EDAN65: Compilers — Reference Sheet
Lund University

Karl Hallsby

Last Edited: March 10, 2020

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).

```
1    int r( {
2      return 3;
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).

```
1    integer q() {
2      return 3;
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.

```
1    integer q() {
2      return 3;
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.

```
1    int p(int x) {
2      int y;
3      y = x * 2;
4    }
```

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 through Regular Expressions.

2.1 Regular Expressions

Defn 4 (Regular Expression). A *regular expression*, sometimes called a *regex* is a way to define a sequence of characters to form strings.

There are 2 types of Regular Expressions, based on the features available to make the regular expression.

1. Core Notation
2. Extended Notation

Defn 5 (Core Notation). The *core notation* of a Regular Expression has a small number of features available. These are shown in Table 2.1.

Regular Expression	Read As	Called
a	a	Symbol
$M|N$	M or N	Alternative
MN	M followed by N	Concatenation
ϵ	The Empty String	Epsilon
$M*$	Zero or more M	Repetition (Kleene Star)
(M)		Scope

Table 2.1: Regular Expression Core Notation

Where a is a symbol in the alphabet. M and N are Regular Expressions.

Defn 6 (Extended Notation). The *extended notation* of a Regular Expression contains all the features of the Core Notation, and some additional features. These additional features *can* be represented in the Core Notation, but are confusing to read and write.

The Core Notation features are shown in Table 2.1, the additional features added by the Extended Notation are shown in Table 2.2.

Extended Regular Expression	Read As	Means
$M+$	One or more	$MM*$
$M?$	Optional	$\epsilon|M$
$[aou]$	One of ... (a character class)	$a|o|u$
$[a-zA-Z]$		$a|b|\ldots|z|A|B\ldots|Z$
$[\wedge 0-9]$	Not	One character, but any one of those listed
Appel Notation: $\sim [0-9]$		
"a+b"	The string	a b

Table 2.2: Regular Expression Extended Notation

2.2 Finite State Automata

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

Defn 7 (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 7.1. The plural of Finite State Automaton is Finite State Automata.

Defn 8 (Deterministic Finite Automata (DFA))**.** In a *deterministic finite automaton* or *DFA*, 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, ϵ. A deterministic finite automaton will eventually terminate when it steps through all of its states necessary to reach the accepting state.

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 9 (Non-deterministic Finite Automata (NFA))**.** A *non-deterministic finite automaton*, or *NFA*, 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 ϵ, 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.

The key difference between a Non-deterministic Finite Automata (NFA) and a Deterministic Finite Automata (DFA) is that you cannot always determine the exact path that the Finite State Automaton will will take.

2.2.1 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 ϵ 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.

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

	$LL(k)$	$LR(k)$
Parses Input	Left-to-Right	
Derivation	Leftmost	Rightmost
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

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 a lookup token (either a terminal or nonterminal) and go "up" the production rules based on the given terminals and the lookup token.

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

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   Context-Free Grammars

Defn 10 (Context-Free Grammar)**.** A *context-free grammar* or *CFG* is a way to define a set of *strings* that form a Language. Each string is a finite sequence of Terminal Symbol taken from a finite Alphabet. This is done with one or more Productions, where each production can have both Nonterminal Symbol and Terminal Symbol.

More formally, a Context-Free Grammar is defined as $G = (N, T, P, S)$, where

- N, the set of Nonterminal Symbols
- T, the set of Terminal Symbols
- P, the set of production rules, each with the form

$$X \to Y_1 Y_2 \ldots Y_n \text{ where } X \in N, x \geq 0, \text{and } Y_k \in N \cup T$$

- S, the start symbol (one of the Nonterminal Symbols, N). $S \in N$.

Remark 10.1*.* It is important to note that there are 3 forms of Context-Free Grammars:

1. Canonical Form

2. Backus-Naur Form
3. Extended Backus-Naur Form

Defn 11 (Language). A *language* is the set of **all** strings that can be formed by the Productions in the Context-Free Grammar.

Defn 12 (Production). A *production* is a rule that defines the relation between a single Nonterminal Symbol and a string comprised of Nonterminal Symbols, Terminal Symbols, and the Empty String.
The are denoted as shown below:

$$p_0 : A \rightarrow \alpha \tag{3.1}$$

Defn 13 (Nonterminal Symbol). A *nonterminal symbol* is a symbol that is used in the Context-Free Grammar as a symbol for a Production.

Defn 14 (Terminal Symbol). A *terminal symbol* is a symbol that cannot be derived any further. This is a symbol that is part of the Alphabet that is used to form the Language.

Defn 15 (Empty String). The *empty string* is a special symbol that is neither a Nonterminal Symbol nor a Terminal Symbol. The empty string is a *metasymbol*. It is a unique symbol meant to represent the lack of a string. It is denoted with the lowercase Greek epsilon, ϵ or ε.

Defn 16 (Alphabet). The finite set of Nonterminal Symbols that can be used to form a Language.

Defn 17 (Ambiguous). A Context-Free Grammar is said to be *ambiguous* or has *ambiguities* if there is more than one way to derive the same string in a grammar.
The grammar below is ambiguous because there are multiple ways to parse the string: "statement;statement;statement".

$$
\begin{aligned}
&p_0 : \text{start} \rightarrow \text{program } \$ \\
&p_1 : \text{program} \rightarrow \text{statement} \\
&p_2 : \text{statement} \rightarrow \text{statement ";" statement} \\
&p_3 : \text{statement} \rightarrow \text{ID "=" INT} \\
&p_4 : \text{statement} \rightarrow \epsilon
\end{aligned}
\tag{3.2}
$$

3.1.1 Context-Free Grammar Forms

Defn 18 (Canonical Form). The *canonical form* of a Context-Free Grammar is the most formal use of a Context-Free Grammar.

$$
\begin{aligned}
A &\rightarrow B \; d \; e \; C \; f \\
A &\rightarrow g \; A
\end{aligned}
\tag{3.3}
$$

The Canonical Form is:

- The core formalism for Context-Free Grammars
- Useful for proving properties and explaining algorithms

