# EDAN65: Compilers - Reference Sheet

Karl Hallsby

Last Edited: September 28, 2019

## Contents

# 1 Introduction

There are numerous steps in the compilation process of a standard program. Each phase converts the program from one representation to another.

1. Lexical Analysis/Scanning
2. Syntactic Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Optimization
6. Target Code Generation

**Defn 1** (Syntactic Analysis (Parsing))**.** *Syntactic Analysis* or *Parsing* is the process where tokens are input and an AST (Abstract Syntax Tree) is created. This AST is generated based on the input source code and the Lexical Analysis (Scanning) that occurs.

This code would generate an error during the Syntactic Analysis (Parsing).

```
int r( {
  return 3;
}
```

This wouldn't fail during Lexical Analysis (Scanning) because the scanner doesn't care that the parentheses don't match. All that it cares about is that there are parentheses that it needs to mark. During the Syntactic Analysis (Parsing) we find out that the syntax would be wrong. This would happen because we can't line our tokens up correctly in our AST.

*Remark* 1.1. Syntactic Analysis (Parsing) *ONLY* handles the reading in of tokens and creating an Abstract Syntax Tree. It *DOES NOT* attach any meaning to anything. Therefore, this does not return an error during Syntactic Analysis (Parsing).

```
integer q() {
  return 3;
}
```

However, it does return an error during Semantic Analysis.

**Defn 2** (Semantic Analysis)**.** *Semantic Analysis* is the phase of the compilation process that takes the AST (Abstract Syntax Tree) and attaches some semblance of meaning to the tokens in the tree. We determine what each "phrase" means, relate the uses of variables to their definitions, check types of expressions, and request translations of each "phrase". This is the point in the compilation process where the strings that were read in by the scanner and organized by the parser have any meaning. Before this, the only things that can be caught are token errors, and the like. So, this will generate an error that is caught during Semantic Analysis.

```
integer q() {
  return 3;
}
```

Because `integer` isn't a valid keyword in the Java language, at least not by default, and not capitalized like that, it gets caught during Semantic Analysis.

This would also generate an error during Semantic Analysis.

```
int p(int x) {
  int y;
  y = x * 2;
}
```

Both of these wouldn't be caught before the Semantic Analysis because the tokens read in during Lexical Analysis (Scanning) and organized during Syntactic Analysis (Parsing) do not have any meaning any earlier.

# 2 Lexical Analysis/Scanning

**Defn 3** (Lexical Analysis (Scanning))**.** *Lexical Analysis* or *Scanning* is the phase of the compilation process that reads in the source code text. It breaks the things it reads into *tokens*.

*Remark* 3.1. Lexical Analysis (Scanning) *ONLY* handles the reading IN of source code and the outputting of tokens. It *DOES NOT* attach any meaning or put anything together.

This means that these are the ***ONLY*** types of errors that will be caught.

```
1  int #s() {
2    return 3;
3  }
```

Because the # token isn't understood by the scanner, the whole thing fails. The Scanner is just a simple look up device. It can only find things that it knows about. If it sees something that it has no clue about, it fails.

There are several ways to implement a scanner. One of the most common ways is the use of a Finite State Automaton or Finite State Machine.

## 2.1 Finite State Automata

Finite state automata are used for regular expressions (regex's) to determine a matching word.

**Defn 4** (Finite State Automaton)**.** A *finite state automaton* or *finite state machine* is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

There are 2 types of finite statue automata:

1. Deterministic Finite Automata (DFA)
2. Non-deterministic Finite Automata (NFA)

A deterministic finite state automata can be constructed to be equivalent to any non-deterministic one.

*Remark* 4.1. The plural of Finite State Automaton is Finite State Automata.

**Defn 5** (Deterministic Finite Automata (DFA))**.** In a *deterministic finite automaton*, no two edges leaving from the same state are labeled with the same symbol. Additionally, there cannot be an edge that matches the empty string, $\epsilon$. A deterministic finite automaton will eventually terminate when it steps through all of its states necessary to reach the accepting state. However, the key difference between a Deterministic Finite Automata (DFA) and a Non-deterministic Finite Automata (NFA) is that you can always figure out the path that a deterministic finite automaton will take.

**Defn 6** (Non-deterministic Finite Automata (NFA))**.** A *non-deterministic finite automaton* is one that has multiple edges leaving a single state that have the same symbol. It may also have special edges labeled with the empty string $\epsilon$, which is when a state is followed without "eating" any of the input string. A non-deterministic finite automaton may eventually terminate when it steps through all its states necessary to reach its accepting state. However, the key difference between a Non-deterministic Finite Automata (NFA) and a Deterministic Finite Automata (DFA) is that you can always figure out the path that a deterministic finite automaton will take.

