COMP3173 Compiler Construction
Top-Down Parsing

Dr. Zhiyuan Li

Outline

- Recursive Descent Parser
- $LL(1)$ grammar
- Left factoring
- Left Recursion Elimination
- Limitations

Parser

- We have learned context-free grammars from the last lecture.
- Next, we want to implement a parser which
  - takes a sequence of tokens as input;
  - analyze the structure of the input;
  - generates a parse tree; and
  - indicates the syntax errors if there is any.
- There are two types of parsing: top-down and bottom-up.
  - Top-down parsing tries to construct a sequence of tokens from the grammar which is same as the input.
  - Bottom-up parsing tries to match the input tokens with grammar rules.
- We introduce top-down parsing first.
- Top-down parsing is only for a subset of context-free grammars.

Top-down parsing

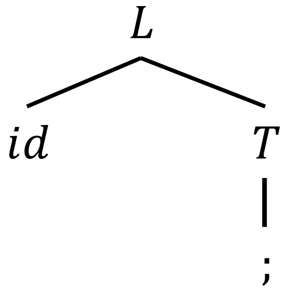
- Top-down parsing means that we generate the parse tree from top to bottom.
- Remember that the internal vertices in a parse tree are the nonterminals (variables), while the leaves are the terminals (tokens).
- The parse tree construction depends on the production rules in the grammar.
- If we want to construct the parse tree from top to bottom, the production rules have to be nicely designed.

Top-down parsing

- For example, the following grammar parses single variable declarations (if we do not care types here).

$$
\begin{aligned}
L &\rightarrow idT \\
T &\rightarrow ;
\end{aligned}
$$

  Suppose the input is $\boldsymbol{id}$;
- The input tokens can be uniquely parsed into



- The RHS of each production rule always starts from a different terminal.
- The parsing on the left branch always finishes before the right branch.
- This is a ***leftmost derivation***.

Recursive-Descent Parser

- However, our life cannot be that easy in most of cases.
- The previous grammar does not allow multiple variable declarations in one statement.
- So, consider this grammar.

$$L \rightarrow id\ ; \tag{1}$$

$$L \rightarrow id, L \tag{2}$$

  and try to parse $\boldsymbol{id, id, id}$;
- We have two different choices for the first derivation.

$$L \Rightarrow id; \text{ or } L \Rightarrow id, L$$

- We cannot decide which production rule can be used for the derivation. The parser will try to pick a valid rule randomly.
- If the guess is wrong, the parser cannot proceed parsing on some input tokens and it will ***backtrack***.

Recursive-Descent Parser

- Consider this grammar

$$L \rightarrow id\ ; \qquad (1)$$

$$L \rightarrow id, L \qquad (2)$$

and try to parse $\boldsymbol{id, id, id};$

- Suppose our parser always choses rule 1 first. The first derivation becomes

$$L \Rightarrow id;$$

- The first token "id" matches with the one in the input stream. But the second token "," does not match. Thus, to continue the parsing, the parser has to
  1. ***roll back*** the first derivation,
  2. **resume** the token "id", and
  3. try the second production rule.
- The roll back is called ***backtracking***.

Recursive-Descent Parser

- Consider this grammar

$$L \to id\ ; \qquad (1)$$

$$L \to id, L \qquad (2)$$

and try to parse $\boldsymbol{id, id, id};$

- The whole parsing is

S_1	L	$\Rightarrow id;$	Try Rule (1)
S_2		$\Rightarrow L$	Mismatch at token "," and backtrack
S_3		$\Rightarrow id, L$	Try Rule (2)
S_4		$\Rightarrow id, id;$	Try Rule (1)
S_5		$\Rightarrow id, L$	Mismatch at token "," and backtrack
S_6		$\Rightarrow id, id, L$	Try Rule (2)
S_7		$\Rightarrow id, id, id;$	Try Rule (1)

All tokens are matched. The parsing is finished.

