sidBison: A Stepwise Interactive Debugger for Bison

Tufts University

Advisor: Prof. Kathleen Fisher

Siddhartha Prasad

23rd March 2016

Abstract

Parser-generator systems specifications are generally large and hard to debug. Generated code often does not semantically match the programmers intent. Debugging machine-generated parsers is a non-trivial task, requiring knowledge of underlying implementations. This has led to the decline of bottom-up parser-generators as staples in language design and syntactic analysis workbenches.

Stepwise Interactive Debugger for Bison is a step-through debugger that preserves an abstraction boundary between Bison specifications and the underlying parser tables. It allows a user to debug a Bison 2.3 specification at the grammar level, requiring minimal knowledge of bottom-up parsing. The ultimate goal of this project is to make parser-generating tools more accessible to the average programmer.

Contents

A Quick Note

This thesis is aimed at audiences who have previously encountered formal grammars and/or concrete syntax trees. Appendix A contains a quick introduction to formal and context-free grammars.

Familiarity with parser-generators like `Yacc` or `Bison` is not required, but will allow for deeper understanding of the problem. I recommend reading the Bison user manual at `http://www.gnu.org/software/bison/manual/`.

Chapter 1

Introduction

1.1 Machine generated parsers

A parser is a machine that examines a string of characters and acts in response to those characters according to the rules of a formal grammar. In computing, parsers have widespread applications, ranging from Natural Language Processing to compiler generation. Programmers often use these systems to translate information into formats about which they can reason more easily.

Early parsers were common well before a theory of formal grammars was developed, and were painstakingly written into punchcards. With Backus and Naur's formalization of *Extended Backus Naur Form* (EBNF) notation for describing languages, programs were soon contracted for this tedious job. Recursive-descent parser generating machines soon became the standard, allowing programmers to put in grammars and get out entire parsers [11]. While these top-down methods were easily understandable and debuggable, the generated machines could not deal with several everyday scenarios, including left-recursion. This led to the generation of **Bottom Up** parsing methods and their generators. These systems could process most common grammars, and preserved the EBNF abstraction afforded earlier to programmers. However, in doing so, these programs became very complex, involving several tables, automata and stacks. Programmers soon could not easily explain the underlying mechanisms of these systems.

Modern parser-generators produce lots of hard-to-understand code. If the resulting parser does not semantically match the programmer's intent, debugging the system is a non-trivial task. Examining errors in specifications requires knowledge of the underlying code-generating implementation. This thesis introduces a tool that attempts to make the parser-generation process more accessible to the average programmer by offering the ability to step-through and debug grammar specifications.

1.2 Bison

Bison is a very popular bottom-up (LALR) parser generator that was built for the GNU project as an alternative to Yacc. According to the GNU website:

> Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.- *GNU Bison*

1.2.1 Basic Actions

The `Bison` parser-generator employs a *bottom-up* parsing mechanism [6]. The program uses an input specification to generate a push-down automaton as well as a token stack. It executes transitions between states on based on the tokens encountered. `Bison` has three basic actions [2] [4]:

1. **shift**: As Bison encounters input tokens, it pushes them onto the token stack.
2. **reduce**: When the last *k* shifted tokens match a rule, they are merged to form the non-terminal specified in the left hand side of the rule. This symbol is now stored on the token stack. The push-down automata then pops back to a previous state.
3. **lookahead**: `Bison` often *looks ahead* at the next coming token before performing a shift or reduce action, in order to better ascertain what to do.

`Bison` attempts to use shifts and reductions (with the aid of lookaheads) to match input to a specified *start* symbol in the specification [4].

1.2.2 Lookaheads

While the need for a lookahead may not be apparent, the following example shows its effectiveness.

```
        Digit : 1 | 2 | 3

        Number : Digit
                 | Digit Number
```

On input `12` the parser requires the look-ahead to know that after the digit 1 has been shifted, it should not immediately be reduced to the rule `Number`.

1.2.3 Limitations

While Bison is able to produce parsers for a wide range of grammars, its underlying LALR(1) parser table implementation is limited to certain forms. Certain specifications can cause shift-reduce and reduce-reduce conflicts, where there is ambiguity in what action should be executed [8].

Shift-Reduce Conflicts

```
    E -> a E
          | a
```

In the grammar above, we see that when parsing terminal `a`, there could be a reduce action to `E` and a shift action in order to parse the rule `a E`. In such a situation, an LR parser would not be able to decide on a 'correct'option. `BISON` shifts in such a situation, which may not echo the programmers intent [4].

Reduce-Reduce conflicts

```
    X -> Y a | Z b
    Y -> a
    Z -> a
```

In the grammar above, we see that the terminal `a` could be reduced to `Y` or `Z`. Bottom-up parsers like `BISON` do not have a resolution strategy in such a situation [3].

1.3 Programmer Errors

Real-world Bison specifications are often very large. Much like any other language, larger specifications create greater room for programmer error. As a result, while generated parsers may accept input, they often do not behave in a way intended by programmers. Debugging these machine-generated parsers is a non-trivial task. While Bison provides error messaging in terms of the underlying parser implementation, it is not easy to examine the semantics of a specification. A user needs to be familiar not only with the specified grammar, but also LR parsing to correct errors in their code.

The following are `Bison` specifications that do not accept the exact set of strings a programmer might expect.

1.3.1 Small Specifications: Calculator Language

This section looks at a grammar specification for a simple calculator language [3] that accepts integers and performs addition, subtraction, multiplication and division. While this grammar may seem to specify the calculator language at first, it cannot parse any string with more than one operator. For instance, the string `1 + 1 + 1` is not accepted.

```
Digit : 1
    | 2
    | 3
    | 4
    | 5
    | 6
    | 7
    | 8
    | 9
    | 0

Number : Digit
         | Digit Number

Operator : +
          | -
          | *
          | /

Expression : Number
           | Number Operator Number
```

1.3.2 Large Specifications: JSON

Larger grammars, by virtue of their size, are hard to debug. The specification in **Appendix 6.3** is intended to describe JSON strings [12]. Upon closer inspection, one might notice that the grammar can parse only 2 members with value separators. Identifying which rule causes the error requires traversing almost the the whole grammar. A step-though debugging approach can be hard to execute manually.

As a result of the complexity of these systems, parser generators like Bison are quickly losing ground to other parsing approaches. An influential paper by Matthew Might and David Darais claims that Yacc is dead [10]. Increasing the accessibility of these systems could help bottom up parser generators re-emerge as a staple of the average language designer's workbench.

Chapter 2

Stepwise Interactive Debugger for Bison

iBison is a version of Bison 2.3 that was built by S.K. Popuri at the University of Illinois at Chicago. It generates an interactive interface that allows a user to step through the parsing process, presenting information in terms of a push-down automaton and its state and token stacks [5]. Stepwise Interactive Debugger for Bison (sidBison), leverages this responsive design to allow for debugging at the grammar level. The system is modelled on gdb-style debuggers, allowing for not only the identification of errors in Bison specifications but also those in input strings, maintaining an abstraction barrier between grammars and parser-generated state tables.