### 2.1.1 Converting a NFA to a DFA

There are a few steps for converting a non-deterministic finite automaton to a deterministic one.

1. Start at the start state and enter it
2. Follow all the states that accept the empty string $\epsilon$ and combine them with the start state.
3. After that, read in the first character/word from the input and follow all the states that you combined.
   - This means that you will be following multiple states or edges at the same time.
4. Continue doing this until you combine all the states down to a deterministic finite automaton.
   - You can have multiple instances of the same state, i.e., you can have state 5 in two different state bubbles, so long as the list of states inside is unique.
5. The end states are found by taking the end states from the non-deterministic finite automaton and placing them in the deterministic finite automaton.
   - This means that if state 3 is and end state in the non-deterministic finite automaton, then every occurrence of state 3 in the deterministic finite automaton will be and end state.

# 3 Syntactic Analysis/Parsing

There are 2 kinds of parsing techniques:

1. LL Parsing
2. LR Parsing

The table below will help characterize the differences between them.

| | $LL(k)$ | $LR(k)$ |
|---|---|---|
| Parses Input | *L*eft-to-Right | |
| Derivation | *L*eftmost | *R*ightmost |
| Lookahead | $k$ Symbols | |
| Build the Tree | Top Down | Bottom Up |
| Select Rule | After seeing its first $k$ tokens | After seeing all its tokens, and an addition $k$ tokens |
| Left Recursion | No | Yes |
| Unlimited Common Prefix | No | Yes |
| Resolve Ambiguities Through Rule Priority | Dangling Else | Dangling Else, Associativity, Priority |
| Error Recovery | Trial-and-Error | Good Algorithms Exist |
| Implement by Hand? | Possible | Too complicated. Use a generator. |

Table 3.1: LL vs. LR Parsing

The block below will show the difference in derivation between LL Parsing and LR Parsing.

Say you have a set of productions as follows:

$$p1 : X \to YZV$$
$$p2 : Y \to ab$$
$$p2 : Z \to c$$
$$p3 : V \to de$$

The LL derivation will be as follows:

$$\underline{X} \Rightarrow \underline{Y}ZV \Rightarrow ab\underline{Z}V \Rightarrow abc\underline{V} \Rightarrow abcde$$

The LR derivation has 2 options, both of which achieve the same thing.

1. The way it works in practice, you have the terminals and have to go "up" the production rules.

$$\underline{ab}cde \Rightarrow Y\underline{c}de \Rightarrow YZ\underline{de} \Rightarrow \underline{YZV} \Rightarrow X$$

2. The way it works in theory, you have a starting nonterminal and work your way down by deriving the right-most side of the input string.

$$\underline{X} \Rightarrow YZ\underline{V} \Rightarrow Y\underline{Z}de \Rightarrow \underline{Y}cde \Rightarrow abcde$$

## 3.1 LL Parsing

There are 5 basic steps in constructing an LL(1) parser.

1. Write the grammar in canonical form.
2. Compute Nullable, FIRST, and FOLLOW.
3. Use them to contruct a table. It shows what production to select given the current lookahead token.
4. Check for conflicts.
   (a) If there *are* conflicts, then the grammar is not LL(1).

(b) If there are *no* conflicts, then there is a straight-forward implementation using table-driven parser or recursive descent.

Many times, you will encounter Fixed-Point Problems.

**Defn 7** (Fixed-Point Problems)**.** *Fixed-point problems* have the form:

$$x == f(x) \tag{3.1}$$

Can we find a value $x$ for which the equation holds (i.e., a solution)? $x$ is then called the *fixed point* of the function $f(x)$. Fixed-Point Problems can (sometimes) be solved using iteration. The steps involved in *fixed-point iteration* are:

1. Guess an initial value $x_0$
2. Apply the function iteratively
3. Iterate until the fixed point is reached.

$$\begin{aligned} x_1 &= f(x_0) \\ x_2 &= f(x_1) \\ &\vdots \\ x_n &= f(x_{n-1}) \end{aligned}$$

You continue this iteration until $x_n = x_{n-1}$, and $x_n$ is called the *fixed point*.

