Learning Programming Language Parsing Rules

James Cheney

May 13, 1999

1 Introduction

Modern programming languages have some natural language properties, such as scoping/precedence ambiguities regarding arithmetic operators (e. g. + vs. *) and word (symbol) sense ambiguities (e. g. unary vs. binary +). Typically these ambiguities are resolved according to well-known conventions so that programs do not need to contain too many parentheses. These ambiguities are not necessary in programming languages, but they are common because people seem to find languages which allow such ambiguities more readable.

Efficient parsers for such languages are difficult to construct by hand. Usually this problem is solved by using parser generation tools (such as `yacc` [9, 7]) which transform a context-free grammar in some restricted form (such as LR or LALR; see [1]) into an efficient (linear time) parser. However, these tools are also difficult to use. It is not too difficult to write LALR grammars to parse individual ambiguous programming language (PL) constructs correctly, but it is not trivial to combine all of these parsing rules while obeying the LALR restrictions.

In this project I considered the problem of improving the "user interface" to constructing efficient parsers for modern programming languages. Because programming language parsing has natural language aspects, I attempted to apply techniques from natural language research to this problem. The long term goal of such research would be a system which divorces the human-readable *concrete syntax* of a programming language from the *abstract syntax* which is the proper input to the compiler. Due to time constraints, I focused on learning parsing rules for the arithmetic expression sublanguage of the popular programming language Java.

I implemented a system based on the CHILLIN inductive logic programming algorithm which learns the control rules for a shift-reduce parser from examples of correct parses of these expressions. This system can learn how to parse the expressions in the source code of the JDK 1.1 Java language library (which contains about 2000 expressions) with exact match, accuracy, and precision rates all exceeding 99%. However, there is reason to believe that the approach of using a large corpus of examples gathered from nature is overkill in this setting, and that instead it suffices to construct a relatively small but rich set of examples. For example, it appears to be possible to learn the correct rules for parsing expressions using only the + and * operators and parentheses from about 50 randomly generated examples.

2 Related Work

There is substantial research on parsing in the NLP literature, but there are only a few papers devoted to the problem of grammar acquisition from examples. One line of research in this direction is the inside-outside algorithm[2, 8, 11], a statistical algorithm which attempts to estimate the probabilities of a stochastic CFG which best fit the data in a corpus of correct parses. This approach has potential disadvantages for PL parsing. PL parsers must be fast, and so the superlinear parsing time of SCFGs is unacceptable. Also, it would be an advantage if the acquired grammar was human-readable and alterable, but stochastic/probabilistic grammars are not easy to modify or correct by hand.

In another approach, machine learning techniques are used to infer the control rules for a shift-reduce parser, again using examples drawn from correct parses. The research of Zelle and Mooney[13] in applying this approach to English parsing, case-role mapping, and database query translation suggests that this approach may be viable for the somewhat easier problem of PL parsing. Shift-reduce parsers are fast, and Zelle and Mooney's system CHILL produces learned control rules which are human-readable Prolog programs, so this approach may sidestep the possible problems with statistical methods.

The chief machine learning technique which Zelle and Mooney use is *inductive logic programming* (ILP). ILP is a recently developed field which "stands at the intersection of machine learning and logic programming"[10]. In this approach to learning, one attempts to learn a target concept from positive and negative examples of the concept, possibly taking some preexisting background knowledge into account. The approach used in this paper is directly based on the CHILLIN inductive logic programming algorithm of Zelle and Mooney, and so I will first discuss ILP in general and their system in particular.

2.1 Inductive logic programming research

In this section we will give an overview of the concepts underlying ILP and the capabilities of such systems, and describe some other problems to which ILP has been successfully applied. However, ILP is still an evolving field. For a more comprehensive overview, see [10].

In logic programming, one might have many instances of the `parent` and `grandparent` relations, as follows:

```
parent(joe,jan).  parent(jan,ira).  grandparent(joe,ira).
```

However, we might not know that `grandparent(A,B) :- parent(A,C), parent(C,B)`. This knowledge is an important regularity which is present in the data, but which might not be obvious to us if the familiar names and words `parent`, `grandparent` were not used. ILP methods attempt to infer general definitions of concepts, called *target concepts*, from examples. Most implemented ILP systems are able to infer the `grandparent` relationship from a number of *positive examples* such `grandparent(joe,ira)` above, together with some *negative examples* such as

`grandparent(ira,joe).  grandparent(jan, ira).`

In this situation the facts about parenthood are part of what is referred to as *background knowledge*: facts and concepts already accessible. It is not always easy to provide all the background knowledge that a system might need to successfully infer a complicated relationship.