Defn 19 (Backus-Naur Form). The *Backus-Naur form* of a Context-Free Grammar is an extension of the Canonical Form. This form is less formal than the Canonical Form, but is allows for condensation of multiple productions that have the same nonterminal on the left-hand side to the same production. This is done with the | symbol.
For example, Equation (3.4) is equivalent to Equation (3.3).

$$A \rightarrow B \; d \; e \; C \; f \mid g \; A \tag{3.4}$$

Defn 20 (Extended Backus-Naur Form). The *Extended Backus-Naur form* of a Context-Free Grammar is an extension of the *Backus-Naur Form*. This is a more informal implementation of a Context-Free Grammar. This informality allows for some additional constructs in the Production rules.
These include:

1. Repetition with the Kleene Star (*)
2. Optionals
3. Parentheses

The Extended Backus-Naur Form is:

- Compact, easy to read and write
- Common notation for practical use

3.1.2 Chomsky Hierarchy of Formal Grammars

There exists a hierarchy for the definition of Grammars that define Languages. It is called the *Chomsky Hierarchy of Formal Grammars.*

Grammar	Rule Patterns	Type
Regular	$X \to aY$ or $X \to a$ or $X \to \epsilon$	3
Context-Free Grammar	$X \to \gamma$	2
Context-Sensitive	$\alpha X \beta \to \alpha \gamma \beta$	1
Arbitrary	$\gamma \to \delta$	0

Table 3.2: Chomsky Hierarchy of Formal Grammars

Where a is a Terminal Symbol, α, β, γ, and δ are *sequences* of symbols (Terminal Symbols or Nonterminal Symbols).
$\text{Type}(3) \subset \text{Type}(2) \subset \text{Type}(1) \subset \text{Type}(0)$
Regular grammars have the same power as Regular Expressions.

3.2 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 21 (Fixed-Point Problems). *Fixed-point problems* have the form:

$$x == f(x) \tag{3.5}$$

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.

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

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

3.2.1 Nullable

Defn 22 (Nullable). For the production p, where $p : X \to \gamma$, and X and γ are nonterminals; p is *Nullable* if we can derive ϵ from γ.

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

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

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

$$\text{Nullable}(\epsilon) == \text{True} \tag{3.6a}$$

$$\text{Nullable}(t) = \text{False} \tag{3.6b}$$

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

$$\text{Nullable}(X) = \text{Nullable}(\gamma_1) \,\|\, \ldots \,\|\, \text{Nullable}(\gamma_n) \tag{3.6c}$$

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

$$\text{Nullable}(s\gamma) = \text{Nullable}(s) \,\&\&\, \text{Nullable}(\gamma) \tag{3.6d}$$

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

Remark 22.1. The equations (Equations (3.6a) to (3.6d)) 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 γ 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 ϵ.
   - 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 ϵ. 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 ϵ, X **is** Nullable.

3.2.2 FIRST

Defn 23 (FIRST). For the production p, where $p : X \to \gamma$, and X and γ are nonterminals. The FIRST(γ) are the **tokens** that occur *FIRST* in a sentence derived from γ.

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

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

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

$$\text{FIRST}(\epsilon) = \emptyset \tag{3.7a}$$

$$\text{FIRST}(t) = \{t\} \tag{3.7b}$$

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

$$\text{FIRST}(X) = \text{FIRST}(\gamma_1) \cup \ldots \cup \text{FIRST}(\gamma_n) \tag{3.7c}$$

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

$$\text{FIRST}(s\gamma) = \text{FIRST}(s) \cup (\text{if Nullable}(s) \text{ then FIRST}(\gamma) \text{ else } \emptyset \text{ fi}) \tag{3.7d}$$

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

Remark 23.1. The equations (Equations (3.7a) to (3.7d)) 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.
   - $\text{FIRST}(Y) = \{b\}$
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 $\text{FIRST}(Z)$, which yields $\{c, \epsilon\}$. We add both to the $\text{FIRST}(Y)$ list. Our list is now $\text{FIRST}(Y) = \{b, c, \epsilon\}$
   - Since B is an option in $p2$, we compute $\text{FIRST}(V)$, which yields $\{\epsilon\}$. We add this to the $\text{FIRST}(Y)$ list. Our list is now $\text{FIRST}(Y) = \{b, c, \epsilon\}$
6. Since ϵ is an empty string, we can remove it from the list, or just ignore it.
7. Once we have computed all possible $\text{FIRST}(Y)$ occurrences, we are done.

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

3.2.3 FOLLOW

Defn 24 (FOLLOW)**.** For the production p, where $p : X \to \gamma$, and X and γ 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 γ and δ are arbitrary sequences of terminals and nonterminals. You can define an equation system for FOLLOW given that $G = (N, T, P, S)$.

$$\text{FOLLOW}(X) = \bigcup \text{FOLLOW}(Y \to \gamma \underline{X} \delta) \tag{3.8a}$$

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

$$\text{FOLLOW}(Y \to \gamma \underline{X} \delta) = \text{FIRST}(\delta) \cup (\text{if Nullable}(\delta) \text{ then FOLLOW}(Y) \text{ else } \emptyset \text{ fi}) \tag{3.8b}$$

Remark 24.1. Again, the equations (Equations (3.8a) to (3.8b)) are recursive. Therefore, they might not terminate, so you must calculate this as another set of Fixed-Point Problems.

Remark 24.2 (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:

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

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 $X \rightarrow Y$ 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.

3.2.4 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 $p : X \rightarrow \gamma$.
2. Compute the token set FIRST(γ). Add p to each corresponding entry for X.
3. Check if γ 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 3.3. It uses the grammar below.

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

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

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

Table 3.3: LL(1) Table Example

3.2.4.1 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

However, the LL(1) table **DOES NOT** show that a grammar is ambiguous; it only shows that that Context-Free Grammar is not LL(1).

3.2.5 Issues with LL Parsing

There are 2 major issues with LL Parsing.

1. Common Prefix
2. Left Recursion

3.2.5.1 Common Prefix

Defn 25 (Common Prefix). A *common prefix* issue is one in which multiple productions that start from the same Nonterminal Symbol *start* with the same Nonterminal Symbol or Terminal Symbol.