### 3.1.1 Nullable

**Defn 8** (Nullable)**.** For the production $p$, where $p : X \to \gamma$, and $X$ and $\gamma$ are nonterminals; $p$ is *Nullable* if we can derive $\epsilon$ from $\gamma$.

More formally, this can be defined as Nullable($\gamma$) is true iff the empty sequence can be derived from $\gamma$:

$$\text{Nullable}(\gamma) = \begin{cases} \text{True}, & \exists(\gamma \Rightarrow* \epsilon) \\ \text{False}, & \text{Otherwise} \end{cases}$$

You can define an equation system for Nullable given that $G = (N, T, P, S)$.

$$\text{Nullable}(\epsilon) == \text{True} \tag{3.2a}$$

$$\text{Nullable}(t) = \text{False} \tag{3.2b}$$

where $t \in T$, i.e., $t$ is a terminal symbol

$$\text{Nullable}(X) = \text{Nullable}(\gamma_1) \,\|\, \ldots \,\|\, \text{Nullable}(\gamma_n) \tag{3.2c}$$

where $X \to \gamma_1, \ldots, X \to \gamma_n$ are all the productions for $X$ in $P$.

$$\text{Nullable}(s\gamma) = \text{Nullable}(s) \,\&\&\, \text{Nullable}(\gamma) \tag{3.2d}$$

where $s \in N \cup T$, i.e., $s$ is a nonterminal or a terminal

*Remark* 8.1. The equations (Equations (3.2a) to (3.2d)) for Nullable are recursive. Therefore, you can't just calculate these recursively, because you might never terminate if the empty sequence is never reached. This is one set of Fixed-Point Problems.

One way to think about solving a problem asking about Nullable is shown below. I will use this grammar to demonstrate:

$$p1 : X \to Y|Z$$
$$p2 : Y \to a|b|V$$
$$p3 : Z \to c|\epsilon$$
$$p4 : V \to d|Y$$

1. Determine the nonterminal you are interested in, let's say $X$.
2. Find all the productions with $X$ on the ***LEFT-HAND SIDE***.
   - $X$ is present on the left-hand side of $p1$.
3. Follow these productions to their right-hand side.
   - So we are considering the right-hand side of $p1$, which is $Y|Z$.
4. You will evaluate each of the tokens on the ***RIGHT-HAND SIDE*** of the production(s) we are interested in.
5. If the token we are looking at on the ***RIGHT-HAND SIDE*** is a terminal, the nonterminal $\gamma$ is ***NOT*** nullable.
   - This is the case for $Y$. Since either $a$ or $b$ could present, and $V$ can either produce $d$ or recurse back to $Y$, $Y$ can NEVER yield $\epsilon$.
   - In our case, both $X \to Y$ and $X \to Z$; $Y$ and $Z$ are nonterminals, so this step doesn't apply.
6. If the token that we are looking at on the ***RIGHT-HAND SIDE*** is a nonterminal, then follow them.
   - In both $X \to Y$ and $X \to Z$, $Y$ and $Z$ are nonterminals, so we follow both.
     - Since we already calculated $Y$ in the previous step, we know that $Y$ is not nullable. However, we can add the values that $Y$ can produce to a set to make sure we are correct. So, Nullable$(X) = \{a, b, d\}$. Now we move onto $Z$.
     - The production for $Z$ is $Z \to c|\epsilon$. In this case, $Z$ may be $\epsilon$. We can add these values to our Nullable$(X)$ set: Nullable$(X) = \{a, b, c, d, \epsilon\}$
7. Once we have computed all possible Nullable$(X)$ occurrences, we are done.

This leaves us with our Nullable$(X)$ set: $\{a, b, c, d, \epsilon\}$. Since there is an option for $X$ to be $\epsilon$, $X$ **is** Nullable.

### 3.1.2 FIRST

**Defn 9** (FIRST)**.** For the production $p$, where $p : X \to \gamma$, and $X$ and $\gamma$ are nonterminals. The FIRST$(\gamma)$ are the **tokens** that occur *FIRST* in a sentence derived from $\gamma$.