For example, in order for an ILP system to learn `quicksort`, correct definitions of `append`, `components`, and `partition` (which are the main "subroutines" in a logic program for `quicksort`) must be part of the background knowledge. It is impossible for most current ILP systems to learn `quicksort` from pure sorting examples alone. Instead, if any definition is learned at all it is likely to be that of some less efficient version of sort such as $O(n!)$ stupid-sort or $O(n^2)$ insertion-sort.

One explanation for this phenomenon might be that quicksort (with its associated subroutines) is a more complicated explanation for the data than insertion-sort and hence easier for a search algorithm to find. Quicksort's auxiliary routines are not necessary to explain sorting. However, when they are available as background information, the search routine might find it easier to explain how sorting can take place using the auxiliary routines than to use the insertion sort explanation. Consequently one would expect that if the helper routines `split` and `merge` for `mergesort` were provided, an ILP algorithm ought to learn how `mergesort` works. Background knowledge helps an ILP algorithm form a predisposition or bias towards a particular kind of solution (hopefully the desired one). It is interesting to note that this appears to be the reverse of what humans do when discovering or "learning" such algorithms; the basic ideas seem to come first and the helper functions and auxiliary concepts afterwards.

It would be a desirable property for ILP systems to be able to invent new concepts which seem to be relevant to the target concept, and which ought to be part of the background knowledge. For the example of quicksort, it seems unlikely that any efficient procedure could invent the `partition` function that was necessary to guide the search toward quicksort, but in less difficult settings predicate invention can be feasible.

Most ILP algorithms work by generating hypothetical definitions of the target concept, evaluating it in some way with respect to the examples, and then modifying the hypothesis and repeating the process until some sort of convergence is reached. Some algorithms begin with a general hypothesis which covers all the positive examples, and then specialize it to avoid covering the negative examples; these are called *top-down* algorithms. Conversely, *bottom-up* methods start with a specific hypothesis, namely that every positive example is true and no others, and attempt to generalize it. The hypothetical definition is typically evaluated both by how many positive and negative examples it covers, and by its size, because by Occam's razor the simplest explanation is thought to be the best.

In the grandparent example, we could generate positive examples by enumerating all true grandparent relationships within a finite set of people, and take the remaining examples as negatives. Thus there are $O(n^2)$ examples total, and most of them are probably negative. For more complicated target concepts, there might be exponentially many total examples, most of which are negative. Generating and using these examples is likely to be expensive, and most concepts do not require so many negative examples. For this reason some ILP systems contain facilities for learning concepts from only positive examples, based on statistical or counting estimates of the number of negative examples hypotheses are likely to cover, given the positive examples.

2.2 Learning approaches

Zelle and Mooney, in [13], present an approach to parser induction based on machine learning. In particular, they present a framework in which first an overly general parser is obtained by examining a training set of correctly parsed sentences and extracting the grammar rules needed to produce the parses. Also, the correct parses are analyzed to determine the situations in which a shift-reduce parser would shift or reduce. The former become positive examples for the `shift` predicate and the latter become its negative examples. A general definition for `shift` is then inferred from the examples using ILP. Finally, the accuracy of the inferred parser based on `shift` can be tested by using `shift` to control the behavior of a shift-reduce parser while parsing the test sentences. An implementation CHILL is described in the paper, which achieves comparable bracketing results to [3] on a restricted 536-sentence corpus with lexical tags. (CHILL produces full parses, not bracketings as Brill's system does.) CHILL combines top-down and bottom-up search heuristics, and includes a facility for inventing new concepts when appropriate.

In [15], the CHILL system is extended to work using only positive examples. Under certain assumptions on the structure of the target concept and positive examples, the number of negative examples covered by a hypothetical concept can be determined implicitly with enough accuracy that explicit negative examples are unnecessary. This result is for a case-role mapping parser instead of a full parser, but the technique should generalize.

In [14], the approach above in which the shift-reduce control predicate is learned is compared with a naive ILP approach in which the actual parsing predicate is inferred. This paper shows that the results of the shift-reduce framework significantly outperform the parser obtained by naive ILP on identical training sets of examples. Thus, the shift-reduce framework provides a "bias" towards parsing, because it is not necessary to learn the routine operation of the parser. The naive ILP method requires a substantially larger number of examples because it must also (try to) learn how parsing should work before it can learn how to parse correctly.