The below grammar is an example of a *direct* Common Prefix issue:

$$X \to aY$$
$$X \to aZ \tag{3.9}$$

The below grammar is an example of an *indirect* Common Prefix issue:

$$Y \to Zb$$
$$Y \to Uc$$
$$Z \to a$$
$$U \to a \tag{3.10}$$

3.2.5.2 Left Recursion

Defn 26 (Left Recursion). A *left recursion* issue is one where a set of productions can eventually yield the same Nonterminal Symbol as was started with.

Remark 26.1. Every Left Recursion is a special case of Common Prefixs where the Nonterminal Symbols are the common element.

The below grammar is an example of a *direct* Left Recursion issue:

$$X \to XYZ \tag{3.11}$$

The below grammar is an example of an *indirect* Left Recursion issue:

$$X \to YZ$$
$$Y \to XD \tag{3.12}$$

In Equation (3.12), if you perform a left-most derivation, you will recurse down X permanently.

3.2.6 Eliminating Issues with LL Parsing

The goal when eliminating LL(1) parser issues is to generate an Equivalent Context-Free Grammar, while fixing the issues present in the original grammar.

Defn 27 (Equivalent Context-Free Grammar). Two Context-Free Grammars are said to be *equivalent* if they both produce the same Language. Technically, this is another unsolvable problem for Context-Free Grammars that generate an infinitely large Languages. However, you can show that they *may* be equivalent by using example cases.

3.2.6.1 Eliminate Common Prefix Since Common Prefix issues rely in two different productions starting with the same Terminal Symbol, if you combine the productions into one, and create a new production to handle the unique cases, then you can solve the issue.

Example 3.2: Eliminate Common Prefix. Exercise 13, Problem 4

The following grammar has a common prefix problem. Transform the grammar to an Equivalent Context-Free Grammar where the common prefix is eliminated.

$$p_0 : G \rightarrow \text{ElementList}$$
$$p_1 : \text{ElementList} \rightarrow \text{Element ElementList}$$
$$p_2 : \text{ElementList} \rightarrow \epsilon$$
$$p_3 : \text{Element} \rightarrow \text{Node}$$
$$p_4 : \text{Element} \rightarrow \text{Edge}$$
$$p_5 : \text{Node} \rightarrow \text{ID}$$
$$p_6 : \text{Edge} \rightarrow \text{ID ``('' ID ``}\rightarrow\text{'' ID ``)''}$$

- -

The Common Prefix in this grammar is an indirect one. If you follow p_3 and p_4, then p_5 and p_6 are the issue. We start by removing the redundant productions to make things a bit clearer.

$$p_0 : G \rightarrow \text{ElementList}$$
$$p_1 : \text{ElementList} \rightarrow \text{Element ElementList}$$
$$p_2 : \text{ElementList} \rightarrow \epsilon$$
$$p_3 : \text{Element} \rightarrow \text{ID}$$
$$p_4 : \text{Element} \rightarrow \text{ID ``('' ID ``}\rightarrow\text{'' ID ``)''}$$

The Common Prefix issue is more obvious now. Like said earlier, we can remove the issue by factoring the unique terms out to their own productions and leaving the common elements alone. In this case, that means we combine p_3 and p_4 and leave the ID alone. However, we factor out the unique elements in p_3 and p_4 with their own productions.

$$p_0 : G \rightarrow \text{ElementList}$$
$$p_1 : \text{ElementList} \rightarrow \text{Element ElementList}$$
$$p_2 : \text{ElementList} \rightarrow \epsilon$$
$$p_3 : \text{Element} \rightarrow \text{ID ElementRest}$$
$$p_4 : \text{ElementRest} \rightarrow \epsilon$$
$$p_5 : \text{ElementRest} \rightarrow \text{``('' ID ``}\rightarrow\text{'' ID ``)''}$$

3.2.6.2 Eliminate Left Recursion LL(1) parsers 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.

Example 3.3: Eliminate Left Recursion. Exercise 13, Problem 5

The following grammar is left-recursive. Transform the grammar into an Equivalent Context-Free Grammar.

$$p_0 : T \rightarrow T \text{ ``*''} F$$
$$p_1 : T \rightarrow F$$
$$p_2 : T \rightarrow \text{ID}$$
$$p_3 : T \rightarrow \text{``(''} T \text{ ``)''}$$

This is a simple case where we want to Eliminate Left Recursion. Since this is for a LL(1) parser, we can simply flip p_0 to make the new grammar right-recursive.

$$p_0 : T \rightarrow F \text{ ``*''} T$$
$$p_1 : T \rightarrow F$$
$$p_2 : T \rightarrow \text{ID}$$
$$p_3 : T \rightarrow \text{``(''} T \text{ ``)''}$$

This new grammar has a Common Prefix issue, but we leave that for the reader to solve. *Note: It is identical to Example 3.2.*

3.3 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 28 (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 28.1 (Accepting). Shifting over the EOF (End of File) marker, typically denoted $, is called *accepting* and causes the parser to stop successfully.

Defn 29 (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 3.4: LR1 Shift-Reduce Parsing.

Say you have a set of productions as follows:

$$p_1 : X \rightarrow YZV$$
$$p_2 : Y \rightarrow ab$$
$$p_2 : Z \rightarrow c$$
$$p_3 : V \rightarrow de$$

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.

$$\underbrace{\bullet}_{Stack} \underbrace{abcde}_{Queue}$$

2. Then you start pulling tokens off the front of the queue and putting them onto the stack with Shift actions.

$$\text{Shift}: \quad a \bullet bcde$$
$$\text{Shift}: \quad ab \bullet cde$$

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.

$$\text{Reduce}: \quad Y \bullet cde$$

4. Repeat this operation until you reach an Accepting action.

$$\text{Shift}: \quad Yc \bullet de$$
$$\text{Reduce}: \quad YZ \bullet de$$
$$\text{Shift}: \quad YZd \bullet e$$
$$\text{Shift}: \quad YZde\bullet$$
$$\text{Reduce}: \quad YZV\bullet$$
$$\text{Reduce}: \quad X\bullet$$
$$\text{Accept}$$

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.

3.3.1 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 30 (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 (3.13).

$$X \to \alpha \bullet \beta \quad t, s \tag{3.13}$$

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

- A *dot* ($\bullet$), corresponding to the position in the input sentence (and the token at the top of the stack).
- One or more possible *lookahead* terminal symbols: t, s.
    - We will use ? when the lookahead doesn't matter.

Remark 30.1. 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 (3.13)):

- The topmost part of the stack is α
- The first part of the remaining input is expected to match either, in no particular order:
    - βt
    - βs

3.3.2 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.

3.3.2.1 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.)

3.3.2.2 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.)

3.3.2.3 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.)

3.3.2.4 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.)

