously selected repair. At this point no new repair can improve on the one that has already been selected. Any input tokens which were deleted during the search but not accepted as deleted as part of the best repair are returned to the input stream.

Sdtgen then displays the repair as a sequence of deleted tokens, a sequence of inserted tokens, or a replacement where tokens are both deleted and inserted. If the selected insertion string contains a token which is in the deletion string, then that token value is reclaimed from the deletion string for use in the insertion. For example, if the deletion string includes an integer token with the value “42” (available because the token is from the actual input file) and the insertion string contains an integer token which has no value because it was manufactured by the repair process, the “42’ is copied from the deletion to the insertion. This allows repairs like “replaced 42 = with = 42” to be generated.

After the input stream has been modified by the repair sdtgen restores the parse stack and resumes parsing normally.

This error repair method is very lightweight since in the absence of an actual syntax error it incurs no additional overhead beyond one integer per parser state for the value generated to select the continuation automaton actions.

6.1 Example Repair

Introducing a couple of syntax errors into the sdtgen language definition file results in:

```
  37:   "integer"    == digit+                          insert = 1, delete 4 =, install;
                      ^
*****  Deleted: =
                                                         ^
*****  Inserted: ,
                                                                        ^
*****  Replaced: 4 =  with  = 4
```

Fortunately in this case the selected repairs actually produce the correct input. However, semantic routines must always handle the cases where a parse stack entry containing a token which has the Install attribute specified was manufactured entirely by the repair process and therefore contains NULL rather than an actual input string.

7 Sdtgen Table Format

Sdtgen writes a single output file containing all the tables required to scan and parse the input language. The default name for this file is “tables.dat” which may be selected with the -w <Output File> – Select the Output File command line parameter.

7.1 Header Line

The first line of the tables file contains global information about the input language.

Table 3: Tables Header Values

Table type: 0 for tables produced by sdtgen and 1 for tables produced by packtables.
Number of terminals in the language.
Number of tokens in the scanner (number of terminals plus ignored regular expressions).
Number of states in the scanner.
Number of nonterminals in the grammar.
Number of productions in the grammar.
Number of states in the parser.
Error Repair Context value.
Default Error Repair Cost.
The Language Identifier.

7.2 Scanner Tables

7.2.1 End of Token Tables

Because the regular expressions that define tokens can overlap, and because of the lookahead operator, it is possible that a single scanner state can represent the end of more than one token. It is also possible that the state which determines what token has been recognized is different from the state which determine where that token ends. In fact, it is possible that the end of multiple different tokens can be encountered before the scanner knows exactly which token has been recognized.

To support this feature, each state in the scanner contains a set of 0 or more token numbers for which it represents the end. To efficiently encode this sdtgen uses an end of token index table and an table of the end of token values for each state concatenated a single table.

7.2.1.1 End of Token Index

Each entry in this table is the index of the start of the end of token values for this state. This table contains one more entry than there are states with the final entry being the total number of entries in the End of Token Values table. This makes the following properties true for every state in the scanner:

- end_of_token_index[i] is the index of the first End of Token Values entry for state i.
- end_of_token_index[i+1]-1 is the index of the last End of Token Values entry for state i.

If end_of_token_index[i] and end_of_token_index[i+1] are equal then state i has no end of token values.

7.2.1.2 End of Token Values

Each entry in this table is a token number recognized by the scanner.

7.2.2 Final State Table

This table contains one entry for each state in the scanner. The value is the token number which is recognized by this state. Unlike the end of token table, where a single state can record the end of multiple tokens, only one token can be recognized by a single state. If the regular expressions in the grammar overlap more that one token could potentially end in this state. When generating the deterministic finite state automaton for the scanner, sdtgen keeps the token number with the smallest value and discards all those greater. This results in tokens defined earlier in the input taking precedence over those defined later.

7.2.3 Install Flag Table

This table contains one entry for each state in the scanner. The value is 1 if the token recognized by this state has the Install attribute specified and 0 otherwise.

