Introduction to Compilers and Language Design

Paperback ISBN: 979-8-655-18026-0
Second edition.

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at `http://compilerbook.org`

Revision Date: August 24, 2023

Chapter 4 – Parsing

4.1 Overview

If scanning is like constructing words out of letters, then parsing is like constructing sentences out of words in a natural language. Of course, not every sequence of words makes a valid sentence: "horse aircraft conjugate" is three valid words, but not a meaningful sentence.

To parse a computer program, we must first describe the form of valid sentences in a language. This formal statement is known as a context free grammar (CFG). Because they allow for recursion, CFGs are more powerful than regular expressions and can express a richer set of structures.

While a plain CFG is relatively easy to write, it does not follow that it is easy to parse. An arbitrary CFG can contain ambiguities and other problems that make it difficult to write an automatic parser. Therefore, we consider two subsets of CFGs known as LL(1) and LR(1) grammars.

LL(1) grammars are CFGs that can be evaluated by considering only the current rule and next token in the input stream. This property makes it easy to write a hand-coded parser known as a recursive descent parser. However, a language (and its grammar) must be carefully designed (and occasionally rewritten) in order to ensure that it is an LL(1) grammar. Not all language structures can be expressed as LL(1) grammars.

LR(1) grammars are more general and more powerful than LL(1). Nearly all useful programming languages can be written in LR(1) form. However, the parsing algorithm for LR(1) grammars is more complex and usually cannot be written by hand. Instead, it is common to use a parser generator that will accept an LR(1) grammar and automatically generate the parsing code.

4.2 Context Free Grammars

Let's begin by defining the parts of a CFG.

A **terminal** is a discrete symbol that can appear in the language, otherwise known as a token from the previous chapter. Examples of terminals are keywords, operators, and identifiers. We will use lower-case letters to represent terminals. At this stage, we only need to consider the kind (e.g. integer literal) and not the value (e.g. `456`) of a terminal.

A **non-terminal** represents a structure that can occur in a language, but is not a literal symbol. Example of non-terminals are declarations, statements, and expressions. We will use upper-case letters to represent non-terminals: P for program, S for statement, E for expression, etc.

A **sentence** is a valid sequence of terminals in a language, while a **sentential form** is a valid sequence of terminals and non-terminals. We will use Greek symbols to represent sentential forms. For example, α, β, and γ represent (possibly) mixed sequences of terminals and non-terminals. We will use a sequence like $Y_1Y_2...Y_n$ to indicate the individual symbols in a sentential form: Y_i may be either a terminal or a non-terminal.

A **context-free grammar (CFG)** is a list of **rules** that formally describe the allowable sentences in a language. The left-hand side of each rule is always a single non-terminal. The right-hand side of a rule is a sentential form that describes an allowable form of that non-terminal. For example, the rule A $\rightarrow$ xXy indicates that the non-terminal A represents a terminal x followed by a non-terminal X and a terminal y. The right hand side of a rule can be ϵ to indicate that the rule produces nothing. The first rule is special: it is the top-level definition of a program and its non-terminal is known as the **start symbol**.

For example, here is a simple CFG that describes expressions involving addition, integers, and identifiers:

Grammar $\mathbf{G_2}$

1. P $\rightarrow$ E
2. E $\rightarrow$ E + E
3. E $\rightarrow$ ident
4. E $\rightarrow$ int

This grammar can be read as follows: (1) A complete program consists of one expression. (2) An expression can be any expression plus any expression. (3) An expression can be an identifier. (4) An expression can be an integer literal.

For brevity, we occasionally condense a set of rules with a common left-hand side by combining all of the right hand sides with a logical-or symbol, like this:

$$\mathsf{E} \rightarrow \mathsf{E} + \mathsf{E} | \mathsf{ident} | \mathsf{int}$$

4.2.1 Deriving Sentences

Each grammar describes a (possibly infinite) set of sentences, which is known as the **language** of the grammar. To prove that a given sentence is a member of that language, we must show that there exists a sequence of rule applications that connects the start symbol with the desired sentence. A sequence of rule applications is known as a **derivation** and a double arrow ($\Rightarrow$) is used to show that one sentential form is equal to another by applying a given rule. For example:

- E $\Rightarrow$ int by applying rule 4 of Grammar G_2.
- E + E $\Rightarrow$ E + ident by applying rule 3 of Grammar G_2.
- P $\Rightarrow$ int + ident by applying all rules of Grammar G_2.

There are two approaches to derivation: top-down and bottom-up.

In **top-down derivation**, we begin with the start symbol, and then apply rules in the CFG to expand non-terminals until reaching the desired sentence. For example, **ident + int + int** is a sentence in this language, and here is one derivation to prove it:

Sentential Form	**Apply Rule**
P	P $\rightarrow$ E
E	E $\rightarrow$ E + E
E + E	E $\rightarrow$ ident
ident + E	E $\rightarrow$ E + E
ident + E + E	E $\rightarrow$ int
ident + int + E	E $\rightarrow$ int
ident + int + int	

In **bottom-up derivation**, we begin with the desired sentence, and then apply the rules backwards until reaching the start symbol. Here is a bottom-up derivation of the same sentence:

Sentential Form	**Apply Rule**
ident + int + int	E $\rightarrow$ int
ident + int + E	E $\rightarrow$ int
ident + E + E	E $\rightarrow$ E+E
ident + E	E $\rightarrow$ ident
E + E	E $\rightarrow$ E+E
E	P $\rightarrow$ E
P	

Be careful to distinguish between a *grammar* (which is a finite set of rules) and a *language* (which is a set of strings generated by a grammar). It is quite possible for two different grammars to generate the same language, in which case we describe them as having **weak equivalence**.

4.2.2 Ambiguous Grammars

An **ambiguous grammar** allows for more than one possible derivation of the same sentence. Our example grammar is ambiguous because there are two possible derivations for any sentence involving two plus signs. The sentence `ident + int + int` can have the two derivations shown in Figure 4.1.