3.3.3 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)*, *LALR(1)*, parser. For example, compare the same states, but different parser types as shown in Table 3.4a and Table 3.4b.

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

3.3.4 Syntax Versus Semantics

There are some things that Context-Free Grammars cannot describe, and thus **CAN** parsed correctly, but make no sense **SEMANTICALLY**. 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

Defn 31 (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 32 (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.

	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

4.1 Parse Trees

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

Defn 33 (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 34 (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 convient 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 35 (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 35.1 (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).

Method	Pros	Cons
Ordinary Programming	Straightforward Code	No modular extension (All classes to be modified)
	Modular extension of subclasses	Tangled code, many things in same class
Visitors	Modular extension (Add new Visitor)	Lots of boilerplate code
		Limited modular extensibility of language
Aspect-Oriented Programming	Straightforward code	Cannot use plain Java
	Modular extension in operation dimension (new aspect)	No Separate compilation
	Modular extension of language (new subclass)	

Table 5.1: Semantic Analysis Methods

	Aspect-Oriented Programming	Visitors
Factor what?	Instace variables Methods Implements clauses	Only methods
Type safety	full type precision	May need typecasts
Method Parameters	Any number	Only 1
Ease of Use	Very simple	Clumsy, need boilerplate code
Arbitrary Module Composition	Yes	No Hard to extend factories to create visitors
Separate Compilation	Not for JastAdd	Yes
Mainstream OOP Lang?	No. Need JastAdd or equivalent	Yes, can use Java

Table 5.2: Visitors vs. Aspect-Oriented Programming

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 36 (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 5.3: Summary of the Visitors Pattern

5.2 Aspect-Oriented Programming

Visitors offer only a limited solution to our Expression Problem. Aspect-Oriented Programming offers a better, more fully realized solution to this problem.

Defn 37 (Aspect-Oriented Programming). *Aspect-Oriented Programming (AOP)* is similar to normal imperative programming, but where fields and methods can be factored out from classes and placed in aspect files. When the tool that is being used runs over these aspect files, the code that belongs to a class is "woven" into the class.

Remark 37.1 (Static vs. Dynamic Aspect-Oriented Programming). For this course, we are mostly using *static aspect-oriented programming*. Dynamic AOP has some very specific properties that we will not/did not use.

Defn 38 (Full Aspect-Oriented Programming). *Full aspect-oriented programming* focuses on dynamic behavior. There are 3 major things that define full aspect-oriented programming.

1. Joinpoint
2. Pointcut
3. Advice

Some example applications of full aspect-oriented programming are:

- Add logging of method calls in an aspect (Instead of adding print statements all over your code)
- Add synchronization code for basic code that is unsynchronized (Multi-threading/Multicore)

Defn 39 (Joinpoint). A *joinpoint* is a point execution where Advice code can be added.

Defn 40 (Pointcut). A *pointcut* is a set of Joinpoints that can be described in a simple way.

- All calls to the method `m()`
- All accesses of a variable `v`

Defn 41 (Advice). *Advice* is code you can specify in an aspect that can be added to Joinpoints, either before, after, or around the Joinpoint.

6 Reference Attribute Grammars

First, for this course, Karl Hallsby used the JastAdd program to use Reference Attribute Grammars. This means that if you use something different, then the syntax of these might change, but the concept stays the same. Most Reference Attribute Grammars support all of the elements (Sections 6.1 to 6.8) that are shown in this section.

Defn 42 (Reference Attribute Grammar). A *reference attribute grammar* or *RAG* is a formal way to define Attributes for nodes in the Abstract Parse Tree and associating them with values. In the case of JastAdd, these are written in a Declarative Programming style.

Defn 43 (Attribute). An *attribute* is similar to a key-value pair from a Hash-based data structure, except the value is an equation that returns some value. These are selectively attached to the various nodes in the Abstract Parse Tree. For example, there may be an attribute that attaches a reference from a variable use to the variable's declaration. There are many types of attributes, each of which is explained in this chapter.

Each attribute's equation does not necessarily have to be a small expression, it can be a whole method.

Defn 44 (Declarative Programming). *Declarative programming* is a programming paradigm that expresses the logic of a computation without describing its control flow. In short, this means programs describe their desired results without explicitly listing commands or steps that must be performed.

There are a few more points of interest when it comes to Declarative Programming.

- Declarative Programming describes what a computation should perform, not necessarily how it does it.

- Declarative Programming lacks side effects. Functions cannot change data that is visible outside of the function itself. For example:
    - Array elements cannot be updated. This can be skirted around by making a copy of the array.
    - Strings cannot be changed. Similar way to skirt the issue as with arrays.
    - Counters cannot be updated which is visible outside of the function.

    This means that every time a function is run, if the input is the same, the output will be the same ***EVERY TIME***.
- Declarative Programming resembles mathematical logic.

6.1   Synthesized Attributes

Defn 45 (Synthesized Attribute). A *synthesized attribute* is an Attribute where the equation is defined in the same node as the Attribute. These can be used when all the information to calculate the Synthesized Attribute is present in *that* node.

If there is information from other nodes that is needed, then there must either be a Reference Attribute, or the equation needs to be defined by something that knows about both objects required to calculate the Synthesized Attribute, which means it is an Inherited Attribute.

Synthesized Attributes are declared as:

```
1   syn T A.x();
```

- `syn`: Declare that this Attribute is a Synthesized Attribute
- `T`: Declare the type that this Synthesized Attribute should have
- `A`: Declare what type of Abstract Parse Tree node this Synthesized Attribute is on
- `.x()`: The name of this Synthesized Attribute
- These must be defined by an equation, as shown in Section 6.1.1.

Synthesized Attributes can also be declared as:

```
1   syn T A.x() = Java-expression;
```

- `syn`: Declare that this Attribute is a Synthesized Attribute
- `T`: Declare the type that this Synthesized Attribute should have
- `A`: Declare what type of Abstract Parse Tree node this Synthesized Attribute is on
- `.x()`: The name of this Synthesized Attribute
- `Java-expression`: An expression or method that is evaluated to calculate the value of this Synthesized Attribute.
    - This must return a value that is of the same type, `T`, as declared.

6.1.1   Defining Equations for Synthesized Attributes

Defining an equation for a Synthesized Attribute is done by:

```
1   eq A.x() = Java-expr;
```

- `eq`: Declare that this is an equation
- `A`: Declare what type of Abstract Parse Tree node this equation is on
- `.x()`: The name of the Synthesized Attribute that this equation belongs to
- `Java-expression`: An expression or method that is evaluated to calculate the value of this equation
    - This must return a value that is of the same type, `T`, as declared.

In order to define an equation for a Synthesized Attribute, all the information **must** be available in this node. If the information needed to calculate this Synthesized Attribute's value, then use:

1. A Reference Attribute.
2. An Inherited Attribute with an equation defined by a common parent.

6.2   Inherited Attributes

Defn 46 (Inherited Attribute). An *inherited attribute* is one that is declared on a node in the Abstract Parse Tree, but its equation is defined by a parent (ancestor) higher in the Abstract Parse Tree.

Inherited Attributes are declared as:

```
1   inh T A.y();
```

- `inh`: Declare that this Attribute is an Inherited Attribute.

- `T`: Declare the type that this Inherited Attribute holds
- `A`: Declare the Abstract Parse Tree node type that this Inherited Attribute is bound to
- `.y()`: Declare the name of this Inherited Attribute

and the Inherited Attribute must be defined by some common parent. This is shown in Section 6.2.1.

6.2.1   Defining Equations for Inherited Attributes

```
1  eq C.getA().y() = Java-expr;
```

- `eq`: Declare that this Attribute is an Inherited Attribute.
- `C`: Declare the parent Abstract Parse Tree node type that this Inherited Attribute is bound to
- `.getA()`: Declare the child Abstract Parse Tree node that is receiving this equation
- `.y()`: The name of the equation that matches some Inherited Attribute's name
- `Java-expression`: An expression or method that is evaluated to calculate the value of this equation
  - This must return a value that is of the same type, `T`, as declared.

6.3   Broadcasting Attributes

Defn 47 (Broadcasting). *Broadcasting* an Attribute is when an equation holds for some node and all of its children.

Remark 47.1. In JastAdd, Broadcasting only occurs on Attributes that declare an attribute for that equation. This means that if you declare an equation for an attribute in a parent node, it *will not* propagate down to the children in the tree unless the children have also declared this attribute/equation.

6.4   Reference Attributes

Defn 48 (Reference Attribute). A *reference attribute* is a reference/pointer to another Abstract Parse Tree node. These are done by using equations for Synthesized Attributes or Inherited Attributes. However, instead of returning a string, integer, etc., an Abstract Parse Tree node (Java object) is returned. This gives us a pointer to that "Java object".

6.5   Parameterized Attributes

Defn 49 (Parameterized Attribute). A *parameterized attribute* is one where the Attribute takes in some parameters to achieve a goal. They are defined the same way as Synthesized Attributes and Inherited Attributes, but they also take parameters.

6.6   Nonterminal Attributes

Defn 50 (Nonterminal Attribute). A *nonterminal attribute* is one in which the Attribute in "nonterminal". This means that a whole new node, or potentially a whole new subtree, is created.

There are some perks of using a nonterminal attribute in the Reference Attribute Grammar:

- Nonterminal Attributes can be Inherited Attributes or Synthesized Attributes.
- The value in the equation should be a freshly built Abstract Parse Tree subtree. It should be complete in the sense that all its children should also be freshly created nodes (i.e., they are not allowed to be initialized to null).
- The Nonterminal Attribute can itself have Attributes that can be accessed like normal.
- If the Nonterminal Attribute has inherited attributes, there must be equations for those attributes in some ancestor, as for normal children.

In JastAdd they are defined as:

```
1  syn nta C A.anNTA() = new C();
```

- `syn`: Declares that this Nonterminal Attribute is a Synthesized Attribute
- `nta`: Declares that this Attribute is a Nonterminal Attribute
- `C`: Declare the type that this Nonterminal Attribute is to have
- `A`: Declare the type of Abstract Parse Tree node that this Nonterminal Attribute is to be on
- `.anNTA()`: The name of this Nonterminal Attribute
- `new C()`: The starting value that the Nonterminal Attribute is to have

6.6.1 Parameterized Nonterminal Attributes

Nonterminal Attributes can have parameters, just like normal Parameterized Attributes. These are declared as:

```
syn nta C A.anNTA(P param) = new C();
```

- `syn`: Declares that this Nonterminal Attribute is a Synthesized Attribute
- `nta`: Declares that this Attribute is a Nonterminal Attribute
- `C`: Declare the type that this Nonterminal Attribute is to have
- `A`: Declare the type of Abstract Parse Tree node that this Nonterminal Attribute is to be on
- `.anNTA()`: The name of this Nonterminal Attribute
- `P param`: The type and name of the parameter that is passed to the Nonterminal Attribute.
- `new C()`: The starting value that the Nonterminal Attribute is to have.

6.7 Collection Attributes

Defn 51 (Collection Attribute). A *collection attribute* is an Attribute that is defined by a set of Contributions instead of an equation. These are special attributes that are attached to specific nodes in the Abstract Parse Tree that can have specific things added to them with the Contribution equations.

