Chapter 4 Grammars and Parsing

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: August 16, 2004

current version: January 25, 2011

©January 25, 2011 by Wuu Yang. All rights reserved.

Chapter outline: Grammars and parsing

1. Context-free grammars
2. Properties of CFGs
3. Transforming extended grammars
4. Parsers and recognizers
5. Grammar analysis algorithms: useless nonterminals, ambiguity, *first* sets, *follow* sets

§4.1. Context-free grammars

The input to a parser is a sequence of tokens. The parser has to find the structure in the token sequence.

The structure (or syntax) is specified by a context-free grammar.

A *context-free grammar* (CFG) consists of four ingredients: $G = (V_t, V_n, S, P)$, where V_t (or Σ) is a finite, non-empty set of *terminals*, V_n (or N) is a finite, non-empty set of *nonterminals*, $S \in V_n$ (called the *start symbol*), P is a finite, non-empty set of *production rules*.

We assume that V_t and V_n are disjoint ($V_t \cap V_n = \emptyset$). The union of V_t and V_n is called the *vocabulary*, i.e., $V_t \cup V_n = V$.

A rule has the form

$$A \rightarrow X_1 X_2 \ldots X_m$$

where $A \in V_n$, $m \geq 0$, and each $X_i \in V_t \cup V_n$. That is, the left-hand

side is a single nonterminal and the right-hand side is a (possibly empty) string of terminals and nonterminals.

A rule whose left-hand side is N is called an N-rule. E.g., $N \rightarrow \ldots$.

We start from a single start symbol and keep on replacing a nonterminal N with the right-hand side of an N-rule until there is no non-terminal left. The resulting string of terminals is called a *sentence* of the grammar. The set of all the sentences defined by a grammar is called the *language* defined by the grammar.

The *language* defined by the grammar G is defined as

$$L(G) = \{u \in V_t^* | S \Rightarrow^* u\}$$

Note that a language is a (possibly infinite) set of finite-length strings that are composed of terminals.

Some terminologies:

- sentence: $u \in V_t^*$ such that $S \Rightarrow^* u$.
- sentential form: $\alpha \in (V_t \cup V_n)^*$ such that $S \Rightarrow^* \alpha$.
- phrase: $\alpha \in (V_t \cup V_n)^*, A \in V_n$ such that $A \Rightarrow^* \alpha$.
- simple phrase: $\alpha \in (V_t \cup V_n)^*, A \in V_n$ such that $A \Rightarrow^1 \alpha$.
- handle of a sentential form: leftmost simple phrase of the sentential form.

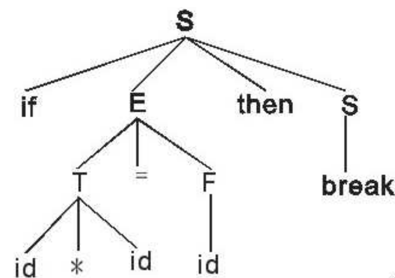
Parsing a sentence is essentially finding the *derivation* of the sentence.

Any string of terminals and nonterminals derived from the start symbol is called a sentential form. A sentence is a sentential form that contains only terminals (and hence cannot be further derived). A sentential form may contain one or more phrases. These phrases are *properly nested* in the sense that either one phrase is totally

contained in another or two phrases are disjoint.

A simple phrase is identical to the right-hand side of a rule. A sentential form may contain one or more simple phrases. These simple phrases are disjoint from one another. The leftmost (first) simple phrase is called the handle of a sentential form. The LR parser works by finding the handles.

Ex.