Another interesting development of this research is in [12], in which a technique called *active learning* is used to guide the choice of training examples used in the CHILL system. In active learning, the system is trained using a small number of examples and then determines which of the remaining examples would be most informative. The informative examples are usually the ones mispredicted or partially predicted (that is, partially parsed correctly) by the current hypothesis. When this method of example selection was applied to CHILL, it was found that significantly fewer examples were needed to achieve a given level of performance.

3 Experiment

I implemented a system based on Zelle and Mooney's CHILLIN ILP algorithm which learns how to parse expressions in the programming language Java. In order to reduce the complexity of the problem, only the sublanguage of expressions was used, not the full Java language. Java was chosen because its LALR grammar is fixed in the language definition[6], and because many compilers are available which parse this language.

The CHILLIN algorithm was designed to learn control rules for shift-reduce natural language parsers. Programming languages are typically parsed using shift-reduce parsers. In this project, therefore, I attempted to use CHILLIN to learn control rules for parsing Java expressions. CHILLIN is available from the authors electronically in source code form, and I used it without modifications.

The system consists of several components, in several programming languages. One component reads in Java files and parses the expressions (using an existing Java compiler). The second part splits the parses into training and test examples. The training examples are converted into positive and negative instances of the `shift` predicate, which determines when to shift and when to try to reduce. CHILLIN is then run using these examples. Finally, the resulting definition for the `shift` predicate is used as the control of a shift-reduce parser to parse the test file. The following sections discuss these components in slightly greater detail.

3.1 The reference parser

The Jikes compiler is a Java compiler available in source code form from IBM Research[1]. Jikes is intended to correctly implement the Java specification in all its glorious detail, so I have more confidence in the correctness of Jikes' parses in strange situations than I might in other free compilers.

Since CHILLIN is written in Prolog, it is necessary to get Prolog terms representing parses out of Jikes. I modified Jikes' abstract syntax tree print function to recursively print the appropriate Prolog terms. My implementation was not perfect and so in rare cases some of the results were not Prolog terms. In such cases these problematic examples were deleted.

[1] http://www.research.ibm.com/jikes

3.2 The example generator

The example builder's task is to read the example parse trees produced by Jikes and produce the positive and negative examples of the `shift` predicate which correspond to a correct shift-reduce parse producing the input parse tree. This is possible because it is possible to reconstruct the correct sequence of shifts and reduces from the parse tree using the reductions. For this to work requires that the reductions be one-to-one; that is, it must not be possible to reduce two different stacks to the same parse tree. This can be thought of as "unparsing" the parse tree.

Since in this case the possible reductions are all known (from the grammar in the Java specification), it is not a problem to build these into the unparser. The reductions needed are those of the general, ambiguous grammar, not the unambiguous LALR grammar needed by `yacc`. For example, the reductions for + and * are:

```
reduce([B,'+',A|S],['+':[A,B]|S]).
reduce([B,'*',A|S],['*':[A,B]|S]).
```

Precedence ambiguities are resolved using the shift-reduce control rules, not the grammar.

There is one additional complication concerning these grammar rules. In my system, the obvious rule for *unary* + is

```
reduce([A,'+'|S], [pre:['+',A]|S]).
```

The unary + rule matches the stack whenever the binary rule does. If these rules are used for parsing then the unary reduction is either always used or only used when there are exactly two constituents on the stack. Neither is the correct behavior, so an additional test is needed to determine when these rules can be used. I used a simple test to determine whether A (and optionally B) are expressions or operators (such as the "==" operator of Java). Only when A (and B for binary +) are expressions is it safe to reduce using either "+" rule.

This is an example of a "word" sense disambiguation problem in PL. Here I was forced to handle it manually because the ambiguity was in the actual reductions, so the learning algorithm wasn't aware of it. However, this problem was simple enough to fix by hand that I don't think it creates a usability problem.

3.3 CHILLIN

CHILLIN is available in Prolog source code form from the authors[13]. It reads in the examples of the `shift` predicate and performs alternating top-down, bottom-up and compressing stages to try to learn the smallest definition for the `shift` predicate it can. I used it without modification.

