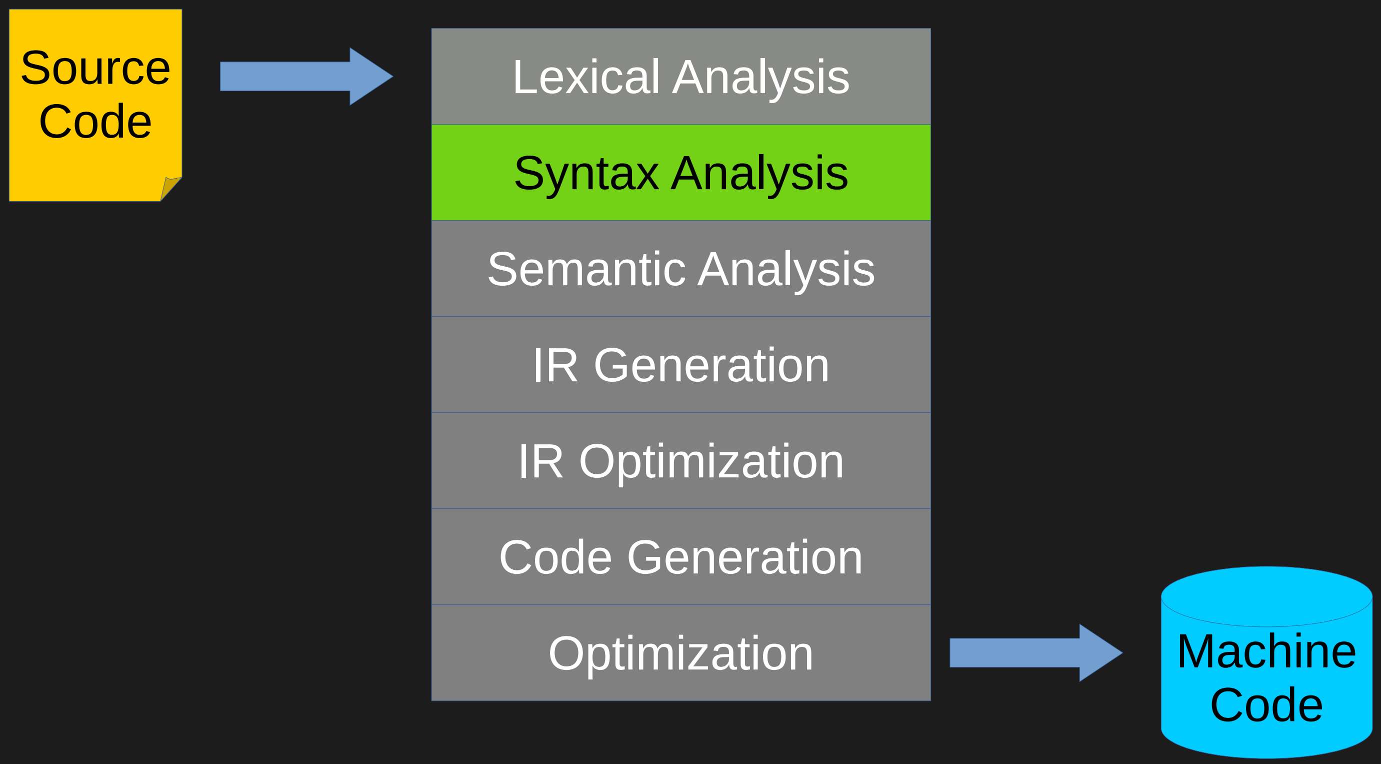
Syntax Analysis – Top-down Parsing

Wei Wang

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Textbook Chapters

- Dragon book
  - Chapter 4.4

Review from Last Time

- Goal of syntax analysis: recover the intended structure of the program.
- Idea: Use a context-free grammar to describe the programming language.
- Given a sequence of tokens, look for a parse tree that generates those tokens.
- Recovering this parse tree is called parsing and is the topic for the next few lectures.

Different Types of Parsing

- Top-Down Parsing
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- Bottom-Up Parsing
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Predictive Top-down Parsing Algorithm

Challenges in Top-down Parsing

- Top-down parsing begins with virtually no information.
  - It begins with the top of the parse tree (the start symbol) and gradually grow the parse tree downwards by applying productions.
- But how can we know which productions to apply?
  - In general, we can’t.
  - Pushdown automaton (PDA) is a variant of top-down parsing. Recall that there is no guarantee that a deterministic PDA exists for an arbitrary CFG.

