COMP3173 Compiler Construction
Context-free

Dr. Zhiyuan Li

Outline

- Context-free grammar
- Ambiguous
- Push-down automata

Overview

- Syntax analysis is the second phase of a compiler.
- Take a stream of tokens from the lexer as input.
- Analyze the structure of the tokens.
- Detect syntax errors.
- Out put a parsing tree to present the structure.
- Use ***context-free grammar*** (CFG) for analysis.

Overview

- Why CFG?
  - Remember this language $\{a^i b^i | i \geq 0\}$?
  - Also, the syntax for binary operators, like $a + b$. a and b have to be nicely paired.
  - Regular languages are not enough for this syntax.
  - CFGs are not the most powerful formal language (see Chomsky hierarchy), but CFGs are enough to describe the structures for most of the languages.
- In practice, we will create efficient parsers for many CFGs (but not all), because parsers for more powerful grammars can be very inefficient.

Context-free grammar

- A ***context-free grammar*** is a 4-tuple (N, T, R, S), where
  - N is a finite set called ***variables***,
  - T is a finite set, disjoint from N, called the ***terminals***,
  - R is a finite set of ***production rules***, with each rule being a variable v and a string s of variables and terminals in the form $v \to s$, and
  - $S \in N$ is the start variable.

- The production rules are same as the rules in the regular definition.
- We can use the rules to ***derive*** the strings in a context-free language.
- Terminals are the symbols which cannot produce more symbols. In a compiler, they are ***tokens***.
- Variables only exists when the production is unfinished. They are not symbols in the alphabet, neither tokens. Thus, they are not in the strings of the language.

Context-free grammar

- For example,
  - $N = \{E\}$
  - $T = \{+, *, (,), -, id\}$
  - $R = \{E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow -E, E \rightarrow id\}$
  - $S = E$

- The production rules can also be “combined” and rewritten as
  - $R = \{E \rightarrow E + E | E * E | (E) | - E | id\}$
  - The symbol “|” is same as it is in regular expressions, meaning that there are multiple options.
  - The rules can be combined only if the LHS are the same.

Context-free grammar

- Sometimes the set of production rules is enough to define a grammar because
  - variables can be obtained by looking at the LHS of each production rule,
  - terminals are the symbols on the RHS of the rules excluding the variables, and
  - the start symbol is usually denoted by E.
- Then, the above grammar can be

$$E \rightarrow E + E | E * E | (E) | -E | id$$

Derivations

- Given a grammar G, we can generate strings in the language $L(G)$ by using a ***derivation***.
- A derivation (informally) is a procedure to apply the production rules from the start symbol to a string which only contains terminals.
- Each production is denoted by $\Rightarrow$.
- For example, given the grammar

$$E \rightarrow E + E | E * E | (E) | - E | id$$

- A derivation can be

	E	start symbol
$\Rightarrow$	$E + E$	using rule 1
$\Rightarrow$	$id + E$	using rule 5
$\Rightarrow$	$id + E * E$	using rule 2
$\Rightarrow$	$id + id * E$	using rule 5
$\Rightarrow$	$id + id * id$	using rule 5

Derivations

- The above derivation is called ***left-most derivation***, meaning that each iteration we replace the left most nonterminal.
- Similarly, the ***right-most derivation*** replaces the right most nonterminal in each iteration.
- For example, the derivation below is a right-most derivation which produces the same string.

$$
\begin{array}{lll}
 & E & \text{start symbol} \\
\Rightarrow & E + E & \text{using rule 1} \\
\Rightarrow & E + E * E & \text{using rule 2} \\
\Rightarrow & E + E * id & \text{using rule 5} \\
\Rightarrow & E + id * id & \text{using rule 5} \\
\Rightarrow & id + id * id & \text{using rule 5}
\end{array}
$$

Derivation

- Each sequence of nonterminals and tokens that we derive at each step is called a ***sentential form***.
- The last sentential form only contains tokens and is called ***sentence*** which is a syntactically correct string in the programming language.
- If w is a sentence and S is the start symbol, we can write $S \overset{*}{\Rightarrow} w$.
- $\overset{*}{\Rightarrow}$ means derive in zero or more steps. Also means RHS is derivable by LHS.
- From the above example, $E \overset{*}{\Rightarrow} id + id * id$