More formally, this can be defined as: FIRST$(\gamma)$ is the set of tokens that can occur *first* in the sentences derived from $\gamma$.

$$\text{FIRST}(\gamma) = \{t \in T | \gamma \Rightarrow* t\delta\}$$

You can define an equation system for FIRST given that $G = (N, T, P, S)$.

$$\text{FIRST}(\epsilon) = \emptyset \tag{3.3a}$$

$$\text{FIRST}(t) = \{t\} \tag{3.3b}$$

where $t \in T$, i.e., $t$ is a terminal symbol

$$\text{FIRST}(X) = \text{FIRST}(\gamma_1) \cup \ldots \cup \text{FIRST}(\gamma_n) \tag{3.3c}$$

where $X \to \gamma_1, \ldots, X \to \gamma_n$ are all the productions for $X$ in $P$.

$$\text{FIRST}(s\gamma) = \text{FIRST}(s) \cup (\text{if Nullable}(s) \text{ then FIRST}(\gamma) \text{ else } \emptyset \text{ fi}) \tag{3.3d}$$

where $s \in N \cup T$, i.e., $s$ is a nonterminal or a terminal

*Remark* 9.1. The equations (Equations (3.3a) to (3.3d)) for FIRST are recursive. Therefore, they might not terminate, so you must calculate this as another set of Fixed-Point Problems.

One way to think about solving a problem asking about FIRST is shown below. I will use this grammar to demonstrate:

$$p1 : X \to YZa$$
$$p2 : Y \to b|Z|V$$
$$p3 : Z \to c|\epsilon$$
$$p4 : V \to \epsilon$$

1. Determine the nonterminal you are interested in, let's say $Y$.
2. Find all productions with $Y$ on the left-hand side.
   - $Y$ is present on the left-hand side of $p2$.
3. Follow each of these productions to their right-hand side.
   - So we are considering the right-hand side of $p2$, which is $b|Z|V$.
4. If the first token on the ***RIGHT-HAND SIDE*** is a terminal, add it to the FIRST set.
   - FIRST$(Y) = \{b\}$
5. If the first token on the ***RIGHT-HAND SIDE*** is a nonterminal, go to that production and compute FIRST on that.
   - Since $Z$ is an option in $p2$, we compute FIRST$(Z)$, which yields $\{c, \epsilon\}$. We add both to the FIRST$(Y)$ list. Our list is now FIRST$(Y) = \{b, c, \epsilon\}$
   - Since $B$ is an option in $p2$, we compute FIRST$(V)$, which yields $\{\epsilon\}$. We add this to the FIRST$(Y)$ list. Our list is now FIRST$(Y) = \{b, c, \epsilon\}$
6. Since $\epsilon$ is an empty string, we can remove it from the list, or just ignore it.
7. Once we have computed all possible FIRST$(Y)$ occurrences, we are done.

This leaves us with our FIRST$(Y)$ set: $\{b, c\}$, which is the solution.

### 3.1.3 FOLLOW

**Defn 10** (FOLLOW)**.** For the production $p$, where $p : X \to \gamma$, and $X$ and $\gamma$ are nonterminals. The FOLLOW$(X)$ are the **tokens** that *FOLLOW* immediately after an $X$-sentence.

More formally, this can be defined as: FOLLOW$(X)$ is the set of tokens that can occur as the *first* token *following* $X$, in any Sentential Form derived from the start symbol $S$:

$$\text{FOLLOW}(X) = \{t \in T | S \Rightarrow* \alpha X t \beta\}$$

The nonterminal $X$ occurs on the right-hand side of a number of productions.

Let $Y \to \gamma X \delta$ denote such an occurrence, where $\gamma$ and $\delta$ are arbitrary sequences of terminals and nonterminals. You can define an equation system for FOLLOW given that $G = (N, T, P, S)$.

$$\text{FOLLOW}(X) = \bigcup \text{FOLLOW}(Y \to \gamma \underline{X} \delta) \tag{3.4a}$$

over all occurrences $Y \to \gamma X \delta$, and where

$$\text{FOLLOW}(Y \to \gamma \underline{X} \delta) = \text{FIRST}(\delta) \cup (\text{if Nullable}(\delta) \text{ then FOLLOW}(Y) \text{ else } \emptyset \text{ fi}) \tag{3.4b}$$

*Remark* 10.1. Again, the equations (Equations (3.4a) to (3.4b)) are recursive. Therefore, they might not terminate, so you must calculate this as another set of Fixed-Point Problems.

*Remark* 10.2 (Sentential Form). Sequence of terminal and nonterminal symbols.

