Recap of Terminal Substitution Formalism

• For simple tasks, it is usually sufficient
• Speech recognizer just outputs your tokens rather than the words
• Easy to do; just add extra symbols to speech recognition grammar
  - This approach:
    • Semantics formed by substituting terminals for meaning

Overview

• Bottom-up Parsing
• Derivations and Parse Trees
• Grammar Formalism
• Parsing & Semantics
  • Terminal Substitution
Overview

- Bottom-Up Parsing
- Derivations and Parse Trees
- Grammar Formalism
- Parsing & Semantics
- Terminal Substitution

Limitations

- Tokens in semantics must be in same order as the words
- Cannot insert tokens that do not correspond to a word
- Hard to have semantics of form turn('state', device, room)
- Cannot have variations that are in different orders
- Tokens in semantics must be in same order as the words
- Tokens in semantics must be in same order as the words
- Tokens in semantics must be in same order as the words
Compositional Meaning

• Can decompose a sentence into hierarchical groupings of words.

Semantic Interpretation needs to consist of:
- Recursive procedure starting at word meaning and reaching sentence meaning
- Constituents might be words, or phrases themselves
- Meaning of each grouping is derived from meaning of its constituents

Towards a New Semantic Building Formalism

• Need a more powerful semantic building formalism
• Semantic interpretation are often black boxes, that you cannot change
• With more complex semantics, could slow recognizer down too much
• We should build semantics outside of speech recognizer
• Assumes speech recognizer built the semantics
• Up to now, semantics were part of the grammar rules
• Need a more powerful semantic building formalism
Overview

Terminal Substitution

• Parsing & Semantics
  ⇒ Grammar Formalism
  • Derivations and Parse Trees

Split Up Parsing & Semantics

• Separate recognition from interpretation

- Algorithm should not be specific to the exact rules we write
- Need algorithms that will do parsing and semantics

- Syntax (Parsing):
  + Use a regular grammar
  + Determine hierarchical sequential grouping of words
  + Derive meaning of whole from meaning of parts

- Semantics:
  + Use parallel rules for semantic building formalism
  + Derive meaning of whole from meaning of parts

Split UP Parsing & Semantics

But be able to work with any set of rules
Ensuring Regularity

- Rules are ordered
- Non-terminal can occur on both sides of more than one rule
- Non-terminal on left hand side can appear on right hand side, but must be on one of the ends (not embedded)
- Non-terminal cannot appear on right hand side in any rule after its own definition
- Not allowing optional or repetitions
  - Not because they would alter the expressibility
  - Easier to write algorithms if they don’t have to handle this

### Yet Another Regular Grammar Notation

- Start state
- Non-terminals
  - For clarity: tokens that start with uppercase letter
- Terminals
  - For clarity: tokens that start with lowercase letter
- Non-terminals can appear on left hand side of more than one rule
- Rules are ordered

So far, almost identical to CSLU speech recognition grammar

\[
\text{Non-terminal} \rightarrow \text{sequence of terminals and non-terminals}
\]

\[
\text{All rules are of form:}
+ \text{for clarity: tokens that start with lowercase letter}
\]

\[
\text{Terminals}
\]

\[
\text{Non-terminals}
\]

\[
\text{Start state}
\]

Since we need to write a parsing algorithm, let’s simplify the regular grammar notation...
Transforming Grammar to New Notation

Old Grammar For Amounts

OneToNine <- one | ... | nine

Tys <- ( twenty | ... | ninety ) [ OneToNine ]

Tens <- Tys | Teens | OneToNine

Hundreds <- OneToNine hundred [ Tens ]

Thousands <- OneToNine thousand [ Hundreds ]

Amount <- ( Thousands | Hundreds | Tens ) dollars

New Grammar For Amounts

OneToNine <- one

Tys <- twenty OneToNineOpt

Tens <- Tys | Teens | OneToNine

Hundreds <- OneToNine hundred [ Tens ]

Thousands <- OneToNine thousand [ Hundreds ]

Amount <- ( Thousands | Hundreds | Tens ) dollars
Overview

- Bottom-Up Parsing
- Derivations and Parse Trees
- Grammar Formalism
- Parsing & Semantics
- Terminal Substitution

New Grammar For Amounts

OneToNine <- one
...
OneToNine <- nine
Teens <- ten
...
Teens <- nineteen
OneToNineOpt <- OneToNine
OneToNineOpt <-
Tys <- twenty OneToNineOpt
...
Tys <- ninety OneToNineOpt
Tens <- Tys
Tens <- Teens
Tens <- OneToNine
TensOpt <- Tens
TensOpt <- ǫ
Hundreds <- OneToNine hundred TensOpt
HundredsOpt <- Hundreds
HundredsOpt <- ǫ
Thousands <- OneToNine thousand HundredsOpt
Amount <- Thousands dollars
Amount <- Hundreds dollars
Amount <- Tens dollars
Parse Tree