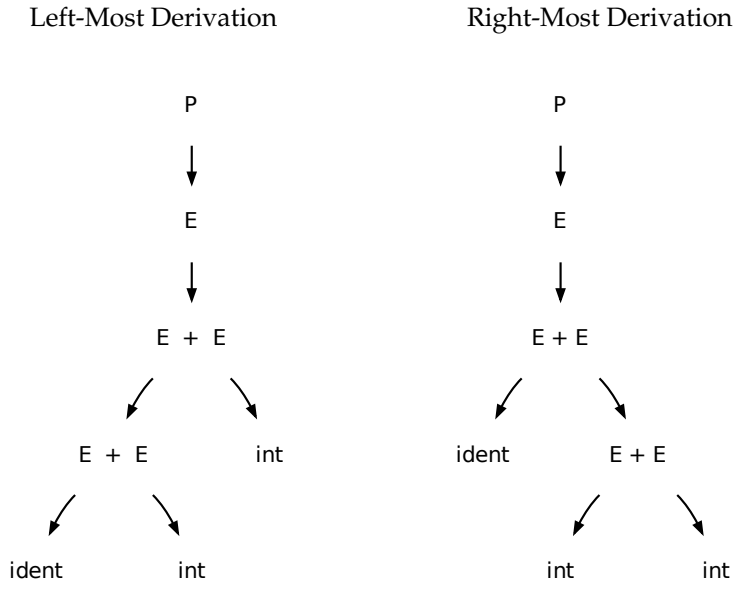

Figure 4.1: Two Derivations of the Same Sentence

Ambiguous grammars present a real problem for parsing (and language design in general) because we do not want a program to have two possible meanings.

Does it matter in this example? It certainly does! In a language like Java, the + operator indicates not only addition between integers, but also concatenation between strings. If the identifier is `hello` and the two integers have the value `5`, then the left-most derivation would concatenate all three together into `hello55`, while the right-most derivation would compute `5+5=10` and concatenate the result into `hello10`.

Fortunately, it is usually possible to re-write a grammar so that it is not ambiguous. In the common case of binary operators, we can require that one side of the expression be an atomic term (T), like this:

Grammar G_3

1. P $\rightarrow$ E
2. E $\rightarrow$ E + T
3. E $\rightarrow$ T
4. T $\rightarrow$ ident
5. T $\rightarrow$ int

With this change, the grammar is no longer ambiguous, because it only allows a left-most derivation. But also note that it *still accepts the same language as Grammar* G_2. That is, any sentence that can be derived by Grammar G_2 can also be derived by Grammar G_3, but there exists only one derivation (and one meaning) per sentence. (Proof is left as an exercise to the reader.)

Now suppose that we would like to add more operators to our grammar. If we simply add more rules of the form E $\rightarrow$ E $*$ T and E $\rightarrow$ E $\div$ T, we would still have an unambiguous grammar, but it would not follow the rules of precedence in algebra: each operator would be applied from left to right.

Instead, the usual approach is to construct a grammar with multiple levels that reflect the intended precedence of operators. For example, we can combine addition and multiplication by expressing them as a sum of terms (T) that consist of multiplied factors (F), like this:

Grammar G_4

1. P $\rightarrow$ E
2. E $\rightarrow$ E + T
3. E $\rightarrow$ T
4. T $\rightarrow$ T * F
5. T $\rightarrow$ F
6. F $\rightarrow$ ident
7. F $\rightarrow$ int

Here is another common example that occurs in most programming languages in some form or another. Suppose that an `if` statement has two variations: an if-then which takes an action when an expression is true, and an if-then-else that takes a different action for the true and false cases. We can express this fragment of the language like this:

Grammar G_5

1. P $\rightarrow$ S
2. S $\rightarrow$ if E then S
3. S $\rightarrow$ if E then S else S
4. S $\rightarrow$ other

Grammar G_5 is ambiguous because it allows for two derivations of this sentence: `if E then if E then other else other`. Do you see the problem? The `else` part could belong to the outer `if` or to the inner `if`. In most programming languages, the `else` is defined as belonging to the inner `if`, but the grammar does not reflect this.

Do this now:
Write out the two possible parse trees for this sentence:
`if E then if E then other else other`.

4.3 LL Grammars

LL(1) grammars are a subset of CFGs that are easy to parse with simple algorithms. A grammar is LL(1) if it can be parsed by considering only one non-terminal and the next token in the input stream.

To ensure that a grammar is LL(1), we must do the following:

- Remove any ambiguity, as shown above.
- Eliminate any left recursion, as shown below.
- Eliminate any common left prefixes, as shown below.

Once we have taken those steps, then we can prove that it is LL(1) by generating the FIRST and FOLLOW sets for the grammar, and using them to create the LL(1) parse table. If the parse table contains no conflicts, then the grammar is clearly LL(1).

4.3.1 Eliminating Left Recursion

LL(1) grammars cannot contain **left recursion**, which is a rule of the form $A \rightarrow A\alpha$ or, more generally, any rule $A \rightarrow B\beta$ such that $B \Rightarrow A\gamma$ by some sequence of derivations. For example, the rule E → E + T is left-recursive because E appears as the first symbol on the right hand side.

You might be tempted to solve the problem by simply re-writing the rule as E → T + E. While that would avoid left recursion, it would not be an equivalent grammar because it would result in a right-associative plus operator. Also, it would introduce the new problem of a common left prefix, discussed below.

Informally, we must re-write the rules so that the (formerly) recursive rule begins with the leading symbols of its alternatives.

Formally, if you have a grammar of the form:

$$\text{A} \rightarrow A\alpha_1|A\alpha_2|...|\beta_1|\beta_2|...$$

Substitute with:

$$\text{A} \rightarrow \beta_1 A'|\beta_2 A'|...$$

$$\text{A}' \rightarrow \alpha_1 A'|\alpha_2 A'|...|\epsilon$$

Applying this rule to grammar Grammar G_3, we can re-write it as:

Grammar G_6

1. P → E
2. E → T E'
3. E' → + T E'
4. E' → ϵ
5. T → ident
6. T → int

While Grammar G_6 is perhaps slightly harder for a person to read, it no longer contains left recursion, and it satisfies all the LL(1) properties. A parser considering an E in rule 2 will immediately consider the T non-terminal, and then look at `ident` or `int` on the input to decide between rule 5 and 6. After considering T, the parser moves on to consider E' and can distinguish between rule 3 and 4 by looking for either a + or any other symbol on the input.

4.3.2 *Eliminating Common Left Prefixes*

A simpler problem to solve is grammars that have multiple rules with the same left hand side and a common prefix of tokens on the right hand side. Informally, we simply look for all common prefixes of a given nonterminal, and replace them with one rule that contains the prefix and another that contains the variants.

Formally, look for rules of this form:

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|...$$

And replace with:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1|\beta_2|...$$

For example, these rules describing an identifier, array reference, and function call all share the same prefix of a single identifier:

Grammar G_7

1. P → E
2. E → id
3. E → id [E]
4. E → id (E)

If a parser is evaluating E and sees an `id` on the input, that information is not sufficient to distinguish between rules 2, 3, and 4. However, the grammar can be salvaged by factoring out the common prefix, like this:

Grammar G_8

1. P → E
2. E → id E'
3. E' → [E]
4. E' → (E)
5. E' → ϵ

In this formulation, the parser always consumes an `id` when evaluating an E. If the next token is `[`, then rule 3 is applied. If the next token is `(`, then rule 4 is applied; otherwise, rule 5 is applied.