CHILLIN also contains a facility for predicate invention. This means that CHILLIN sometimes will try to make a new predicate which considers only the relevant parts of the arguments to the `shift` predicate (namely, the stack and input stream). For example, typically when learning how to resolve +,* ambiguities, CHILLIN will invent a predicate `ipred2` which is true of * and +, and state that `shift([A,B|C],[D|E])` holds when `ipred2(D,B)` holds.

Once it decides it can't do any better, CHILLIN outputs a Prolog readable definition of the `shift` predicate.

3.4 The test parser

Using the definition for the `shift` predicate produced by CHILLIN, the test module attempts to parse each of the expressions of the test set. Because the expressions are in parsed form already, it is first necessary to unparse them (using the same unparsing method as the example generator.) Then the `shift` predicate is read in and used as the control of a shift-reduce parser on the input streams which resulted from unparsing the expressions. If the shift-reduce parser gets stuck, that is, `shift` is false and no reduction applies, then the parser returns `stuck(S,I)` where `S` is the stack at that point and `I` is the input stream. This partial parsing information is used later to obtain accuracy and precision information.

However, in a program there are bound to be many "expressions" which are trivial to parse, for example, single numbers and variables. These are filtered out by the test program, since counting them inflates the results for no good reason.

Once all the test examples have been (re-)parsed, they are compared with the original parses in a number of ways. First we count how many parsings *exactly match* the originals. We also calculate the *precision* (#correct found / # found) and *accuracy* (#correct constituents found/#possible correct constituents) of the parses. Finally we calculate the percentage of *false positives* (# fully parsed examples / # incorrectly parsed examples); that is, rate at which incorrectly parsed examples are fully parsed. In an interactive parser learning system, these are the examples which would cause the most difficulty.

4 Results

The JDK1.1 class library was fed into the above system. (Some of the parses produced by my modified Jikes were broken; these were deleted.) 30% of the expression parses were reserved as test examples. CHILLIN was used to learn a definition of the `shift` predicate consistent with the training examples. This took almost exactly 6 minutes on virgo (which is the only CS machine that runs the right version of Prolog to run CHILLIN). The resulting predicate was used to parse the test examples.

The exact match rate, accuracy, and precision were 99.23%, 99.45%, and 99.39% respectively. Recall that these numbers do not include the trivial one-symbol parses caused by single numbers and variable names. The false positive rate was 66.67%, but this is not too significant because there were only three missed examples. (out of about 390). Also, inspecting the results shows that the two false positives were correct except some + operators were misassociated. Except in numerically intensive applications, it is usually safe for + to be fully associative.

For comparison, two other experiments were performed. In the first, the test set was parsed using the trivial "always shift" control rule (which actually works in many simple cases). This rule obtained an exact match rate of 88.21%, an accuracy of 90.44%, a precision of 91.44%, and a false positive rate of 41.30%.

In the second additional experiment, the predicate learned from the training set was used to parse the expressions of two different Java applets, one a graphical "dining philosophers problem" applet from Sun's sample applets web page, and one a Java version of the Linpack benchmark. For the first applet, the exact match rate was 90.2% vs. 83.66% using the "always shift" rule. For the second applet, these numbers were 98.59% vs. 89.44% respectively.

5 Analysis

The first observation we can make about the experiments is that the default parsing rule of "always shift" is correct most of the time. This is perhaps because many of the expressions in the test data were simple assignments of one variable to another, or simple additions or multiplications with no scope ambiguity. However, in programming language parsing it is necessary to handle all cases correctly. For languages in which the grammar is formally specified, it is necessary for the learned parser to be 100% correct in order for it to be acceptable.

Based on the experiments with the large training corpus, we can conclude that this learning method does learn the correct parsing rules for parsing the test set with above 99% exact match rate, accuracy and precision. The missed examples probably contain symbols or usages not found anywhere in the training data. There were very few such examples since the training and test examples were drawn from the same body of Java programs.

Although the system was able to learn how to parse a test set that was similar to the training set, the resulting parser was not general enough to correctly parse other Java programs, even though the training set was large. This suggests that the examples in the training set chosen were not sufficiently rich. Other experiments I have conducted suggest that programs people typically write do not cover all of the parsing situations in enough detail for this system to learn exact parsing.