7.2.4 Scanner Transitions Table

This table contains one entry for each state in the scanner. Each entry consists of the number of transitions out of the state followed by that many pairs of character and next state numbers. “Character”, in this sense, is a number between 0 and 256 where 0-255 represent an 8 bit character and 256 represents end of file.

7.3 Sdtgen Parser Tables

7.3.1 Terminal Insertion Costs

This table contains one entry for each terminal in the language. Each entry is the Insertion Cost for that terminal.

7.3.2 Terminal Deletion Costs

This table contains one entry for each terminal in the language. Each entry is the Deletion Cost for that terminal.

7.3.3 Left Hand Side Token Numbers

This table contains one entry for each production in the grammar. Each entry is the token number of the left hand side symbol for that production. When performing a reduce this table provides the token that replaces the right hand side on the top of the parse stack.

7.3.4 Right Hand Side Lengths

This table contains one entry for each production in the grammar. Each entry is the number of symbols on the right hand side of that production. When performing a reduce this table provides the number of entries to pop from the parse stack.

7.3.5 Semantic Routine Numbers

This table contains one entry for each production in the grammar. Each entry is the semantic routine number to be invoked when this production is reduced, or 0 if there is no associated semantic routine.

7.3.6 Error Repair Values

This table contains the values generated by specifying the ERRORREPAIR parser option. There is one entry for each state in the parser. A positive value is a token number used to access an action in the current parser state as well as to be appended to the continuation string. A negative number is the negative production number of the reduce that should be performed when in this state.

7.3.7 String Tables

In order to be able to automatically generate error messages containing tokens which were inserted and deleted, the parser requires a table of all the terminal and nonterminal name strings. Since these names are variable length an index table and concatenated string table are generated using the same method as the End of Token Tables.

7.3.7.1 String Table Index

Each entry in this table is the index of the first character of the name of a token in the String Table Values table. This table contains one more entry than there are terminals plus nonterminals with the final entry being the total length of the String Table Values table. This makes the following properties true for every state in the scanner:

- string_tables_index[i] is the index of the first character in the String Table Values table for token i.
- string_tables_index[i+1]-1 is the index of the last character in the String Table Values table for token i.

7.3.7.2 String Table Values

This table is a concatenated string containing all the terminal and nonterminal names in order by token number. This table consists of just the names. There are no quotes or angle brackets surrounding them (except of course for the special <Goal> and “’$’” tokens which were generated containing those characters).

7.3.8 Parser Action Tables

This table contains one entry for each state in the parser. Each entry consists of the number of actions in the state followed by that many pairs of token number and corresponding action. Parsing actions are encoded as follows:

Table 4: Parsing Action Encoding

n > 10000	Shift to state n - 10000
0 < n <= 10000	Shiftreduce by production n
-10000 < n < 0	Reduce by production -n
n = -10000	Accept

Unused entries are 0 and indicate error.

8 Packtables

8.1 Packtables Command Line

packtables [<input file> [<output file>]]

Packtables reads the Sdtgen Table Format input file as produced by sdtgen and writes a compressed output file. If the input file is specified as "-" input will be read from standard input. If the output file is specified as "-" output will be written to standard output. Likewise, if the output file name is omitted the output will be written to standard output. If both input and output file names are omitted input will be read from standard input and output will be written to standard output.

Packtables writes a summary of the input table size and the compression achieved to standard error.

8.2 Scanner Table Packing

The scanner tables produced by sdtgen are a two-dimensional array indexed by state and input character. There are 256 possible input characters plus end of file, so this table tends to be fairly large. Additionally, the normal regular expressions defined for a language frequently have many states whose entries are quite similar. Taking advantage of this characteristic greatly enhances the amount of compression which can be achieved.

The scanner tables are compressed using the method described in Aho, Lam, Sethi, and Ullman's book "Compilers – Principles, Techniques, & Tools" in section 3.9.8 entitled "Trading Time for Space in DFA Simulation."