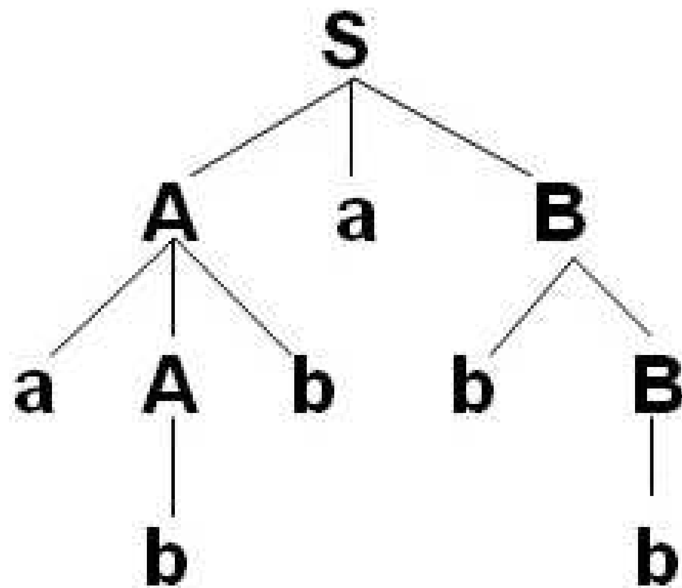
a sentence: $if\ id * id = id\ then\ break$

a sentential form: $if\ id * id = F\ then\ break$

a phrase: $id * id = F$

two simple phrases: $id * id$; $break$

Ex.



abbabb - sentence

aAbabB
- sentential form

aAb - handle

bB - simple phrase

abb - phrase of A

Example. $S \Rightarrow AB \Rightarrow pXpB \Rightarrow pXpqYq \Rightarrow prpqYq \Rightarrow prpqrq$

§4.1.1 Derivations and parse trees

Each sentence may have several derivations; but it should have a unique parse tree, e.g, $ID + (ID + ID)$

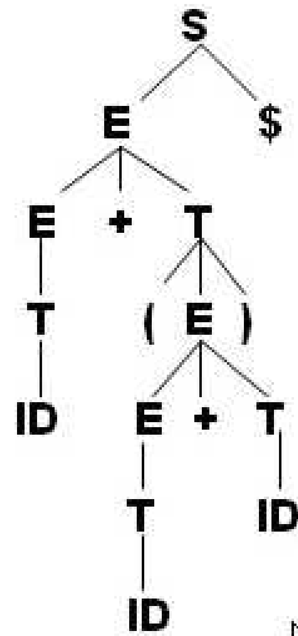
$$
\begin{aligned}
\text{leftmost: } S &\Rightarrow E\$ \\
&\Rightarrow E + T\$ \\
&\Rightarrow T + T\$ \\
&\Rightarrow ID + T\$ \\
&\Rightarrow ID + (E)\$ \\
&\Rightarrow ID + (E + T)\$ \\
&\Rightarrow ID + (T + T)\$ \\
&\Rightarrow ID + (ID + T)\$ \\
&\Rightarrow ID + (ID + ID)\$
\end{aligned}
$$

$$
\text{rightmost: } S \Rightarrow E\$
$$

$$
\begin{aligned}
&\Rightarrow \quad E + T\$ \\
&\Rightarrow \quad E + (E)\$ \\
&\Rightarrow \quad E + (E + T)\$ \\
&\Rightarrow \quad E + (E + ID)\$ \\
&\Rightarrow \quad E + (T + ID)\$ \\
&\Rightarrow \quad E + (ID + ID)\$ \\
&\Rightarrow \quad T + (ID + ID)\$ \\
&\Rightarrow \quad ID + (ID + ID)\$
\end{aligned}
$$

Both leftmost and rightmost derivations take the same number of steps, but different order.

The rightmost derivation makes use of the handles in every step.



Exercise. Consider the grammar:

$$S \rightarrow E \ \ \$$$
$$E \rightarrow E \ \ + \ \ T \ \mid \ T$$
$$T \rightarrow ID \ \mid \ (\ \ E \ \ )$$

Show the leftmost and rightmost derivation of the sentence "$(ID) + (ID)\$$" (and also a third derivation).

§4.1.4 Other types of grammars

Σ^* is the set of all finite-length strings composed of terminals. So Σ^* is also a language. Note that every language is a subset of Σ^* according to our definition. There are many classes of languages. A language that can be defined with a context-free grammar is called a context-free language. There are languages that are not context-free.

