CS502: Compilers & Programming Systems

Top-down Parsing

Zhiyuan Li
Department of Computer Science
Purdue University, USA

PURDUE
UNIVERSITY

- There exist two well-known schemes to construct deterministic top-down parsers:
  - Using a parsing-stack and an LL(1) parsing table.
  - Recursive-descent, using recursive procedures.
  - Both kinds of top-down parsers can be automatically generated by compiler-construction. However, it is still important to study how the parser is constructed and how it works due to two reasons:
    - The grammar must be written in a proper way in order for the parser-generation to succeed. We need to understand what kind of grammars are acceptable.
    - We need to add semantic actions to the production rules in order to build the rest of the compiler. We won't know how (or where) to add such actions if we do not understand how the parser is generated and how the parser functions

Table driven top-down parsing

- It maintains a parsing stack during the parsing time
- At each derivation step (or parsing step), it consults a parsing table
- The stack contains grammar symbols, so the parsing actions may be
  - Matching a token, if top of stack is a terminal
  - Applying a production rule, if top of stack is a nonterminal
  - If the production rule is selected based on the next k tokens in the remaining input, then the parsing is LL(k)
    - The parsing table is then called a LL(k) table
    - We like it to be LL(1), at least for most of the entries

Construction of the parsing table

- After understanding how the LL(1) table-driven parsing works, we study how to construct the LL(1) table
- This requires the computation of several pieces of information
- Nullability of nonterminals
- First sets
- Follow sets

PURDUE
UNIVERSITY

Nullability

- A grammar symbol is said to be nullable if it can eventually derive null
- To compute nullability for all symbols in a grammar:
  - Initially assume all symbols A to be nonnullable
  - Repeat the following until there is no change to the nullability of any A
    - For each production rule A → <right-hand side>
      - If right hand side is ε, then mark A as nullable.
      - If right hand side is X1X2 … Xn and all Xi is nullable, then mark A as nullable.

PURDUE UNIVERSITY

The FIRST sets

- Suppose α is a string of tokens and nonterminals. By expanding the nonterminals in α, various strings can be derived.
  - FIRST(α) is the set of tokens each of which can become the leading token in some string derived from α.
  - If α => ε , then we say α is nullable.
- At each parsing step, we want to choose the production rule whose right-hand side has its FIRST set containing the next token in the input

- If two or more production rules (for the same nonterminal) have their right-hand side phrases whose FIRST sets overlap, then we will have *parsing conflicts*
  - *That is, we do* not know which rule to apply
- If the next token in the input does not belong to the FIRST set of the nonterminal on the top of stack, and if there eixst no ε production rule for the nonterminal, we have a parsing error
  - The input is incorrect.

How to compute FIRST sets?

- For each nonterminal A, initialize FIRST(A) to empty.
- For each terminal a, define FIRST(a) = { a }.
- Repeat the following until there is no change to the FIRST(A) set for any A:
  - For each production rule p: A → <right-hand side>
    - If the right hand side is X1X2 … Xn, add FIRST(X1) to FIRST(<right-hand side>).
  - For each i such that X1 through Xi-1 are all nullable, add FIRST(Xi) to FIRST(<right-hand side>).
  - Add FIRST(<right-hand side>) to FIRST(A)
- Define FIRST(p) = FIRST(<right-hand side>), where p→<right-hand side> is a production rule

PURDUE UNIVERSITY

The FOLLOW sets

- Given a nonterminal A, FOLLOW(A) is the set of terminals each of which can immediately follow A in a certain sentential form
- The use of the FOLLOW sets:
  - If the top of stack nonterminal A has a rule A→ <right hand side> that is nullable, then we check to see whether the next token in the input belongs to FOLLOW(A). If so, then the A → <right-hand side> may be applied

How to compute the FOLLOW sets

- Place $ in FOLLOW(S), where S is the start nonterminal of G, $ is the end marker for the input. Initialize FOLLOW(B) as empty for all other nonterminal B.
- Examine each production, p, in G. For each nonterminal B which appears in the right-hand side of p,
- suppose p is in the form of A => α B β, add FIRST(β) to FOLLOW(B). In addition, if β is null or nullable, then add FOLLOW(A) to FOLLOW(B).

How to Construct the LL(1) Parsing Table?

- Each row of the parsing table corresponds to a nonterminal in the grammar.
- Each column corresponds to a terminal (i.e. the look-ahead token).
- For each production rule: A → α, do the following:
  - 1. For each terminal t in FIRST(), enter "A → α" as the entry [A, t]
  - 2. If α is nullable, for each terminal *t* in FOLLOW(A), enter "A → ε" as [A, t].
- If any entry has two or production rules, we have a parsing conflict. The grammar is not LL(1).
- An empty entry indicates a syntax error in the input

How to Deal with Non-LL(1) Grammars?

(1) Consider LL(k) parsing, k > 1

- Use up to *k* look-ahead tokens. In worst case, how many columns are there in the parsing table?
- Use as small *k* as possible in most columns.
- Compute the FIRST(k) sets and FOLLOW(k) sets.
- If the LL(k) parsing table does not show any parsing conflicts, the grammar is LL(k)

- An LL(k) grammar is always unambiguous
- There exist tools to analyze a CFG and construct the LL(k) parsing table or identify LL(k) parsing conflicts.

(2) Transform the CFG into an equivalent LL(1) grammar.

- Left factoring

  $A \rightarrow \alpha\ \beta$

  $A \rightarrow \alpha\ \gamma$

  Converted into

  $A \rightarrow \alpha\ A'$

  $A' \rightarrow \beta$

  $A' \rightarrow \gamma$

- Left-recursion removal algorithm
  - A CFG is called left-recursive if it has a nonterminal A such that A → + A α for some string α.
  - If a CFG is left-recursive, then it is definitely not LL(k) for any k > 0
  - An algorithm for removing direct left-recursion
  - Example:
    - < expr > → < expr > + < term >
    - <expr> → < term >
  - We can transform those rules to
    - < expr > → < term > (+ < term>)*
  - Or equivalently
    - < expr > → < term >< etail >
    - < etail> → + < term >< etail >
    - <etail> → ε

PURDUE UNIVERSITY

General Cases

- $A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots \mid A\ \alpha_m$
- $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$, where no β_i begins with an A.
- We transform the above into

  $A \rightarrow \beta_1\ A' \mid \beta_2\ A' \mid \ldots \mid \beta_n\ A'$

  $A' \rightarrow \alpha_1\ A' \mid \alpha_2\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$

- Indirect left recursion: $A \rightarrow B\ \beta \rightarrow \ldots \rightarrow A\ \gamma$
  - To remove, substitute B by its own production rules' right-hand sides. Keep substituting until all left-recursions become direct:

    $A \rightarrow A\ \gamma$, etc

PURDUE UNIVERSITY

- A lot of other non-LL(1) cases can be reduced to a form to which either left-factoring or left-recursion (or both) can be applied
- Important: an ambiguous grammar, in general, cannot be transformed into LL(k) by simple transformations such as left-factoring and left-recursion.
  - Instead, to remove the ambiguity, the compiler designer decides what derivations are truly wanted and then introduce new production rules (and new nonterminals) to enforce such desired derivations.
  - There are also cases in which ambiguity is removed by removal of undesired production rules (without changing the set of accepted sentences).
  - Some common examples of ambiguous CFG and their corrections.