Introduction to Bottom-Up Parsing

Outline

- The strategy: shift-reduce parsing
- LR(0) example

Predictive Parsing Summary

- First and Follow sets are used to construct predictive tables
  - For non-terminal A and input t, use a production $A \rightarrow \alpha$ where $t \in \text{First}(\alpha)$
  - For non-terminal A and input t, if $\varepsilon \in \text{First}(A)$ and $t \in \text{Follow}(\alpha)$, then use a production $A \rightarrow \alpha$ where $\varepsilon \in \text{First}(\alpha)$

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method in practice

An Introductory Example

- Bottom-up parsers don’t need left-factored grammars
- Hence we can revert to a “natural” grammar for our example:
  $E \rightarrow T + E \mid T$
  $T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)$
- Consider the string: $\text{int} \ast \text{int} + \text{int}$

The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions:

- $\text{int} \ast \text{int} + \text{int}$ $T \rightarrow \text{int}$
- $\text{int} \ast T + \text{int}$ $T \rightarrow \text{int} \ast T$
- $T \ast \text{int}$ $T \rightarrow \text{int}$
- $T + T$ $E \rightarrow T$
- $T + E$ $E \rightarrow T + E$
- $E$
Observation

- Read the productions found by bottom-up parse in reverse (i.e., from bottom to top)
- This is a rightmost derivation!

\[
\begin{align*}
E & \rightarrow T + ET + E \\
E & \rightarrow TT + T \\
T & \rightarrow \text{int}T + \text{int} \\
T & \rightarrow \text{int} * T \\
T & \rightarrow \text{int} * \text{int} + \text{int} \\
T & \rightarrow \text{int} \\
E &
\end{align*}
\]

Important Fact #1

Important Fact #1 about bottom-up parsing:

*A bottom-up parser traces a rightmost derivation in reverse*

A Bottom-up Parse

\[
\begin{align*}
\text{int} * \text{int} + \text{int} & \\
\text{int} * \text{T} + \text{int} & \\
\text{T} & \\
\text{T} & \\
\text{int} & \\
\text{int} * \text{int} + \text{int} & \\
\text{E} & \\
\end{align*}
\]

A Bottom-up Parse in Detail (1)

\[
\begin{align*}
\text{int} * \text{int} + \text{int} & \\
\text{int} * \text{T} + \text{int} & \\
\text{T} & \\
\text{T} & \\
\text{int} & \\
\text{int} * \text{int} + \text{int} & \\
\text{E} & \\
\end{align*}
\]

A Bottom-up Parse in Detail (2)

\[
\begin{align*}
\text{int} * \text{int} + \text{int} & \\
\text{int} * \text{T} + \text{int} & \\
\text{T} & \\
\text{T} & \\
\text{int} & \\
\text{int} * \text{int} + \text{int} & \\
\text{E} & \\
\end{align*}
\]

A Bottom-up Parse in Detail (3)

\[
\begin{align*}
\text{int} * \text{int} + \text{int} & \\
\text{int} * \text{T} + \text{int} & \\
\text{T} & \\
\text{T} & \\
\text{int} & \\
\text{int} * \text{int} + \text{int} & \\
\text{E} & \\
\end{align*}
\]
A Bottom-up Parse in Detail (4)

int * int * int
int * int
T + int
T + T

A Bottom-up Parse in Detail (5)

int * int * int
int * int
T + int
T + T
T + E

int * int + int

A Bottom-up Parse in Detail (6)

int * int * int
int * int
T + int
T + T
T + E

E

int * int + int

Question

How do we choose the substring to reduce at each step?