One way to think about solving a problem asking about FOLLOW is shown below. I will use this grammar to demonstrate:

$$p1 : S \to Xa$$
$$p2 : X \to Y|Yb$$
$$p3 : Y \to YZc|\epsilon$$
$$p4 : Z \to d|\epsilon$$

1. Determine the nonterminal you are interested in, let's say $Y$.
2. Find all occurrences of that nonterminal on the ***RIGHT-HAND SIDE*** of the productions. If the nonterminal is ***NOT*** present anywhere on the right-hand side, it yields the empty set, $\{\emptyset\}$. In this case our nonterminal occurs in:
   - $p2$
   - $p3$
3. Find the terminals that can directly follow your nonterminal *in the same production.*
   - $b$, from $p2$
   - $c$, from $p3$
4. If there are nonterminals after the nonterminal you are interested in, follow them. In this case, we follow $Z$. Then, find the nonterminals that can directly follow that nonterminal and add them to you're list.
   - $b$, from $p2$
   - $c$, from $p3$
   - $d$, from $p4$
5. If nothing (no nonterminal AND no terminal) follows the nonterminal you want, then go "backwards" through the production.
   (a) Since $p2$ has $X \to Y$ as an option and nothing follows this $Y$
   (b) Go backwards, up to the production(s) that produces $X$ on the right-hand side
   (c) Compute Follow on the right-hand side of that production. This produces:
      - $a$, from $p1$

This leaves us with our FOLLOW($Y$) set: $\{a, b, c, d\}$, which is the solution.

### 3.1.4 Constructing an LL(1) Table

Using the information that was gathered from the Nullable, FIRST, and FOLLOW calculations, you can construct an LL(1) parse table with the following steps.

1. Look at each production $p : X \to \gamma$.
2. Compute the token set FIRST($\gamma$). Add $p$ to each corresponding entry for $X$.
3. Check if $\gamma$ is Nullable.

   (a) If so, compute the token set FOLLOW($X$), and add $p$ to each corresponding entry for $X$.

**Example 3.1: LL1 Parse Table.**

An example of an LL(1) table is shown in Table 3.2. It uses the grammar below.

$p0 : S \rightarrow$ varDecl \$
$p1 :$ varDecl $\rightarrow$ type ID optInit
$p2 :$ type $\rightarrow$ "Integer"
$p3 :$ type $\rightarrow$ "Boolean"
$p4 :$ optInit $\rightarrow$ "="INT
$p5 :$ optInit $\rightarrow \epsilon$

The steps to constructing an LL(1) table are:

1. Look at each production $p : X \rightarrow \gamma$
2. Compute FIRST($\gamma$), and add the corresponding $p$ to the table for $X$
3. If $\gamma$ is Nullable, then compute FOLLOW($\gamma$) and add the $p$ with the Nullable production to each corresponding value for $X$

| | ID | Integer | Boolean | "=" | INT | $ |
|---|---|---|---|---|---|---|
| S | | $p0$ | $p0$ | | | |
| varDecl | | $p1$ | $p1$ | | | |
| type | | $p2$ | $p3$ | | | |
| optInit | | | | $p4$ | $p5$ | |

Table 3.2: LL(1) Table Example

**3.1.4.1 LL(1) Parse Table Conflicts** For every case where an LL(1) grammar fails, it can be illustrated by the parse table for the grammer. The table for any of these grammars with have a *conflict* in one of its cells, as shows when there are multiple productions in the cell. Some of the cases where this may happen are listed below:

1. Ambiguous Grammar
2. Left-Recursive
3. Common Prefix

### 3.1.5 Eliminating Left Recursion

LL(1) cannot support left recursion, however, they can support right recursion. So, if we have a grammar with left recursion, and want to LL(1) parse it, we need to rewrite the grammar. We can introduce a new nonterminal, which allows us to recurse on the right side of the production, but not the left.

## 3.2 LR Parsing

LR($k$) parsing stands for Left-to-right parse, rightmost-derivation, $k$-token lookahead. This parsing technique postpones the decision of which production to use until it sees the entire right-hand side of the production in question (and $k$ more tokens beyond).

An LR parser has a *stack* and and *input*. The first $k$ tokens of the input are the *lookahead*. Based on the contents of the stack and the lookahead, the parser performs 1 of 2 actions.

1. Shift
2. Reduce

**Defn 11** (Shift)**.** A *shift* operation corresponds to moving the first input token onto the top of the stack. This is equivalent to reading the token and moving it onto the stack, and advancing forward through the sentence.

*Remark* 11.1 (Accepting). Shifting over the EOF (End of File) marker, typically denoted \$, is called *accepting* and causes the parser to stop successfully.

**Defn 12** (Reduce)**.** A *reduce* operation corresponds to choosing a grammar rule $X \rightarrow ABC$; pop $C, B, A$ off the top of the stack, and push $X$ onto the stack.

