

Miscellaneous Parsing

Handout written by Maggie Johnson and revised by Julie Zelenski.

Resolving Ambiguity: A Different Approach

Recall that ambiguity means we have two or more leftmost derivations for the same input string, or equivalently, that we can build more than one parse tree for the same input string. A simple arithmetic expression grammar is a common example:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

Parsing input $id + id * id$ can produce two different parse trees because of ambiguity



Earlier we discussed how to fix the problem by re-writing the grammar to introduce new intermediate non-terminals that enforce the desired precedence. Instead, let's consider what happens if we go ahead and create the LR(0) configurating sets for the ambiguous version of this grammar:

$I_0:$	$E' \rightarrow \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$	$I_5:$	$E \rightarrow E * \bullet E$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$
$I_1:$	$E' \rightarrow E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$	$I_6:$	$E \rightarrow (E \bullet)$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$
$I_2:$	$E \rightarrow (\bullet E)$ $E \rightarrow \bullet E + E$ $E \rightarrow \bullet E * E$ $E \rightarrow \bullet (E)$ $E \rightarrow \bullet id$	$I_7:$	$E \rightarrow E + \bullet E$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$
$I_3:$	$E \rightarrow id \bullet$	$I_8:$	$E \rightarrow E * E \bullet$ $E \rightarrow E \bullet + E$ $E \rightarrow E \bullet * E$

$I_4: \quad E \rightarrow E+\bullet E$
 $E \rightarrow \bullet E + E$
 $E \rightarrow \bullet E * E$
 $E \rightarrow \bullet(E)$
 $E \rightarrow \bullet id$

$I_9: \quad E \rightarrow (E)\bullet$

Let's say we were building an SLR(1) table for this grammar. Look carefully at state 7. In the action table, there are two conflicting entries under the column labeled *: s5 and r1, a shift/reduce conflict. Trace the parse of input $id + id * id$ up to the point where we arrive in state 7:

<u>State stack</u>	<u>Remaining input</u>	
$S_0S_1S_4S_7$	$* id \$$	state 7: next input is *

At this point during the parse, we have the handle $E + E$ on top of the stack, and the lookahead is *. * is in the follow set for E , so we can reduce that $E + E$ to E . But we also could shift the * and keep going. What choice should we make if we want to preserve the usual arithmetic precedence?

What about if we were parsing $id + id + id$? We have a similar shift/reduce in state 7, now on next input +. How do we want to resolve the shift/reduce conflict here? (Because addition is commutative, it actually doesn't much matter, but it will for subtraction!)

<u>State stack</u>	<u>Remaining input</u>	
$S_0S_1S_4S_7$	$+ id \$$	state 7: next input is +

Now consider parsing $id * id + id$. A similar shift/reduce conflict comes up in state 8.

<u>State stack</u>	<u>Remaining input</u>	
$S_0S_1S_5S_8$	$+ id \$$	state 8: next input is +

And what about parsing $id * id * id$?

<u>State stack</u>	<u>Remaining input</u>	
$S_0S_1S_5S_8$	$* id \$$	state 8: next input is *

Instead of rearranging the grammar to add all the different precedence levels, another way of resolving these conflicts to build the precedence rules right into the table. Where there are two or more entries in the table, we pick the one to keep and throw the others out. In the above example, if we are currently in the midst of parsing a lower-precedence operation, we shift the higher precedence operator and keep going. If the next operator is lower-precedence, we reduce. How we break the tie when the two

operators are the same precedence determines the associativity. By choosing to reduce, we enforce left-to-right associativity.

There are a few reasons we might want to resolve the conflicts in the parser instead of re-writing the grammar. As originally written, the grammar reads more naturally, for one. In the parser, we can easily tweak the associativity and precedence of the operators without disturbing the productions or adding extra states to the parser. Also, the parser will not need to spend extra time reducing through the single productions ($T \rightarrow F$, $E \rightarrow T$) introduced in the grammar re-write.

Note that just because there are conflicts in LR table does not imply the grammar is ambiguous—it just means the grammar isn't LR(1) or whatever technique you are using. The reverse, though, is true. No ambiguous grammar is LR(1) and will have conflicts in any type of LR table (or LL for that matter).

Dangling Else Is Back!

Another example of ambiguity in programming language grammars is the famous *dangling else*.