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:
- Let αω be a step of a bottom-up parse
- Assume the next reduction is by X → β
- Then ω is a string of terminals

Why? Because αXω → αβω is a step in a rightmost derivation

How to build the house of cards?
Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a |
  - The | is not part of the string
  - Some texts use *
- Initially, all input is unexamined \( |x_1x_2 \ldots x_n \)

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

- **Shift**
- **Reduce**

Shift

- **Shift**: Move | one place to the right
  - Shifts a terminal to the left string

\[ ABC|xyz \Rightarrow ABC\timesyz \]

Reduce

- Apply an *inverse production* at the right end of the left string
  - If \( A \rightarrow xy \) is a production, then

\[ Cbxyijk \Rightarrow CbA\timesijk \]

The Example with Shift-Reduce Parsing

\[
\begin{align*}
| & \text{int} * \text{int} + \text{int} & \text{shift} \\
\text{int} & \text{int} + \text{int} & \text{shift} \\
\text{int} & \text{int} + \text{int} & \text{shift} \\
\text{int} & \text{int} & \text{reduce } T \rightarrow \text{int} \\
\text{int} & T \text{ int} & \text{reduce } T \rightarrow \text{int} * \text{T} \\
T & \text{int} & \text{shift} \\
T & \text{int} & \text{shift} \\
T & \text{int} & \text{reduce } T \rightarrow \text{int} \\
T & T & \text{reduce } E \rightarrow \text{T} \\
T & E & \text{reduce } E \rightarrow T * E \\
E & & \\
\end{align*}
\]

A Shift-Reduce Parse in Detail (1)

\[
\begin{align*}
| \text{int} * \text{int} + \text{int} & \\
\text{int} * \text{int} + \text{int} & \text{reduce } T \rightarrow \text{int} * \text{T} \\
\text{int} * \text{int} + \text{int} & \text{reduce } E \rightarrow T * E \\
\end{align*}
\]

\[ \uparrow \]
A Shift-Reduce Parse in Detail (8)

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
```

A Shift-Reduce Parse in Detail (9)

```
| int * int + int
int | * int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
```

A Shift-Reduce Parse in Detail (10)

```
| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
```

A Shift-Reduce Parse in Detail (11)

```
| int * int + int
E
int | * int + int
int | + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |
```

The Stack

- Left string can be implemented by a stack
- Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off the stack (production rhs) and pushes a non-terminal on the stack (production lhs)

Key Issue (will be resolved by algorithms)

- How do we decide when to shift or reduce?
  - Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake: No way to reduce to the start symbol E
Conflicts

• Generic shift-reduce strategy:
  – If there is a handle on top of the stack, reduce
  – Otherwise, shift

• But what if there is a choice?
  – If it is legal to shift or reduce, there is a shift-reduce conflict
  – If it is legal to reduce by two different productions, there is a reduce-reduce conflict

Source of Conflicts

• Ambiguous grammars always cause conflicts

• But beware, so do many non-ambiguous grammars

Conflict Example

Consider our favorite ambiguous grammar:

\[
E \rightarrow E + E \\
| E * E \\
| (E) \\
| \text{int}
\]

One Shift-Reduce Parse

| int * int + int | shift |
| . . . . . . . . . . . . | . . . . . . . . . . . . |
| E * E | + int | reduce \( E \rightarrow E * E \) |
| E | + int | shift |
| E + int | shift |
| E + int | reduce \( E \rightarrow \text{int} \) |
| E + E | reduce \( E \rightarrow E + E \) |
| E | |

Another Shift-Reduce Parse

| int * int + int | shift |
| . . . . . . . . . . . . | . . . . . . . . . . . . |
| E * E | + int | shift |
| E * E | + int | shift |
| E * E | + int | reduce \( E \rightarrow \text{int} \) |
| E * E | reduce \( E \rightarrow E + E \) |
| E * E | reduce \( E \rightarrow E * E \) |
| E | |

Example Notes

• In the second step \( E * E | + \text{int} \) we can either shift or reduce by \( E \rightarrow E * E \)

• Choice determines associativity of + and *

• As noted previously, grammar can be rewritten to enforce precedence

• Precedence declarations are an alternative
Precedence Declarations Revisited

- Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways
- Declaring "*" has greater precedence than "+" causes parser to reduce at E * E | + int
- More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a +

Precedence Declarations Revisited (Cont.)

- The term "precedence declaration" is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!

Nitty Gritty Algorithms

- See pages 215-257 in the Dragon Book
  - How to determine handles
  - Algorithms to construct a DFA describing a parse
  - LR(0), LR(1), SLR, LALR
- Next class
  - Yacc does most of it for you