Parse Tree

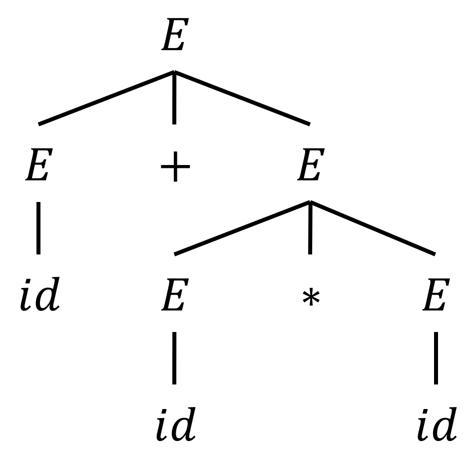
- Given a derivation of a sentence, we can construct a corresponding ***parse tree***, which describes the structure of the sentence.
- The construction is done recursively.
- Initially, the parse tree has only one vertex – the start variable.
- For each iteration of the derivation, every variable and terminal in RHS of the production rule is a new vertex being added to the parse tree as a child of the LHS variable.

- After a parse tree is fully constructed, the leaves are tokens, internal vertices are variables, and the root is the start variable.

Parse Tree

For example, $E \overset{*}{\Rightarrow} id + id * id$

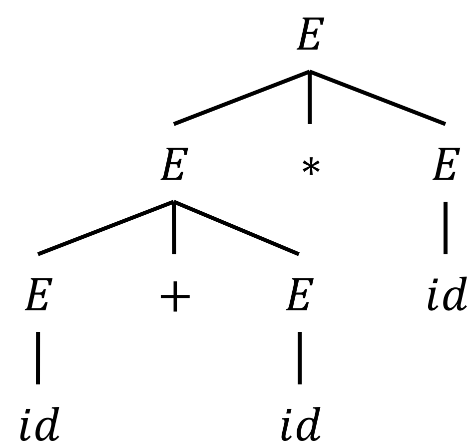
	E	start symbol
$\Rightarrow$	$E + E$	using rule 1
$\Rightarrow$	$id + E$	using rule 5
$\Rightarrow$	$id + E * E$	using rule 2
$\Rightarrow$	$id + id * E$	using rule 5
$\Rightarrow$	$id + id * id$	using rule 5



Ambiguous

- The grammar G is ***ambiguous*** if there is a sentence in $L(G)$ from which it is possible to construct multiple parse trees (using any type of derivation).
- For example, we use $E \overset{*}{\Rightarrow} id + id * id$ again. Even we also use left-most derivation,

$$
\begin{array}{lll}
 & E & \text{start symbol} \\
\Rightarrow & E * E & \text{using rule 2} \\
\Rightarrow & E + E * E & \text{using rule 1} \\
\Rightarrow & id + E * E & \text{using rule 5} \\
\Rightarrow & id + id * E & \text{using rule 5} \\
\Rightarrow & id + id * id & \text{using rule 5}
\end{array}
$$



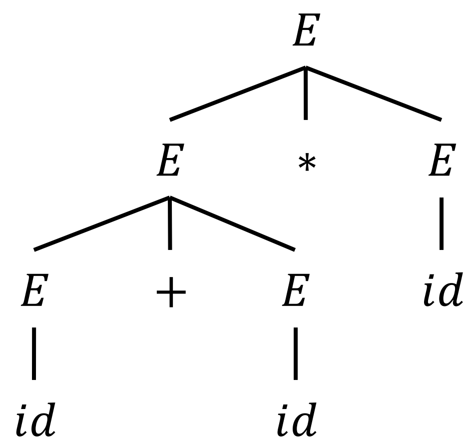
which is different from the previous parse tree.

- The ambiguity is caused by lacking parenthesis, which is omitted by the precedence of operations.

Ambiguous

- Here is another way to produce the second parse tree using the right-most derivation.

	E	start symbol
$\Rightarrow$	$E * E$	using rule 2
$\Rightarrow$	$E * id$	using rule 5
$\Rightarrow$	$E + E * id$	using rule 1
$\Rightarrow$	$E + id * id$	using rule 5
$\Rightarrow$	$id + id * id$	using rule 5



Some properties

- Each derivation (left-most, right-most, or otherwise) corresponds to exactly one parse tree, whether the grammar is ambiguous or not.
- Each parse tree corresponds to multiple derivations, whether the grammar is ambiguous or not.
- Each parse tree corresponds to exactly one left-most derivation and exactly one right-most derivation, whether the grammar is ambiguous or not.
- All derivations of the same sentence correspond to the same parse tree if the grammar is not ambiguous.
- Multiple derivations of the same sentence may not correspond to the same parse tree if the grammar is ambiguous.
- In general, deciding a grammar is ambiguous or not is ***undecidable***, or ***uncomputable*** (like the halting problem).