4.3.3 First and Follow Sets

In order to construct a complete parser for an LL(1) grammar, we must compute two sets, known as FIRST and FOLLOW. Informally, FIRST(α) indicates the set of terminals (including ϵ) that could potentially appear at the beginning of any derivation of α. FOLLOW(A) indicates the set of terminals (including \$) that could potentially occur after any derivation of the non-terminal A. Given the contents of these sets, an LL(1) parser will always know which rule to pick next.

Here is how to compute FIRST and FOLLOW:

Computing First Sets for a Grammar G

FIRST(α) is the set of terminals that begin all strings given by α, including ϵ if $\alpha \Rightarrow \epsilon$.

For Terminals:
For each terminal $a \in \Sigma$: FIRST(a) = $\{a\}$

For Non-Terminals:
Repeat:
    For each rule $X \rightarrow Y_1Y_2...Y_k$ in a grammar G:
        Add a to FIRST(X)
            if a is in FIRST(Y_1)
            or a is in FIRST(Y_n) and $Y_1...Y_{n-1} \Rightarrow \epsilon$
        If $Y_1...Y_k \Rightarrow \epsilon$ then add ϵ to FIRST(X).
until no more changes occur.

For a Sentential Form α:
For each symbol $Y_1Y_2...Y_k$ in α:
    Add a to FIRST(α)
        if a is in FIRST(Y_1)
        or a is in FIRST(Y_n) and $Y_1...Y_{n-1} \Rightarrow \epsilon$
    If $Y_1...Y_k \Rightarrow \epsilon$ then add ϵ to FIRST(α).

Computing Follow Sets for a Grammar G

FOLLOW(A) is the set of terminals that can come after non-terminal A, including \$ if A occurs at the end of the input.

FOLLOW(S) = {\$} where S is the start symbol.

Repeat:
    If A $\rightarrow \alpha B \beta$ then:
        add FIRST(β) (excepting ϵ) to FOLLOW(B).
    If A $\rightarrow \alpha B$ or FIRST(β) contains ϵ then:
        add FOLLOW(A) to FOLLOW(B).
until no more changes occur.

Here is an example of computing FIRST and FOLLOW for Grammar G_9:

Grammar G_9

1. P $\rightarrow$ E
2. E $\rightarrow$ T E'
3. E' $\rightarrow$ + T E'
4. E' $\rightarrow \epsilon$
5. T $\rightarrow$ F T'
6. T' $\rightarrow$ * F T'
7. T' $\rightarrow \epsilon$
8. F $\rightarrow$ (E)
9. F $\rightarrow$ int

First and Follow for Grammar G_9

	P	E	E′	T	T′	F
FIRST	(int	(int	+ ϵ	(int	* ϵ	(int
FOLLOW	\$	) \$	) \$	+) \$	+) \$	+ *) \$

Once we have cleaned up a grammar to be LL(1) and computed its FIRST and FOLLOW sets, we are ready to write code for a parser. This can be done by hand or with a table-driven approach.

4.3.4 Recursive Descent Parsing

LL(1) grammars are very amenable to writing simple hand-coded parsers. A common approach is a **recursive descent parser** in which there is one simple function for each non-terminal in the grammar. The body of the function follows the right-hand sides of the corresponding rules: non-terminals result in a call to another parse function, while terminals result in considering the next token.

Three helper functions are needed:

- `scan_token()` returns the next token on the input stream.
- `putback_token(t)` puts an unexpected token back on the input stream, where it will be read again by the next call to `scan_token`.
- `expect_token(t)` calls `scan_token` to retrieve the next token. It returns true if the token matches the expected type. If not, it puts the token back on the input stream and returns false.

Figure 4.2 shows how Grammar G_9 could be written as a recursive descent parser. Note that the parser has one function for each non-terminal: `parse_P`, `parse_E`, etc. Each function returns true (1) if the input matches the grammar, or false (0) otherwise.

Two special cases should be considered. First, if a rule X cannot produce ϵ and we encounter a token not in FIRST(X), then we have definitely encountered a parsing error, and we should display a message and return failure. Second, if a rule X *could* produce ϵ and we encounter a token not in FIRST(X), then we accept the rule X $\rightarrow \epsilon$ put the token back on the input, and return success. Another rule will expect to consume that token.

There is also the question of what the parser should actually *do* after matching some element of the grammar. In our simple example, the parser simply returns true on a match, and serves only to verify that the input program matches the grammar. If we wished to actually evaluate the expression, each `parse_X` function could compute the result and return it as a `double`. This would effectively give us a simple interpreter for this language. Another approach is for each `parse_X` function to return a data structure representing that node of the parse tree. As each node is parsed, the result is assembled into an abstract syntax tree, with the root returned by `parse_P`.

```
int parse_P() {
    return parse_E() && expect_token(TOKEN_EOF);
}

int parse_E() {
    return parse_T() && parse_E_prime();
}

int parse_E_prime() {
    token_t t = scan_token();
    if(t==TOKEN_PLUS) {
        return parse_T() && parse_E_prime();
    } else {
        putback_token(t);
        return 1;
    }
}

int parse_T() {
    return parse_F() && parse_T_prime();
}

int parse_T_prime() {
    token_t t = scan_token();
    if(t==TOKEN_MULTIPLY) {
        return parse_F() && parse_T_prime();
    } else {
        putback_token(t);
        return 1;
    }
}

int parse_F() {
    token_t t = scan_token();
    if(t==TOKEN_LPAREN) {
        return parse_E() && expect_token(TOKEN_RPAREN);
    } else if(t==TOKEN_INT) {
        return 1;
    } else {
        printf("parse error: unexpected token %s\n",
               token_string(t));
        return 0;
    }
}
```

Figure 4.2: A Recursive-Descent Parser

4.3.5 Table Driven Parsing

An LL(1) grammar can also be parsed using generalized table driven code. A table-driven parser requires a grammar, a parse table, and a stack to represent the current set of non-terminals.

The **LL(1) parse table** is used to determine which rule should be applied for any combination of non-terminal on the stack and next token on the input stream. (By definition, an LL(1) grammar has exactly one rule to be applied for each combination.) To create a parse table, we use the FIRST and FOLLOW sets like this:

LL(1) Parse Table Construction.
Given a grammar G and alphabet Σ, create a parse table $T[A, a]$ that selects a rule for each combination of non-terminal $A \in G$ and terminal $a \in \Sigma$.