Predictive Top-down Parsing

- For a class of CFGs, we are able to guess what productions should be used by reading a terminal symbol from the input strings.
- For example, which productions should we use first for the following input string and CFG?

```
E → INT_E
E → DOUBLE_E
INT_E → int Op int
DOUBBL_E → Double Op Double
Op → +
Op → -
Op → *
Op → /
```

String: int + int

Predictive Top-down Parsing cont.

- For a class of CFGs, we are able to guess what productions should be used by reading a terminal symbol from the input strings.
- For example, which productions should we use first for the following input string and CFG?

```
E → INT_E
E → DOUBLE_E
INT_E → int Op int
DOUBBL_E → Double Op Double
Op → +
Op → -
Op → *
Op → /
```

String: int + int

We should use
`E → INT_E`
and
`INT_E → int Op int`
because they are the only rules that produces strings starting with an “int”

Predictive Top-down Parsing Algorithms

- Intuitively, we read a few symbols in the input string, then pick the production that can generate strings starting with these symbols.
- Predictive top-down parsing algorithms are:
  - Similar as PDA, they use a stack which starts with "**S**$".
  - Similar as PDA, if the top of the stack is a terminal symbol **t**, pop **t** if the current input symbol is also **t**.
  - If the top of the stack is a non-terminal symbol **N** and the current input string starts with ω,
    - pick the rule $N \rightarrow \alpha$ that can produce strings starting with ω
    - Pop **N**, and push α.

An Example of Predictive Top-down Parsing

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack: E$

Begin stack with start symbol E

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont.

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack:

E$
T + E $

Replace “E” with “T+E”, b/c E → T+E is the only rule that can produce strings starting with “int +”.

Look two symbols ahead.

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack:

E$
T + E $
int + E $

Replace “T” with “int”, b/c T → int is the only rule that can produce strings starting with “int”.

Look one symbol ahead

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont.

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack:

E$
T + E $
int + E $
+ E $

Pop “int” and advance the input string pointer.

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont.

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack:

E$
T + E $
int + E $
+ E $
E $

Pop “+” and advance the input string pointer.

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont.

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

Stack:

E$
T + E $
int + E $
+ E $
E $
T $

Replace “E” with “T”, b/c E → T is the only rule that can produce strings starting with “(”.

Look one symbol ahead.

int	+	(	int	+	int	)

An Example of Predictive Top-down Parsing cont.

- Consider the following CFG and string:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

int	+	(	int	+	int	)

Stack:

E$
T + E $
int + E $
+ E $
E $
T $
(E) $
E) $
T+E) $
int+E) $
E) $
T) $
int) $
$

The rest of the stack

LL(1) Parser

A Simple Predictive Parser: LL(1)

- Top-down, predictive parsing:
  - L: Left-to-right scan of the tokens
  - L: Leftmost derivation.
  - (1): One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a nonterminal, we predict the production to use by looking at the next token of the input. The decision is forced.

LL(1) Parse Tables

- For LL(1) parser, which production to use (for poping and pushing) depends on,
  - The non-terminal symbol on top of the stack.
  - The next input terminal symbol in the input string.
- Typically, a parse table is used to represent the production usages for a CFG.

An Example of LL(1) Parsing

- An example grammar and its parsing table:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

This cells means when the top of the stack is "E" and the next character in the string is "(", we should pop "E" and replace it with the right hand of rule 2, which is (E Op E).

	int	(	)	+	*
E	Rule 1 E → int	Rule 2 E → (E Op E)			
Op				Rule 3 Op → +	Rule 4 Op → *

An Example of LL(1) Parsing cont.

- To parse string “(int + (int * int))”:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack: E$

Begin stack with start symbol E

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

An Example of LL(1) Parsing cont.

- To parse string “(int + (int * int))”:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$
(E Op E)$

Since stack top is “E” and next char is “(“, we use rule 2 as specified by the parsing table.

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

An Example of LL(1) Parsing cont.

- To parse string “(int + (int * int))”:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$
(E Op E)$
E Op E)$

Pop “(“ and advance to next char in the input string

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

↑

An Example of LL(1) Parsing cont.

