Predictive Parsing

The Front End

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic
The Front End - Scanner

**Scanner**
- Maps stream of characters into words
  - Basic unit of syntax
  - \( x = x + y ; \) becomes \(<id,x><eq,><plus><id,y><semi>;<>\)
  - Characters that form a word are its *lexeme*
  - Its *part of speech* (or *syntactic category*) is called its *token type*
  - Scanner discards white space & (often) comments

**The Front End - Parser**

**Parser**
- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code
Roadmap (Where are we?)

In CMSC 330 we studied scanners & parsers
- Specifying tokens
  - Regular expressions
- Specifying syntax
  - Context-free grammars

Now we’ll look at more advanced parsers
- Predictive top-down parsing
  - FIRST, FOLLOW, FIRST+
  - The LL(1) condition
  - Table-driven LL(1) parsers
- Bottom-up shift-reduce parsers

Parsing Techniques

Top-down parsers (LL(1), recursive descent)
- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” ⇒ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)
- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars
Parsing Techniques: Top-down parsers

**LL(1), recursive descent**

1 input symbol lookahead
construct leftmost derivation (forwards)
input: read left-to-right

\[ S \Rightarrow^*_{lm} A \beta \Rightarrow^*_{lm} \delta \beta \Rightarrow^*_{lm} y \]

**LR(1), operator precedence**

1 input symbol lookahead
construct rightmost derivation (backwards)
input: read left-to-right

\[ S \Rightarrow^*_{rm} B \gamma \Rightarrow^*_{rm} \alpha \gamma \Rightarrow^*_{rm} y \]

CS430  Lecture 4  7

CS430  Lecture 4  8
Top-down Parsing

A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:
- Construct the root node of the parse tree
- Repeat until the fringe of the parse tree matches the input string
  1. At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child
  2. When a terminal symbol is added to the fringe and it doesn’t match the fringe, backtrack
  3. Find the next node to be expanded (label ∈ NT)

- The key is picking the right production in step 1
  → That choice should be guided by the input string

Picking the “Right” Production

If it picks the wrong production, a top-down parser may backtrack
Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?
- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley’s algorithm

Fortunately,
- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) and LR(1) grammars
Predictive Parsing

Basic idea

Given $A \rightarrow \alpha | \beta$, the parser should be able to choose between $\alpha$ & $\beta$

We can try to predict the correct choice by calculating

FIRST$(\alpha)$ sets

The set of tokens that appear as the first symbol in some string that derives from $\alpha$

That is, $a \in$ FIRST$(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$

FOLLOW$(A)$ sets

The set of tokens that appear immediately to the right of $A$ in some sentential form

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha | \beta$, the parser should be able to choose between $\alpha$ & $\beta$

FIRST sets

For some rhs $\alpha \in G$, define FIRST$(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from $\alpha$

That is, $a \in$ FIRST$(\alpha)$ iff $\alpha \Rightarrow^* a \gamma$, for some $\gamma$

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

FIRST$(\alpha) \cap$ FIRST$(\beta) = \emptyset$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!
The FIRST Set

\[ a \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build FIRST(X) for all grammar symbols X:

1. if X is a terminal (token), FIRST(X) := \{ X \}
2. if X ::= \varepsilon, then \varepsilon \in \text{FIRST}(X)
3. iterate until no more terminals or \varepsilon can be added to any FIRST(X):
   if X ::= Y_1 Y_2 \ldots Y_k then
      a \in \text{FIRST}(X) if a \in \text{FIRST}(Y_j) and
      \varepsilon \in \text{FIRST}(Y_j) for all 1 \leq j < i
      \varepsilon \in \text{FIRST}(X) if \varepsilon \in \text{FIRST}(Y_i) for all 1 \leq i \leq k
   end iterate

Note: if \varepsilon \in \text{FIRST}(Y_j), then \text{FIRST}(Y_i) is irrelevant, for 1 < i

CS430 Lecture 4 13

---

The FIRST Set

\[ a \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* a \gamma, \text{ for some } \gamma \]

To build FIRST(\alpha) for \alpha = X_1 X_2 \ldots X_n:

1. \varepsilon \in \text{FIRST}(\alpha) if \varepsilon \in \text{FIRST}(X_j) and
   \varepsilon \in \text{FIRST}(X_j) for all 1 \leq j < i
2. \varepsilon \in \text{FIRST}(\alpha) if \varepsilon \in \text{FIRST}(X_i) for all 1 \leq i \leq n
## LL(1) Example - First Sets