For each rule $A \to \alpha$ in G:
    For each terminal a (excepting ϵ) in FIRST(α):
        Add $A \to \alpha$ to $T[A, a]$.
    If ϵ is in FIRST(α):
        For each terminal b (including \$) in FOLLOW(A):
            Add A $\to \alpha$ to $T[A, b]$.

For example, here is the parse table for Grammar G_9. Notice that the entries for P, E, T, and F are straightforward: each can only start with `int` or `(`, and so these tokens cause the rules to descend toward F and a choice between rule 8 ($F \to int$) and rule 9 ($F \to (E)$). The entry for E' is a little more complicated: a + token results in applying $E' \to +TE'$, while `)` or `$` indicates $E' \to \epsilon$.

Parse Table for Grammar G_9:

	int	**+**	*****	**(**	**)**	**$**
P	1			1		
E	2			2		
E′		3			4	4
T	5			5		
T′		7	6		7	7
F	9			8		

Now we have all the pieces necessary to operate the parser. Informally, the idea is to keep a stack that tracks the current state of the parser. In each step, we consider the top element of the stack and the next token on the input. If they match, then pop the stack, accept the token, and continue. If not, then consult the parse table for the next rule to apply. If we can continue until the end-of-file symbol is matched, then the parse succeeds.

LL(1) Table Parsing Algorithm.
Given a grammar G with start symbol P and parse table T,
parse a sequence of tokens and determine whether they satisfy G.

Create a stack S.
Push \$ and P onto S.
Let c be the first token on the input.

While S is not empty:
    Let X be the top element of the stack.
    If X matches c:
        Remove X from the stack.
        Advance c to the next token and repeat.
    If X is any other terminal, stop with an error.
    If $T[X, c]$ indicates rule $X \to \alpha$:
        Remove X from the stack.
        Push symbols α on to the stack and repeat.
    If $T[X, c]$ indicates an error state, stop with an error.

Here is an example of the algorithm applied to the sentence `int * int`:

Stack	Input	Action
P \$	`int * int $`	apply 1: P ⇒ E
E \$	`int * int $`	apply 2: E ⇒ T E'
T E′ \$	`int * int $`	apply 5: T ⇒ F T'
F T′ E′ \$	`int * int $`	apply 9: F ⇒ int
int T′ E′ \$	`int * int $`	match `int`
T′ E′ \$	`* int $`	apply 6: T' ⇒ * F T'
* F T′ E′ \$	`* int $`	match `*`
F T′ E′ \$	`int $`	apply 9: F ⇒ int
int T′ E′ \$	`int $`	match `int`
T′ E′ \$	`$`	apply 7: T' ⇒ ϵ
E′ \$	`$`	apply 4: E' ⇒ ϵ
\$	`$`	match `$`

4.4 LR Grammars

While LL(1) grammars and top-down parsing techniques are easy to work with, they are not able to represent all of the structures found in many programming languages. For more general-purpose programming languages, we must use an LR(1) grammar and associated bottom-up parsing techniques.

LR(1) is the set of grammars that can be parsed via shift-reduce techniques with a single token of lookahead. LR(1) is a super-set of LL(1) and can accommodate left recursion and common left prefixes which are not permitted in LL(1). This enables us to express many programming constructs in a more natural way. (An LR(1) grammar must still be non-ambiguous, and it cannot have shift-reduce or reduce-reduce conflicts, which we will explain below.)

For example, Grammar G_{10} is an LR(1) grammar:

Grammar G_{10}

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

We need to know the FIRST and FOLLOW sets of LR(1) grammars as well, so take a moment now and work out the sets for Grammar G_{10}, using the same technique from section 4.3.3.

	P	**E**	**T**
FIRST			
FOLLOW			

4.4.1 Shift-Reduce Parsing

LR(1) grammars must be parsed using the **shift-reduce** parsing technique. This is a bottom-up parsing strategy that begins with the tokens and looks for rules that can be applied to reduce sentential forms into non-terminals. If there is a sequence of reductions that leads to the start symbol, then the parse is successful.

A **shift** action consumes one token from the input stream and pushes it onto the stack. A **reduce** action applies one rule of the form A $\rightarrow \alpha$ from the grammar, replacing the sentential form α on the stack with the non-terminal A. For example, here is a shift-reduce parse of the sentence `id(id+id)` using Grammar G_{10}:

Stack	**Input**	**Action**
	id (id + id) $	shift
id	(id + id) $	shift
id (	id + id) $	shift
id (id	+ id) $	reduce T $\rightarrow$ id
id (T	+ id) $	reduce E $\rightarrow$ T
id (E	+ id) $	shift
id (E +	id) $	shift
id (E + id	) $	reduce T $\rightarrow$ id
id (E + T	) $	reduce E $\rightarrow$ E + T
id (E	) $	shift
id (E)	$	reduce T $\rightarrow$ id(E)
T	$	reduce E $\rightarrow$ T
E	$	reduce P $\rightarrow$ E
P	$	accept

While this example shows that there exists a derivation for the sentence, it does not explain how each action was chosen at each step. For example, in the second step, we might have chosen to reduce `id` to T instead of shifting a left parenthesis. This would have been a bad choice, because there is no rule that begins with `T(`, but that was not immediately obvious without attempting to proceed further. To make these decisions, we must analyze LR(1) grammars in more detail.

4.4.2 The LR(0) Automaton

An **LR(0) automaton** represents all the possible rules that are currently under consideration by a shift-reduce parser. (The LR(0) automaton is also variously known as the **canonical collection** or the **compact finite state machine** of the grammar.) Figure 4.6 shows a complete automaton for Grammar G_{10}. Each box represents a state in the machine, connected by transitions for both terminals and non-terminals in the grammar.

Each state in the automaton consists of multiple **items**, which are rules augmented by a **dot** (.) that indicates the parser's current position in that rule. For example, the configuration E $\rightarrow$ E . + T indicates that `E` is currently on the stack, and `+ T` is a possible next sequence of tokens.

The automaton is constructed as follows. State 0 is created by taking the production for the start symbol (P $\rightarrow$ E) and adding a dot at the beginning of the right hand side. This indicates that we expect to see a complete program, but have not yet consumed any symbols. This is known as the **kernel** of the state.

Kernel of State 0

P $\rightarrow$. E

Then, we compute the **closure** of the state as follows. For each item in the state with a non-terminal X immediately to the right of the dot, we add all rules in the grammar that have X as the left hand side. The newly added items have a dot at the beginning of the right hand side.