- To parse string "(int + (int * int))":

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$
(E Op E)$
E Op E)$
int Op E)$

Use rule 1 as specified by the parsing table.

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

An Example of LL(1) Parsing cont.

- To parse string “(int + (int * int))”:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$
(E Op E)$
E Op E)$
int Op E)$
Op E)$

Pop “int“ and advance to next char in the input string.

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

An Example of LL(1) Parsing cont.

- To parse string "(int + (int * int))":

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$
(E Op E)$
E Op E)$
int Op E)$
Op E)$
+ E)$

Apply rule 3.

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

(	int	+	(	int	*	int	)	)

An Example of LL(1) Parsing cont.

- To parse string "(int + (int * int))":

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

	int	(	)	+	*
E	R1	R2			
Op				R3	R4

Stack:

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	Int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$
$	$

The rest of the stack.

LL(1) Error Detection

- If the symbols in the input string (“`t`”) and stack (“`N`”) directs to an empty cell in the parsing table, there is an error.
  - This indicates that there is no derivations that generates a string starting with “`t`” from “`N`”
- If the non-terminal symbols on top of the stack and current input do not match, there is an error.
  - If the stack is empty but there are still unread input symbols, there is an error.

An Example of LL(1) Parsing Error

- To parse string “(int (int))”:

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$	(int (int))$
(E Op E)$	(int (int))$
E Op E)$	int (int))$
int Op E)$	int (int))$$
Op E)$	(int))$$

No rules to apply for “Op” and “(“

	int	(	)	+	*
E	R1	R2			
Op		X		R3	R4

Another Example of LL(1) Parsing Error

- To parse string "int + int":

```
(1) E → int
(2) E → ( E Op E )
(3) Op → +
(4) Op → *
```

Stack:

E$	int + int$
int$	int + int$
$	+ int$

Stack is empty, but input string is not empty.

	int	(	)	+	*
E	R1	R2			
Op		X		R3	R4

LL(1) Parsing Algorithm

- Let `T[A,t]` represent a cell in the parsing table `T` on row "`A`" and column "`t`".
- Suppose a grammar has start symbol `S` and LL(1) parsing table `T`. We want to parse string ω.
- Initialize a stack containing `S$`.
- Repeat until the stack is empty:
  - Let the next character of ω be `t`.
  - If the top of the stack is a terminal `r`:
    - If `r` and `t` don't match, report an error.
    - Otherwise consume the character `t` and pop `r` from the stack.
  - Otherwise, the top of the stack is a nonterminal `A`:
    - If `T[A,t]` is undefined, report an error.
    - Replace the top of the stack with `T[A,t]`.

Parsing Table Generation

Parsing Table Generation

- Obviously, the key part of LL(1) parsing algorithm is the parsing table.
- How do we generate paring table?
  - Recall that, if the top of the stack is `A` and the next input symbol is `t`, we only apply production `A`→`α`, if and only if, `α` may start with `t`.
  - That is, if `T[A,t] = A`→`α`, then `α` may start with `t`.

Parsing Table Generation cont.

- How about **ε**-productions?
  - **A**→**ε** does not produce any real symbol. In other words, it does not start with any real symbol.
  - We will never see **ε** in any input string.
  - Then, when should we apply **A**→**ε**?
- Production **A**→**ε** can be applied,
  - when top of the stack is **A**, and
  - when the next input symbol, **t**, is a symbol that might come after **A**. That is, for some string in {ω| **A**⇒* ω}, **t** might come after ω.

FIRST Sets

- We want to tell if a particular non-terminal symbol **A** derives a string starting with a particular terminal **t**.
- We can formalize this with FIRST sets.
  - FIRST(**A**) = { **t** | **A** $\Rightarrow^*$ **t**ω for some ω }
- Intuitively, FIRST(**A**) is the set of terminals that can be at the start of a string produced by **A**.
- We can generalize FIRST to strings with FIRST(ω) being the set of all terminals (or **ε**) that can appear at the start of a string derived from ω.

FOLLOW Sets

- With ε-productions in the grammar, we may have to "look past" the current nonterminal to what can come after it.
- The FOLLOW set represents the set of terminals that might come after a given nonterminal.
- Formally:
  - FOLLOW(**A**) = { **t** | **S** ⇒* α**At**ω for some α, ω } where **S** is the start symbol of the grammar.
- Informally, every nonterminal that can ever come after **A** in a derivation.