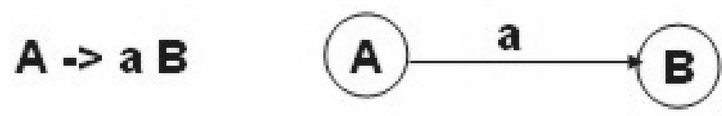
regular grammars

All rules are of the form:

$$A \rightarrow aB$$
$$A \rightarrow$$

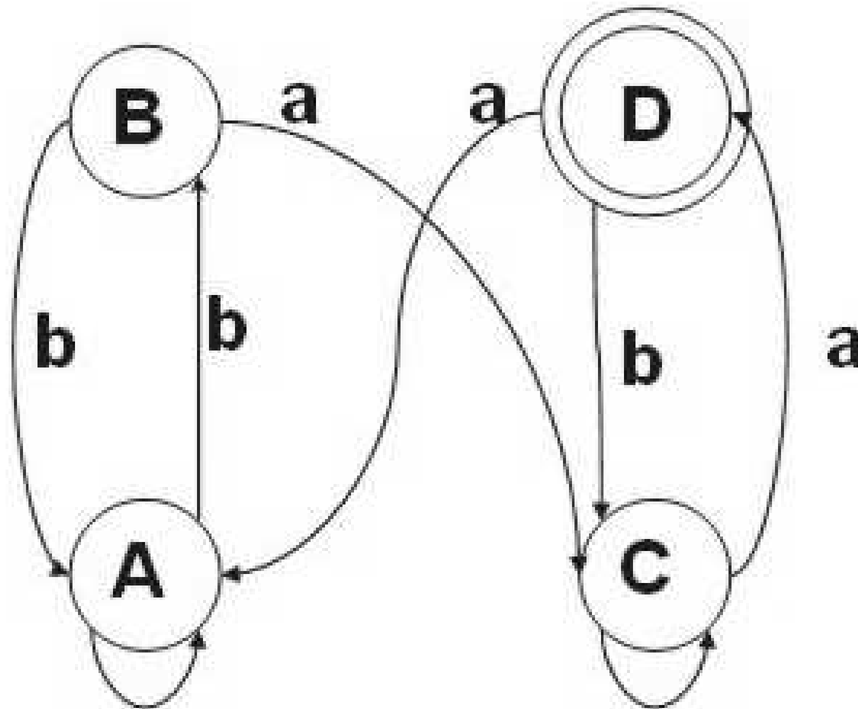
Lemma. Regular grammar = regular expression.

Why? Transform a regular grammar into an fa. Each nonterminal as a state.



However, CFG can define the set of balanced parentheses. $\{(^i)^i | i \geq 1\}$. Therefore, CFG $\supset$ regular expressions.

Ex. Derive a CFG from an FA.



A -> aA | bB
B -> aC | bA
C -> aD | bC
D -> aA | bC | λ

A is the Initial state.

Not all syntactic rules are expressible with CFG, e.g. variables must be declared before being used. These are considered static semantics.

Context-sensitive grammars: Production rules may have the form: $\alpha A \beta \rightarrow \alpha \delta \beta$, where $A \in V_n$ and $\alpha, \beta, \delta \in (V_t \cup V_n)^*$.

Chomsky's hierarchy

type 0 grammars: no restrictions, $\alpha \rightarrow \beta$.
type 1 grammars: context-sensitive grammars, $\alpha A \beta \rightarrow \alpha \delta \beta$.
type 2 grammars: context-free grammars, $A \rightarrow \alpha$.
type 3 grammars: regular grammars, $A \rightarrow aB$.

Though context-sensitive and type-0 grammars can define many languages that cannot be defined with a context-free grammar, they lack efficient parsers and it is more difficult to infer their properties. So they are seldom used in practice.

CFG is a nice balance between expressive power and ease of

parsing.

§4.2 Properties of CFGs

useless nonterminals: nonterminals that derive no terminal strings or are unreachable from the start symbol.

Consider the grammar:

$$
\begin{aligned}
&S \rightarrow A \ \mid \ B \\
&A \rightarrow a \\
&B \rightarrow b \ \ B \\
&C \rightarrow a
\end{aligned}
$$

B and C are useless.