Each scanner state has a default state number and a base index into a pair of tables which contain the entries for that state. The pair of tables used to store table entries consist of a next state value and a check state value for every entry in the tables. The base index value for a state plus the input character is the index in the next and check state tables where the indicated table entry should be found. If the check state entry at that index contains the current state number then the next state value at that index is the correct result for the current

state and character. If the check state value does not match the current state then we try again using the default state number for the current state. That is:

```
while (check[base[state] + character] != state) state = default[state];
return(next[base[state] + character]);
```

This allows us to store only the table entries which differ from the default state entries and rely on the default state to fill in the rest.

Packtables begins by reading the entire scanner table into a two dimensional array. Then it compares every state to every other state counting up the number of entries which do not match between the states. Then it computes the weighted-distance mean difference between each state and all the other states. This allows packtables to sort the states based on how different from other states they are. States are inserted into the compressed tables beginning with the state which is most similar to other states and ending with the state that is most different from other states.

Every table entry for the state which is most similar to other states is inserted into the compressed tables to establish a base state for the default state chain. Subsequent states are inserted by finding the previously inserted state which is most like the state being inserted. This previous state will be the default state for the newly inserted state. Then starting from the beginning of the next and check arrays packparse searches for a place where the entries which differ from the default state will fit into unused entries. That index will be the base index for the new state. The differing entries are inserted into next and check and the insertion of the new state is complete.

After all the states have been inserted there may still be unused next and check table entries. Packparse sorts the states by how long their default state chain is. Then starting with the state that has the longest chain it checks to see if any of the next and check table entries indexed by that state are unused. If there are packparse fills in those entries in the table. It continues this process until all states have been checked. This decreases the number of table entries belonging to the states with the longest default state chains that actually have to refer to the default state.

8.3 Parser Table Packing

The parser tables produced by sdtgen are a two-dimensional array indexed by state and token. These tables are generally much smaller and more sparse than the scanner tables so the use of a default state is not as much a benefit.

Similarly to the compressed scanner tables, each parser state has a base index into a pair of next and check tables which contain the entries for the state. The next table contains the parsing action corresponding to the indexed token and the check table contains the state number. For these tables, the access method is:

```
if (check[base[state] + token] == state)
  return(next[base[state] + token]);
else
  return(ERROR);
```

Packtables begins by reading the entire parser table into a two dimensional array. It sorts the states by the number of actions each state contains. States are inserted into the compressed tables beginning with the state which has the most actions and ending with the state with the fewest.

Starting from the beginning of the next and check arrays packparse searches for a place where the state's non-error entries do not overwrite previous state's entries. That index will be the base index for the new state. The non-error entries in the new state are written into unoccupied entries of the next and check arrays. The error entries in the new state are either unused entries which are truly errors or used entries which are errors because the check state does not match the current state.

8.4 Packparse Table Format

The output tables written by packparse are identical to the Sdtgen Table Format written by sdtgen with the following changes:

The table type on the Header Line is changed from 0 for sdtgen to 1 for packparse.

The Scanner Transitions Table and the Parser Action Tables are replaced with the packed versions of those tables.

8.4.1 Packed Scanner Table

8.4.1.1 Scanner Default State Table

This table has one entry for every scanner state and contains the default state number for the state. A default state number of 0 means that this state has no default and every entry in this state has a value in the Scanner Check State Table and Scanner Next State Table.

8.4.1.2 Scanner Base Index Table

This table has one entry for every scanner state and contains the index in the Scanner Check State Table and Scanner Next State Table where the entries for the state begin. Every state has 257 entries in those tables, one for every character 0x00 through 0xFF plus one for end of file.

8.4.1.3 Scanner Check State Table

This table has enough entries to store the deterministic finite state automaton transitions for each state in the scanner. The exact number of entries will depend on how well the

transitions can be packed. When accessing a table entry the Scanner Base Index Table provides a base index for the current state and the current input character provides an offset. If the Scanner Check State Table entry at that index is equal to the current state the Scanner Next State Table entry at that index is the state number to transition to. If the Scanner Check State Table entry is not the current state then the table lookup continues using the Scanner Default State Table value for the current state.

8.4.1.4 Scanner Next State Table

This table has the same number of entries as the Scanner Check State Table. The exact number of entries will depend on how well the transitions can be packed. The corresponding Scanner Check State Table entry is the state for which this Scanner Next State Table entry is valid. If valid, the entry is the number of the state in the DFA to be transitioned to.