Exercises

- Add parenthesis to the grammar $E \to E + E|E * E|(E)| - E|id$ to eliminate ambiguity.
- Given the context-free grammar $S \to SS + |SS * |a$ and the string $aa + a *$
  - Give a left-most derivation for the string.
  - Give a right-most derivation for the string.
  - Give a parse tree for the string.
  - Is the grammar ambiguous? Why?
  - Use English to describe the language defined by the grammar.
- Some grammars are ambiguous, but the left-most derivation and the right-most derivation correspond to the same parse tree. Try to construct an example for such ambiguous grammars.

Ambiguity Elimination

- A compiler cannot use an ambiguous grammar because each input program must be parsed to a unique parse tree to show the structure.
- To remove ambiguity in a grammar, we can transform it by hand into an unambiguous grammar. This method is possible in theory but not widely used in practical because of the difficulty.
- Most compilers use additional information to avoid ambiguity.

Ambiguity Elimination

- Here is an ambiguous grammar for “if-else”.

$$
\begin{array}{rcl}
S & \rightarrow & \text{if } (E)\ S \\
 & | & \text{if } (E)\ S \text{ else } S \\
 & | & \text{other}
\end{array}
$$

- The sentence “if (E_1) if (E_2) else S_1” can be parsed differently.
- The grammar can be transformed into the one below without ambiguity but hard to be understood.

$$
\begin{array}{rcl}
S & \rightarrow & M|U \\
M & \rightarrow & \text{if } (E)\ M \text{ else } M \\
 & | & \text{other} \\
U & \rightarrow & \text{if } (E)\ S \\
 & | & \text{if } (E)\ M \text{ else } U
\end{array}
$$

- In practice, we can tell the compiler that “else” is associated with the nearest “if” to eliminate ambiguity.

CFG vs Regular Definition

- From the definition of CFG and Regular Definition, we can easily tell CFG is the superset.
- The RHS of each production rule in a CFG has no restriction.
- But the RHS of the rules in a regular definition is a regular expression which cannot contain the LHS.
- Simulating regular expressions in CFGs is simple.
- Thus, using CFGs for lexical analysis is possible in theory but not efficient in practical.

Push-down Automata

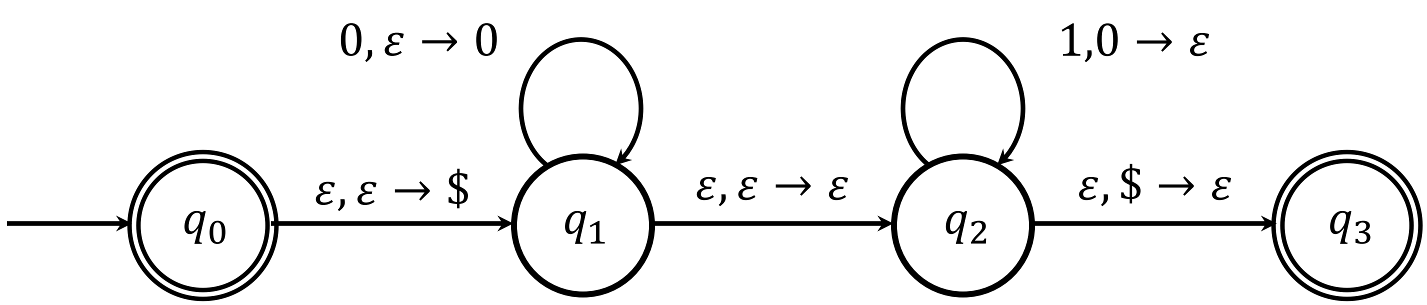
- Keep in mind that machines, languages, and grammars are equivalent.
- Same as NFAs/DFAs recognize regular languages, there are some machines called ***push-down automata*** (PDA) recognize context-free languages.
- Intuitively, **a PDA is a finite automata plus a stack** with some modifications on the transitions to fit stack behaviors.
- A nondeterministic PDA is more powerful than a deterministic PDA. Here we only discuss NPDAs.