Useless symbols may be removed without affecting the language defined by the grammar. CFGs without useless nonterminals are called *reduced* grammars.

ambiguous grammars: A sentence may have more than one parse tree. Consider the grammar:

$$E \rightarrow E - E$$
$$E \rightarrow ID$$

The sentence $ID - ID - ID$ has two distinct parse trees.

Ambiguous grammars are not used in defining a programming language since modern programming languages are defined in terms of its syntax.

In general, it is not possible to decide whether a CFG is ambiguous. However, for certain classes of CFGs, we can.

grammar equality (similar)

We will frequently transform a grammar into another (for ease of parsing), hoping that the transformation will not affect the language defined by the grammar.

In general, it is undecidable whether two grammars are equivalent. However, for simple transformations, we can decide two grammars are equivalent.

§4.3. Extended BNF $\Rightarrow$ BNF

For $\boxed{A \rightarrow \alpha\ [\beta]\ \gamma}$, use

$$A \rightarrow \alpha N \gamma$$
$$N \rightarrow \beta$$
$$N \rightarrow \lambda$$

For $\boxed{A \rightarrow \alpha\ \{\beta\}\ \gamma}$, use

$$A \rightarrow \alpha N \gamma$$
$$N \rightarrow \beta N$$
$$N \rightarrow \lambda$$

§4.4 Parsers and Recognizers

recognizers: answer whether the input is a sentence (YES/NO)

parsers: identify the structure of a sentence

top-down, predictive, pre-order, lm, LL

bottom-up, post-order, rm(reverse), LR

1 lookahead

Recognizers are usually smaller and easier.

Consider the following grammar:

```
Prog  ::= begin Stmts end $
Stmts ::= Stmt ; Stmts
Stmts ::= λ
Stmt  ::= simplestmt
Stmt  ::= begin Stmts end
```

Assume the input is

```
begin simplestmt ; simplestmt ; end $
```

The following two figures show top-down and bottom-up parsing techniques.