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Production</th>
<th>FIRST Sets</th>
<th>Nonterminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal → Expr</td>
<td>{ num, id }</td>
<td>Goal</td>
<td>{ num, id }</td>
</tr>
<tr>
<td>Expr → Term Expr’</td>
<td>{ num, id }</td>
<td>Expr</td>
<td>{ num, id }</td>
</tr>
<tr>
<td>Expr’ → + Expr’</td>
<td>{ + }</td>
<td>Expr’</td>
<td>{ +, -, e }</td>
</tr>
<tr>
<td></td>
<td>- Expr’</td>
<td>Term</td>
<td>{ num, id }</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>Term’</td>
<td>{ *, /, e }</td>
</tr>
<tr>
<td>Term → Factor Term’</td>
<td>{ num, id }</td>
<td>Factor</td>
<td>{ num, id }</td>
</tr>
<tr>
<td>Term’ → * Term</td>
<td>{ * }</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/ Term</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factor → num</td>
<td>{ num }</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>id</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### The FOLLOW Set

For a non-terminal A, define FOLLOW(A) as:

\[
\text{FOLLOW}(A) := \text{the set of terminals that can appear immediately to the right of } A \text{ in some sentential form.}
\]

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set.
**The FOLLOW Set**

To build FOLLOW(X) for all non-terminal X:

1. Place $ in FOLLOW( <goa ) // $ = EOF
   
   iterate until no more terminals or $ can be added
   to any FOLLOW(X):
2. If $ -> $ then
   put {FIRST($) - $} in FOLLOW(B)
3. If $ -> $ then
   put FOLLOW(A) in FOLLOW(B)
4. If $ -> $ and $ \in $ FIRST($) then
   put FOLLOW(A) in FOLLOW(B)

---

**LL(1) Example - Follow Sets**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>FIRST Sets</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>{ num, id }</td>
<td>{ $ }</td>
</tr>
<tr>
<td>Expr</td>
<td>{ num, id }</td>
<td>{ $ }</td>
</tr>
<tr>
<td>Expr’</td>
<td>{ +, - }</td>
<td>{ $ }</td>
</tr>
<tr>
<td>Term</td>
<td>{ num, id }</td>
<td>{ +, - }</td>
</tr>
<tr>
<td>Term’</td>
<td>{ *, / }</td>
<td>{ +, - }</td>
</tr>
<tr>
<td>Factor</td>
<td>{ num, id }</td>
<td>{ *, / }</td>
</tr>
</tbody>
</table>

1. Place $ in FOLLOW( <goa )
2. If $ -> $ then
   put {FIRST($) - $} in FOLLOW(B)
3. If $ -> $ then
   put FOLLOW(A) in FOLLOW(B)
4. If $ -> $ and $ \in $ FIRST($) then
   put FOLLOW(A) in FOLLOW(B)
Predictive Parsing

If $A \to \alpha$ and $A \to \beta$ and $\epsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(A)$, too.

**Define $\text{FIRST}^*(\delta)$ for rule $A \to \delta$ as**

- $\text{FIRST}(\delta) \cup \text{FOLLOW}(A)$, if $\epsilon \in \text{FIRST}(\delta)$
- $\text{FIRST}(\delta)$, otherwise

---

**Predictive Parsing**

**The LL(1) Property**

A grammar is LL(1) iff $A \to \alpha$ and $A \to \beta$ implies $\text{FIRST}^*(\alpha) \cap \text{FIRST}^*(\beta) = \emptyset$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

**Question:** Can there be two rules $A \to \alpha$ and $A \to \beta$ in a LL(1) grammar such that $\epsilon \in \text{FIRST}(\alpha)$ and $\epsilon \in \text{FIRST}(\beta)$?
Predictive Parsing

Given a grammar that has the $LL(1)$ property
- Problem: NT $A$ needs to be replaced in next derivation step
- Assume $A \rightarrow \beta_1 | \beta_2 | \beta_3$, with
  $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \emptyset$

/* find rule for $A$ */
if (current token $\in$ FIRST$^+(\beta_1)$)
  select $A \rightarrow \beta_1$
else if (current token $\in$ FIRST$^+(\beta_2)$)
  select $A \rightarrow \beta_2$
else if (current token $\in$ FIRST$^+(\beta_3)$)
  select $A \rightarrow \beta_3$
else
  report an error and return false

Grammars with the $LL(1)$ property are called **predictive grammars** because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the $LL(1)$ property are called **predictive parsers**.

One kind of predictive parser is the **recursive descent parser**. The other is a table-driven parser **table-driven parser**.

$LL(1)$ Parser Example

Is the following grammar $LL(1)$?

$S ::= a \ S \ b | \epsilon$

First($aSb$) = \{a\}
First($\epsilon$) = \{\epsilon\}

First$^*(aSb)$ = \{a\}
First$^*(\epsilon)$ = (First ($\epsilon$) - \{\epsilon\}) $\cup$ Follow ($S$) = \{$\$, b\}

$LL(1)$? YES, since \{a\} $\cap$ \{$\$, b\} = \emptyset
**LL(1) Parser Example**

Table-driven LL(1) parser

- **current input symbol**
- **rules for non-terminal**
- **non-terminal on top of the stack**

---

**Building Table-driven Top Down Parsers**

Building the complete table

- Need a row for every NT & a column for every T
- Need an algorithm to build the table

Filling in TABLE[X,y], X ∈ NT, y ∈ T

- entry is the rule X ::= β, if y ∈ FIRST+(β)
- entry is error otherwise (can treat empty entry as implicit error)