P $\rightarrow$. E
E $\rightarrow$. E + T
E $\rightarrow$. T

The procedure continues until no new items can be added:

Closure of State 0

P $\rightarrow$. E
E $\rightarrow$. E + T
E $\rightarrow$. T
T $\rightarrow$. id (E)
T $\rightarrow$. id

You can think of the state this way: It describes the initial state of the parser as expecting a complete program in the form of an E. However, an E is known to begin with an E or a T, and a T must begin with an id. All of those symbols could represent the beginning of the program.

From this state, all of the symbols (terminals and non-terminals both) to the right of the dot are possible outgoing transitions. If the automaton takes that transition, it moves to a new state containing the matching items, with the dot moved one position to the right. The closure of the new state is computed, possibly adding new rules as described above.

For example, from state zero, E, T, and id are the possible transitions, because each appears to the right of the dot in some rule. Here are the states for each of those transitions:

Transition on E:

P $\rightarrow$ E .
E $\rightarrow$ E . + T

Transition on T:

E $\rightarrow$ T .

Transition on id:

T $\rightarrow$ id . (E)
T $\rightarrow$ id .

Figure 4.3 gives the complete LR(0) automaton for Grammar G_{10}. Take a moment now to trace over the table and be sure that you understand how it is constructed.

No, really. Stop now and study the figure carefully before continuing.

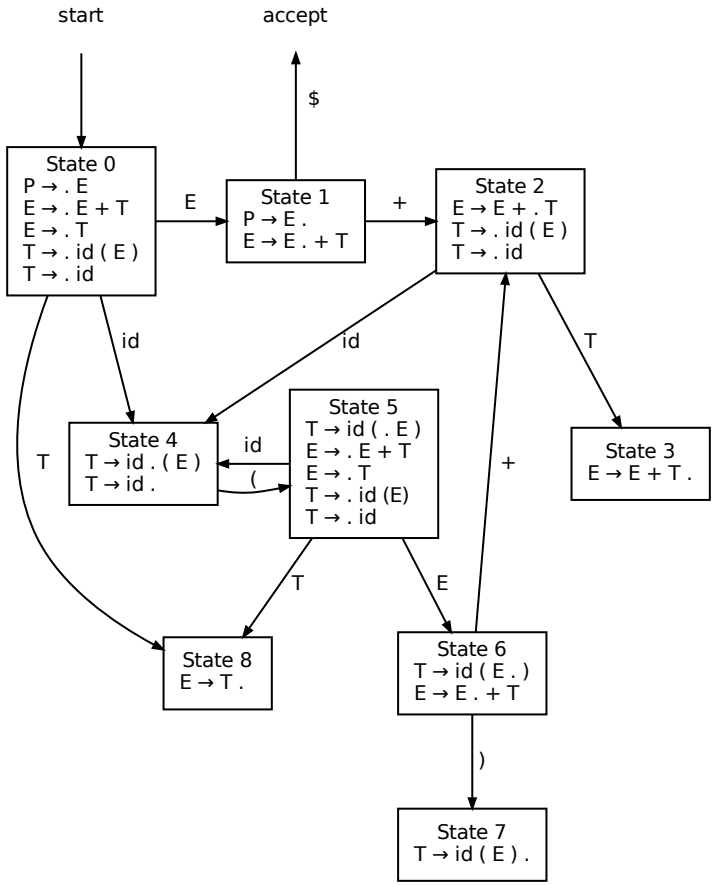

Figure 4.3: LR(0) Automaton for Grammar G_{10}

The LR(0) automaton tells us the choices available at any step of bottom up parsing. When we reach a state containing an item with a dot at the end of the rule, that indicates a possible reduction. A transition on a terminal that moves the dot one position to the right indicates a possible shift. While the LR(0) automaton tells us the available actions at each step, it does not always tell us *which* action to take.[1]

Two types of conflicts can appear in an LR grammar:

A **shift-reduce conflict** indicates a choice between a shift action and a reduce action. For example, state 4 offers a choice between shifting a left parenthesis and reducing by rule five:

Shift-Reduce Conflict:

T → id . (E)
T → id .

A **reduce-reduce conflict** indicates that two distinct rules have been completely matched, and either one could apply. While Grammar G_{10} does not contain any reduce-reduce conflicts, they commonly occur when a syntactic structure occurs at multiple layers in a grammar. For example, it is often the case that a function invocation can be a statement by itself or an element within an expression. The automaton for such a grammar would contain a state like this:

Reduce-Reduce Conflict:

S → id (E) .
E → id (E) .

The LR(0) automaton forms the basis of LR parsing, by telling us which actions are available in each state. But, it does not tell us *which* action to take or how to resolve shift-reduce and reduce-reduce conflicts. To do that, we must take into account some additional information.

[1] The 0 in LR(0) indicates that it uses zero lookahead tokens, which is a way of saying that it does not consider the next token before making a reduction. While it is possible to write out a grammar that is strictly LR(0), such a grammar has very limited utility.

4.4.3 SLR Parsing

Simple LR (SLR) parsing is a basic form of LR parsing in which we use FOLLOW sets to resolve conflicts in the LR(0) automaton. In short, we take the reduction A $\rightarrow \alpha$ only when the next token on the input is in FOLLOW(A). If a grammar can be parsed by this technique, we say it is an **SLR grammar**, which is a subset of LR(1) grammars.

For example, the shift-reduce conflict in state 4 of Figure 4.6 is resolved by consulting FOLLOW(T). If the next token is $+$, $)$ or \$, then we reduce by rule T $\rightarrow$ id. If the next token is $($, then we shift to state 5. If neither of those is true, then the input is invalid, and we emit a parse error.

These decisions are encoded in the **SLR parse tables** which are known historically as GOTO and ACTION. The tables are created as follows:

SLR Parse Table Creation.

Given a grammar G and corresponding LR(0) automaton,
create tables ACTION[s, a] and GOTO[s, A] for all states s,
terminals a, and non-terminals A in G.

For each state s:
  For each item like A $\rightarrow \alpha$. a β
    ACTION[s, a] = **shift** to state t according to the LR(0) automaton.
  For each item like A $\rightarrow \alpha$. B β
    GOTO[s, B] = **goto** state t according to the LR(0) automaton.
  For each item like A $\rightarrow \alpha$.
    For each terminal a in FOLLOW(A):
      ACTION[s, a] = **reduce** by rule A $\rightarrow \alpha$

All remaining states are considered error states.

Naturally, each state in the table can be occupied by only one action. If following the procedure results in a table with more than one state in a given entry, then you can conclude that the grammar is not SLR. (It might still be LR(1) – more on that below.)

Here is the SLR parse table for Grammar G_{10}. Note carefully the states 1 and 4 where there is a choice between shifting and reducing. In state 1, a lookahead of $+$ causes a shift, while a lookahead of \$ results in a reduction P $\rightarrow$ E because \$ is the only member of FOLLOW(P).