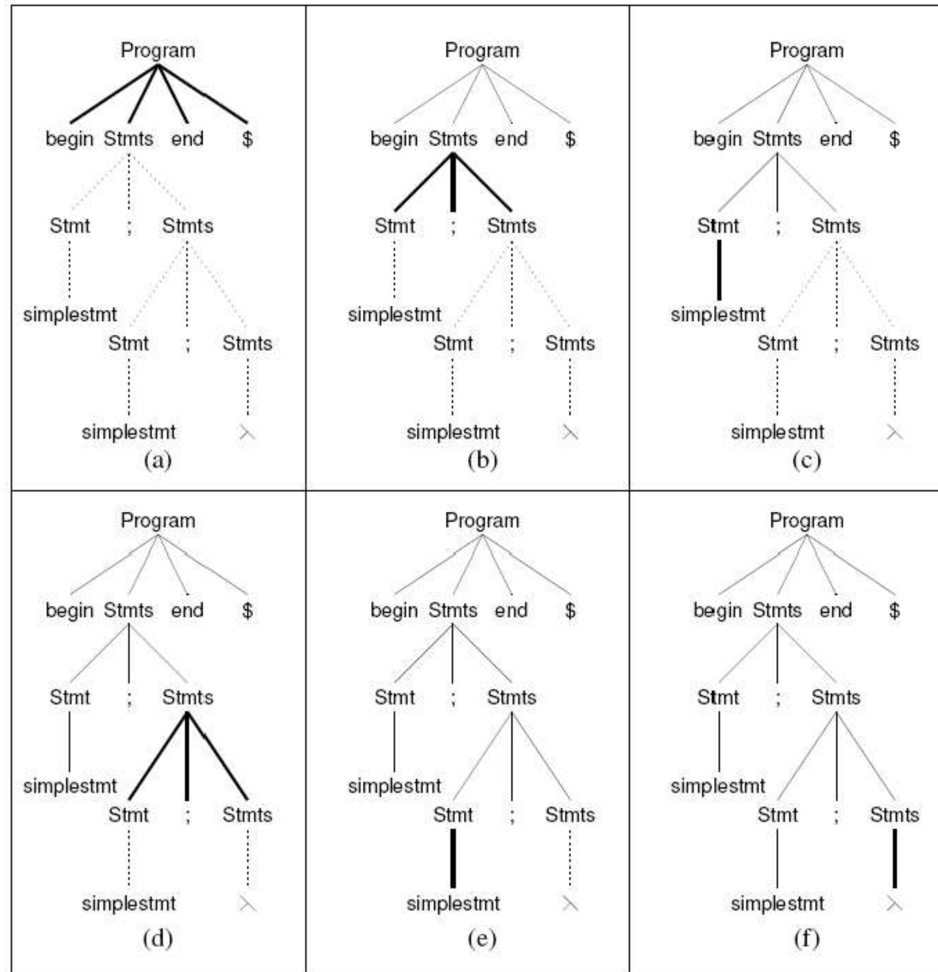


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

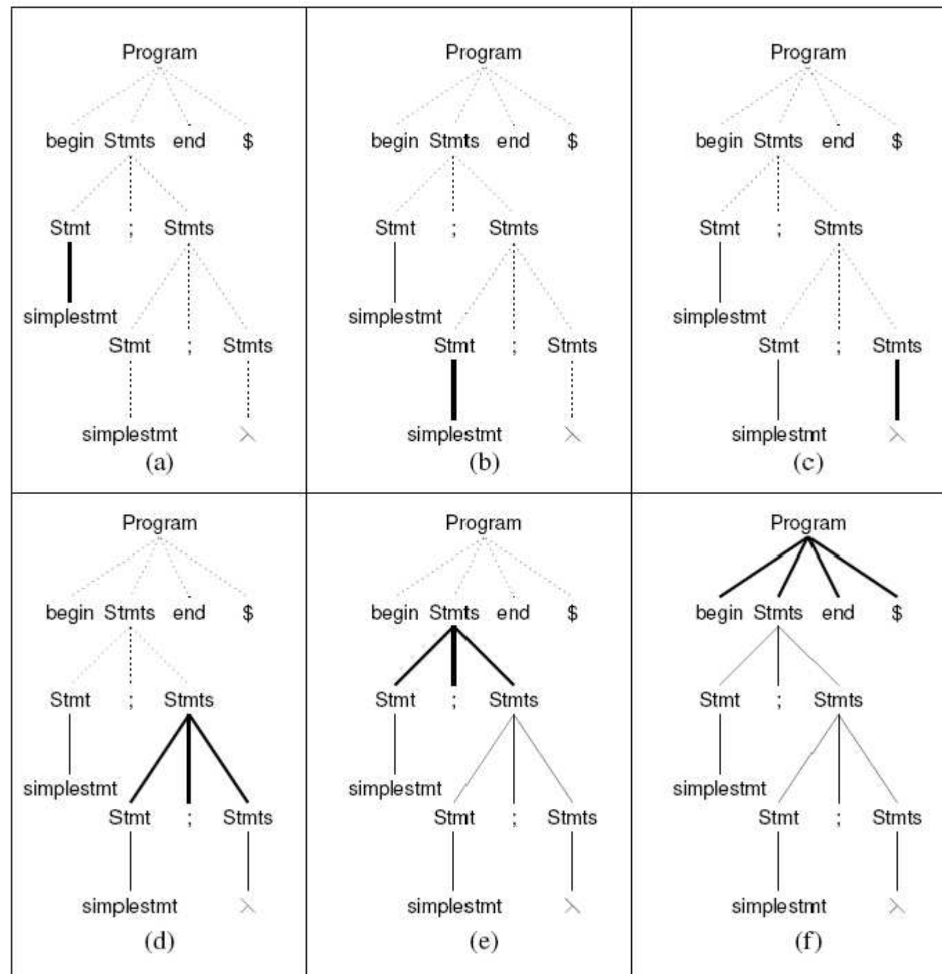


Figure 4.6: Parse of “begin simplestmt ; simplestmt ; end $” using the bottom-up technique. Legend explained on page 126.

