CSC 4181
Compiler Construction

Parsing

Introduction
Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.
We already learned how to describe the syntactic structure of a language using (context-free) grammar.
So, a parser only needs to do this?

Top–Down Parsing
A parse tree is created from root to leaves.
The traversal of parse trees is a preorder traversal.
Two types:
- Backtracking parser
- Predictive parser

Bottom–Up Parsing
A parse tree is created from leaves to root.
The traversal of parse trees is a reversal of postorder traversal.
Two types:
- Shift-reduce parsers
- Finite automata of items
- Error recovery

Parse Trees and Derivations

E \Rightarrow E + E
  \Rightarrow id + E
  \Rightarrow id + E * E
  \Rightarrow id + id * E
  \Rightarrow id + id * id
  \Rightarrow E + E
  \Rightarrow E + E * E
  \Rightarrow E + E * id
  \Rightarrow E + id * id
  \Rightarrow id + id * id

Top-down parsing

Bottom-up parsing

TOP DOWN PARSING
**Top-down Parsing**

- What does a parser need to decide?
  - Which production rule is to be used at each point of time?

- How to guess?
- What is the guess based on?
  - What is the next token?
  - Reserved word if, open parentheses, etc.
  - What is the structure to be built?
  - If statement, expression, etc.

**Top-down Parsing**

- Why is it difficult?
  - Cannot decide until later
  - Next token: if
  - Structure to be built: St
  - St → MatchedSt | UnmatchedSt
  - UnmatchedSt →
    - if (E) St | if (E) MatchedSt else UnmatchedSt
  - MatchedSt → if (E) MatchedSt else MatchedSt [...]
  - Production with empty string
  - Next token: id
  - Structure to be built: par
    - par → parList | λ
    - parList → exp , parList | exp

**Recursive-Descent**

- Write one procedure for each set of productions with the same nonterminal in the LHS
- Each procedure recognizes a structure described by a nonterminal.
- A procedure calls other procedures if it needs to recognize other structures.
- A procedure calls match procedure if it needs to recognize a terminal.

**Recursive-Descent: Example**

- For this grammar:
  - We cannot decide which rule to use for E, and
  - If we choose \( E \to E \ O \ F \), it leads to infinitely recursive loops.

- Rewrite the grammar into EBNF

**Match procedure**

```c
procedure match(expTok)
{
    if (token==expTok)
        getToken
    else
        error
}
```

- The token is not consumed until `getToken` is executed.

**Problems in Recursive-Descent**

- Difficult to convert grammars into EBNF
- Cannot decide which production to use at each point
- Cannot decide when to use \( \lambda \)-production \( A \to \lambda \).
LL(1) Parsing

- **LL(1)**
  - Read input from (L) left to right
  - Simulate (L) leftmost derivation
  - 1 lookahead symbol

- Use stack to simulate leftmost derivation
  - Part of sentential form produced in the leftmost derivation is stored in the stack.
  - Top of stack is the leftmost nonterminal symbol in the fragment of sentential form.

---

Example of LL(1) Parsing

| E → TX | n |
| N → FNX | N |
| X → (n+(n))nS | X |
| 18 18 17 16 |

**Finished**

| E → TX | X → ATX;λ |
| A → t1;λ |
| T → FN |
| N → MFX;λ |
| M → t |
| F → (E ) | n |

---

LL(1) Parsing Algorithm

Push the start symbol into the stack

WHILE stack is not empty ($ is not on top of stack) and the stream of tokens is not empty (the next input token is not $)

SWITCH (Top of stack, next token)

CASE (terminal a, a):
  - Pop stack; Get next token

CASE (nonterminal A, terminal a):
  - IF the parsing table entry M[A, a] is not empty  THEN
    - Get A → X1 X2 ... Xn from the parsing table entry M[A, a] Pop stack;
    - Push X1, X2 ... Xn into stack in that order
  - ELSE Error

CASE ($,$): Accept

OTHER: Error

---

LL(1) Parsing Table

If the nonterminal N is on the top of stack and the next token is t, which production rule to use?

- Choose a rule N → X such that
  - X → * tY or
  - X → * λ and S → * WNY

| t | X |
| Y | t |
| Q | Y |

---

First Set

Let X be λ or be in V or T.

First(X) is the set of the first terminal in any sentential form derived from X.