```
coll T A.c() [fresh]
    [with m]
    [root R];
```

- `T`: type of the attribute. Usually T is a subtype of `java.lang.Collection`.
- `A`: Abstract Parse Tree class on which the attribute is evaluated (put on).
- `.c()`: Declares the attribute name, in this case c.
- `fresh` (Optional): How the Collection Attribute is initialized. The Java expression `fresh` creates an empty instance of the result type, `T`. This part is optional if `T` is a concrete type with a default constructor, if it is omitted the default constructor of the type `T` is used, i.e. `new T()`.
- `with m` (Optional): Specifies the name of a method to be used for updating the Collection Attribute object. This part is optional and the default method `add` is used if no method `m` is specified. The update method must fulfill these requirements:
  - The method `m`, should be a one-argument method of `T`. It only takes one argument, the thing being updated.
  - The method `m` should mutate the `T` object by adding *one* object to it.
  - The method `m` should be commutative, in the sense that the order of calling `m` for different contributions should yield the same resulting `T` value.
- `root R` (Optional): Declares the Collection Attribute root type, `R`. The collection mechanism starts by finding the nearest ancestor node of type `R` for the `A` node which the collection attribute is evaluated on. The subtree rooted at that nearest `R` ancestor is searched for contributions to `A.c()`, this means that the collection is scoped to the subtree of `R`, and contributions outside that tree are not visible.
  - This allows you to have multiple Collection Attributes in a single Abstract Parse Tree by placing the attribute on an arbitrary root type.
  - If you do this, only the tree beneath that node type will be able to access the Collection Attribute.

Remark 51.1. These are the main way to introduce mutable objects to the Reference Attribute Grammar. However, because the contributions cannot have side-effects, this technically fits into the Declarative Programming paradigm.

Defn 52 (Contribution). A *contribution* is a special equation that adds or mutates a value inside of the Collection Attribute. There are 3 main ways to contribute to a Collection Attribute:

1. Contribute a single value to a single Collection Attribute. This is the most commonly used one.

```
N1 contributes value-expression
    [when conditional-expression]
    to A.c()
    [for A-reference-expression];
```

- `N1`: The type of Abstract Parse Tree node that is providing the Contribution.
- `value-expression`: Java expression that evaluates to an object to be added to the intermediate collection of the target collection attribute.
- `when conditional-expression` (Optional): the contribution is only added to the target collection attribute if the Java expression cond-exp evaluates to `true`.

- **A**: Node type where the target collection attribute is declared. Matches the node type declared for the Collection Attribute.
- **.c()**: The name of the target Collection Attribute.
- **for A-reference-expression** (Optional): Java expression which evaluates to a reference to the AST node that owns the collection attribute this contribution is contributing to. This is the target expression, and it can be omitted if the target node is identical to the collection root node.
  - Essentially, if the Collection Attribute is not located on the `N1` node type, then we have to give a pointer to what node the Collection Attribute is actually contained.
  - This means there must be an Attribute that has a pointer to the node type (Reference Attribute) that does contain the Collection Attribute.
  - It is likely that the Pointer will have to be an Inherited Attribute.

2. Contribute a single value to multiple target Collection Attributes.

```
1    N1 contributes value-expression
2        [when cond-expression]
3        to A.c()
4        [for each A-reference-set];
```

- **N1**: The type of Abstract Parse Tree node that is providing the Contribution.
- **value-expression**: Java expression that evaluates to an object to be added to the intermediate collection of the target collection attribute.
- **when conditional-expression** (Optional): the contribution is only added to the target collection attribute if the Java expression cond-exp evaluates to `true`.
- **A**: Node type where the target collection attribute is declared. Matches the node type declared for the Collection Attribute.
- **.c()**: The name of the target Collection Attribute.
- **for each A-reference-set** (Optional): Java expression that evaluates to an `Iterable<A>` containing references for the set of contribution target nodes.
  - This is just an extension of the `A-reference-expression`.
  - Instead of a single `A-reference-expression`, there is an `Iterable` collection of `A-reference-expressions`.

3. Contribute multiple values to multiple target Collection Attributes.

```
1    N1 contributes each value-iterable
2        [when conditional-expression]
3        to A.c()
4        [for each A-reference-set];
```

- **N1**: The type of Abstract Parse Tree node that is providing the Contribution.
- **each value-expression**: This syntax works if `value-expression` has the type `Iterable<E>` where E is the element type of the Collection Attribute.
  - For example, if the collection attribute is declared as `coll LinkedList<String>` ... then `value-iterable` should have the type `Iterable<String>`.
  - Meaning the `value-iterable` should be an `Iterable` collection.
- **when conditional-expression** (Optional): the contribution is only added to the target collection attribute if the Java expression cond-exp evaluates to `true`.
- **A**: Node type where the target collection attribute is declared. Matches the node type declared for the Collection Attribute.
- **.c()**: The name of the target Collection Attribute.
- **for each A-reference-set** (Optional): Java expression that evaluates to an `Iterable<A>` containing references for the set of contribution target nodes.
  - This is just an extension of the `A-reference-expression`.
  - Instead of a single `A-reference-expression`, there is an `Iterable` collection of `A-reference-expressions`.

6.8   Circular Attributes

Defn 53 (Circular Attribute). A *circular attribute* is one in which an Attribute depends on itself, either directly or indirectly. Since these are Fixed-Point Problems, circular attributes are calculated iteratively. Thus, they must have some starting value.

In JastAdd, circular attributes are defined as:

```
1    syn T A.x([P param]) circular [bv];
2    eq A.x([P param]) = rv;
```

or