Computing FIRST Sets

- To compute FIRST sets for all non-terminal symbols:
  - Step 1: initially, for all non-terminals **A**, set FIRST(**A**) = { **t** | **A** → **t**ω for some ω }
  - Step 2: for all non-terminals **A** where **A** → **ε** is a production, add ε to FIRST(**A**).
  - Step 3: repeat the following until no changes occur to FIRST sets,
    - For each production **A** → $Y_1Y_2Y_3Y_4 \ldots Y_k$, where Y_i can be any terminal and non-terminal symbol, set
      - FIRST(**A**) = FIRST(**A**) ∪ (FIRST(Y_i) – {**ε**}), if **ε** is in all of FIRST(Y_1)… FIRST(Y_{i-1}). Note that if **ε** is in FIRST(Y_j), then $Y_j \Rightarrow^* \varepsilon$.
        - More detailed operations: first add (FIRST(Y_1) – {**ε**}) to FIRST(**A**); if **ε** is in FIRST(Y_1), add (FIRST(Y_2) – {**ε**}) to FIRST(**A**);
      - If **ε** is in all of FIRST(Y_1)… FIRST(Y_k), **ε** is also in FIRST(**A**).

Computing FOLLOW Sets

- Intuition: if $x \rightarrow A\omega$, the First(ω) $\subseteq$ Follow(A).
  - Little trickier because the possibility that $\omega \Rightarrow^* \varepsilon$.
- To compute FOLLOW sets for all non-terminal symbols:
  - Step 1: initially, for each nonterminal A, set FOLLOW(A) = $\{ t \mid B \rightarrow \alpha A t \omega$ is a production $\}$
  - Step 2: Add $\$$ to FOLLOW(S), where S is the start symbol. That is, FOLLOW(S) = $\{\$\}$.
  - Step 3: Repeat the following until no changes occur to FOLLOW sets:
    - If $B \rightarrow \alpha A \omega$ is a production, set FOLLOW(A) = FOLLOW(A) $\cup$ FIRST(ω) – $\{\varepsilon\}$.
    - If $B \rightarrow \alpha A \omega$ is a production and $\varepsilon \in$ FIRST(ω), set FOLLOW(A) = FOLLOW(A) $\cup$ FOLLOW(B).
    - If $B \rightarrow \alpha A$ is a production, set FOLLOW(A) = FOLLOW(A) $\cup$ FOLLOW(B).

+, ε

An Example of FIRST Sets

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FIRST sets after Step 1:

Symbol	FIRST Set
E	
E'	+
T	
T'	*
F	(, id

An Example of FIRST Sets cont.

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FIRST sets after Step 2:

Symbol	FIRST Set
E	
E'	+, ε,
T	
T'	*, ε,
F	(, id

An Example of FIRST Sets cont.

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- First sets after Step 3 for one iteration:

Symbol	First Set
E	
E'	+, ε
T	(, id
T'	*, ε
F	(, id

An Example of FIRST Sets cont.

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FIRST sets after Step 3 for two iterations:

Symbol	FIRST Set
E	(, id
E’	+, ε
T	(, id
T’	*, ε
F	(, id

An Example of FIRST Sets cont.

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FIRST sets after Step 3 for three iterations:
  - No changes to FIRST sets, can stop.

Symbol	FIRST Set
E	(, id
E'	+, ε
T	(, id
T'	*, ε
F	(, id

An Example of FOLLOW Sets

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FOLLOW sets after Step 1:

Symbol	FOLLOW Set
E	)
E’	
T	
T’	
F	

An Example of FOLLOW Sets

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FOLLOW sets after Step 2:

Symbol	FOLLOW Set
E	), $
E’	
T	
T’	
F	

An Example of FOLLOW Sets

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FOLLOW sets after Step 3 for one iteration:

Symbol	FOLLOW Set
E	), $
E'	), $
T	+,), $
T'	+,), $
F	*,+,), $

An Example of FOLLOW Sets

- Consider the following grammar:

```
(1) E  → T E'
(2) E' → + T E' | ε
(3) T  → F T'
(4) T' → * F T' | ε
(5) F  → ( E ) | id
```