§4.5. Grammar analysis algorithms

§4.5.2 Nullable nonterminals

Does $A \Rightarrow^* \lambda$? (Show an example.)

1. $Nullable := \emptyset$;
2. **repeat** {
3. **for** each rule P **do** {
4. **if** RHS of P contains no terminals **and** all
5. nonterminals on P's RHS are in $Nullable$
6. **then** add P's LHS to $Nullable$
7. }
8. } **until** no more nonterminal can be added to $Nullable$;

Here S is the set of nonterminals that may become null (*nullable nonterminals*).

Ex. Decide the nonterminals that derive λ for the following grammar.

$$T \rightarrow ABC$$
$$A \rightarrow DE$$
$$B \rightarrow$$
$$B \rightarrow bA$$
$$C \rightarrow BE$$
$$D \rightarrow dBE$$
$$D \rightarrow$$
$$E \rightarrow$$

Add the following nonterminals to S, in the order listed:
B, D, E; A, C; T.
The grammar is examined four times.

In general, at most how many times will the `repeat` loop be executed?

Answer. At most $|V_n|$ times.

Computing *first* sets and *follow* sets

$first(A) = \{b \in V_t | A \Rightarrow^* b \ldots\} \cup \{\lambda \mid \text{if } A \Rightarrow^* \lambda\}$

$b \in first(A)$ means that the terminal b is the first symbol in a phrase derived from the nonterminal A. We may extend the definition of $first$ to a string of symbols:

$first(\alpha) = \{b \in V_t | \alpha \Rightarrow^* b \ldots\} \cup \{\lambda \mid \text{if } \alpha \Rightarrow^* \lambda\}$

$follow(A) = \{b \in V_t | S \Rightarrow^* \ldots Ab \ldots\} \cup \{\lambda \mid \text{if } S \Rightarrow^* \ldots A\}$

$b \in follow(A)$ means that the terminal b follows the nonterminal A in a sentential form.

$first$ and $follow$ sets are all (finite) subset of V_t.

The $first$ and $follow$ sets are useful for providing lookaheads.

Computing $first$ sets

1. Initially, **foreach** terminal $X \in V_t$ **do** $first(X) := \{X\}$
2. **foreach** nonterminal $X \in V_t$ **do**
3. **if** $X \Rightarrow^* \lambda$ **then** $first(X) := \{\lambda\}$
4. **else** $first(X) := \emptyset$
5. **repeat** $\{$
6. **for** each rule $X \rightarrow \alpha$ **do** $\{$
7. $first(X) := first(X) \cup first(\alpha)$
8. $\}$
9. $\}$ **until** no more change to any $first$ sets;

Let $\alpha = X_0 X_1 X_2 X_3 X_4 \ldots X_k$.

$$
\begin{aligned}
first(\alpha) \quad = \quad & first(X_0) - \{\lambda\} \\
& \cup(\text{if } X_0 \Rightarrow^* \lambda \text{ then } first(X_1) - \{\lambda\}) \\
& \cup(\text{if } X_0 X_1 \Rightarrow^* \lambda \text{ then } first(X_2) - \{\lambda\}) \\
& \cup \ldots \\
& \cup(\text{if } X_0 X_1 \ldots X_{k-1} \Rightarrow^* \lambda \text{ then } first(X_k))
\end{aligned}
$$

Note that $first(\lambda) = \{\lambda\}$.

Computing $follow$ sets

1. Initially, $follow(START_SYMBOL) = \{\lambda\}$
2. All other $follow$ sets are empty sets.
3. **repeat** {
4. **for** each rule $A \rightarrow \ldots B\beta$ (where $B \in V_n$) **do** {
5. $follow(B) := follow(B) \cup (first(\beta) - \{\lambda\})$
6. **if** $\lambda \in first(\beta)$ **then**
7. $follow(B) := follow(B) \cup follow(A)$
8. }
9. } **until** no more change to any $follow$ sets;