```
1    syn T A.x([P param]) circular [bv] = rv;
```

- `T`: The type of things being stored/calculated in the Circular Attribute.
- `A`: The type of Abstract Parse Tree node that that the Circular Attribute is on.
- `.x()`: The name of the Circular Attribute.
- `[P param]`: Parameters that may be needed to calculate the value in the definition
- `circular`: Declare that this attribute is *supposed* to be circular and inform JastAdd that this is supposed to be calculated with fixed-point iteration.
- `[bv]`: The starting value that is used for the fixed-point iteration.
- `rv`: The expression that will define the Circular Attribute.

7   Runtime Systems

Before we start discussing Code Generation, we should look at what the end result of our generation should be.

Remark. It is important to note that *function*, *method*, and *procedure* may be used interchangeably. While they are all technically different, for our case here, they all act the same. They are just named blocks of code that may take in some additional arguments that act as if they were local variables. **In this case, pass-by-reference and pointers are not handled, discussed, or supported in any way!**

There are 2 different portions of code:

1. A Read-Only portion

   - The read-only portion is called the "text" of the program.
   - This is the actual code in the program.

2. A Read/Write portion

   - This is the "data" portion of the program.
   - Global variables are here.
     - Global in this sense means global to *this* program, not global to **all** programs.
   - The Heap is here
   - The Call Stack is here
   - The Class Descriptor (If the programming language supports object-oriented programming).

Defn 54 (Heap)**.** The *heap* is a memory organization construct that helps visualize the way memory is used during the execution of a program. It is **_SIGNIFICANTLY_** larger than the Call Stack.

In an object-oriented language, this is where instances of classes exist. These objects are allocated on a first-come first-serve basis. When the object is allocated, the amount of memory that the object requires is allocated **_in a continuous block_**. The problem is that if the heap has been extensively used, then there might be enough total memory required to allocate an object, but because it is not in a continuous block, the object cannot be allocated.

This makes deallocation more complex, because when they are deallocated, the program might still view the memory as in-use. Additionally, because of the continuous block allocation nature of the heap, it must be organized every once in a while. This is called Garbage Collection.

Remark 54.1. In an object-oriented language, like Java, when objects are stored on the heap, **ONLY** their class fields are stored there. The class methods are stored elsewhere, in the Class Descriptor.

Defn 55 (Call Stack)**.** The *call stack* is an imaginary construct that resembles the traditional stack data structure. It is a way to visualize and organize the way memory is used during the execution of a program and its function/methods. It is filled with Stack Frames. This **does not** hold the code that is used in the function, rather it is everything that is needed for the function to be able to run.

Defn 56 (Stack Frame)**.** A *stack frame*, *activation frame*, or *activation record* are objects that represent necessary portions of a function. These include:

- A Dynamic Link
- Local Variables
- Temporary Variables
- Static Links
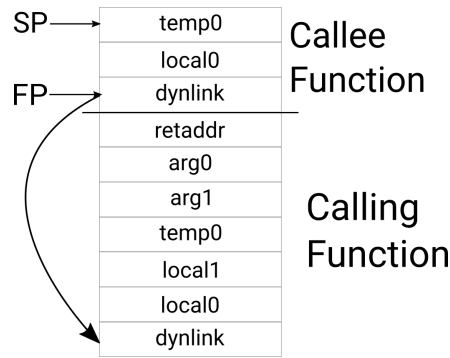- Function Arguments
- Return Address

Figure 7.1: Stack Frame

Additionally, there are 2 registers used as pointers to move around and interact with the stack frame.

1. FP is in register %rbp. It is the Frame Pointer.
2. SP is in register %rsp. It is the Stack Pointer.

Defn 57 (Dynamic Link). The *dynamic link* or *dynlink* is a memory address pointer and sits at the bottom of a Stack Frame. It is a pointer back to the previous function's dynamic link. This ensures that any function can find its parent/calling function.

The dynamic link also serves as a means to access any variable that might be needed by this function. To access any variable in *this* function, you can subtract a byte multiple that you need to access the proper value. To access any variable in the calling function, you can add a byte multiple that would correspond to the proper variable.

In the x86_ 64 architecture, instruction set, and convention that we used, the register %rbp was the dynamic link.

Remark 57.1. Note: Due to the conventions we used, when accessing arguments passed to the function, we treated them as local variables, just further down in the call stack. This also means that we need to skip over the Return Address block in memory.

Remark 57.2. Because we used the %rbp register to store our current Dynamic Link's address, the dynamic link might also be called the *base pointer*.

Defn 58 (Local Variable). *Local variables* are handled very simply. They get an appropriate amount of memory allocated to them on the stack, and that is it.

There is no way to give a variable a name in assembly. (Usually. Depends on the architecture and instruction set). However, there is no way to name something in memory. But, because the size of all the objects is known at compile-time, allocating the proper amount of memory required by each variable is possible.

Remark 58.1. This holds true for strongly-typed, static, compiled languages, like Java, C, C++, etc. However, Python is slightly different in this regard, and handles it differently. That is discussed further in Section 8.1, Intermediate Code.

Defn 59 (Temporary Variable). A *temporary variable* is one that is allocated on this function's Stack Frame while calculating values. Once the calculations are completed, these values are deallocated. These temporary variables can also point to objects on the Heap. When the function has finished running, then these values are deallocated, along with all other Local Variables in use.

For example, since assembly-level addition only allows for 2 operands, but in general, addition can have more than 2 operands in use, there needs to be a way to store the value used in the addition. While we can accumulate and use that value in the addition, the values being added together are *not* modified.

Defn 60 (Static Link). The *static link* is an implicit argument, meaning it ***ALWAYS*** gets pushed onto the stack as a Function Argument, when appropriate.

Appropriate in this context could mean several things:

- When an object is alive, and when it is being acted on by a function.
  - In this case, the static link points to an instance of a class on the Heap.
- When a language allows for nested function declarations.
  - Then the static link points to the Dynamic Link of the outer function
  - This allows access to the outer function's Local Variables like normal, and allows us to go back later.

Defn 61 (Function Argument). *Function argument*s are handled very simply. If a function call takes an argument, then the argument is calculated, and then that argument is pushed onto the stack in *this* Stack Frame.