- FOLLOW sets after Step 3 for two iterations:
  – No changes to FOLLOW sets, can stop

Symbol	FOLLOW Set
E	), $
E’	), $
T	+,), $
T’	+,), $
F	*,+,), $

Parsing Table Generation From FIRST and FOLLOW Sets

- Compute FIRST(A) and FOLLOW(A) for all nonterminals A.
- For each rule A → ω, for each terminal t ∈ FIRST(ω), set T[A, t] = A → ω.
  - Note that ε is not a terminal in parsing table.
- For each rule A → ω, if ε ∈ FIRST(ω), set T[A, t] = A → ω for each t ∈ FOLLOW(A).

An Example of Parsing Table Generations

- Consider the same grammar,

```
(1) E → T E'
(2) E'→ + T E' | ε
(3) T → F T'
(4) T'→ * F T' | ε
(5) F → ( E ) | id
```

- The parsing table is,

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E'→ +TE'			E'→ ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T'→ ε	T'→ ε
F	F → id			F → (E)		

Limitations of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **Ab** | **c**

- FIRST(**A**) = {**c**} and FIRST(**Ab**) = {**c**}
- However, we cannot build an LL(1) parsing table.
  - There is a FIRST/FIRST conflict in column **c**.

	b	c
A		A → Ab A → c

Eliminating Left Recursion

- In general, left recursion can be converted into right recursion by a mechanical transformation.
- Consider the grammar

  $A \to A\omega \mid \alpha$
- This will produce α followed by some number of ω's.
- Can rewrite the grammar as

  $A \to \alpha A'$

  $A' \to \varepsilon \mid \omega A'$

Another Non-LL(1) Grammar

- Consider the following grammar:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

- FIRST(**E**) = { **int**, **(** } and FIRST(**T**) = { **int**, **(** }
- `T[E, int]` and `T[E, (]` can be either `E → T` or `E → T + E`.
  - Also a FIRST/FIRST conflict.

Left Factoring

- The second FIRST/FIRST conflict can be removed with Left factoring.
- Left factoring:
  - For production:
    $A \rightarrow \alpha\beta \mid \alpha\gamma$
  - Convert to
    $A \rightarrow \alpha A'$
    $A' \rightarrow \beta \mid \gamma$

Left Factoring Example

- Consider the following grammar:

```
(1) E → T
(2) E → T + E
(3) T → int
(4) T → ( E )
```

- Can be converted to:

```
(1) E → T E'
(2) E' → + E | ε
(3) T → int
(4) T → ( E )
```

A Formal Characterization of LL(1)

- Given a grammar G, G is LL(1), if for every production $A \rightarrow \alpha \mid \beta$,
  - There exist no terminal t, such at $t \in$ FIRST(α) and $t \in$ FIRST(β).
  - At most one of the α and β can derive the ε.
  - If β derives ε, then α does not derive any string beginning with a terminal in FOLLOW(A). That is, FIRST(α) $\cap$ FOLLOW(A) = $\varnothing$.
- These conditions are equivalent to saying that there are no conflicts in the table.

The Advantages and Disadvantages of LL(1) Parser

LL(1) is Straightforward

- Can be implemented quickly with a table-driven design.
- Can also be implemented by recursive-descent:
  - Define a function for each non-terminal.
  - Have these functions call each other based on the lookahead token.

LL(1) is Fast

- Both table-driven and recursive-descent LL(1) is fast.
- Can parse in $O(n |G|)$ time, where n is the length of the string and $|G|$ is the size of the grammar.

The Disadvantages of LL(1)

- For most programming languages, using LL(1) requires eliminating left-recursive and left factoring.
- By changing the grammar, it might make the other phases of the compiler more difficult.
  - Hard to determine semantics and generate code.

Summary

- Top-down parsing tries to derive the user's program from the start symbol.
- LL(1) parsing scans from left-to-right, using one token of look-ahead to find a leftmost derivation.
- FIRST sets contain terminals that may be the first symbol of a production.
- FOLLOW sets contain terminals that may follow a non-terminal in a production.
- Left recursion and left factorability cause LL(1) to fail and can be mechanically eliminated in some cases.

Acknowledgement

- This lectures is partially based on the Compiler slides of Keith Schwarz.