Push-down Automata

- An NPDA is (formally) a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, f)$, where
  - Q is a finite set of states,
  - Σ is a finite set of the input alphabet,
  - Γ is a finite set of the stack alphabet,
  - $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
  - $q_0 \in Q$ is the start state, and
  - $F \subseteq Q$ is the set of accept states.
- For example, the context-free language $L = \{0^n 1^n | \text{n} \geq 0\}$ has grammar

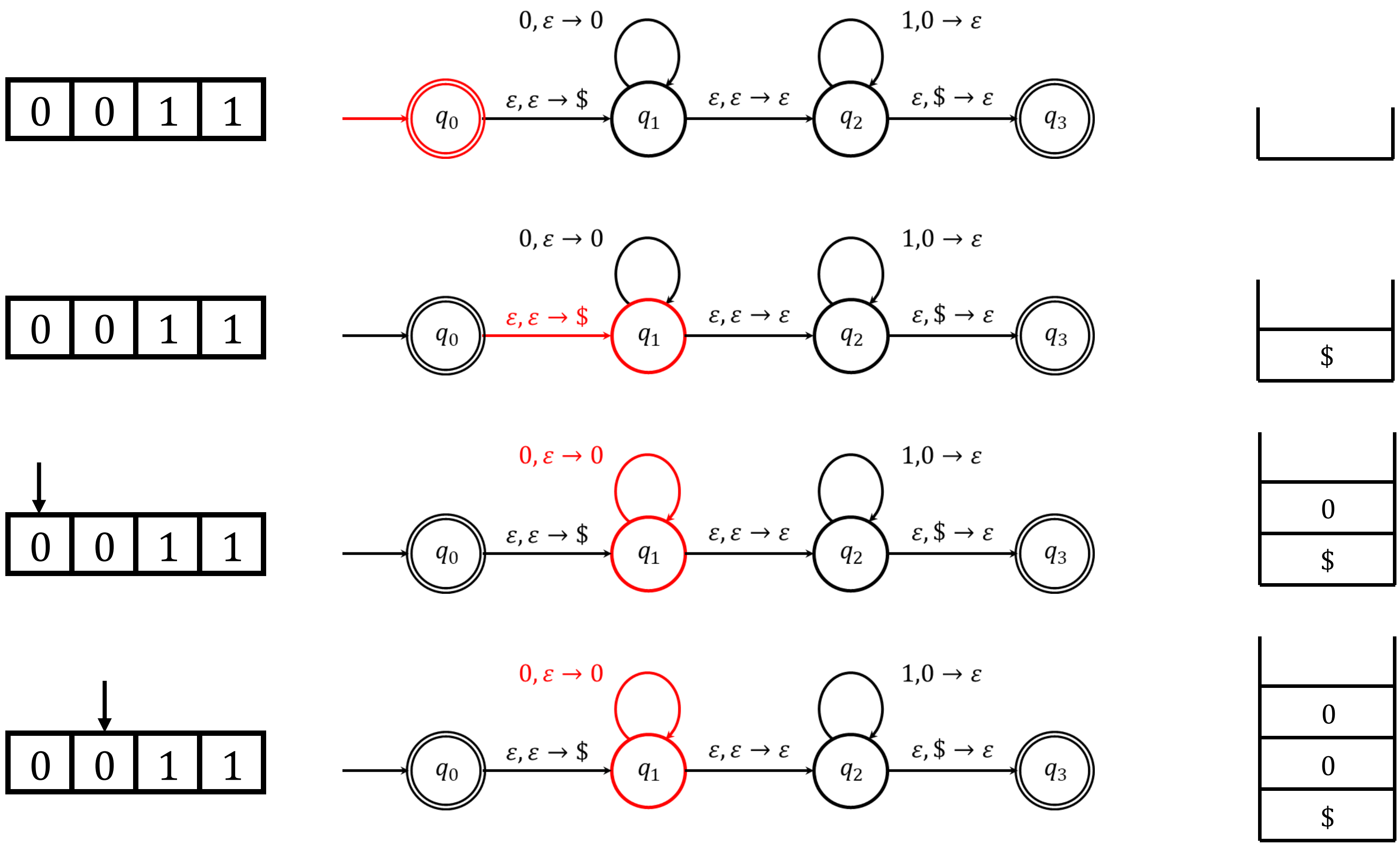
$$E \to 0E1|\varepsilon$$

and the corresponding PDA (in a transition graph)