Initially, the stack is empty and the parser is sitting at the beginning of the input.

**Example 3.2: LR(1) Shift-Reduce Parsing.**

Say you have a set of productions as follows:

$$p1 : X \to YZV$$
$$p2 : Y \to ab$$
$$p2 : Z \to c$$
$$p3 : V \to de$$

and an input string of

$$abcde$$

to parse. Assume that there is a production to handle the end-of-file character \$.

---

You start by constructing a "queue" and "stack" of your input tokens. The front of the queue is after the dot, and the top of the stack is before the dot. So, to parse the above string:

1. Construct your data structures.
$$\bullet abcde$$
2. Then you start pulling tokens off the front of the queue and putting them onto the stack with Shift actions.
$$\text{Shift } a \bullet bcde$$
$$\text{Shift } ab \bullet cde$$
3. Whenever you have a set of terminals and/or nonterminals on the top of the stack, you pop them, perform a Reduce action, and push the resulting nonterminal back onto the top of the stack.
$$\text{Reduce } Y \bullet cde$$
4. Repeat this operation until you reach an Accepting action.
$$\text{Shift } Yc \bullet de$$
$$\text{Reduce } YZ \bullet de$$
$$\text{Shift } YZd \bullet e$$
$$\text{Shift } YZde\bullet$$
$$\text{Reduce } YZV\bullet$$
$$\text{Reduce } X\bullet$$
$$\text{Accept}$$

In practice, $k > 1$ is not used. These would generate incredibly large tables that would be hard to used and hard to make. Most reasonable programming languages can be described by LR(1) grammars.

### 3.2.1 LR Finite State Automata

These automata are Deterministic Finite Automata (DFA). They are used in the parser to decide when to Shift and when to Reduce, and are applied to the stack.

They can be used to generate an LR Parse Table.

Each of the states consists of one ore more LR Items.

**Defn 13** (LR Item)**.** The parser uses a Deterministic Finite Automata (DFA) to decide whether to shift or reduce the expression that is has seen so far. The *states* in the DFA are sets of *LR Items*.

An example of an LR Item is shown in Equation (3.5).

$$X \to \alpha \bullet \beta \quad t, s \tag{3.5}$$

An LR(1) item is a production extended with:

- A *dot* ($\bullet$), corresponding to the position in the input sentence.
- One or more possible *lookahead* terminal symbols: $t, s$.

- We will use ? when the lookahead doesn't matter.

*Remark* 13.1. The stack that is referenced in this section is left side of the dot that is present in LR Items.

The LR(1) item corresponds to a state where (using Equation (3.5)):

- The topmost part of the stack is $\alpha$
- The first part of the remaining input is expected to match either, in no particular order:
  - $\beta t$
  - $\beta s$

### 3.2.2 LR Parse Table

This table is generated after making an LR Finite State Automata. To make one, there are 4 actions to note in the table.

1. Shift Actions
2. Reduce Actions
3. Goto Actions
4. Accept Action
5. Errors are denoted by blank entries in the table.

**3.2.2.1 Shift Actions** These are found by reading ***TOKENS***. For each **token edge,** $t$, from state $j$ to state $k$, add a ***shift action*** **s**$k$ to table$[j, t]$. (This corresponds to reading a token and pushing it onto the stack.)

**3.2.2.2 Reduce Actions** These are found when the **dot is at the end**. Add a ***reduce action*** **r**$p$ (reduce $p$) to table$[j, t]$, where $p$ is the production and $t$ is the lookahead token. (This corresponds to popping the right-hand side of a production off the stack and going backwards through the state machine.)

**3.2.2.3 Goto Actions** These are found by reading a ***NONTERMINAL***. For each *nonterminal edge,* $X$, from state $j$ to state $k$, add a ***Goto Action*** $gk$ (goto state $k$) to table$[j, X]$. (This corresponds to pushing the left-hand side of a nonterminal production onto the stack.)

**3.2.2.4 Accept Action** These are found when in a state containing an LR item with your **dot on the left of $**. If this is so, then add an ***accept action*** $a$ to the table with indices table$[j, \$]$. (If we are about to perform a shift action over the EOF (End Of File) token, $, then parsing has succeeded.)