Recursive-Descent Parser

- Actually, you have seen backtracking in other places … Depth First Search.
- We can express the parsing by using a DFS tree.
- Each vertex in the tree is a valid sentential form (a stream of terminals and nonterminals). In this example, the stream to the RHS of ⇒, given by a step number S_i.
- Each edge is defined by applying a production rule.
- The minor difference here is that we don't need to go back to the root (as DFS) when the parsing is finished.
- Note that this DFS is different from the parse tree because a vertex presents a sentential form, which is equivalent to a (partial) parse tree.

Recursive-Descent Parser

- The grammar is

$$L \rightarrow id\ ; \qquad (1)$$
$$L \rightarrow id, L \qquad (2)$$

and try to parse $\boldsymbol{id, id, id};$

- If we follow the DFS tree representation, the parsing will be

$S_0 \quad L \overset{*}{\Rightarrow} L$

$S_1 \quad L \overset{*}{\Rightarrow} id;$

$S_2 \quad L \overset{*}{\Rightarrow} L$ Same as S_0

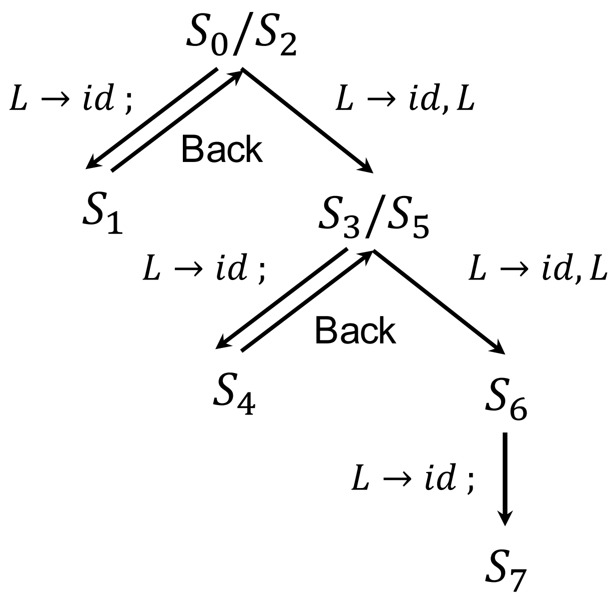
$S_3 \quad L \overset{*}{\Rightarrow} id, L$

$S_4 \quad L \overset{*}{\Rightarrow} id, id;$

$S_5 \quad L \overset{*}{\Rightarrow} id, L$ Same as S_3

$S_6 \quad L \overset{*}{\Rightarrow} id, id, L$

$S_7 \quad L \overset{*}{\Rightarrow} id, id, id;$



Implementation

- To design a recursive-decent algorithm, we can start from the DFS.
- Recall the DFS pseudo code.

Algorithm: Depth First Search

Input: a graph $G = (V, E)$ and a start vertex v

Output: Nil

1. mark v as visited
2. **for each** vertex $u \in adj(v)$
3.   **if** u is not visited
4.   DFS(G, u)
5.   **end if**
6. **end for**
7. mark v as finished

Implementation

- Then, modify the DFS to design our parser.
- Each nonterminal A is implemented by an individual function.
- Iterate on every rule starts with A on LHS.
- Remember to resume the tokens when derivation fails.

Algorithm: match_A()

Input: a stream of tokens

Output: Nil

for each production rule $A \to X_1X_2\cdots X_k$
    for each X_i **and** no error
        if X_i is a nonterminal **and** match_X_i() successful
            continue parsing
        else if X_i is same as the next token
            continue parsing
        else
            error occurs
    resume the parsed tokens

Implementation

- The parse tree generation can be done at Row 4 and Row 6, when the algorithm continues parsing.
- You can also record the derivation rules and generate the parse tree when parsing is finished.
- Resuming the parsed tokens at Row 9 can be implemented by a stack. When a token is matched with a terminal, push the token into the stack. When error occurs, we pop the tokens and put them back to the input.
- The main routine starts from matching the start variable.
- The parser reports syntax error when the main routine has tried all possible production rules. A syntax error can be
  - the parser receives a token which cannot be parsed;
  - after processing all tokens, the parse is incomplete (some leaves are nonterminals); or
  - when the parsing is complete (all leaves are terminals), there are some tokens remained in the input.