8.4.2 Packed Parser Table

8.4.2.1 Parser Base Index Table

This table has one entry for every scanner state and contains the index in the Parser Check State Table and Parser Action Table where the entries for the state begin. Every state has one entry in those tables for every terminal and nonterminal token.

8.4.2.2 Parser Check State Table

This table has enough entries to store the parsing actions for each state in the parser. The exact number of entries will depend on how well the actions can be packed. When accessing a table entry the Parser Base Index Table provides a base index for the current state and the current input token provides an offset. If the Parser Check State Table entry at that index is equal to the current state the Parser Action Table entry at that index is the Parsing Action Encoding to be executed. If the Parser Check State Table entry is not the current state then the action to be executed is ERROR.

8.4.2.3 Parser Action Table

This table has the same number of entries as the Parser Check State Table. The exact number of entries will depend on how well the actions can be packed. The corresponding Parser Check State Table entry is the state for which this Parser Action Table entry is valid. If valid, the entry is the Parsing Action Encoding of the parsing action to be executed otherwise the parsing action is ERROR.

9 Tableformat

Tableformat reads the Packed Parser Table format as produced by Packtables and writes a C module which defines a variable of type struct sdt_tables that is declared in the include file

“tables_definitions.h”. This structure contains elements for every one of the tables defined in the Packparse Table Format described above. Tableformat uses the Language Identifier which is stored in the table Header Line as the variable name.

This C module is compiled and linked with the sdtgen driver program and a pointer to the sdt_tables variable is passed to the parser to control the scanning and parsing of the input language.

9.1 Tableformat Command Line

```
tableformat [ <input file> [ <output file> ] ]
```

If the input file is specified as “-” input will be read from standard input. If the output file is specified as “-” output will be written to standard output. Likewise, if the output file name is omitted the output will be written to standard output. If both input and output file names are omitted input will be read from standard input and output will be written to standard output.

10 Using the Parser

The C module created by tableformat can be compiled to produce a variable of type sdt_tables which can be linked with a program to scan and parse a file in the defined language. A program using the parser must define the parser data types and functions as well as the sdt_tables data type:

```
#include “parser_definitions.h”
#include “tables_definitions.h”

#include “parser_functions.h
```

Then the parsing tables can be referenced with

```
extern sdt_tables ptables;
```

assuming “ptables” is the name defined by the Language Identifier in the sdtgen input file

To parse an input file using these tables they must be initialized with the file descriptor of the file to be parsed and pointers to the perform_action Callback function which performs semantic routine actions and the previously described install_token Callback:

```
void perform_action(sdt_tables *tables, int semno);
void install_token(sdt_tables *tables, tokenentry *token);
```

perform_action is called with a pointer to the sdt_tables structure and the integer semantic routine number. Install_token is called with the sdt_tables pointer and a pointer to the newly scanned token entry. These functions must be defined, if only as stubs, to initialize the parser.

```
init_parser(&ptables, fd, &perform_action, &install_token);
```

This will configure the members of ptables to read from the file descriptor fd as well as provide the callback function pointers for perform_action and install_token. It will also initialize all the members of ptables which are required to scan and parse the input file.

The call:

parse_input(&ptables);

will scan and parse the provided input file and call the perform_action and install_token functions to do whatever is necessary to interpret the language. If a syntax error is encountered and ERRORREPAIR was specified for the language the syntax error will be repaired and the parse will continue, otherwise the parser will call exit(1) to terminate the program.

Finally, a call to

```
free_parser(&ptables);
```

will free any remaining data structures that were allocated by the parser.

10.1 perform_action Callback

When a reduce action has been signaled to the parser but before the production right hand side has been popped from the parse stack and replaced with the left hand side symbol, the parser calls the perform_action Callback passing it the semantic routine number of the production which has been recognized.

```
void perform_action(sdt_tables *tables, int semno);
```

The tables pointer can be used to access any member of the structure used by the parser. In particular, this gives us access to the parse stack. The parser_definitions.h file provides a number of macros to access these members provided we specify the sdt_tables pointer variable name to be “tables”.