### 3.2.3 LALR(1) Parsing Tables

Since LR(1) tables can get quite large, a smaller table can be made by merging any 2 states whose items are identical except for lookahead sets. The resulting parser is called a *Lookahead LR*(1) parser. For example, compare the same states, but different parser types as shown in Table 3.3a and Table 3.3b.

| Productions | Lookahead Token |
|---|---|
| $S' \to \bullet S\$$ | ? |
| $S \to \bullet V = E$ | $ |
| $S \to \bullet E$ | $ |
| $E \to \bullet V$ | $ |
| $V \to \bullet x$ | $ |
| $V \to \bullet * E$ | $ |
| $V \to \bullet x$ | = |
| $V \to \bullet * E$ | = |

(a) LR(1) Table State

| Productions | Lookahead Token |
|---|---|
| $S' \to \bullet S\$$ | ? |
| $S \to \bullet V = E$ | $ |
| $S \to \bullet E$ | $ |
| $E \to \bullet V$ | $ |
| $V \to \bullet x$ | $,= |
| $V \to \bullet * E$ | $,= |

(b) LALR(1) Table State

Table 3.3: LR(1) Table State vs LALR(1) Table State

### 3.2.4 Syntax Versus Semantics

There are some things that context-free grammars cannot describe, and thus cannot be parsed correctly. For instance, the expression

$$a + 5 \,\&\&\, b$$

The precedence here has the mathematical addition in greater priority than the logical AND operator. But logical and mathematical operators are not allowed in the same expression, because the types of the variables and operations don't make sense together. However, the context-free grammar and parser have no knowledge of the *types* of the variables and oeprators in play here. So, the solution is to let this expression pass through the parser, but it should be caught later, during the Semantic Analysis.

# 4 Abstract Syntax and Abstract Syntax Trees

| | **Concrete Syntax** | **Abstract Syntax** |
|---|---|---|
| What does it Describe? | The concrete representation of the programs | The abstract structure of the programs |
| Main Use | Parsing text to trees | Model representing program inside compiler |
| Underlying formalism | Context-Free Grammar | Recursive Data Types |
| What is Named? | Only non-terminals. (Productions usually anonymous) | Nonterminals and Productions |
| What tokens occur in the grammar? | All tokens corresponding to "words" in the text | Usually tokens with values (identifiers, literals) |
| | Independent of abstract structure | Independent of parser and parser algorithm |

Table 4.1: Concrete Syntax vs. Abstract Syntax

**Defn 14** (Concrete Syntax)**.** The *concrete syntax* is more verbose than that of an Abstract Syntax. It contains the necessary information, grammar transformations, and elimination of ambiguity required to parse the program. However, they can be unwieldy to use in the later stages of compilation.

**Defn 15** (Abstract Syntax)**.** The *abstract syntax* is less verbose than that of the Concrete Syntax, but it contains all the of the same information. This makes a clean interface between the parser and later stages of a compiler (or other kinds of program-analysis tools).

The *abstract syntax* conveys the phrase structure of the source program, with all the parsing issues resolved, but no semantic interpretation yet.

Early compilers did not use these because of memory issues.

## 4.1 Parse Trees

One method of doing this is for the parser to produce a *parse tree*.

**Defn 16** (Parse Tree)**.** A *parse tree* is a data structure for later phases of the compiler to traverse. It describes the entirety of the program within a tree structure. Here, there is exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse. This is called a Concrete Parse Tree

**Defn 17** (Concrete Parse Tree)**.** A *concrete parse tree* is one that has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse. This represents the *Concrete Syntax* of the source language, which may be difficult to use internally. Much of the punctuation that is borken up from the input string conveys no information to the compiler, but are useful for the programmer. However, once the Parse Tree is built, the structure of the tree conveys the structuring information in a more convienent way.

Additionally, the structure of the Concrete Parse Tree may depend too much on the grammar and Concrete Syntax. Grammar transformations and the elimination of ambiguity should only take place during parsing, and no later.

**Defn 18** (Abstract Parse Tree)**.** An *abstract parse tree* is also called an *Abstract Syntax Tree*. Here, the Concrete Parse Tree is represented only by the operations present in the program. The precedence and formatting of the tree is handled by the Concrete Syntax and Concrete Parse Tree.

*Abstract Parse/Syntax Trees* are data structures within the compiler program, and are not going to be used outside of it.