State	GOTO		ACTION				
	E	T	id	(	)	+	$
0	G1	G8	S4				
1						S2	R1
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

Figure 4.4: SLR Parse Table for Grammar G_{10}

Now we are ready to parse an input by following the SLR parsing algorithm. The parse requires maintaining a stack of states in the LR(0) automaton, initially containing the start state S_0. Then, we examine the top of the stack and the lookahead token, and take the action indicated by the SLR parse table. On a shift, we consume the token and push the indicated state on the stack. On a reduce by $\mathrm{A} \to \beta$, we pop states from the stack corresponding to each of the symbols in β, then take the additional step of moving to state GOTO[t, A]. This process continues until we either succeed by reducing to the start symbol, or fail by encountering an error state.

SLR Parsing Algorithm.

Let S be a stack of LR(0) automaton states. Push S_0 onto S.
Let a be the first input token.

Loop:
- Let s be the top of the stack.
- If ACTION[s, a] is **accept**:
  - Parse complete.
- Else if ACTION[s, a] is **shift** t:
  - Push state t on the stack.
  - Let a be the next input token.
- Else if ACTION[s, a] is **reduce** $\mathrm{A} \to \beta$:
  - Pop states corresponding to β from the stack.
  - Let t be the top of stack.
  - Push GOTO[t, A] onto the stack.
- Otherwise:
  - Halt with a parse error.

Here is an example of applying the SLR parsing algorithm to the program `id ( id + id )`. The first three steps are easy: a shift is performed for each of the first three tokens `id ( id`. The fourth step reduces T $\rightarrow$ id. This causes state 4 (corresponding to the right hand side `id`) to be popped from the stack. State 5 is now at the top of the stack, and GOTO$[5, T] = 8$, so state 8 is pushed, resulting in a stack of `0 4 5 8`.

Stack	**Symbols**	**Input**	**Action**
0		id (id + id) $	shift 4
0 4	id	(id + id) $	shift 5
0 4 5	id (	id + id) $	shift 4
0 4 5 4	id (id	+ id) $	reduce T $\rightarrow$ id
0 4 5 8	id (T	+ id) $	reduce E $\rightarrow$ T
0 4 5 6	id (E	+ id) $	shift 2
0 4 5 6 2	id (E +	id) $	shift 4
0 4 5 6 2 4	id (E + id	) $	reduce T $\rightarrow$ id
0 4 5 6 2 3	id (E + T	) $	reduce E $\rightarrow$ E + T
0 4 5 6	id (E	) $	shift 7
0 4 5 6 7	id (E)	$	reduce T $\rightarrow$ id(E)
0 8	T	$	reduce E $\rightarrow$ T
0 1	E	$	accept

(Although we show two columns for "Stack" and "Symbols", they are simply two representations of the same information. The stack state `0 4 5 8` represents the parse state of `id ( T` and vice versa.)

It should now be clear that SLR parsing has the same algorithmic complexity as LL(1) parsing. Both techniques require a parsing table and a stack. At each step in both algorithms, it is necessary to only consider the current state and the next token on the input. The distinction is that each LL(1) parsing state considers only a single non-terminal, while each LR(1) parsing state considers a large number of possible configurations.

SLR parsing is a good starting point for understanding the general principles of bottom up parsing. However, SLR is a subset of LR(1), and not all LR(1) grammars are SLR. For example, consider Grammar G_{11} which allows for a statement to be a variable assignment, or an identifier by itself. Note that FOLLOW(S) = {\$} and FOLLOW(V) = {=]\$}.

Grammar G_{11}

1. S → V = E
2. S → id
3. V → id
4. V → id [E]
5. E → V

We need only build part of the LR(0) automaton to see the problem:

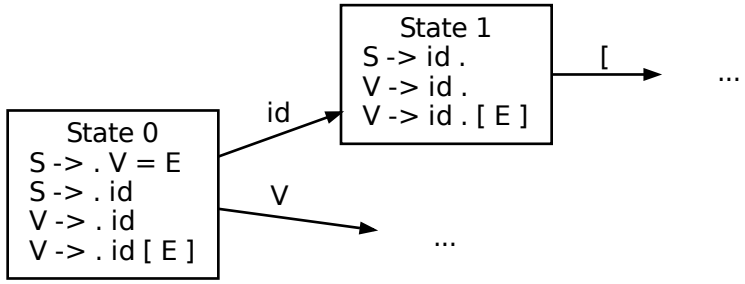

Figure 4.5: Part of LR(0) Automaton for Grammar G_{11}

In state 1, we can reduce by S → id or V → id. However, both FOLLOW(S) and FOLLOW(V) contain \$, so we cannot decide which to take when the next token is end-of-file. Even using the FOLLOW sets, there is still a reduce-reduce conflict. Therefore, Grammar G_{11} is not an SLR grammar.

But, if we look more closely at the possible sentences allowed by the grammar, the distinction between the two becomes clear. Rule S → id would only be applied in the case where the complete sentence is `id $`. If any other token follows a leading `id`, then V → id applies. So, the grammar is not inherently ambiguous: we just need a more powerful parsing algorithm.

4.4.4 LR(1) Parsing

The LR(0) automaton is limited in power, because it does not track what tokens can actually follow a production. SLR parsing accommodates for this weakness by using FOLLOW sets to decide when to reduce. As shown above, this is not sufficiently discriminating to parse all LR(1) grammars.

Now we give the complete or "canonical" form of LR(1) parsing, which depends upon the LR(1) automaton. The LR(1) automaton is like the LR(0) automaton, except that each item is annotated with the set of tokens that could potentially follow it, given the current parsing state. This set is known as the **lookahead** of the item. The lookahead is always a subset of the FOLLOW of the relevant non-terminal.

The lookahead of the kernel of the start state is always $\{\$\}$. When computing the closure of a state, we consider two cases:

- For an item like A $\rightarrow \alpha.B$ with a lookahead of $\{L\}$, add new rules like B $\rightarrow .\gamma$ with a lookahead of $\{L\}$.
- For an item like A $\rightarrow \alpha.B\beta$, with a lookahead of $\{L\}$, add new rules like B $\rightarrow .\gamma$ with a lookahead as follows:
  - If β **cannot** produce ϵ, the lookahead is FIRST(β).
  - If β **can** produce ϵ, the lookahead is FIRST(β) $\cup\{L\}$.

As before, the rules like B $\rightarrow .\gamma$ to be added to the state correspond to all of the rules in the grammar with B on the left hand side.