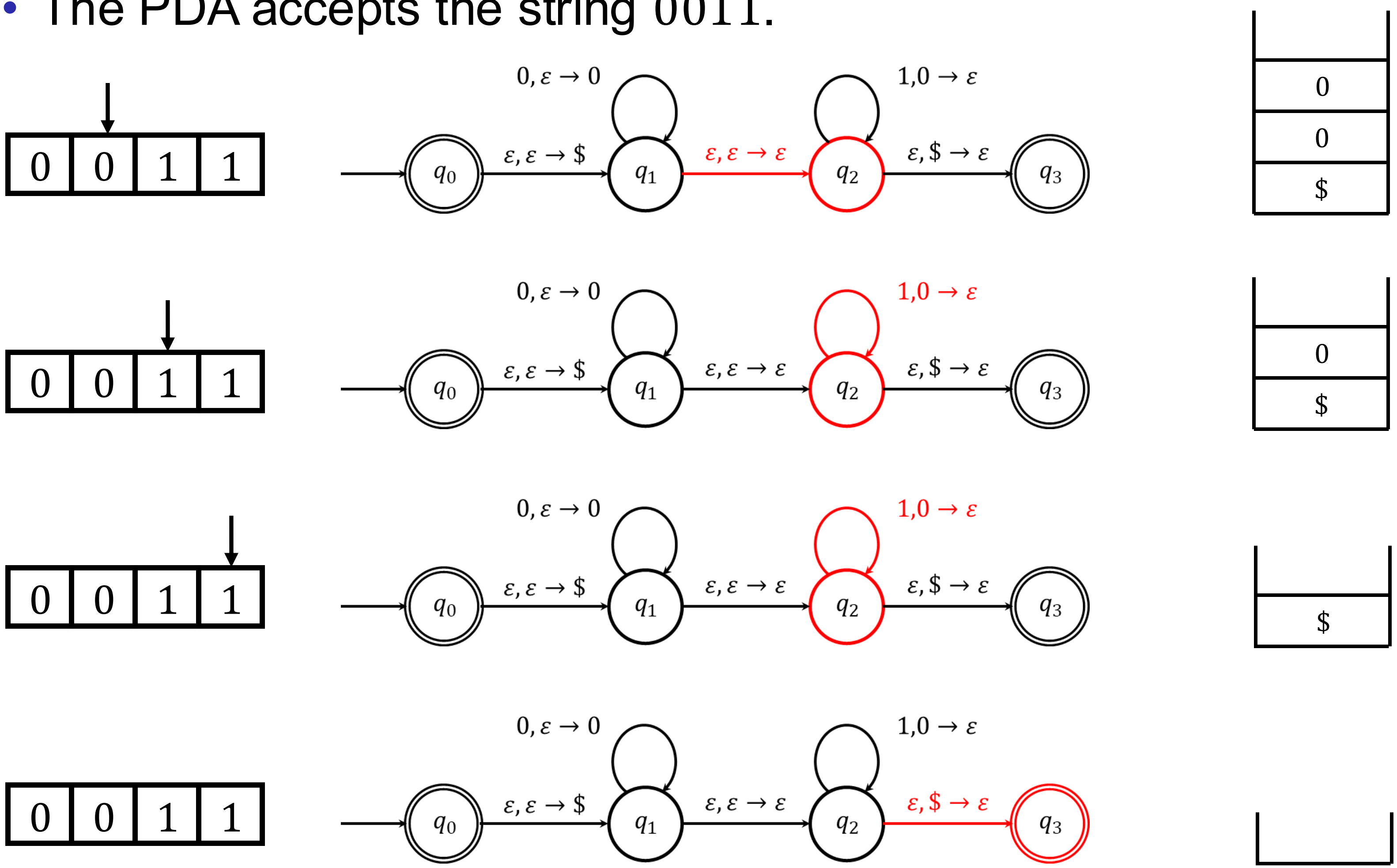
Push-down Automata

- The PDA accepts the string 0011.



Push-down Automata

- The PDA accepts the string 0011.



Limit of CFG

- Same as regular languages/expressions/definitions/DFAs/NFAs (remember they are all equivalent), CFGs are not ultimate even CFGs are more powerful than regular expressions.
- There are some languages which cannot be defined by CFGs.
- For example, $L = \{a^i b^j c^k | 0 \leq i \leq j \leq k\}$ over the alphabet $\Sigma = \{a, b, c\}$.
- To prove a language is not context-free, you need pumping lemma for context free languages.
- To powerup the CFG/PDA, we can try again to add another stack to the machine, same as what we did to finite automata.
- This upgrade is ultimate. A finite automata with two stacks is as powerful as a Turing machine, which is the most powerful computational model that human can implement right now.

The End of Lecture 4