The goal of this project is to simplify the parser-generation process for programmers who are not well-versed with the underlying implementation. Debuggers preserving such abstraction barriers could be particularly useful in the process of democratizing language design workbenches.

2.1 Using sidBison

The sidBison system requires a Bison specification and lexer shared object in order to debug a string. A more detailed setup process as well as examples are described in Appendix B.

2.2 Commands

The sidBison command set is designed to reinforce the abstraction barrier between a Bison specification and the underlying parser implementation.

1. **crule**: Returns the current non-terminal rule being parsed in the Bison specification.

2. **steprule** : Steps to the next rule in the Bison specification
3. **str**: Identifies the current position in the entire parsing process.
4. **br** : Allows the user to break when a particular token is encountered.
5. **step** : Steps to the next action taken by the parser
6. **ctkn** : Displays the current token being looked at by the parser.
7. **rulepos** : Identifies current position in the rule being parsed.
8. **test <filename>** : Accepts the input string as a file.
9. **quit** : Ends the sidBison program

The implementation of these commands is described in the Chapter 3: *Implementing sidBison Commands.*

2.3 Usability

`sidBison`'s success hinges on its usability. It should be able to aid a user with the examples presented earlier. This is predicated on the idea that:

1. Humans generally find EBNF easier to understand than push-down automata.
2. The ability to step through grammar specifications presents bottom-up parsing in an understandable, linear manner.

These criteria are evaluated in the case study presented in Chapter 5: *Case Study: Usability.*

Chapter 3

Implementing sidBison Commands

`sidBison` is designed to be a step-through debugger at the grammar level, and is built on top of `iBison`'s interactive debugging mechanism. The mathematical underpinnings of the system involve mapping `Bison` and `iBison` constructs like

1. Push-Down Automata
2. State Stacks
3. Token Stacks
4. Lookaheads

to Extended Backaus Naur Form (EBNF) grammar rules. As a result, each command takes the form of a mathematical function in terms of these variables. Thus, `sidBison` commands can be thought of as a set $\{f_i\}$ of functions.

step The `step` command allows the user to step forward to the next action taken by the underlying parser debugger. It is the simplest command for the `sidBison` system. The rule is of the form $f_1(ss, ts, ct) \to (ss', ts', tc')$, where:

1. ss and ss' are state stacks, where ss' is a new state stack created by taking a step.
2. ts and ts' are token stacks, where ts is a new token stack created by taking a step.
3. ct and ct' are current tokens, where ct' is a new current token obtained either by a new lookahead or from ss'.

`step` provides the basis of mapping actions in terms of the underlying parser to those involving `Bison` specifications. It maps a new *lookahead* to encountering a new token, a shift to adding the token to the string, and a reduce to parsing a string. Several of the more complicated `sidBison` commands leverage `step`.

steprule The `steprule` command takes the user to the next rule encountered by the parser. Thus, it presents a step-wise debugging abstraction in terms of the user-provided grammar specification instead of parser-generator options.

The command is implemented by stepping through `iBison` state changes till a reduce action is executed. As a result it has the mathematical form $f_2 : \texttt{Parser Automaton} \rightarrow (ss`, ts`)$, where :

1. ss' represents a new state stack, and by extension a new current state.

2. ts' represents a new token stack.

crule The `crule` command returns the current rule being parsed by Bison generated parser. It takes the form of a mathematical function
f_3 : (Parser Automaton,cs,ss) $\rightarrow$ (Backus Naur Form Rule), where cs is the current state, and ss is the state stack.

In general, a bottom-up parser cannot predict the non-terminal to which a partially known sequence of tokens will be reduced. Even judging if a reduction will ever take place is an undecidable problem. As a result, the command cannot be implemented using a single instance of a single-pass parser like `Bison`. `crule` therefore utilizes a time-travelling heuristic.

This approach is concurrent in nature. A secondary `iBison` process is spawned and is placed in the same state as the primary one. This secondary debugger is then stepped forward to the first parser *reduce* action where the state stack has either shrunk, or has the same size with a different top element. The reduced rule is precisely the production being parsed.

rulepos `rulepos` enumerates current positions the parser may be in within grammar rules. It is implemented by examining the rules and positions associated with the current rule in the underlying automaton. It has the mathematical form $f_4 : cs \rightarrow [(\text{Backus Naur Form Rule}, p)]$, where cs represents the current state and p represents a position in the rule.

str The `str` command returns the current position in the overall parsing process. It is implemented by displaying the contents of the `iBison` token stack. It has mathematical form $f_5 : cs \rightarrow str_{ss}$, where cs represents the current state and str_{ss} is a string representation of the state stack.

ctkn The `ctkn` command displays the token with which the parser is currently dealing. It is implemented by presenting the newer of the look-ahead token and the top element of the token stack.

br The `br` command allows the user to break when a particular token is encountered. This functionality is provided by stepping till the `ctkn` equals the token provided. Thus it has mathematical form $f_6 : tkn \rightarrow (cs', ss')$, where:

1. ss' represents a new state stack, and by extension a new current state.
2. ts' represents a new token stack.
3. tkn represents the input token.

The **quit** and **test** commands are not involved in the actual debugging process, and so are not explained above.

Chapter 4

Interactivity

The `sidBison` tool cannot be considered a successful stepwise debugger if it is not truly interactive. As a result, each command must execute in a resonable amount of time. Since a command's execution time may be context dependent, an understanding of the system's time-complexity may help users improve their use of the system. These analytical formulae, along with empirical run-times, also allow for the evaluation of the interactivity of `sidBison`.

4.1 Complexity

Complexity of Bison It is necessary to understand the runtime of Bison before reasoning about `sidBison`'s runtime. According to the Bison user manual, "a GLR parser can take quadratic or cubic worst-case time, and the current Bison parser even takes exponential time and space for some grammars" [1]. However, for most grammars parsing time is $O(|G|n)$ [1][9]. $|G|$ is the number of tokens in the grammar and n is the length of the string to be parsed.

Reading from iBison `sidBison` is built on top of Satya Kiran Popuri's `iBison` system. As a result, much of `sidBison`'s resources and run-time are invested in inter-process communication. Several commands read information from the iBison process. As a result, it is important to compute the asymptotic time needed to communicate with iBison.

1. The size of information output from `iBison` is a function of its state and token stack sizes.
2. `sidBison` sets an upper limit of $1KB$ on the sizes of the state and token stacks.
3. As a result, there is a fixed upper bound on the information that needs to be read from iBison.

Thus, while the constants in question may be large (at most $2KB$), reading information from iBison takes $O(1)$ time. This communication is implemented in the function `read_from_ibison`.

step The `step` command is the most basic of `iBison` commands. It executes a single Bison step. Since reading from iBison takes constant time, `step`'s runtime has complexity $O(n|G|)$ where $n = 1$. Since the number of tokens in a grammar is constant for any debugging instance, step runs in constant time.

steprule The `steprule` command steps repeatedly till a reduce action is executed. Thus, it may call `step` up to n times, where n is the length of the input string. As a result, since the number of tokens in a grammar is constant for any debugging instance, it has complexity $O(n)$.

ctkn The `ctkn` command accesses the newer of the top element of the token stack and the lookahead to return the current token. As both these operations take constant time, `ctkn` runs in $O(1)$.

str The `str` command returns the contents of the token stack. As the size of this stack is bounded, it runs in $O(1)$.