However, it might be possible to create fairly compact, artificial programs or sets of examples which do cover all of the informative parsing situations. For example, it appears to be possible for the system to completely learn how to parse expressions involving +, *, and parentheses from between 10 and 50 randomly generated examples. The resulting parser could be modified by hand to make the right decisions for all the other operator precedence decisions, or additional similar examples could be written by hand to cover these cases. In Java there are about 35 operators and about 10 precedence levels, so it might be necessary to consider between 100 and 1000 possible interactions. However, there is no theoretical obstruction to doing so, possibly using automatically generated examples.

It might also be possible to introduce background knowledge to the effect that "+ has lower precedence that *", etc., and allow the learning algorithm to figure out how these facts relate to the shift operator. For example, if the background knowledge contains complete precedence information and CHILLIN decides that the same action should be taken when either a * or a / is seen, it might generalize this behavior to the precedence level of * and /. Since the mod operator % occupies the same precedence level, the resulting parser would handle % the same way it handles * and /, even if no % operators were seen in the training data. This could lead to increased performance with less training data.

The current parsing system is guaranteed to get stuck if it encounters a symbol not seen in the training data, because for the learned `shift` predicate to be different from the "always shift" rule, it cannot include a rule that lets it shift whenever it sees a new symbol. This would actually be a a desirable behavior in an interactive system based on this approach, because it ensures that the user is notified whenever the parser needs more training examples. Along the same lines, it would also be an advantage if the false positive rate was generally low, since this would also ensure that the user is made aware of parsing problems because the parser fails whenever they occur.

6 Conclusions and Future Work

In this project I have implemented a system based on the CHILLIN inductive logic programming algorithm of Zelle and Mooney[13] which learns how to disambiguate some "natural language" features of programming languages. This system was restricted to the expression sublanguage of Java, but its implementation is general enough that it should be possible to extend the system to cover all of Java or some other programming language. The system can learn how to parse the expressions collected from the JDK class library with exact match rate, accuracy, and precision exceeding 99%.

There is evidence that even large corpora such as the Java class library do not cover all of the parsing situations needed for an exact parser to be learned. Instead, it seems likely that a more interactive approach might lead to quicker parser construction, possibly involving some form of boosting or active learning in which additional examples are chosen to be as informative as possible.

It has yet to be shown that a system like this one could be used as a replacement for current parser construction tools such as `yacc` which require the user to specify an unambiguous grammar in a special, unintuitive form. Showing this would require comparing the speed of the learned parsers with those generated by `yacc`, building a more usable, complete and robust parser learning system, and performing usability and productivity studies comparing such a system with `yacc`.

There may be some additional applications of this technique to related problems in software engineering research. One example might be *reverse engineering*: given a program which reads some (presumably context-free) language (for which the source is no longer available or obtuse), what language does it accept? This would probably work only if the general, ambiguous form of the language was known, and informative examples could be constructed and given to the unknown program automatically. A further application of this technology might be more systematic verification of existing programming language parsers. In [4], Cohen uses machine learning in a similar situation, that of recovering database specifications from implementation source code.

Another possible application is in programming language error detection and correction. Currently, error detection and message systems evolve over time until they become usable. This is why some compilers and languages (like gcc for C) have much more informative warning messages than others (like, for example, the SML compiler SML/NJ). Is there some way of using learning to decrease the human effort involved, and accelerate the rate at which error handling systems become helpful to users? Cohen, in [5], applies machine learning to a similar problem, that of detecting faults in large software systems from the source code.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools.* Addison-Wesley, Reading, MA, 1986.

[2] J. K. Baker. Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547–550, 1979.

[3] Eric Brill. Automatic grammar induction and parsing free text: A transformation-based approach. In *Proceedings of the 31 st ACL*, pages 259–265, 1993.

[4] William W. Cohen. Recovering software specifications with inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 142–148, 1994.

[5] William W. Cohen and Prem Devanbu. A comparative study of inductive logic programming methods for software fault prediction. *Machine Learning*, ?(?), 1997.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[7] S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report \#32, Bell Laboratories, Murray Hill, NJ, 1975.

[8] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[9] Tony Mason and Doug Brown. `lex` *&* `yacc`. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1990.

[10] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[11] F. Pereira and Y. Schabes. Inside-outside re-estimation from partially bracketed corpora. In *Proceedings of the 30th Annual meeting of the ACL*, pages 128–135, Newark, 1992.