If any entry is defined multiple times, G is not LL(1)

This is the LL(1) table construction algorithm
**LL(1) Skeleton Parser**

```
token ← next_token()
push $ onto Stack // $ used to mark EOF
push the start symbol, $, onto Stack
TOS ← top of Stack
loop forever
  if TOS = $ and token = $ then
    break & report success (accept)
  else if TOS is a terminal then
    if TOS matches token then
      pop Stack // recognized TOS
token ← next_token()
    else report error looking for TOS
  else // TOS is a non-terminal
    if TABLE[TOS, token] is A → B₁B₂...Bᵦ then
      pop Stack // get rid of A
      push Bᵦ, Bᵦ₋₁, ... B₁ // in that order
    else report error expanding TOS
    TOS ← top of Stack
```
Table-driven $\text{LL}(1)$ Parser Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>S $\Rightarrow aSb$</td>
<td>S $\Rightarrow \varepsilon$</td>
<td>S $\Rightarrow \varepsilon$</td>
<td>error</td>
<td></td>
</tr>
</tbody>
</table>

Stack | Remaining Input | Action
--- | --- | ---
[$, S$] | aabb$b$, | S $\Rightarrow aSb$
[$, b, S, a$] | aabb$b$, | next input+pop
[$, b, S$] | abbb$, | S $\Rightarrow aSb$
[$, b, b, S, a$] | abbb$, | next input+pop
[$, b, b, S$] | abbb$, | next input+pop
[$, b, b, a, S$] | abbb$, | S $\Rightarrow \varepsilon$
[$, b, b, b, S$] | abbb$, | next input+pop
[$, b, b, b$] | bbb$b$, | next input+pop
[$, b, b$] | b$b$, | next input+pop
[$, b$] | b$$. | next input+pop
[$$] | $$. | accept

LL(1) Example - LL(1) Table

<table>
<thead>
<tr>
<th>Grammar</th>
<th>FIRST Sets</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal $\Rightarrow$ Expr</td>
<td>(num, id)</td>
<td>Goal ($\varepsilon$)</td>
</tr>
<tr>
<td>Expr $\Rightarrow$ Term Expr'</td>
<td>(num, id)</td>
<td>Expr ($\varepsilon$)</td>
</tr>
<tr>
<td>Expr' $\Rightarrow$ * Expr'</td>
<td>(+)</td>
<td>Expr' ($\varepsilon$)</td>
</tr>
<tr>
<td></td>
<td>- Expr'</td>
<td>Term' ($\varepsilon$, $\varepsilon$)</td>
</tr>
<tr>
<td></td>
<td>- (Expr)</td>
<td>Term' ($\varepsilon$, $\varepsilon$)</td>
</tr>
<tr>
<td>Term $\Rightarrow$ Factor Term'</td>
<td>(num, id)</td>
<td>Factor' ($\varepsilon$, $\varepsilon$)</td>
</tr>
<tr>
<td>Term' $\Rightarrow$ * Term</td>
<td>(*$\varepsilon$)</td>
<td>Factor' ($\varepsilon$, $\varepsilon$)</td>
</tr>
<tr>
<td></td>
<td>/ Term</td>
<td>(/)</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon$</td>
<td>($\varepsilon$)</td>
</tr>
<tr>
<td>Factor $\Rightarrow$ num</td>
<td>(num)</td>
<td>entry is the rule X $\Rightarrow$ $\beta$, if $y \in$ FIRST$(\beta)$</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>(id)</td>
</tr>
</tbody>
</table>

| num | id | . | . | . | / | $\varepsilon$
|-----|----|---|---|---|---|---
| Goal | Goal $\Rightarrow$ Expr | Goal $\Rightarrow$ Expr |
| Expr | Expr $\Rightarrow$ Term Expr' | Expr $\Rightarrow$ Term Expr' |
| Expr' | Expr $\Rightarrow$ * Expr' | Expr' $\Rightarrow$ * Expr' |
| Term | Term $\Rightarrow$ Factor Term' | Term $\Rightarrow$ Factor Term' |
| Term' | Term' $\Rightarrow$ * Term | Term' $\Rightarrow$ * Term |
| Factor | Factor $\Rightarrow$ num | Factor $\Rightarrow$ id |
LL(1) Languages

Question
By eliminating left recursion and left factoring, can we transform an arbitrary CFG to a form where it meets the LL(1) condition? (and can be parsed predictively with a single token lookahead?)

Answer
Given a CFG that doesn’t meet the LL(1) condition, it is undecidable whether or not an equivalent LL(1) grammar exists.

Example
\{a^n b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\} has no LL(1) grammar

Language that Cannot Be LL(1)

Example
\{a^n b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\} has no LL(1) grammar

\begin{align*}
G & \rightarrow aAb \\
& \mid aAbbb \\
A & \rightarrow aAb \\
& \mid 0 \\
B & \rightarrow aAbbb \\
& \mid 1
\end{align*}

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.