Remark 61.1. If more than one argument is passed to a function, there are 2 ways to push values onto the Stack Frame:

1. In the order they are passed to the function

   - Say a function with 3 arguments is called, then the stack would have arguments in this order
   (a) argument0 (Lowest memory address)
   (b) argument1
   (c) argument2 (Highest memory address)

2. In reverse order

   - Say the same function is called with the same 3 arguments, then the stack would have arguments in this order
   (a) argument2 (Lowest memory address)
   (b) argument1
   (c) argument0 (Highest memory address)

When the values that were passed need to be accessed, and if the memory sizes of things are known at compile time, then we can calculate how far down we need to go in the stack to find the value. This is done by adding a positive value to the `%rbp`

Defn 62 (Return Address). The *return address* is used by the `%rip` register. It is calculated and pushed onto the Stack Frame stack when the `CALL` macro is used. It is the thing that allows us to jump around in the code from the `text` area of our program.

Remark 62.1. The `%rip` register is the *register instruction pointer*. It holds the value of the *next* instruction to execute. Technically, it is the Program Counter's value.

Defn 63 (Garbage Collection). *Garbage collection* is the act of deallocating objects that may still be on the heap and organizing the heap. Since the heap allocates continuous "blocks" of memory required by an object, the heap may have the necessary memory to allocate an object, but in discontinuous locations.

Defn 64 (Frame Pointer). The *frame pointer* is a pointer that **ALWAYS** points to the current function's Dynamic Link. The frame pointer is commonly abbreviated as *FP*. This is the thing that allows us to access Local Variables, Function Arguments, and everything else inside of the Stack Frame. The value is held in the `%rbp` register.

Defn 65 (Stack Pointer). The *stack pointer* is a pointer that **ALWAYS** points to the top of the current Stack Frame. The stack pointer is commonly abbreviated as *SP*. This value is held in the `%rsp` register. This is the pointer that allows us to push and pop onto this function's Stack Frame.

Defn 66 (Class Descriptor). The *class descriptor* is a portion of memory set aside for the methods that are in an object.

7.1 Making a Function Call

There are several steps that must be followed:

1. Push the arguments onto the calling function's Stack Frame.
2. Push the Return Address. This is handled by the `CALL` instruction.
3. Jump to the called method. This is also handled by the `CALL` instruction.
4. Push the Frame Pointer's current value onto the Stack.
5. Move the Stack Pointer to the newly pushed Frame Pointer.
6. Run the code for the called function.
7. Put the value to be returned in the `%rax` register
8. Move the Stack Pointer to the Frame Pointer to deallocate all the values on the called function's stack
9. Move the value where the Frame Pointer is into the Frame Pointer, moving the Frame Pointer to the calling function's Dynamic Link.
10. Pop the called function's Dynamic Link off. (The Stack Pointer is pointing to this value).
11. Pop the Return Address off and put the value into the `%rip` register. (The `RETURN` instruction handles this).
12. Pop the Function Arguments off.
13. Continue the execution of the calling function.

7.2 What Does the Compiler Compute?

The compiler has to compute a couple of things:

1. For uses of locals and arguments

   - The offsets to use (Relative to the Frame Pointer).

2. For methods

- The space needed for Local Variables and Function Arguments
- We typically use `PUSH` and `POP` for the allocation and deallocation of variable of Temporary Variables.

3. If nested methods are supported

- The number of Static Link levels to use for variable accesses (0 for Local Variables)
- The number of Static Link levels to use for method calls (0 for local methods)

7.3 Compiler Function Calling Conventions

- Caller-save Register: The calling function must save their registers before calling a function
- Callee-save Register: The called function must save registers that it will use, and restore them before the called function exits.
- Argument Order: Put Function Arguments onto the calling function's Stack Frame in what order? Forwards or Backwards?
- Direction: Let the Call Stack grow towards larger or smaller addresses?
- Allocation of Space for Local Variables and Temporary Variables: Push the variables in one big chunk, or push them one at a time?

7.4 Optimizing the Generated Program

There are multiple ways to optimize the assembly program that the compiler generates:

1. Store data in registers instead of in memory

- The return value (Which we are already doing)
- As many function arguments as possible
- The Static Link
- The Return Address

- If a function call is made, then the registers must not be corrupted

2. Inline some portions of code.

- Some variable values that are not modified can have their values inlined to later instructions

8 Code Generation

Code generation is the last step of the compilation process. The assembly code is also optimized in this step, before the code is generated, in the Intermediate Code portion.

8.1 Intermediate Code

There are 2 common forms of intermediate code generation:

1. Three-Address Code
2. Stack Code

8.1.1 Three-Address Code

The Three-Address Code is the type of Intermediate Code used for static, compiled languages. An example of Three-Address Code is shown below. They have these elements on common:

- Each instruction has 3 operands, in the form of

  `op src1 src2 dest`
- Temporary Variables are frequently used
- Very related to a register-based machine
- ***VERY GOOD FOR OPTIMIZATION***
  - Removing unnecessary instructions for operations
  - Keep as many values in registers rather than memory

```
1      ADD v1 v2 t1
2      JEQ t1 v3 L1
3      SUB v3 1 t2
4      MOV t2 c1
5    L1:
6      MOV v2 v4
```

8.1.2   Stack Code

The other type of Intermediate Code is more commonly used for dynamic languages. These have these elements in common:

- The use of a *value stack* instead of Temporary Variables
- Each instruction needs the necessary operands popped, performs the operation, then pushes the result
- Commonly used for interpreters and virtual machines
- Similar to LR Parsing
    - Push the operands/arguments onto the value stack
    - Pop the appropriate number of arguments off the top of the value stack when performing an operation
    - Perform the operation and push the result

```
1     PUSH v1
2     PUSH v2
3     ADD
4     PUSH v3
5     JEQ L1
6     PUSH v3
7     PUSH 1
8     SUB
9     POP v1
10    L1:
11    PUSH v2
12    POP v4
```

8.2   Target Code Generation

The assembly code that we generate in this course is completely **UNOPTIMIZED**. Thus, the assembly code that we generated was nearly identical to the Three-Address Code.

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.