[12] Cynthia A. Thompson, Mary Elaine Califf, and Raymond J. Mooney. Active learning for natural language parsing and information extraction. In *Proceedings of the Sixteenth International Machine Learning Conference*, Bled, Slovenia, 1999.

[13] John M. Zelle and Raymond J. Mooney. Inducing deterministic prolog parsers from treebanks: A machine learning approach. In *Proceedings of the Twelfth National Conference on AI (AAAI-94)*, pages 748–753, Seattle, WA, July 1994.

[14] John M. Zelle and Raymond J. Mooney. Comparative results on using inductive logic programming for corpus-based parser construction. In E. Riloff S. Wermter and G. Scheler, editors, *Symbolic, Connectionist, and Statistical Approaches to Learning for Natural Language Processing*. Springer Verlag, 1996.

[15] John M. Zelle, Cynthia A. Thompson, Mary Elaine Califf, and Raymond J. Mooney. Inducing logic programs without explicit negative examples. In *Proceedings of the Fifth International Workshop on Inductive Logic Programming*, Sepetember 1995.

A Sample output

A.1 Input Java code

Here is a Java program with a number of interesting expressions in it:

```
class Benchmark {
        public static void main(String args[]) {
        int i = 0, j = 0, k = 0, x = 0;
        x = i +(j+k);
        x = (i+j)*k;
        x = (i*j)+k;
        x = i * -j;
        x = +i +j+k;
        x = -i+j*k;
        x = +i*-j+(-k);
        }
```

```
}
```

A.2 Parse trees

Some of the parse trees generated by the above program are:

```
assn:[assnop:'=', v:vx, '+':[v:vi, par:['+':[v:vj, v:vk]]]] .
assn:[assnop:'=', v:vx, '+':[par:['*':[v:vi, v:vj]], v:vk]] .
```

These correspond to the first two assignments to `x`.

A.3 `shift` predicate examples

Here are the positive and negative examples of the `shift` predicate learned from the second parse tree. The negative examples are marked `reduce`. The first argument to the predicate is the stack state `[Top|Rest]` and the second argument is the input stream state. Shifting consists of moving the next input token to the top of the stack. Note that the order of the tokens is thereby reversed.

```
shift ([], [v:vx, assnop: (=), '(', v:vi, +, v:vj, ')', *, v:vk]).
shift ([v:vx],   [assnop: (=), '(', v:vi, +, v:vj, ')', *, v:vk]).
shift ([assnop: (=), v:vx],   ['(', v:vi, +, v:vj, ')', *, v:vk]).
shift (['(', assnop: (=), v:vx],   [v:vi, +, v:vj, ')', *, v:vk]).
shift ([v:vi, '(', assnop: (=), v:vx],   [+, v:vj, ')', *, v:vk]).
shift ([+, v:vi, '(', assnop: (=), v:vx],   [v:vj, ')', *, v:vk]).
reduce([v:vj, +, v:vi, '(', assnop: (=), v:vx],   [')', *, v:vk]).
shift ([+ :[v:vi, v:vj], '(', assnop: (=), v:vx], [')', *, v:vk]).
reduce([')', + :[v:vi, v:vj], '(', assnop: (=), v:vx], [*, v:vk]).
shift ([par:[+ :[v:vi, v:vj]], assnop: (=), v:vx],     [*, v:vk]).
shift ([*, par:[+ :[v:vi, v:vj]], assnop: (=), v:vx],     [v:vk]).
reduce([v:vk, *, par:[+ :[v:vi, v:vj]], assnop: (=), v:vx],   []).
reduce([* :[par:[+ :[v:vi, v:vj]], v:vk], assnop: (=), v:vx], []).
```

A.4 Learned `shift` predicate

The shift predicate learned from the above examples is:

```
ipred5('(').
ipred5(A:B).
ipred8(=).
ipred8([-,v:vi]).
shift(A,[B:C|D]).
shift([A,B,C:D,E:F|G],[H,v:I]) :-
        ipred8(D).
```

```
shift([A:B,C|D],[E,F|G]) :-
        ipred5(C).
```

This set of examples is to small to learn correct rules for dealing with unary and binary + and -, *, and parentheses. Instead, it comes up with special conditions (like `ipred8([-,v:vi])`) which actually only help parse the training data. This indicates that overtraining is taking place.

B How to use the system

The system is provided as a gzipped Unix tar file. Untar it into an empty directory. There should now be four subdirectories: bin, examples, java, and prolog.