Disadvantage of Recursive Descent Parser

- The implementation (in code) can be very complicated/ugly because it needs to try many production rules, imaging many try-catch scopes nested with each other.
- It waste a lot of time on backtracking.
- It needs additional space to store the tokens for resuming purpose.
- In practical, nobody uses recursive descent parser.

Predictive Parser

- To avoid backtracking, we want to design a parser which can guess the next token from the input.
- A correct guess can let the parser uniquely choose a production rule for parsing.
- The idea is using a temporary space to store some tokens in advanced, like a buffer. Then, use the buffered tokens to make decisions.
- This technique is very similar to space-time trade-off in the dynamic programming, what we have learned in algorithms.
- The time complexity of recursions with backtracking can go up to $O(2^n)$. But, if we can choose rules uniquely, the time complexity becomes $O(n)$.

$LL(1)$

- Consider the following grammar

$$L \rightarrow id\ T \tag{1}$$
$$T \rightarrow ,id\ T \tag{2}$$
$$T \rightarrow ; \tag{3}$$

which is equivalent to the above one and try to parse $\boldsymbol{id, id, id}$;

- In the first iteration, the parser has to use the rule $L \rightarrow id\ T$.
- In the second iteration, there are two production rules "$T \rightarrow, id\ T$" or "$T \rightarrow;$"
- This uncertainty can be easily solved by just "looking" at the next token from the lexer. This token is called ***lookahead token***.
- Normally, when lexer returns a token to parser, the token is consumed. But the lookahead token remains in the input.
- The lookahead token in this example is ",". So, the parser knows the rule is $T \rightarrow, id\ T$.

$LL(1)$

- Because the parser reads tokens from left to right, does leftmost derivation, and looks at most one lookahead token; this parser/grammar is called $\boldsymbol{LL(1)}$.
- Some grammars may not be $LL(1)$. For example, the one we have seen.

$$L \rightarrow id\ ; |\ id, L$$

- Looking one token ahead is insufficient for this grammar.
- $L \rightarrow id;$ and $L \rightarrow id, L$ agree on the first token. If the lookahead token is "id", we cannot decide which rule will be used. Thus, to parse this grammar without recursions, the parser needs to look 2 tokens ahead.
- This is $LL(2)$ grammar.

Left-Factoring

- In general, we can design $LL(k)$ grammar, for constant k.
- In practical, more lookahead tokens make no big differences but increase the implementation difficulties.
- More importantly, $LL(1)$ grammar is already powerful enough.
- ***Left-factoring*** is a grammar transformation to convert a grammar to $LL(1)$**.**

Left-Factoring

- Back to the example,

$$L \rightarrow id\,;\,|\,id, L$$

  is not $LL(1)$ because the first token of the two rules are the same. One lookahead token is not enough to the two productions.
- We can solve this issue by introducing a new nonterminal L'.
- And convert the grammar to

$$\begin{aligned} L &\rightarrow id\ L' \\ L' &\rightarrow ;\,|\,, L \end{aligned}$$

- The new grammar is $LL(1)$, which is equivalent to

$$\begin{aligned} L &\rightarrow id\ T \\ T &\rightarrow , id\ T\,|; \end{aligned}$$

- This conversion is called ***left-factoring***.