Stmt \rightarrow if Expr then Stmt else Stmt | if Expr then Stmt | Other...

which we rewrite for brevity as:

S \rightarrow iSeS | iS | a

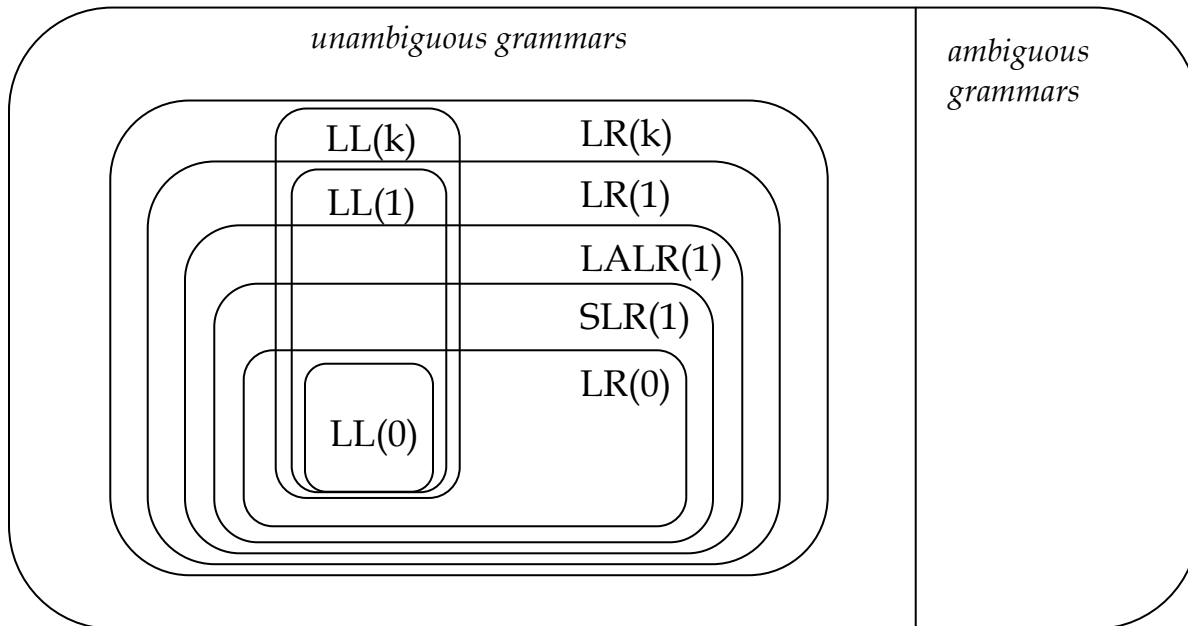
Here are the LR(0) configurating sets. Where is the conflict in the following collection?

I ₀ :	S' \rightarrow •S S \rightarrow •iSeS S \rightarrow •iS S \rightarrow •a	I ₃ :	S \rightarrow a•
I ₁ :	S' \rightarrow S•	I ₄ :	S \rightarrow iS•eS S \rightarrow iS•
I ₂ :	S \rightarrow i•SeS S \rightarrow i•S S \rightarrow •iSeS S \rightarrow •iS S \rightarrow •a	I ₅ :	S \rightarrow iSe•S S \rightarrow •iSeS S \rightarrow •iS S \rightarrow •a
		I ₆ :	S \rightarrow iSeS•

Say we are parsing: if S e S. When we arrive in state 4, we have if S on the stack and the next input is e. Do we shift the e or do we reduce what is on the stack? To follow C and Java's rules, we should want the e to associate it with the nearest if. Which action do we keep to get that behavior?

LL(1) versus LR(k)

A picture is worth a thousand words:



Note this diagram refers to **grammars**, not **languages**, e.g. there may be an equivalent LR(1) grammar that accepts the same language as another non-LR(1) grammar. No ambiguous grammar is LL(1) or LR(1), so we must either re-write the grammar to remove the ambiguity or resolve conflicts in the parser table or implementation.

The hierarchy of LR variants is clear: every LR(0) grammar is SLR(1) and every SLR(1) is LALR(1) which in turn is LR(1). But there are grammars that don't meet the requirements for the weaker forms that can be parsed by the more powerful variations.

We've seen several examples of grammars that are not LL(1) that are LR(1). But every LL(1) grammar is guaranteed to be LR(1). A rigorous proof is fairly straightforward from the definitions of LL(1) and LR(1) grammars. Your intuition should tell you that an LR(1) parser uses more information than the LL(1) parser since it postpones the decision about which production is being expanded until it sees the entire right side rather than attempting to predict after seeing just the first terminal.