For the semantic routines, the most important of these are

- PARSTACK(i) – the i^{th} element on the parse stack.
- PARCOUNT – the number of elements on the parse stack.

Therefore PARSTACK(PARCOUNT – 1) is the top element of the stack and the rightmost token of the recognized production’s right hand side. PARSTACK(PARCOUNT – 2) is the next token to the left, etc.

Parse stack entries are defined by:

```
struct parseentry               /* One entry on the parse stack */
```

```
{
  int                state;        /* State number */
  location           where;        /* Start of token which created this entry */
  int                token;        /* Token number */
  unsigned char *symbol;       /* Token string (if installed) */
};
```

The code to process the indicated semantic routine number can examine the parse stack entries corresponding to the recognized production's right hand side and perform actions depending on their values.

10.2 Generating Error Messages

While the scanner and parser will automatically generate error messages for unrecognized characters by the scanner and syntax errors by the parser, it is sometimes necessary to generate semantic errors from the perform_action Callback.

Error messages are created with the function:

```
void record_error(sdt_tables *tables, location *point, char *fmt, …);
```

The location parameter point indicates where to write a caret to point at the location of the error on the input line. Fmt and the variable arguments are sprintf parameters to format the error message.

Because of the way the parser performs input buffer handling, the specified location should always be &PARSTACK(PARCOUNT – 1).where. While it would be more aesthetic to point at the first token of the production's right hand side rather than the last, because free-form input could allow previous tokens to be many lines back in the input file it is no longer possible to point to an error location on those lines. Only the line of the last token is guaranteed to be usable.

The sdtgen scanner and parser automatically manage input buffers by allocating new buffers as needed to parse ahead. When the parser shifts a token it considers all input lines preceding the line containing the shifted token to be complete and advances past them, writing them out if requested or if they contain the location of an error message. When a line is written the parser follows it with any error messages for that line. If writing the line advanced from one input buffer to the next the previous input buffer is no longer needed and is freed. Once a line is written along with its error messages, it is no longer possible to append more error messages to it. An attempt to do so will result in an error message on the next line which won't point to the correct location.

11 Example Driver

The example subdirectory contains a minimal main program which can be linked with the libstd.a library which contains the parser and support routines and a compiled sdt_tables structure to demonstrate sdtgen's functionality.

Assuming you've digested this user's guide and produced an input file named language.sdt in the grammars subdirectory, the following steps will create a program to scan and parse that language.

The Makefile in the main sdtgen directory will compile the sdtgen library, the main sdtgen program, and the packtables and tableformat tools:

```
/usr/bin/make
```

Run sdtgen to create the scanner and parser tables in a file named language.dat.

```
./sdtgen -w language.dat grammars/language.sdt
```

If there are issues, make corrections and rerun the sdtgen command otherwise run packtables to produce the packed version of the tables in language.pak.

```
./packtables language.dat language.pak
```

Convert the packed tables into C and write them into the example subdirectory.

```
./tableformat language.pak example/tables.c
```

When the language.sdt input file was created it specified a name for these tables with the Language Identifier statement. Edit the file example/driver.c and change the definition

```
#define LANGUAGE_IDENTIFIER     ptables
```

from ptables to whatever identifier was specified with the Language Identifier statement.

There should now be three files in the example subdirectory in addition to its Makefile. The driver.c program which has been modified with the name of the sdt_tables variable created from the language.sdt input file, the tables_definitions.h file which defines the minimal structure for those tables, and the tables.c file created from the input language definition by sdtgen -> packtables -> tableformat which contains the actual sdt_tables variable.

Back in the main sdtgen directory, compile the driver program.

```
/usr/bin/make driver
```

Run the driver.

```
./driver [ -l ] <input file>
```

The example driver doesn't have any semantic routines, only a stub function to invoke them, so it won't do more than generate a listing if requested with -l on the command line and repair syntax errors if requested with ERRORREPAIR in the language.sdt input file. Sdtgen was written using itself so it can serve as inspiration for how to build on the example. The rest is up to you.