The bin directory contains binary files. All of these run on Solaris 2.6 machines except `chillin-nr` which runs only on the machine virgo. The examples directory contains some auxiliary files needed by CHILLIN and also `always.r`, an artificial CHILLIN "result" file which encodes the default "always shift" rule. The java directory contains several Java programs and the .ast files containing Prolog terms for parses of the expressions in them. The prolog directory contains prolog sources for the example generator and test parser.

My system is much more fun to use if the bin directory is in your PATH.

B.1 Generating the examples

In what follows, `root` is an arbitrary file name which will have many different extensions.

The Jikes compiler is too large to fit on a floppy disk. So instead I have included the .ast files produced by Jikes. If you had my modified version of Jikes, called `pjikes`, the command line for creating an ast file from a Java source would look like this:

```
pjikes +A root.java > root.ast
```

The +A flag tells Jikes to print out the abstract syntax tree. I modified the print routine to print out (mostly correct) Prolog terms representing the parsed expressions.

The resulting file contains Prolog terms, but for Prolog to be able to read them the lines must be terminated with dots. There might be empty lines, which should not be terminated with dots. The Perl script `dots.pl` does this automatically. To do this, do

```
dots.pl < root.ast > root.examples
```

Now that Prolog can read the example parses, we can split the data set into training and test examples. The program `splitex` (which is a Prolog standalone executable) does this. Use the command line

```
splitex root root.examples
```

Finally you run `genshift` on the training set (or the test set, if you want) to produce examples for CHILLIN.

`genshift root root.train`

where root is the root file name you want the resulting .f (positive example) and .n (negative example) files to have.

Also, you need to copy either `examples/trace.i` or `examples/basic.i` to `examples/root.i`.

Luckily, there is a shell script `doast` which performs all these steps for you. In the root project directory, if you do `doast root` then all of the above steps should happen automagically. However, you might need to edit it so that it sets the path correctly for your environment.

B.2 CHILLIN

Now we can run CHILLIN on the examples we generated. CHILLIN expects three files: `root.i`, `root.f`, and `root.n`. The .i file contains special information for CHILLIN, such as how much trace information to print and what background knowledge to use (in our case, none.) The .f file contains positive (foreground) examples and the .n file contains negative examples.

To run CHILLIN on the generated examples, go to the machine virgo and change to the examples subdirectory. Do

`chillin-nr root`

CHILLIN should now do its thing. It will print trace information and eventually the total time and resulting predicate. (If you don't want the trace, comment out the trace lines in `root.i`).

The "-nr" is for the non-recursive version of CHILLIN. CHILLIN can learn recursive predicates, but that turns out to be overkill for learning shift-reduce control rules, and the recursive version sometimes draws the wrong conclusions.

The reason you must be using virgo to run CHILLIN is that the version of Prolog that the CHILLIN source uses is only installed on that machine. It should be possible to rewrite it to run on standard Prolog systems, especially now that I know Prolog, but I didn't have time.

B.3 Testing the results

Once CHILLIN is done, there should be a file called `root.r` in the examples subdirectory. This is a Prolog file containing a definition for `shift`. (CHILLIN also printed this out when it terminated.) To test the results on a set of parsed examples, say `root.test`, use the `stest` program as follows:

`stest root.test root.r`

This uses `root.r`'s definition of `shift` to try to parse each of the examples in `root.test`. The resulting inputs, correct parses, and attempted parses are printed out, and once all of the parsing is done the statistics on correctness are calculated and printed out.

Any .r file can be used as the second argument. So, for example, to use the "always shift" artificial predicate, you could do

`stest root.test examples/always.r`

C Source code guide

The Jikes compiler sources are not included because they are large, and there is nothing particularly enlightening about the modifications I made.

Everything else I did was in Prolog. There are three Prolog source files in the project directory `prolog`: `parser.pl`, `reducej.pl`, and `test.pl`. I used SWI-Prolog 3.2 to write these files.

The parser source file contains code to shift-reduce parse and unparse expressions given arbitrary definitions for `shift` and `reduce`. It also contains code to read in parses and write out examples of the `shift` predicate, and to split the examples into test and training sets.

The `reducej` source file contains the reductions used in a shift-reduce parser for Java expressions. This is the only part of the system that is dependent on Java.

Finally, the `test` source file contains code for reading in the test examples, unparsing them and then parsing them using the learned predicate, and computing the statistics for the results.