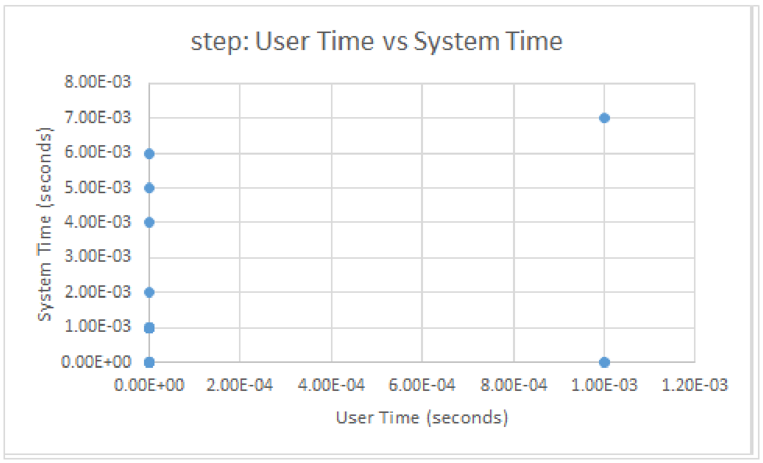
br The `br` command steps repeatedly till the current token equals a specified token. Thus, it may call `step` and `ctkn` up to n times each, where n is the length of the input string. As both `step` and `ctkn` take constant time, it has complexity $O(n)$.

crule The time-travelling `crule` command spawns a new `iBison` process and steps first to the current state, and then till the current rule is reduced. As a result, it steps and reads from iBison up to n times, where n is the length of the input string. As a result, it runs in $O(n)$. However, given that it has to spawn a new process, the constants in the linear describing its execution time are large.

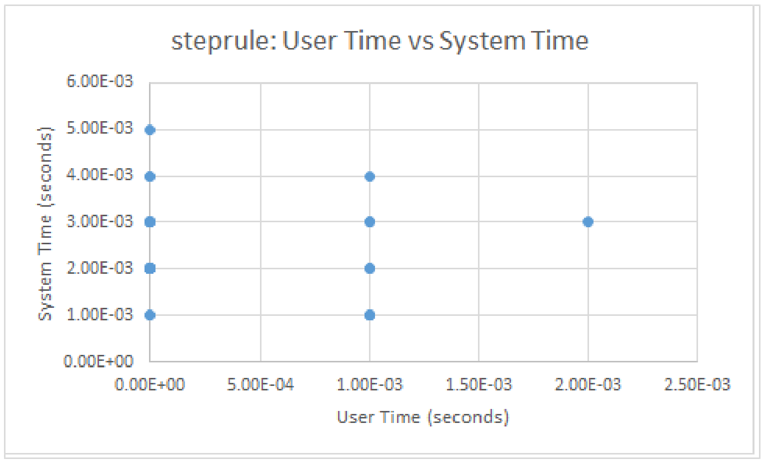
rulepos The `rulepos` command helps identify the user's current position in the rule they are parsing. It first looks up the rules associated with the current parser automaton state and then calls `crule`. As looking up rules is a constant time operation, and `crule` is linear, the `rulepos` command runs in $O(n)$.

4.2 Empirical analysis

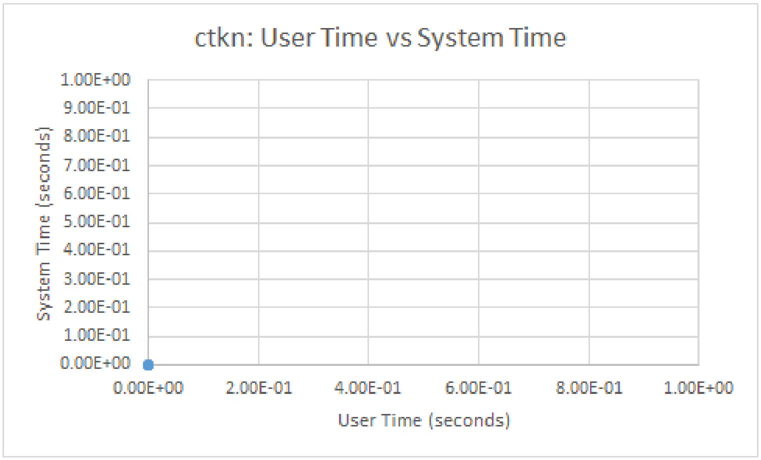
Compiling `sidBison` with the `DDEBUGLOG=1` flag enables a timing routine with least count 10^{-6} seconds . This feature writes system and user time taken by a command to `stderr`. Runtimes for each command were logged while debugging examples in Appendix C. Raw data can be found in Appendix D.



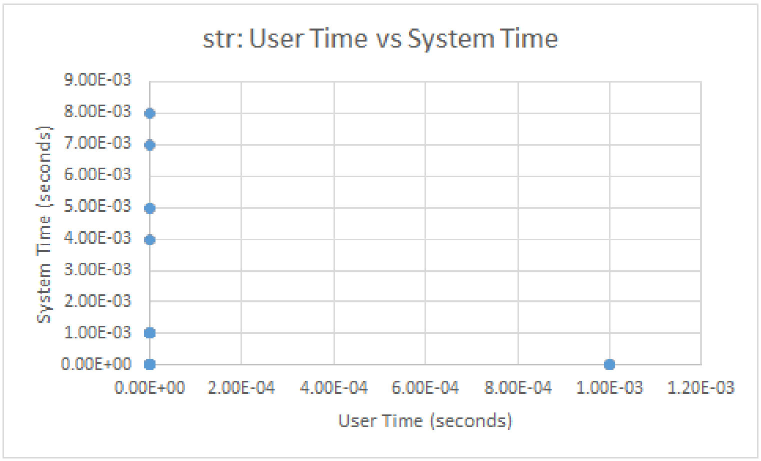
The average user time taken by the **step** command was $1.07 * 10^{-4}s$ with standard deviation $3.150*10^{-4}s$. The average system time taken was $1.29*10^{-3}s$ with standard deviation $1.883 * 10^{-3}s$
.



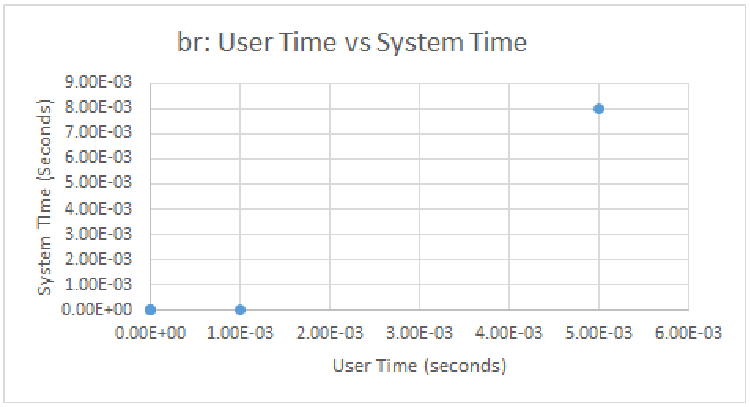
The average user time taken by the **steprule** command was $3.33*10^{-4}s$ with standard deviation $5.34*10^{-4}s$. The average system time taken was $2.31*10^{-3}s$ with standard deviation $9.80 * 10^{-4}s$
.