$E \rightarrow P \ (\ E \ )$

$E \rightarrow v \ T$

$P \rightarrow f$

$P \rightarrow$

$T \rightarrow + \ E$

$T \rightarrow$

$first[v] = \{v\}$

$first[(] = \{(\}$

$first[)] = \{)\}$

$first[+] = \{+\}$

$first[f] = \{f\}$

first/follow	E	P	T	E	P	T
initially	$\emptyset$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$	$\emptyset$	$\emptyset$
	$\{(, v\}$			$\{), \lambda\}$	$\{(\}$	
		$\{f, \lambda\}$				$\{), \lambda\}$
			$\{+, \lambda\}$			
	$\{(, v, f\}$					

$S \rightarrow ABc$

$A \rightarrow a$

$A \rightarrow$

$B \rightarrow b$

$B \rightarrow$

$first(a) = \{a\}$

$first(b) = \{b\}$

$first(c) = \{c\}$

first/follow	S	A	B	S	A	B
initially	$\emptyset$	$\{\lambda\}$	$\{\lambda\}$	$\{\lambda\}$	$\emptyset$	$\emptyset$
	$\{c\}$				$\{b, c\}$	
		$\{a, \lambda\}$				$\{c\}$
			$\{b, \lambda\}$			
	$\{a, b, c\}$					

$$S \rightarrow aSe$$
$$S \rightarrow B$$
$$B \rightarrow bBe$$
$$B \rightarrow C$$
$$C \rightarrow cCe$$
$$C \rightarrow d$$

$$first(a) = \{a\}$$
$$first(b) = \{b\}$$
$$first(c) = \{c\}$$
$$first(d) = \{d\}$$
$$first(e) = \{e\}$$

first/follow	S	B	C	S	B	C
initially	$\emptyset$	$\emptyset$	$\emptyset$	$\{\lambda\}$	$\emptyset$	$\emptyset$
	$\{a\}$			$\{e, \lambda\}$		
		$\{b\}$			$\{e, \lambda\}$	
			$\{c, d\}$			$\{e, \lambda\}$
	$\{a, b\}$	$\{b, c, d\}$				
	$\{a, b, c, d\}$					

A Comparison of Parsers

See Figure 1.

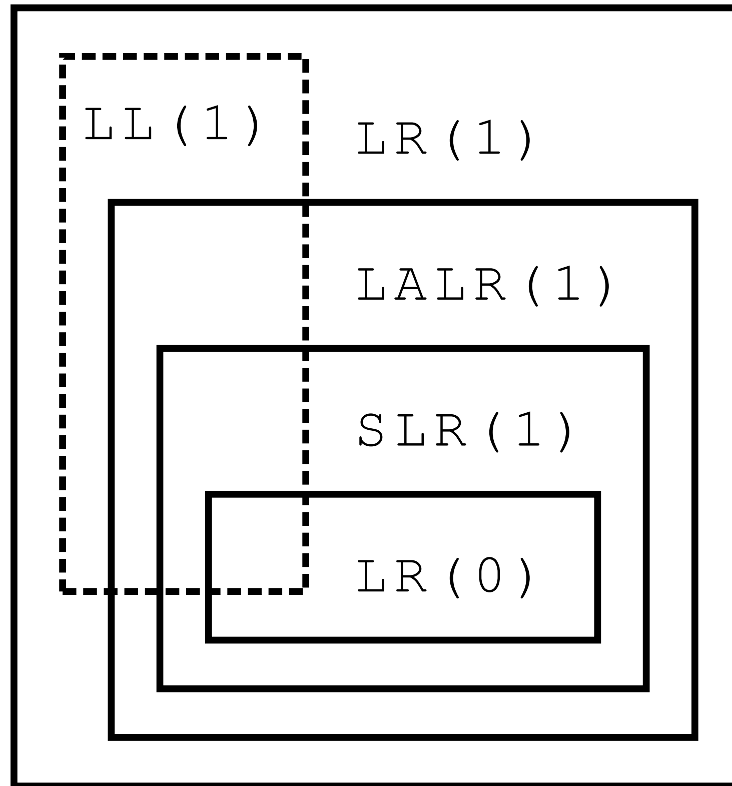


Figure 1: A Comparison of Parsers.

A Pictorial