- If X is a terminal or λ, then First(X) = {X}.
- If X is a nonterminal and X → X1 X2 ... Xn is a rule, then
  - First(Xi) - {λ} is a subset of First(X)
  - First(X) - {λ} is a subset of First(X) if for all Xj<1 First(Xi) contains {λ}
  - λ is in First(X) if for all j≤n First(Xj) contains λ.
Parsing

Examples of First Set

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | num
First(exp) = {0, 1}
First(addop) = {+, -}
First(term) = {, num}
First(addop) = {+, -}
First(term) = {, num}
First(exp) = {0, 1, num}
First(addop) = {+, -}
First(term) = {, num}
First(exp) = {0, 1, num}

Algorithm for finding First(A)

For all terminals a, First(a) = {a}
For all nonterminals A, First(A) := {}
While there are changes to any First(A)
For each rule A → X₁ X₂ ... Xᵣ
If for all j < i First(Xᵢ) contains λ,
Then add First(Xᵢ) - {λ} to First(A)
For each Xᵢ in {X₁, X₂, ..., Xᵣ}
If some i < n, First(Xᵢ), First(Xᵢ₊₁), ..., First(Xᵣ) contain λ,
Then First(A) contains First(Xᵢ) - {λ}
If First(Xᵢ) is a rule, First(Xᵢ₊₁) → A
And First(Xᵢ₊₁) contains λ,
Then First(A) also contains λ.

Finding First Set: An Example

exp → term exp' exp' → addop term exp' | λ addop → + | -
term → factor term' term' → mulop factor term' | λ mulop → *
factor → (exp) | num

Follow Set

Let $ denote the end of input tokens
If A is the start symbol, then $ is in Follow(A)
If A is a rule B → X A Y, then First(Y) - {λ} is in Follow(A)
If there is a rule B → X A Y and λ is in First(Y), then Follow(A) contains Follow(B).

Algorithm for Finding Follow(A)

Follow(S) = {$}
For each A in V - {$}
Follow(A) = {};
While change is made to some Follow sets
For each production A → X₁ X₂ ... Xᵣ
For each nonterminal Xᵢ
Add First(Xᵢ+₁ Xᵢ₊₂ ... Xᵣ) - {λ} into Follow(Xᵢ)
(Note: if i = n, then Xᵢ₊₁ Xᵢ₊₂ ... Xᵣ = λ)
If λ is in First(Xᵢ+₁ Xᵢ₊₂ ... Xᵣ)
Add Follow(A) to Follow(Xᵢ)

Finding Follow Set: An Example

exp → term exp' exp' → addop term exp' | λ addop → + | -
term → factor term' term' → mulop factor term' | λ mulop → *
factor → (exp) | num

exp (num $)
exp' (λ | - $)
Constructing LL(1) Parsing Tables

FOR each nonterminal A and a production A → X
FOR each token a in First(X)
   A → X is in M(A, a)
   IF λ is in First(X) THEN
      FOR each element a in Follow(A)
         Add A → X to M(A, a)

Example: Constructing LL(1) Parsing Table

<table>
<thead>
<tr>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp (num)</td>
<td>( )</td>
</tr>
<tr>
<td>exp (num)</td>
<td>+, -</td>
</tr>
<tr>
<td>addop λ</td>
<td>*</td>
</tr>
<tr>
<td>term λ</td>
<td>exp'</td>
</tr>
<tr>
<td>term λ</td>
<td>exp'</td>
</tr>
<tr>
<td>mulop λ</td>
<td>exp'</td>
</tr>
<tr>
<td>factor  λ</td>
<td>exp'</td>
</tr>
</tbody>
</table>

LL(1) Grammar

A grammar is an LL(1) grammar if its LL(1) parsing table has at most one production in each table entry.

CAUSES OF NON-LL(1) GRAMMAR

- What causes grammar being non-LL(1)?
  - Left-recursion
  - Left factor

- Immediate left recursion

- General left recursion

- Can be removed very easily

- Can be removed when there is no empty-string production and no cycle in the grammar
Removal of Immediate Left Recursion

exp → exp + term | exp - term | term
term → term * factor | factor
factor → ( exp ) | num

Remove left recursion
exp → term exp’
exp’ → + term exp’ | - term exp’ | λ
term → factor term’
term’ → * factor term’ | λ
factor → ( exp ) | num

General Left Recursion

Bad News!
• Can only be removed when there is no empty-string production and no cycle in the grammar.

Good News!!!!
• Never seen in grammars of any programming languages

Left Factoring

Left factor causes non-LL(1)
• Given A → X Y | X Z. Both A → X Y and A → X Z can be chosen when A is on top of stack and a token in First(X) is the next token.

A → X Y | X Z
   can be left-factored as
A → X A’ and A’ → Y | Z

Example of Left Factor

ifSt → if ( exp ) st else st | if ( exp ) st can be left-factored as
ifSt → if ( exp ) st elsePart
elsePart → else st | λ

seq → st ; seq | st
   can be left-factored as
seq → st seq’
seq’ → ; seq | λ

Bottom-up Parsing

Use explicit stack to perform a parse
Simulate rightmost derivation (R) from left (L) to right, thus called LR parsing
More powerful than top-down parsing
• Left recursion does not cause problem
Two actions
• Shift: take next input token into the stack
• Reduce: replace a string B on top of stack by a nonterminal A, given a production A → B
### Example of Shift-reduce Parsing