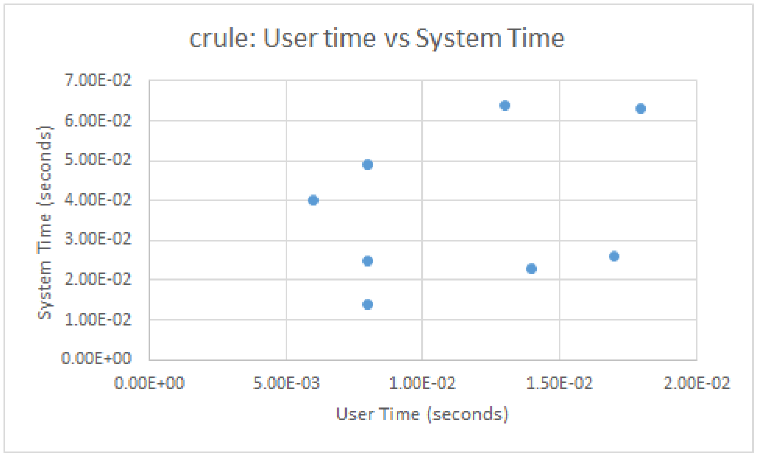
The average user time taken by the **ctkn** command was $0s$ with standard deviation $0s$. The average system time taken was $0s$ with standard deviation $0s$
.



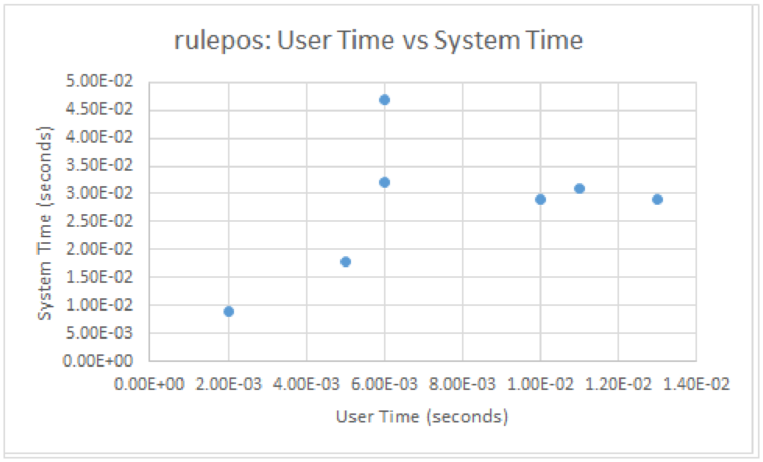
The average user time taken by the **str** command was $1.18 * 10^{-4}s$ with standard deviation $3.270*10^{-4}s$. The average system time taken was $1.12*10^{-3}s$ with standard deviation $2.1 * 10^{-3}s$
.



The average user time taken by the **br** command was $1.20 * 10^{-3}s$ with standard deviation $2.168*10^{-3}s$. The average system time taken was $1.60*10^{-3}s$ with standard deviation $3.577 * 10^{-3}s$
.



The average user time taken by the **crule** command was $1.15 * 10^{-2}s$ with standard deviation $4.597*10^{-3}s$. The average system time taken was $3.80*10^{-2}s$ with standard deviation $1.9046 * 10^{-2}s$
.



The average user time taken by the **rulepos** command was $7.57 * 10^{-3}s$ with standard deviation $3.866 * 10^{-3}s$. The average system time taken was $2.79 * 10^{-2}s$ with standard deviation $1.18930^{1}0^{-3}s$
.

4.3 Conclusion

All `sidBison` commands run in either constant or linear time in the average case. While constants might be large, this guarantees that the system's response time will not grow too large as input string sizes increase. Maximum average execution times for grammar are of the order $10^{-3}s$ for $O(1)$ commands and $10^{-2}s$ for $O(n)$ commands, with low standard deviations. If grammar sizes increase, `sidBison`'s command execution time will only grow linearly – the same as GNU Bison. Thus, debugging tool promises to deliver reasonable execution times for reasonably sized grammars.

Chapter 5

Case Study : Usability

Note: This is a study proposal

The ultimate goal of the `sidBison` project is to make parser-generating tools more accessible to the average programmer. The proposed study will study the effectiveness of Stepwise Interactive Debugger for Bison as a tool that aids this accessibility. This will be achieved by studying its helpfulness in performing tasks with the Bison parser-generator. While ease of use and intuitiveness are very hard to quantify, user case studies are a good way to get a sense for the system's effectiveness.

The proposed study will involve student volunteers from the Computer Science department at Tufts University. In addition to graduate students, undergraduate upperclassmen, especially those who are in or have taken Comp 181: Compilers and Comp 105: Programming Languages will also be approached to participate in the study via the online class forums. The only identifying information that will be collected will be the subject's level of familiarity with parser-generator systems.

Subjects will be provided with incorrect parser-generator specifications and inputs, and will be asked to debug these programs with and without `sidBison`. A questionnaire will then be administered, evaluating the intuitiveness and ease of use of the tool.

The anonymous, statistical results of the survey will be included in the final thesis.

Chapter 6

Using sidBison

This section presents a small example of using `sidBison`. Given a `Flex` file `lexer.l`, and a `Bison` file `parser.y`, a lexer shared object `lex.so` can be prepared for input to `sidBison` as follows:

```
ibison -d parser.y
flex lexer.l
gcc -c -fPIC lex.yy.c
gcc -shared -o lex.so lex.yy.o
```

This requires the `LD_LIBRARY_PATH` environment variable to include the directory in which `lexer.so` is available and the `BISON_PKGDATADIR` variable points to the `ibison\data` directory. `sidBison` can now be run with `parser.y` and `lex.so` as command-line arguments.

6.1 Lambda Calculus

The lambda calculus example described in Appendix B involves files lambdacalc.l and lambdacalc.y.

Compilation These files can be compiled to create a lexer shared object file for input for `sidBison` as described below..