LL(1) versus LALR(1)

The two dominant parsing techniques in real compilers are LL(1) and LALR(1). These techniques are the ones to stash away in your brain cells for future reference. Here are some thoughts on how to weigh the two approaches against one another:

Implementation: Because the underlying algorithms are more complicated, most LALR(1) parsers are built using parser generators such as `yacc` and `bison`. LL(1) parsers may be implemented via hand-coded recursive-descent or via LL(1) table-driven predictive parser generators like `LLgen`. There are those who like managing details and writing all the code themselves, no errors result from misunderstanding how the tools work, and so on. But as projects get bigger, the automated tools can be a help, and `yacc/bison` can find ambiguities and conflicts that you might have missed doing the work by hand. The implementation chosen also has an effect on maintenance. Which would you rather do: add new productions into a grammar specification being fed to a generator, add new entries into a table, or write new functions for a recursive-descent parser?

Simplicity: Both techniques have fairly simple drivers. The algorithm underlying LL(1) is simpler, so it's easier to visualize and debug. The details of the LALR(1) configurations can be messy and when trying to debug can be a bit overwhelming.

Generality: All LL(1) grammars are LR(1) and virtually all are also LALR(1), although there are some fringes grammars that can be handled by one technique or the other exclusively. This isn't much of an obstacle in practice since simple grammar transformation and/or parser tricks can usually resolve the problem. As a rule of thumb, LL(1) and LALR(1) grammars can be constructed for any reasonable programming language.

Grammar conditioning: An LL(1) parser has strict rules on the structure of productions, so you will need to massage the grammar into the proper form first. If extensive grammar conditioning is required, you may not even recognize the grammar you started out with. The most troublesome area for programming language grammars is usually the handling of arithmetic expressions. If you can stomach what is needed to transform those to LL(1), you're through the hard part—the rest of the grammar is smoother sailing. LALR(1) parsers are much less restrictive on grammar forms, and thus allow you to express the grammar more naturally and clearly. The LALR(1) grammar will also be smaller than its LL(1) equivalent because LL(1) requires extra nonterminals and productions to factor the common prefixes, rearrange left recursion, and so on.

Error repair: Both LL(1) and LALR(1) parsers possess the *valid prefix* property. What is on the stack will always be a valid prefix of a sentential form. Errors in both types

of parsers can be detected at the earliest possible point without pushing the next input symbol onto the stack. LL(1) parse stacks contain symbols that are predicted but not yet matched. This information can be valuable in determining proper repairs. LALR(1) parse stacks contain information about what has already been seen, but do not have the same information about the right context that is expected. This means deciding possible continuations is somewhat easier in an LL(1) parser.

Table sizes: Both require parse tables that can be big. For LL(1) parsers, the uncompressed table has one row for each non-terminal and one column for each terminal, so the total table size is $|T| \times |N|$. An LALR table has a row for each state and a column for each terminal and each non-terminal, so the total table size is $|S| \times (|N| + |T|)$. The number of states can be exponential in the worst case. (i.e., states form the power set of all productions). So for a pathologically designed grammar, the LALR(1) could be much, much larger. However, for average-case inputs, the LALR(1) table is usually about twice as big as the LL(1). For a language like Pascal, the LL(1) table might have 1500 non-error entries, the LALR(1) table has around 3000. This sort of thing used to be important, but with the capabilities of today's machines, a factor of 2 is not likely to be a significant issue.

Efficiency: Both require a stack of some sort to manage the input. That stack can grow to a maximum depth of n , where n is the number of symbols in the input. If you are using the runtime stack (i.e. function calls) rather than pushing and popping on a data stack, you will probably pay some significant overhead for that convenience (i.e. a recursive-descent parser takes that hit). If both parsers are using the same sort of stack, LL(1) and LALR(1) each examine every non-terminal and terminal when building the parse tree, and so parsing speeds tend to be comparable between the two.

How Do Real Compilers Do Parsing?

As we saw earlier, the introduction of ALGOL-60 and its specification using BNF was highly influential. Soon after, it was realized that BNF and context-free grammars were equivalent (remember Chomsky defined CFG's in the late 1950's). Consequently, there was a lot of theoretical work done on programming languages in the early 1960's.

Parsing methods became much more systematic as the theoretical work progressed. Several general techniques for parsing any CFG were invented in the 1960's. LR grammars and parsers were first introduced by Knuth (1965) who described the construction of LR parsing tables. Korenjak (1969) was the first to show parsers for programming languages could be produced using these techniques. DeRemer (1969, 1971) devised the more practical SLR and LALR techniques. This laid the groundwork for automatic parser generators. YACC was built by S.C. Johnson in the early 1970's. Today, most compilers are created using automatic parser generators.

The idea of LL(1) grammars was introduced by Lewis and Stearns (1968). Predictive parsers were first discussed by Knuth (1971). Recursive-descent and table-driven parsing techniques were shown to be useful for programming languages by Lewis, Rosenkrantz and Stearns (1976). These techniques were widely used in early compilers but nowadays bottom-up seems more dominant.