**Derivation**

- We will directly use grammar, rather than convert to FSM
- Grammar specifies what is or is not in language
- Word sequence is in grammar if there is derivation of it from start symbol
- Derivation: sequence $s_1s_2\ldots s_n$ where $s_i$ is derived from $s_{i-1}$
- Rule: $A \rightarrow X \mid Y$ if $A$ is a non-terminal and $X \mid Y$ is a terminal or non-terminal
- We will directly use grammar, rather than convert to FSM
- Parse tree captures how derivation grouped things into phrases
- Does not fully capture order that rules were applied, but that doesn’t really matter for semantic analysis

- Amount
  - ✟✟✟✟✟
  - ❍❍❍❍❍
- Thousands
  - ✟✟✟✟✟✟✟
  - ❍❍❍❍❍❍❍
- OneToNine
- Three
- Ten
- Twenty

- Parse tree captures how derivation grouped things into phrases
- Does not fully capture order that rules were applied, but that doesn’t really matter for semantic analysis
Overview

• Terminal Substitution
• Parsing & Semantics
• Grammar Formalism
• Derivations and Parse Trees

⇒ Bottom-Up Parsing

Parsing Algorithm

We want an algorithm that will work with any rule set and sound.

- As long as it is complete and sound:
  - Can substitute any algorithm in
  - Sound: Don't find a parse if one doesn't exist
  - Complete: Find parse if it exists

Must be guaranteed to do the right thing:

Wanted an algorithm that will work with any rule set and sound.

We want an algorithm that will take a sequence of words and
Bottom-Up Parsing

Start with word sequence to be parsed
While token sequence is not start symbol
  For each rule
    For each token in sequence
      If rule can be applied ending at that token
        Rewrite token sequence with rule
  Next while
Halt with yes

Will it always halt?
- If we couldn't rewrite the sequence, then we should halt
- If no parse, will keep looping

Bottom-Up Parsing Motivation

Grammar

String: tell me my savings account balance

Det  →  the
Det  →  my
Det  →  the
Type  →  savings
Type  →  checking
Type  →  line of credit
Account  →  Det Type account
Account  →  Det Type account
Account  →  Det account
Account  →  Det account
Account  →  Det Type account
Command  →  tell me the balance of account
Command  →  tell me the account balance
Command  →  report account balance
Command  →  what is the balance of account
Command  →  what is the account balance
Command  →  checking account balance
Command  →  line of credit account balance
Command  →  savings account balance
Recursive Rules and Halting

- Problem with empty body rules
  \[ A \rightarrow A \]

- Problem with right recursive rules?
  \[ A \rightarrow aA \]

- Problem with left recursive rules?
  \[ A \rightarrow Aa \]

Start with word sequence to be parsed

While token sequence is not start symbol
  \[ \text{rewritten} \rightarrow \text{no} \]

For each rule
  For each token in sequence
    If rule can be applied ending at that token
      Rewrite token sequence with rule
      \[ \text{rewritten} \rightarrow \text{yes} \]
      Next while
    \[ \text{rewritten} \rightarrow \text{no} \]

If rewritten is no
  Halt with no

Halt with yes
While list of alternatives is not empty
For each rule
For each token in sequence
If rule can be applied ending at that token
Rewrite sequence with rule
If new sequence is start symbol
Halt with yes
Otherwise
Store new sequence at end of list of alternatives
Halt with no

Revision 2: Breath First Version

Add word sequence to list of alternatives

Completeness of Algorithm

Make it Breath First

• Now stick
• Withdraw any dollars from bank
• Withdraw any dollars from checkings

Parsing Progress

• Transfer to another account
• Command withdraw Amount from Act
• Command withdraw Amount from Act

Grammar

Could get stuck
• Even if we standardized order of applying rules, still inefficient
  • "an arrow" is always parsed as NP
  • Not leveraging locality in the parsing
  • Should just have to do this once, not once for each different derivation

More inefficiencies

CSE560 Class 02  26
© P. Heeman, 2003

• Searching over derivation sequences
  • Order of a lot of decisions does not matter
    • "Time flies like an arrow"
    • Doesn't matter if we pick "time" as a noun before or after "flies" is a verb
  • No simple way to correct algorithm for this
    • Doing "leftmost" derivation, only works in top-down approach

Inefficient

CSE560 Class 02  25
© P. Heeman, 2003
Recap

- Want to build semantic representations
- Parse to determine hierarchical grouping
- Semantic analysis requires
- More complex ones need to be done as a separate process
- Regular grammar
- Simple semantic building formulas are built into speech recognizer’s

• Recap