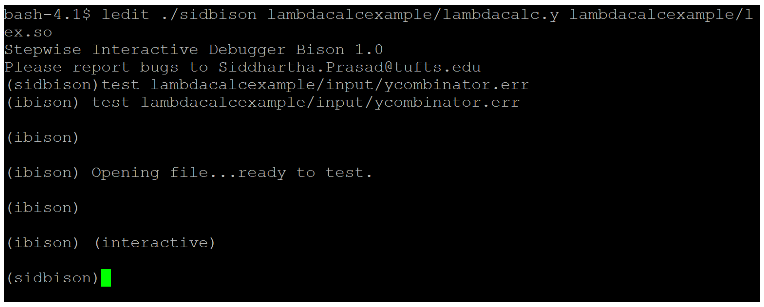
```
ibison -d lambdacalc.y
flex lambdacalc.l
gcc -c -fPIC lex.yy.c
gcc -shared -o lex.so lex.yy.o
```

After compilation, the generated files can be loaded into `sidBison` as command line arguments.

```
./sidbison lambdacalc.y lex.so
```

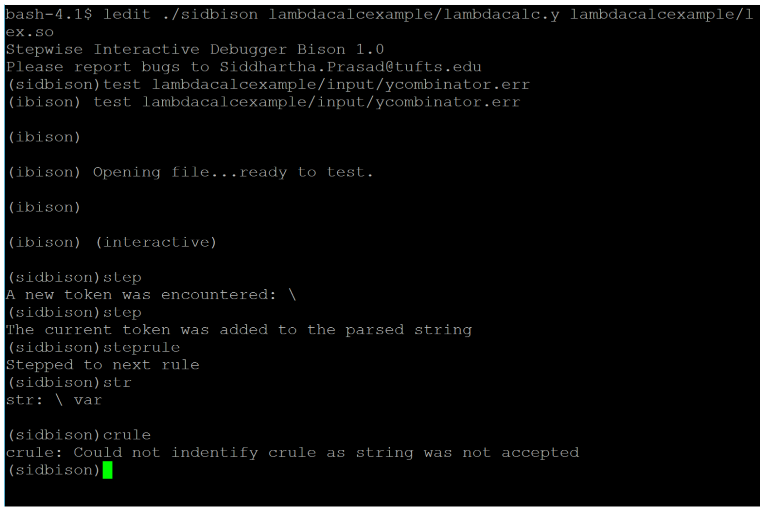
6.1.1 Debugging input

The file ycombinator.err contains the string `\h.(\h.(\x.(h (x x))) \x.(h (x x)))`. This is not accepted by the specified lambda calculus grammar. An example step-through debug is described below.



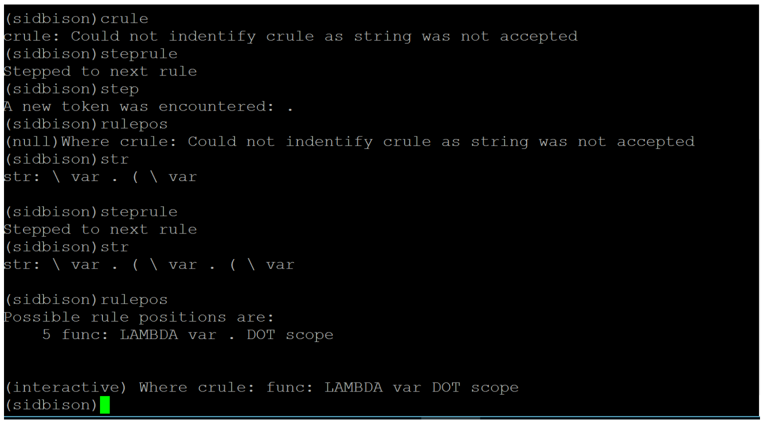
```
bash-4.1$ ledit ./sidbison lambdacalcexample/lambdacalc.y lambdacalcexample/l
ex.so
Stepwise Interactive Debugger Bison 1.0
Please report bugs to Siddhartha.Prasad@tufts.edu
(sidbison)test lambdacalcexample/input/ycombinator.err
(ibison) test lambdacalcexample/input/ycombinator.err

(ibison)

(ibison) Opening file...ready to test.

(ibison)

(ibison) (interactive)

(sidbison)
```

`ycombinator.err` was loaded into `sidBison` using the `test` command.



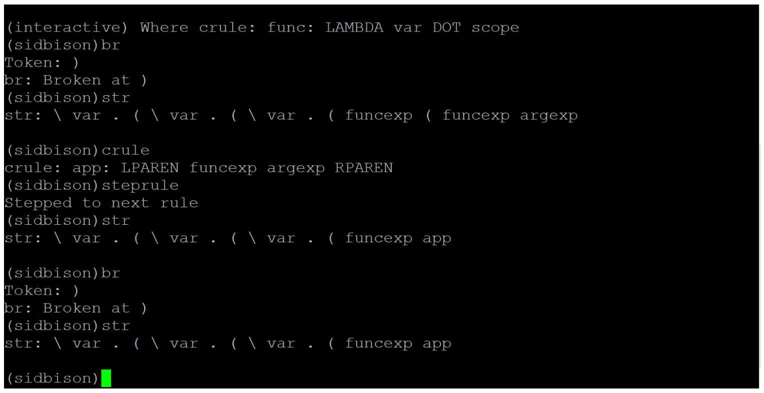
```
bash-4.1$ ledit ./sidbison lambdacalcexample/lambdacalc.y lambdacalcexample/l
ex.so
Stepwise Interactive Debugger Bison 1.0
Please report bugs to Siddhartha.Prasad@tufts.edu
(sidbison)test lambdacalcexample/input/ycombinator.err
(ibison) test lambdacalcexample/input/ycombinator.err

(ibison)

(ibison) Opening file...ready to test.

(ibison)

(ibison) (interactive)

(sidbison)step
A new token was encountered: \
(sidbison)step
The current token was added to the parsed string
(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var

(sidbison)crule
crule: Could not indentify crule as string was not accepted
(sidbison)
```

The `step` command executes a single parser step. The first `step` executes the action of *looking ahead* to the \ character. The `steprule` command is then used to step to the next rule being parsed. The `str` command shows that this rule was named `var`. The current rule being parsed could not be identified since the entire string was not accepted. `iBison` is trying to parse the entire subsequent string, alongside the current results of `str` to a top level rule. However, the string does not match any rules in the grammar.



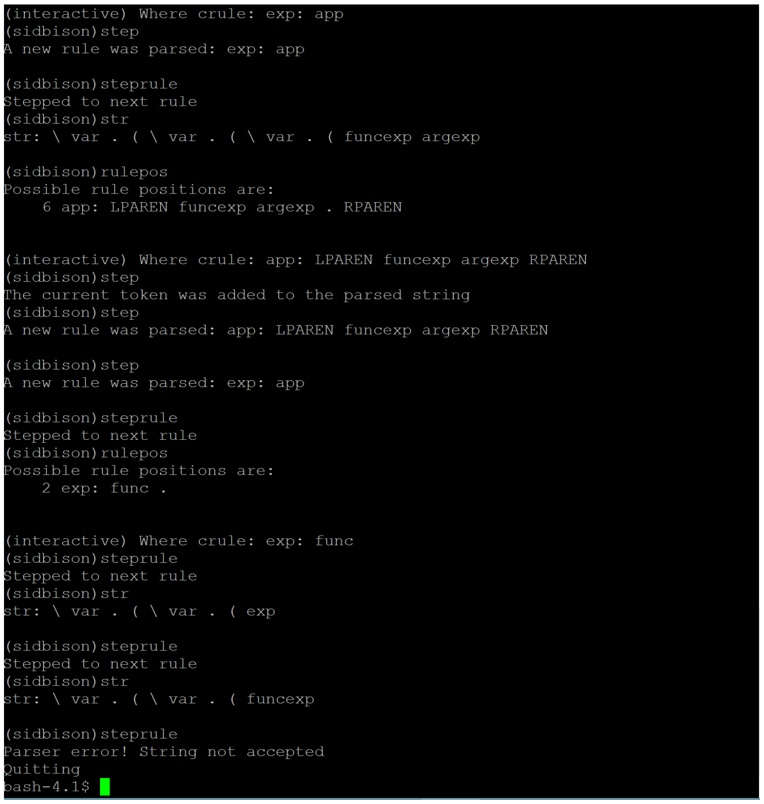
```
(sidbison)crule
crule: Could not indentify crule as string was not accepted
(sidbison)steprule
Stepped to next rule
(sidbison)step
A new token was encountered: .
(sidbison)rulepos
(null)Where crule: Could not indentify crule as string was not accepted
(sidbison)str
str: \ var . ( \ var

(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var . ( \ var . ( \ var

(sidbison)rulepos
Possible rule positions are:
    5 func: LAMBDA var . DOT scope


(interactive) Where crule: func: LAMBDA var DOT scope
(sidbison)
```