Here is an example for Grammar G_{11}. The kernel of the start state consists of the start symbol with a lookahead of \$:

Kernel of State 0

S $\rightarrow$. V = E	{\$}
S $\rightarrow$. id	{\$}

The closure of the start state is computed by adding the rules for V with a lookahead of $=$, because $=$ follows V in rule 1:

Closure of State 0

S $\rightarrow$. V = E	{\$}
S $\rightarrow$. id	{\$}
V $\rightarrow$. id	{=}
V $\rightarrow$. id [E]	{=}

Now suppose that we construct state 1 via a transition on the terminal `id`. The lookahead for each item is propagated to the new state:

Closure of State 1

S → id .	{$}
V → id .	{=}
V → id . [E]	{=}

Now you can see how the lookahead solves the reduce-reduce conflict. When the next token on the input is $, we can only reduce by S → id. When the next token is =, we can only reduce by V → id. By tracking lookaheads in a more fine-grained manner than SLR, we are able to parse arbitrary LR(1) grammars.

Figure 4.6 gives the complete LR(1) automaton for Grammar G_{10}. Take a moment now to trace over the table and be sure that you understand how it is constructed.

One aspect of state zero is worth clarifying. When constructing the closure of a state, we must consider *all* rules in the grammar, including the rule corresponding to the item under closure. The item E → . E + T is initially added with a lookahead of {$}. Then, evaluating that item, we add all rules that have *E* on the left hand side, adding a lookahead of {+}. So, we add E → . E + T *again*, this time with a lookahead of {+}, resulting in a single item with a lookahead set of {$, +}

Once again: Stop now and study the figure carefully before continuing.

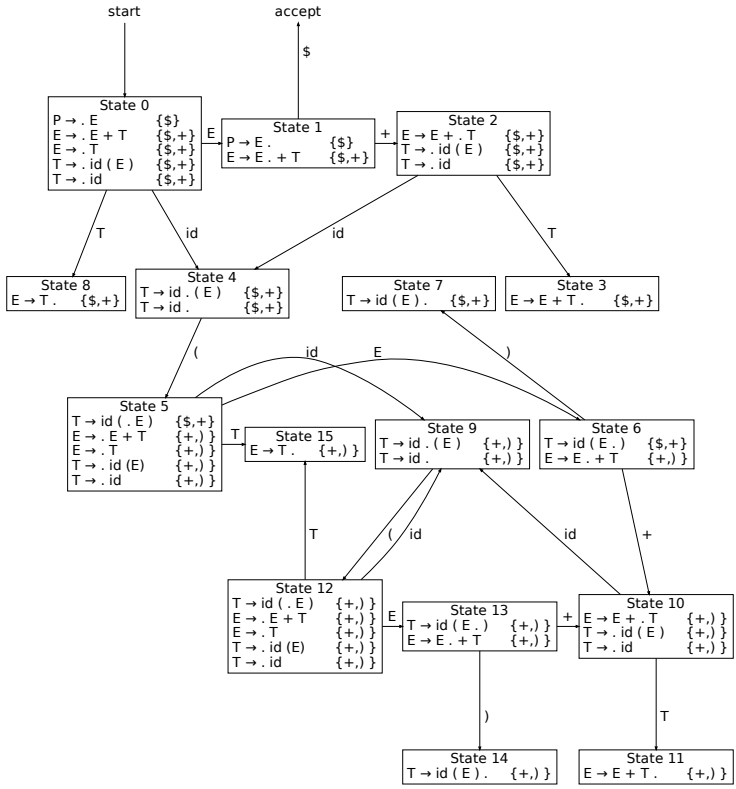

Figure 4.6: LR(1) Automaton for Grammar G_{10}

4.4.5 LALR Parsing

The main downside to LR(1) parsing is that the LR(1) automaton can be *much* larger than the LR(0) automaton. Any two states that have the same items but differ in lookahead sets for *any* items are considered to be different states. The result is enormous parse tables that consume large amounts of memory and slow down the parsing algorithm.

Lookahead LR (LALR) parsing is the practical answer to this problem. To construct an LALR parser, we first create the LR(1) automaton, and then merge states that have the same core. The **core** of a state is simply the body of an item, ignoring the lookahead. When several LR(1) items are merged into one LALR item, the LALR lookahead is the union of the lookaheads of the LR(1) items.

For example, these two LR(1) states:

E → . E + T {$+}	E → . E + T {)+}
E → . T {$+}	E → . T {)+}

Would be merged into this single LALR state:

E → . E + T {$)+}
E → . T {$)+}

The resulting LALR automaton has the same number of states as the LR(0) automaton, but has more precise lookahead information available for each item. While this may seem a minor distinction, experience has shown this simple improvement to be highly effective at obtaining the efficiency of SLR parsing while accommodating a large number of practical grammars.

4.5 Grammar Classes Revisited

Now that you have some experience working with different kinds of grammars, let's step back and review how they relate to each other.

$$LL(1) \subset SLR \subset LALR \subset LR(1) \subset CFG \tag{4.1}$$

CFG: A context-free grammar is any grammar whose rules have the form A $\rightarrow \alpha$. To parse any CFG, we require a finite automaton (a parse table) and a stack to keep track of the parse state. An arbitrary CFG can be ambiguous. An ambiguous CFG will result in a non-deterministic finite

automaton, which is not practical to use. Instead, it is more desirable to re-write the grammar to fit a more restricted class.

LR(k): An LR(k) parser performs a bottom-up **L**eft to right scan of the input and provides a **R**ight-most parse, deciding what rule to apply next by examining the next k tokens on the input. A canonical LR(1) parser requires a very large finite automaton, because the possible lookaheads are encoded into the states. While strictly a subset of CFGs, nearly all real-world language constructs can be expressed adequately in LR(1).

LALR: A Lookahead-LR parser is created by first constructing a canonical LR(1) parser, and then merging all itemsets that have the same core. This yields a much smaller finite automaton, while retaining some detailed lookahead information. While less powerful than canonical LR(1) in theory, LALR is usually sufficient to express real-world languages.

SLR: A Simple-LR parser approximates an LR(1) parser by constructing the LR(0) state machine, and then relying on the FIRST and FOLLOW sets to select which rule to apply. SLR is simple and compact, but there are easy-to-find examples of common constructs that it cannot parse.

LL(k): An LL(k) parser performs a top-down **L**eft to right scan of the input and provides a **L**eft-most parse, deciding what rule to apply next by examining the next k tokens on the input. LL(1) parsers are simple and widely used because they require a table that is only $O(nt)$ where t is the number of tokens, and n is the number of non-terminals. $LL(k)$ parsers are less practical for $k > 1$ because the size of the parse table is $O(nt^k)$ in the worst case.[2] However, they often require that a grammar be rewritten to be more amenable to the parser, and are not able to express all common language structures.