Early Pascal compilers (1971 on) generated absolute machine code for the CDC 6000 series computers. They were one-pass compilers built around a recursive-descent parser. Niklaus Wirth and his colleagues found it was "relatively easy to mold the language according to the restrictions of the parsing method". This worked for a time, but as they added more features to the language, it became difficult to keep the grammar LL(1), and to not blow out the runtime stack (as programs got longer and had more nested structures). Eventually, they went on to use automatic parser generators.

One important event occurred in the history of Pascal as a language: Kenneth Bowles at UCSD developed the UCSD Pascal compiler for use on both micro and mini-computers. It included a text editor, compiler, assembler and linker all in one. It was distributed free to educational institutions and was a primary reason for Pascal's early adoption as an educational language. The parser in the UCSD version was recursive-descent.

C was designed by Dennis Ritchie. Back in the 1960's, Ritchie was a Harvard physics major. In 1968, he went to work for Bell Labs and was teamed up with Ken Thompson, who was an electrical engineer from Berkeley. Given their assignment to "think about interesting problems in computer science", they got intrigued by operating systems. In the early 1970's, researchers at Bell Labs were working on an OS called Multics which was a multiuser, time-share system. The programmers in the labs loved this system because they were used to doing things the old-fashioned way (giving a batch of cards to an operator and waiting). Consequently everyone was using Multics which unfortunately, cost a lot of money to run, which eventually lead the company to abandon Multics because of its expense. Ritchie and Thompson, however, would just not go back to doing things the old way. They decided to design an operating system just for themselves (and their lab colleagues).

Thompson wrote up a proposal and was flatly turned down (management did not need another Multics). He was not discouraged! He managed to find an old PDP-7, and he and Ritchie wrote the first version of UNIX. In time, they realized they would not get too far running their operating system on an obsolete computer. So, they presented a new proposal to create an editing system for office tasks. It was approved, and the two got a brand-new PDP-11 to work with. By 1971, UNIX was completed and its use inside Bell Labs began to grow.

Thompson and Ritchie wrote the first version in assembly and it only worked on PDP-11's. They realized this had to change if people were really going to use their operating system outside Bell Labs. Back in those days, a lot of work was done in England on the concept of "high- and low-level languages in one". A group working at the University of London and Cambridge designed a language that was high-level enough to not be tied to any particular computer, and low-level enough to allow manipulation of bits. The resulting language was CPL (combined programming language). It was never very popular because it was so large, but a stripped-down version called BCPL (basic CPL) did attract some users. Thompson and Ritchie heard about these developments and took BCPL and transformed it first into B and then into C in collaboration with Brian Kernighan. UNIX was then rewritten in C, and UNIX went on to become an industry standard. C established itself as the ultimate "programmer's language", i.e., a language written by serious programmers for serious programmers.

The earliest C parsers were recursive-descent, but eventually they were written with `yacc`. All these early compilers were two-pass, with an optional third pass. The first pass handled lexical and syntax analysis, the second pass was machine-dependent assembly code generation, and the optional last pass handled code optimization.

Bibliography

- A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.
- Appleby, D. Programming Languages: Paradigm and Practice. New York, NY: McGraw-Hill, 1991.
- DeRemer, F. Practical Translators for LR(k) Languages. Ph.D. dissertation, MIT, 1969.
- DeRemer, F. "Simple LR(k) Grammars," Communications of the ACM, Vol. 14, No. 7, 1971.
- Johnson, S.C., "Yacc - Yet Another Compiler Compiler", Computing Science Technical Report 32, AT&T Bell Labs, 1975.
- Korenjak, A. "A Practical Method for Constructing LR(k) Processors," Communications of the ACM, Vol. 12, No. 11, 1969.
- Knuth, D. "On the Translation of Languages from Left to Right," Information and Control, Vol. 8, No. 6, 1965.
- Knuth, D. "Top-Down Syntax Analysis," Acta Informatica, Vol 1., No. 2, 1971.
- Lewis, P. and Stearns, R. "Syntax-Directed Transduction," Journal of the ACM, Vol. 15, No. 3, 1968.
- Lewis, P. Rosenkrantz, D., and Stearns, R. Compiler Design Theory. Reading, MA: Addison-Wesley, 1976.
- MacLennan, B. Principles of Programming Languages. New York, NY: Holt, Rinehart, Winston, 1987.
- Ritchie, D., and Thompson, K. "The UNIX Time-Sharing System," Communications of the ACM, Vol. 17, No. 7, 1974.
- Wirth, N. "The Design of a Pascal Compiler," Software - Practice and Experience, Vol. 1, No. 4, 1971.