After stepping two more rules, the `rulepos` rule uses a `.` to show the current position in the parsing process, as well as the current rule.



```
(interactive) Where crule: func: LAMBDA var DOT scope
(sidbison)br
Token: )
br: Broken at )
(sidbison)str
str: \ var . ( \ var . ( \ var . ( funcexp ( funcexp argexp

(sidbison)crule
crule: app: LPAREN funcexp argexp RPAREN
(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var . ( \ var . ( \ var . ( funcexp app

(sidbison)br
Token: )
br: Broken at )
(sidbison)str
str: \ var . ( \ var . ( \ var . ( funcexp app

(sidbison)
```

The `break` command can be used to step forward till the current token being looked at is a `)`. The `str` command shows the current state in the parsing process.



```
(interactive) Where crule: exp: app
(sidbison)step
A new rule was parsed: exp: app

(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var . ( \ var . ( \ var . ( funcexp argexp

(sidbison)rulepos
Possible rule positions are:
    6 app: LPAREN funcexp argexp . RPAREN


(interactive) Where crule: app: LPAREN funcexp argexp RPAREN
(sidbison)step
The current token was added to the parsed string
(sidbison)step
A new rule was parsed: app: LPAREN funcexp argexp RPAREN

(sidbison)step
A new rule was parsed: exp: app

(sidbison)steprule
Stepped to next rule
(sidbison)rulepos
Possible rule positions are:
    2 exp: func .


(interactive) Where crule: exp: func
(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var . ( \ var . ( exp

(sidbison)steprule
Stepped to next rule
(sidbison)str
str: \ var . ( \ var . ( funcexp

(sidbison)steprule
Parser error! String not accepted
Quitting
bash-4.1$ 
```

The parsing process reaches lower level rules. rulepos— and `crule` show the current rules being parsed. The only rule that starts and ends with parentheses is observed to be `app`. Thus, `iBison` attempts to parse `(\x.(h (x x)))` as a function application with no argument. As a result, ycombinator.err is not accepted.

6.2 Impcore

6.3 Source Code

Source code for `sidBison` be found at `https://github.com/sidprasad/sidbison/`.
Example Flex and Bison specifications can be found in Appendix B.
Source code and instructions for building and installing iBison can be found at `https://www.cs.uic.edu/~spopuri/ibison.html`.

Chapter 7

Additional Work

The ultimate aim of this project is to make parser-generator systems more accessible.

1. The `sidBison` system could be expanded to deal with LL parser generator systems such as ANTLR.
2. `sidBison` could also be ported to `YACC` implementations in languages such as C++, F # and Standard ML.
3. Syntax highlighting and IDE support for Bison files could help improve the readability and debug-ability of grammar specifications.
4. Just-In-Time compiling within an IDE could help highlight potential conflicts in the specification while it is being written.

Appendix A

A Quick Overview of Formal Grammars

A formal grammar is a set of rules that describe how strings might be produced in a language. It can function as both a recognizer and a language generating construct.

Terminals Terminal symbols are actual members of the alphabet that compose the language described by a formal grammar.

Non-terminals Non-terminal symbols can be thought of as textitsyntactic variables that describe groupings or combinations of other symbols, as described by the rules of a formal grammar.

A.1 Context Free Grammars

A context-free grammar is a special kind of formal grammar where each rule is of the form $E \to \alpha$, where E is a non-terminal symbol and α is a string of terminals and non-terminals in the grammar. Such a grammar is not context dependent – the non-terminal symbol E can always be replaced by α, regardless of the situation.

Backus Naur Form (BNF) is a notational method used to describe context-free grammars. It has the basic form $E := \alpha \mid \beta \ldots$ [7]. Here, the non-terminal E could be replaced either by the strings α or β. Thus, the BNF specification for a language $\{a\}^*$ is

```
E  :=  E a
```

```
        | a
```

`Bison` specifications are rooted in Backus Naur Form.

Appendix B

Examples

B.1 Calculator

Presented below are Flex and Bison files that represent a calculator. These specifications were inspired by an iBison example [5]. These files can be debugged with sidBison are presented below.

B.1.1 Lexer file: calcparser.l

```
%{
# include "calcparser.tab.h"
# undef yywrap
# define yywrap() 1
%}

%option noyywrap

int   [0-9]+
blank [ \t]

%%
{blank}+
[\n]+
\+           return OPERATOR_PLUS;
-           return OPERATOR_MINUS;
\*           return OPERATOR_MULT;
\/           return OPERATOR_DIV;
\(           return LPAREN;
\)           return RPAREN;
{int}      return NUMBER;
```

```
.          {fprintf(stderr,"Invalid token!\n");}
%%
```

B.1.2 Grammar file: calcparser.y

```
%{
/* A simple calculator */
void yyerror(const char *s);
%}
%token        NUMBER
%token              OPERATOR_PLUS
%token             OPERATOR_MINUS
%token             OPERATOR_MULT
%token              OPERATOR_DIV
%token             LPAREN
%token              RPAREN
%left OPERATOR_PLUS OPERATOR_MINUS
%left OPERATOR_MULT OPERATOR_DIV
%%
exp: exp OPERATOR_PLUS exp
   | exp OPERATOR_MINUS exp
   | exp OPERATOR_MULT exp
   | exp OPERATOR_DIV exp
   | LPAREN exp RPAREN
   | NUMBER
;
%%
void
yyerror(cosnt char *s)
{
  fprintf(stderr,"Syntax error: %s\n",s);
}
```

B.2 Programmable Computable Functions

PCF is a simple typed functional language proposed by Dana Scott in 1969. Flex and Bison files that can be debugged with `sidBison` are presented below.