*Remark* 18.1 (Abstract Syntax Tree in Java). In Java, because of its Object-Oriented nature, the Abstract Parse Tree is organized in the following way:

- An abstract class for each nonterminal
- A subclass for each production
- etc.

| **Abstract Grammar** | **Object-Oriented Model** | **Other Model** (Algebraic Data Types) |
|---|---|---|
| Programming Language | Java | Haskell |
| Nonterminal | Superclass | Type, Sort |
| Production | Subclass | Constructor, Operator |

Table 4.2: Abstract Syntax Tree Hierarchy in Programming Paradigms

### 4.1.1 Abstract Parse/Syntax Trees

The compiler can use the Abstract Parse Tree to do many things. It can perform:

- Name Analysis: Find the declaration of an identifier
- Type Analysis: Compute the type of an expression
- Expression Evaluation: Compute the value of a constant expression
- Code Generation: Compute an intermediate code representation of the program
- Unparsing: Compute a textual representation of the program

# 5 Semantic Analysis

This is the time where we can start attaching some meaning to the tokens that we have read. We can attach types to the tokens to note that they are number literals, variables, identifiers, expressions, etc. This will begin the portion of the compiler where we have read in our program and now we need to start making sure that it makes sense.

To improve modularity, it is better to seaparate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code).

## 5.1 Visitors

Visitors are an example of "The Expression Problem". This problem states that we would like to:

- Define *language constructs* in a modular way (Java Class Hierarchy)
- Define *computations* in a modular way (On those Classes)
- *Compose* these modules as we like
- Be able to *separately compile* these modules
- Have full *static type safety* (No need for typecasting or instanceof)

The Expression Problem contains:

- *Kinds* of objects: compound statements, assignment statements, print statements, etc.
- *Interpretations* of these objects: type-checking, translate to other code, optimize, interpret, etc.

This means there are 2 "methods" to solve The Expression Problem:

1. Separate your Abstract Syntax from your interpretation. This makes it easy and modular to:
   - Add a new interpretation, because they are all logically grouped together
   - However, it is hard to add a new kind of interpretation, because you need to add new functions to all existing interpretations
2. Tie your Abstract Syntax to your interpretations
   - Easy to add new kind. All the interpretations of that kind are grouped together as methods of the new kind.

- Not modular to add a new interpretation, a new method must be added to every class

These require your language to support static aspects, and Java natively doesn't. You would need another language like AspectJ or JastAdd.

So, to deal with The Expression Problem, there are a few options:

1. Edit the AST classes
   - Doesn't actually solve the problem
   - Non-modular
   - Non-compositional
   - **BAD IDEA TO EDIT GENERATED CODE**
   - However, sometimes this is done in industry
2. Visitors
   - An Object-Oriented design pattern
   - Modularize through clever indirect function/method calls
   - Not full modularization
   - No composition
   - Supported by many parser generators
   - Reasonably useful, commonly used in industry
3. Static Aspect-Oriented Programming (AOP)
   - Also known as *Inter-Type Declarations (ITDs)*
   - Use new language constructs (aspects) to factor out code
   - Solves The Expression Problem in a nice simple way
   - Drawback: You need a new language
     - AspectJ
     - JastAdd
4. Advanced Language Constructs
   - Use more advanced language constructs:
     - Virtual Classes in bgeta
     - Traits in Scala
     - Typeclasses in Haskell
   - Drawbacks:
     - More complex than Static Aspect-Oriented Programming
     - You need an advanced language
     - Not much practical experience (so far)

**Defn 19** (Visitors)**.** *Visitors* are used to modularize compilers in Java, or any other Object-Oriented language without Aspect-Oriented Programming mechanisms. "The Visitor design pattern lets you define a new operation without changing the elements on which it operates" (Gamma et al. 1994).

The Visitor pattern is a technique to use the Abstract Syntax-separate-from-interpretation style.

A visitor implements an interpretation; it is an object which contains a `visit` method for each Abstract Parse Tree class. Each Abstract Parse Tree should contain an `accept` method, which serves a hook for all interpretations. It is called by a visitor and passes control back to an appropriate method of the visitor.

This can be thought of as a dialogue between the Abstract Parse Tree class and the visitor class. The visitor calls the `accept` method of a node and asks "What is your class?" The `accept` method answers by calling the corresponding `visit` method from the visitor.

These `visit` methods are usually overloaded for the various types present in the Abstract Parse Tree, further increasing code modularity.

| | Frequent type-casts? | Frequent recompilation? |
|---|---|---|
| `Instanceof` and type-casts | Yes | No |
| Dedicated methods | No | Yes |
| The Visitor Pattern | No | No |

Table 5.1: Summary of the Visitors Pattern

# References

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd Edition. Cambridge University Press, 2002. ISBN: 052182060X.

[Gam+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 9780201633610.