Left-Factoring

- More formally, for each nonterminal A which has multiple productions starting with the same prefix α:
$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_k \mid \gamma_1 \mid \cdots \mid \gamma_n$$
where α is a non-empty sequence of grammar symbols (terminals and nonterminals), $\beta_1, \cdots, \beta_k$ and $\gamma_1, \cdots, \gamma_n$ are (possibly empty) sequences of grammar symbols.
- Create a new nonterminal A' and transform the rules as
$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_n \\ A' &\rightarrow \beta_1 \mid \cdots \mid \beta_k \end{aligned}$$

Left Recursion Elimination

- Some grammars are not good enough even after left factoring.
- For example, again to parse the variable declaration

$$
\begin{aligned}
L \quad & \rightarrow A; \\
A \quad & \rightarrow id \mid A, id
\end{aligned}
$$

  No production rule has a same prefix on RHS. But lookahead tokens do not work .
- Suppose, the parser at some point needs to derive the nonterminal A.
- One may try: if the lookahead token is id, the parser uses $A \rightarrow id;$. If the lookahead token is A, it uses $A \rightarrow A, id$.
- Be careful! A is a nonterminal but not a token. A lexer can never find such a lookahead token.
- As a result, $A \rightarrow A, id$ will never be used. Obviously, this is wrong.

Left Recursion Elimination

- The problem is cause by $A \rightarrow A, id$. The RHS starts from a nonterminal, which cannot be used to match with a lookahead token. This type of grammar is called ***left recursive***.
- Note that this still left-most derivation. Left-most or right-most derivation is not related to grammar itself. It only relates to in which order we derive the nonterminals.
- Formally, a left recursive grammar has a valid derivation $A \stackrel{*}{\Rightarrow} A\alpha$, where A is a nonterminal and α is a string of grammar symbols.

Left Recursion Elimination

- In general, a left recursive production rule has the form

$$A \to A\alpha \mid \beta$$

- This production can derive β, $\beta\alpha$, $\beta\alpha\alpha \cdots$. The parsing stops at $A \to \beta$, and can produce as many α as possible.
- So, we can let A derives β first and followed by some α.

$$\begin{aligned} A \quad &\to \beta A' \\ A' \quad &\to \alpha A' \mid \varepsilon \end{aligned}$$

- Note that the production from A' can be $A' \to \alpha A' | \alpha$. But this is not $LL(1)$.

Left Recursion Elimination

- Formally, for each nonterminal A which has one or more productions with RHSs starting with the same nonterminal A
$$A \rightarrow A\alpha_1 | \cdots | A\alpha_k | \beta_1 | \cdots | \beta_n$$
where $\alpha_1, \cdots, \alpha_k$ and $\beta_1, \cdots, \beta_n$ are possibly empty sequences of grammar symbols.
- Create a new nonterminal A' and transform the grammar as
$$\begin{aligned} A &\rightarrow \beta_1 A' | \cdots | \beta_n A' \\ A' &\rightarrow \alpha_1 A' | \cdots | \alpha_k A' | \varepsilon \end{aligned}$$

Exercise

- Given the following grammar, try to do left factoring and eliminate left recursions.

$$
\begin{aligned}
E &\quad \rightarrow E + T \mid E - T \mid T \\
T &\quad \rightarrow id \mid (E)
\end{aligned}
$$

Limitations

- In the last example, the grammar is left associative before conversion, but becomes right associative after conversion.
- This totally changes the structure of the parse tree for some expressions, like "$id - id - id$".
- In fact, top-down parsing cannot solve this issue. We need to either use a bottom-up parser or be stuck into the implementation details.

Limitations

- There are also some unambiguous context-free grammar cannot be converted into $LL(1)$ even after doing left factoring and left recursion elimination.
- For example, the one we used to show ambiguity elimination

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

After left factoring U becomes

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

(Recursive) Predictive Parsers

Here are some last words about (recursive) predictive parsers.

- A predictive parser can always correctly predict what it has to do next.
- Predictive parsers can always be implemented by a recursive parser without using lookahead tokens.
- Without further specification, we consider recursive parsers and predictive parsers are the same.
- One major disadvantages of (recursive) predictive parsers is that they are not very efficient in implementations. Each production rule is implemented as a function. The parser needs to make many function calls and returns, which consume a lot of resources.
- To avoid the function calls, we introduce ***nonrecursive predictive parsers***.

The End of Lecture 5

Dr. Zhiyuan Li