B.2.1 Lexer file: pcf.l

```
%{
# include "pcf.tab.h"
# undef yywrap
# define yywrap() 1

%}

%option noyywrap

blank [ \t]
word [a-zA-Z][a-zA-Z0-9]*


%%
{blank}+
[\n]+
0            return ZERO;
:            return COLON;
"true"       return TRUE;
"false"      return FALSE;
\.           return DOT;
\(           return LPAREN;
\)           return RPAREN;
"fix"        return FIX;
"zero"       return ZEROFUNC;
"succ"       return SUCC;
"pred"       return PRED;
"if"         return IF;
"then"       return THEN;
"else"       return ELSE;
"fn"         return FN;
"nat"        return NAT;
"bool"       return BOOL;
"->"         return ARROW;
{word}       return IDEN;
.            {fprintf(stderr, "Invalid token!\n");}

%%
```

B.2.2 Parser file: pcf.y

```
%{
```

```
/* A simple PCF grammar */

%}

%token ZERO
%token TRUE
%token FALSE
%token IDEN
%token FIX
%token ZEROFUNC
%token LPAREN
%token RPAREN
%token SUCC
%token PRED
%token IF
%token THEN
%token ELSE
%token FN
%token COLON
%token DOT
%token NAT
%token ARROW
%token BOOL

%left ARROW

%%


m : ZERO
  | TRUE
  | FALSE
  | var
  | zerofunc
  | succ
  | pred
  | ifelse
  | fun
  | app
  | fix

fix : FIX LPAREN m RPAREN

app : LPAREN callfunc argfunc RPAREN

callfunc : m
```

```
argfunc : m


tau : NAT
    | BOOL
    | funtau
    | LPAREN funtau  RPAREN

funtau : tau ARROW tau


fun : FN var COLON tau DOT m

succ : SUCC LPAREN m RPAREN

pred : PRED LPAREN m RPAREN

zerofunc : ZEROFUNC LPAREN m RPAREN

var : IDEN

ifelse : IF m THEN m ELSE m

%%
/*

app : funcexp argexp

succ : SUCC LPAREN m RPAREN
pred : PRED LPAREN m RPAREN
fix : FIX LPAREN m RPAREN

funcexp : m
argexp : m

fix : LPAREN m RPAREN

func : FN var COLON tau DOT m

*/
void yyerror (const char *s)
{
fprintf(stderr, "Syntax error: %s\n", s);
}
```

B.3 Lambda Calculus

The lambda calculus is a formal system that expresses computation by way of functions. Flex and Bison files that can be debugged with `sidBison` are presented below.

B.3.1 Lexer file: lambdacalc.l

```
%{
# include "lambdacalc.tab.h"
# undef yywrap
# define yywrap() 1

%}

%option noyywrap

blank [ \t]
word [a-zA-Z][a-zA-Z0-9]*
int [0-9]+


%%
{blank}+
[\n]+
{int}       return CONSTANT;
{word}      return IDENT;
\.          return DOT;
\\          return LAMBDA;
\(          return LPAREN;
\)          return RPAREN;
.           {fprintf(stderr, "Invalid token!\n");}

%%
```

B.3.2 Grammar file: lambdacalc.y

```

%{
/* Untyped lambda calculus */
void yyerror(const char *s);

%}

%token  IDENT
```

```
%token  CONSTANT
%token  LPAREN
%token  RPAREN
%token  LAMBDA
%token  DOT

%%

exp : var
    | func
    | app
    | CONSTANT

func : LAMBDA var DOT scope

app : LPAREN funcexp argexp  RPAREN

scope : exp

funcexp : exp

argexp : exp

var : IDENT


;

%%
void
yyerror(const char *s)
{
    fprintf(stderr, "Syntax error: %s\n", s);
}
```

B.4 Impcore

Impcore is a simple imperative language used as a pedagogical tool in Comp 105: Programming Languages at Tufts University. A lexer and parser for impcore are presented below:

B.4.1 Lexer file: imp.l

```
%{
#include "heading.h"
# include "imp.tab.h"
# undef yywrap
# define yywrap() 1

/* Need to include - in word */

%}

%option noyywrap

blank [ \t]
word [a-zA-Z][a-zA-Z0-9\-]*
int [0-9]+


%%
{blank}+
[\n]+
\(              return LPAREN;
\)              return RPAREN;
\+              return PLUS;
-               return MINUS;
\*              return MUL;
\/              return DIV;
"="             return EQ;
\<              return LT;
\>              return GT;
"val"           return VAL;
"define"        return DEFINE;
"use"           return USE;
"check-expect"  return CHECKEXPECT;
"set"           return SET;
"if"            return IF;
"while"         return WHILE;
"begin"         return BEGN;
"print"         return PRINT;
"check-err"     return CHECKERR;
{int}           {return NUMERAL;}
{word}          {return NAME;}
.               {fprintf(stderr, "Invalid token!\n");}

%%
```

B.4.2 Parser file: imp.y

```
%{
/* Leaving unit-test out of def */
#include "heading.h"
#include <stdio.h>
void yyerror(const char *s);
int yylex(void);
%}


%token  VAL
%token  DEFINE
%token  LPAREN
%token  RPAREN
%token  USE
%token  CHECKEXPECT
%token  SET
%token  IF
%token  WHILE
%token  BEGN
%token  PLUS
%token  MINUS
%token  MUL
%token  DIV
%token  EQ
%token  LT
%token  GT
%token  PRINT
%token  CHECKERR
%token  NUMERAL
%token  NAME




%%


program :
        | def program

def : LPAREN VAL variablename exp RPAREN                                          | exp
```

```
    | LPAREN DEFINE functionname formals exp RPAREN
    | LPAREN USE filename RPAREN
    | unittest


unittest : LPAREN CHECKEXPECT exp exp RPAREN
          | LPAREN CHECKERR exp RPAREN

exp : literal
    | variablename
    | LPAREN SET variablename exp RPAREN
    | LPAREN IF exp exp exp RPAREN
    | LPAREN WHILE exp exp RPAREN
    | LPAREN BEGN expstar
    | LPAREN function expstar



expstar : RPAREN
        | exp expstar


formals : LPAREN variablenamestar

variablenamestar : RPAREN
                  | variablename variablenamestar


literal : NUMERAL

function : functionname
         | primitive

primitive : PLUS
          | MINUS
          | MUL
          | DIV
          | EQ
          | LT
          | GT
          | PRINT

variablename : NAME
functionname : NAME
filename : NAME
```

```
;

%%
void
yyerror(const char *s)
{
    fprintf(stderr, "Syntax error: %s\n", s);
}
```

B.5 JSON Grammar

```
    JSON-text : ws value ws

        begin-array     : ws %x5B ws  ; [ left square bracket

        begin-object    : ws %x7B ws  ; { left curly bracket

        end-array       : ws %x5D ws  ; ] right square bracket

        end-object      : ws %x7D ws  ; } right curly bracket

        name-separator : ws %x3A ws  ; : colon

        value-separator : ws %x2C ws  ; , comma


        ws  :                       ; Empty string
                  | %x20 |             ; Space
                  %x09 |               ; Horizontal tab
                  %x0A |               ; Line feed or New line
                  %x0D                 ; Carriage return

        value : false | null | true | object
                        | array | number | string

        false : %x66.61.6c.73.65   ; false

        null  : %x6e.75.6c.6c      ; null

        true  : %x74.72.75.65      ; true

        object : begin-object
```

```
      [ member ( value-separator member ) ]
                end-object

member : string name-separator value

array : begin-array [ value *( value-separator value ) ]
                         end-array

number : [ minus ] int [ frac ] [ exp ]

decimal-point : %x2E       ; .

digit1-9 : %x31-39         ; 1-9

e : %x65 | %x45            ; e E

exp : e [ minus | plus ] 1*DIGIT

frac : decimal-point 1*DIGIT

int : zero | ( digit1-9 *DIGIT )

minus : %x2D               ; -

plus : %x2B                ; +

zero : %x30                ; 0

string : quotation-mark *char quotation-mark

char : unescaped
           | escape (
           %x22 |          ; "    quotation mark  U+0022
           %x5C |          ; \    reverse solidus U+005C
           %x2F |          ; |    solidus         U+002F
           %x62 |          ; b    backspace       U+0008
           %x66 |          ; f    form feed       U+000C
           %x6E |          ; n    line feed       U+000A
           %x72 |          ; r    carriage return U+000D
           %x74 |          ; t    tab             U+0009
           %x75 4HEXDIG )  ; uXXXX                U+XXXX

escape : %x5C              ; \

quotation-mark : %x22      ; "
```

```
        unescaped : %x20-21 | %x23-5B | %x5D-10FFFF
```