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Parsing actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S' \rightarrow S$</td>
<td>$S \rightarrow (S)S \mid \lambda$</td>
</tr>
</tbody>
</table>

#### Reverse of Rightmost Derivation
- from left to right

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$S\rightarrow (S)S$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

### Terminology

- **Right sentential form**
  - sentential form in a rightmost derivation
- **Viable prefix**
  - sequence of symbols on the parsing stack
- **Handle**
  - right sentential form + position where reduction can be performed + production used for reduction
- **LR(0) Item**
  - production with distinguished position in its RHS

### Shift-Reduce parsers

There are two possible actions:
- shift and reduce

Parsing is completed when:
- the input stream is empty and
- the stack contains only the start symbol

The grammar must be **augmented**
- a new start symbol $S'$ is added
- a production $S' \rightarrow S$ is added

To make sure that parsing is finished when $S'$ is on top of the stack because $S'$ never appears on the RHS of any production.

### LR(0) parsing

Keep track of what is left to be done in the parsing process by using finite automata of items
- An item $A \rightarrow w \cdot B \cdot y$ means:
  - $A \rightarrow w$ $B \cdot y$ might be used for the reduction in the future,
  - at the time, we know we already construct $w$ in the parsing process,
  - if $B$ is constructed next, we get the new item $A \rightarrow w \cdot B \cdot y$

### LR(0) items

- **LR(0) Item**
  - production with a distinguished position in the RHS
- **Initial Item**
  - Item with the distinguished position on the leftmost of the production
- **Complete Item**
  - Item with the distinguished position on the rightmost of the production
- **Closure Item of $x$**
  - Item $x$ together with items which can be reached from $x$ via $\lambda$-transition
- **Kernel Item**
  - Original item, not including closure items
Finite automata of items

Grammar:
- $S' \rightarrow S$
- $S \rightarrow (S)S$
- $S \rightarrow \lambda$

Items:
- $S' \rightarrow S$
- $S' \rightarrow s$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
- $S \rightarrow (S)S$
**SLR(1) parsing**

- Simple LR with 1 lookahead symbol
- Examine the next token before deciding to shift or reduce
  - If the next token is the token expected in an item, then it can be shifted into the stack.
  - If a complete item $A \rightarrow x.$ is constructed and the next token is in Follow($A$), then reduction can be done using $A \rightarrow x.$
  - Otherwise, error occurs.
- Can avoid conflict

**SLR(1) parsing algorithm**

<table>
<thead>
<tr>
<th>Item in state</th>
<th>token</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow x. B$ (B is terminal)</td>
<td>$B$</td>
<td>shift $B$ and push state $s$ containing $A \rightarrow x.B$</td>
</tr>
<tr>
<td>$A \rightarrow x.$ (B is terminal)</td>
<td>not $B$</td>
<td>error</td>
</tr>
<tr>
<td>$A \rightarrow x.$ in Follow($A$)</td>
<td></td>
<td>reduce with $A \rightarrow x.$ (i.e. pop $x$, backup to the state $s$ on top of stack) and push $A$ with new state $d(s,A)$</td>
</tr>
<tr>
<td>$A \rightarrow x.$ not in Follow($A$)</td>
<td></td>
<td>error</td>
</tr>
<tr>
<td>$S' \rightarrow S.$</td>
<td>none</td>
<td>accept</td>
</tr>
<tr>
<td>$S' \rightarrow S.$</td>
<td>any</td>
<td>error</td>
</tr>
</tbody>
</table>

**SLR(1) grammar**

**Conflict**

- Shift-reduce conflict
  - A state contains a shift item $A \rightarrow x.Wy$ such that $W$ is a terminal and a complete item $B \rightarrow z$ such that $W$ is in Follow($B$).
- Reduce-reduce conflict
  - A state contains more than one complete item with some common Follow set.
**A grammar is an SLR(1) grammar if there is no conflict in the grammar.**

**SLR(1) Grammar not LR(0)**

**Disambiguating Rules for Parsing Conflict**

- **Shift-reduce conflict**
  - Prefer shift over reduce
    - In case of nested if statements, preferring shift over reduce implies most closely nested rule for dangling else
- **Reduce-reduce conflict**
  - Error in design
**LALR(1) parsing**

- **Goal:** reduce number of states in LR(1) parser.
- **Some states LR(1) automaton have the same core items and differ only in the possible lookahead.**
  - States $I_3$ and $I_3'$, $I_5$ and $I_5'$, $I_7$ and $I_7'$, $I_8$ and $I_8'$
- **We shrink our parser by merging such states.**
- SLR: 10 states, LR(1): 14 states, LALR(1): 10 states

**Conflicts in LALR(1) parsing**

- Most conflicts that existed in LR(1) parser can be eliminated with LALR(1)
- Can LALR(1) parsers introduce conflicts that did not exist in the LR(1) parser?
  - Unfortunately YES.
  - BUT, only reduce/reduce conflicts.
- YACC generates LALR(1) parser