4.6 The Chomsky Hierarchy

Finally, this brings us to a fundamental result in theoretical computer science, known as the **Chomsky hierarchy** [1], named after noted linguist Noam Chomsky. The hierarchy describes four categories of languages (and corresponding grammars) and relates them to the abstract computing machinery necessary to recognize such a language.

Regular languages are those described by regular expressions, as you learned back in Chapter 3. Every regular expression corresponds to a finite automaton that can be used to identify all words in the corresponding language. As you know, a finite automaton can be implemented with the very simple mechanism of a table and a single integer to represent the current state. So, a scanner for a regular language is very easy to implement efficiently.

Context free languages are those described by context free grammars where each rule is of the form A $\rightarrow \gamma$, with a single non-terminal on the

[2] For example, an LL(1) parser would require a row for terminals {`a`, `b`, `c`, ...}, while an LL(2) parser would require a row for pairs {`aa`, `ab`, `ac`, ...}.

Language Class	Machine Required
Regular Languages	Finite Automata
Context Free Languages	Pushdown Automata
Context Sensitive Languages	Linear Bounded Automata
Recursively Enumerable Languages	Turing Machine

Figure 4.7: The Chomsky Hierarchy

left hand side, and a mix of terminals and non-terminals on the right hand side. We call these "context free" because the meaning of a non-terminal is the same in all places where it appears. As you have learned in this chapter, a CFG requires a pushdown automaton, which is achieved by coupling a finite automaton with a stack. If the grammar is ambiguous, the automaton will be non-deterministic and therefore impractical. In practice, we restrict ourselves to using subsets of CFGs (like LL(1) and LR(1) that are non-ambiguous and result in a deterministic automaton that completes in bounded time.

Context sensitive languages are those described by context sensitive grammars where each rule can be of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$. We call these "context sensitive" because the interpretation of a non-terminal is controlled by context in which it appears. Context sensitive languages require a non-deterministic linear bounded automaton, which is bounded in memory consumption, but not in execution time. Context sensitive languages are not very practical for computer languages.

Recursively enumerable languages are the least restrictive set of languages, described by rules of the form $\alpha \rightarrow \beta$ where α and β can be any combination of terminals and non-terminals. These languages can only be recognized by a full Turing machine, and are the least practical of all.

The Chomsky Hierarchy is a specific example of a more general principle for the design of languages and compilers:

The least powerful language gives the strongest guarantees.

That is to say, if we have a problem to be solved, it should be attacked using the least expressive tool that is capable of addressing the problem. If we *can* solve a given problem by employing REs instead of CFGs, then we *should* use REs, because they consume less state, have simpler machinery, and present fewer roadblocks to a solution.

The same advice applies more broadly: assembly language is the most powerful language available in our toolbox and is capable of expressing any program that the computer is capable of executing. However, assembly language is also the most difficult to use because it gives none of the guarantees found in higher level languages. Higher level languages are *less* powerful than assembly language, and this is what makes them more predictable, reliable, and congenial to use.

4.7 Exercises

1. Write out an improvement to Grammar G_5 that does not have the dangling-else problem. Hint: Prevent the inner S from containing an `if` without an `else`.
2. Write a grammar for an interesting subset of sentences in English, including nouns, verbs, adjectives, adverbs, conjunctions, subordinate phrases, and so forth. (Include just a few terminals in each category to give the idea.) Is the grammar LL(1), LR(1), or ambiguous? Explain why.
3. Consider the following grammar:

Grammar G_{12}

1.	P → S
2.	P → S P
3.	S → if E then S
4.	S → if E then S else S
5.	S → while E S
6.	S → begin P end
7.	S → print E
8.	S → E
9.	E → id
10.	E → integer
11.	E → E + E

   (a) Point out all aspects of Grammar G_{12} which are not LL(1).
   (b) Write a new grammar which accepts the same language, but avoids left recursion and common left prefixes.
   (c) Write the FIRST and FOLLOW sets for the new grammar.
   (d) Write out the LL(1) parse table for the new grammar.
   (e) Is the new grammar an LL(1) grammar? Explain your answer carefully.

4. Consider the following grammar:

Grammar G_{13}

1.	S → id = E
2.	E → E + P
3.	E → P
4.	P → id
5.	P → (E)
6.	P → id(E)

(a) Draw the LR(0) automaton for Grammar G_{13}.

(b) Write out the complete SLR parsing table for Grammar G_{13}.

(c) Is this grammar LL(1)? Explain why.

(d) Is this grammar SLR? Explain why.

5. Consider Grammar G_{11}, shown earlier.

   (a) Write out the complete LR(1) automaton for Grammar G_{11}.

   (b) Compact the LR(1) automaton into the LALR automaton for Grammar G_{11}.

6. Write a context free grammar that describes formal regular expressions. Start by writing out the simplest (possibly ambiguous) grammar you can think of, based on the inductive definition in Chapter 3. Then, rewrite the grammar into an equivalent LL(1) grammar.

7. (a) Write a grammar for the JSON data representation language.

   (b) Write the FIRST and FOLLOW sets for your grammar.

   (c) Is your grammar LL(1), SLR, or LR(1), or neither? If necessary, re-write it until it is in the simplest grammar class possible.

   (d) Write out the appropriate parse table for your grammar.

8. Write a working hand-coded parser for JSON expressions, making use of the JSON scanner constructed in the previous chapter.

9. Create a hand-coded scanner and a recursive descent parser that can evaluate first order logic expressions entered on the console. Include boolean values (T/F) and the operators `&` (and), `|` (or), `!` (not), `->` (implication), and `()` (grouping).

   For example, these expressions should evaluate to true:

   ```
   T
   T & T | F
   ( F -> F ) -> T
   ```

   And these expressions should evaluate to false:

   ```
   F
   ! ( T | F )
   ( T -> F ) & T
   ```

10. Write a hand-coded parser that reads in regular expressions and outputs the corresponding NFA, using the Graphviz [2] DOT language.

11. Write a parser-construction tool that reads in an LL(1) grammar and produces working code for a table-driven parser as output.

4.8 Further Reading

1. N. Chomsky, "On certain formal properties of grammars", Information and Control, volume 2, number 2, 1959.
   http://dx.doi.org/10.1016/S0019-9958(59)90362-6

2. J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, "Graphviz – Open Source Graph Drawing Tools", International Symposium on Graph Drawing, 2001.
   http://www.graphviz.org

3. J. Earley, "An Efficient Context Free Parsing Algorithm", Communications of the ACM, volume 13, issue 2, 1970.
   https://doi.org/10.1145/362007.362035

4. M. Tomita (editor), "Generalized LR Parsing", Springer, 1991.