Appendix C

Empirical Data

Table C.1: Raw DDEBUGLOG Data

Command	User Time (s)	System Time (s)
br	5.00E-03	8.00E-03
br	0.00E+00	0.00E+00
br	0.00E+00	0.00E+00
br	0.00E+00	0.00E+00
br	9.99E-04	0.00E+00
crule	8.00E-03	2.50E-02
crule	1.40E-02	2.30E-02
crule	6.00E-03	4.00E-02
crule	8.00E-03	4.90E-02
crule	8.00E-03	1.40E-02
crule	1.70E-02	2.60E-02
crule	1.30E-02	6.40E-02
crule	1.80E-02	6.30E-02
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
ctkn	0.00E+00	0.00E+00
rulepos	6.00E-03	3.20E-02
rulepos	2.00E-03	9.00E-03

Continued on next page

Table C.1 – continued from previous page

Command	User Time (s)	System Time (s)
rulepos	1.00E-02	2.90E-02
rulepos	1.30E-02	2.90E-02
rulepos	6.00E-03	4.70E-02
rulepos	5.00E-03	1.80E-02
rulepos	1.10E-02	3.10E-02
step	0.00E+00	0.00E+00
step	0.00E+00	0.00E+00
step	0.00E+00	1.00E-03
step	0.00E+00	0.00E+00
step	0.00E+00	1.00E-03
step	0.00E+00	5.00E-03
step	0.00E+00	0.00E+00
step	1.00E-03	0.00E+00
step	0.00E+00	1.00E-03
step	0.00E+00	6.00E-03
step	0.00E+00	0.00E+00
step	0.00E+00	1.00E-03
step	9.99E-04	7.00E-03
step	0.00E+00	0.00E+00
step	0.00E+00	4.00E-03
step	0.00E+00	1.00E-03
step	0.00E+00	9.99E-04
step	0.00E+00	1.00E-03
step	0.00E+00	1.00E-03
step	0.00E+00	1.00E-03
step	0.00E+00	0.00E+00
step	0.00E+00	2.00E-03
step	0.00E+00	1.00E-03
step	0.00E+00	1.00E-03
step	0.00E+00	1.00E-03
step	1.00E-03	0.00E+00
step	0.00E+00	0.00E+00
step	0.00E+00	0.00E+00
steprule	1.00E-03	2.00E-03
steprule	0.00E+00	3.00E-03
steprule	0.00E+00	3.00E-03
steprule	0.00E+00	1.00E-03
steprule	1.00E-03	4.00E-03
steprule	0.00E+00	3.00E-03
steprule	1.00E-03	3.00E-03
steprule	0.00E+00	2.00E-03
steprule	9.99E-04	1.00E-03
steprule	0.00E+00	2.00E-03

Continued on next page

Table C.1 – continued from previous page

Command	User Time (s)	System Time (s)
steprule	0.00E+00	2.00E-03
steprule	1.00E-03	1.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	2.00E-03
steprule	1.00E-03	2.00E-03
steprule	1.00E-03	9.99E-04
steprule	0.00E+00	4.00E-03
steprule	1.00E-03	3.00E-03
steprule	0.00E+00	4.00E-03
steprule	0.00E+00	3.00E-03
steprule	1.00E-03	1.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	3.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	2.00E-03
steprule	1.00E-03	1.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	1.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	5.00E-03
steprule	2.00E-03	3.00E-03
steprule	0.00E+00	2.00E-03
steprule	0.00E+00	3.00E-03
stprule	0.00E+00	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	9.99E-04
str	0.00E+00	1.00E-03
str	1.00E-03	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	9.99E-04
str	1.00E-03	0.00E+00
str	0.00E+00	1.00E-03
str	0.00E+00	0.00E+00
str	1.00E-03	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	1.00E-03
str	0.00E+00	5.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	0.00E+00
str	1.00E-03	0.00E+00

Continued on next page

Table C.1 – continued from previous page

Command	User Time (s)	System Time (s)
str	0.00E+00	1.00E-03
str	0.00E+00	7.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	9.99E-04
str	0.00E+00	0.00E+00
str	0.00E+00	4.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	1.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	5.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	0.00E+00
str	0.00E+00	1.00E-03
str	0.00E+00	0.00E+00
str	0.00E+00	8.00E-03
str	0.00E+00	0.00E+00

Bibliography

[1] 2015. URL: http://www.gnu.org/software/bison/manual/html_node/Simple-GLR-Parsers.html (visited on 03/21/2016).

[2] Andrew W Appel. *Modern Compiler Implementation in ML*. Cambridge UP, 1998.

[3] Donnelly Charles and Richard Stallman. *Bison The YACC-compatible Parser Generator*. 1995. URL: http://dinosaur.compilertools.net/bison/index.html (visited on 09/26/2015).

[4] Donnelly Charles and Richard Stallman. *The Bison Parser Algorithm, Bison 1.25*. URL: http://dinosaur.compilertools.net/bison/bison_8.html (visited on 02/17/2016).

[5] Popuri Satya K. *Interactive Mode Bison*. URL: https://www.cs.uic.edu/~spopuri/ibison.html (visited on 02/17/2016).

[6] Popuri Satya K. *Understanding C Parsers Generated by GNU Bison*. URL: https://www.cs.uic.edu/~spopuri/cparser.html (visited on 02/17/2016).

[7] Garshol Lars. *BNF and EBNF: What Are They and How Do They Work?* 1998. URL: http://www.garshol.priv.no/download/text/bnf.html (visited on 02/18/2016).

[8] *LR Parser Wikipedia. Wikimedia Foundation.* URL: https://en.wikipedia.org/wiki/LR_parser (visited on 03/28/2015).

[9] Johnson Mark. "The Computational Complexity of GLR Parsing". In: *Generalized LR Parsing*. Ed. by Masaru Tomita. Boston: Kluwer Academic, 1991. Chap. 3.

[10] Might Matthew and David Darais. "Yacc Is Dead". University Of Utah, Salt Lake City, Utah, USA. 2009.

[11] Bidwell Daniel R. *History of Parsing Methods*. URL: https://www.andrews.edu/~bidwell/456/history.html (visited on 02/17/2016).

[12] Bray Tim. *The JavaScript Object Notation (JSON) Data Interchange Format*. URL: http://rfc7159.net/rfc7159 (visited on 10/07/2015).