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.

Semantics formed by substituting terminals for meaning.

Overview

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

Limitations

- Cannot interpret tokens that do not correspond to a word

- Tokens in semantics must be in same order as the words

- Cannot insert tokens that do not correspond to a word

- Tokens in semantics must be in same order as the words

- Cannot have semantics of form

- Tokens in semantics must be in same order as the words

- Cannot have variations that are in different orders
Compositional Meaning

- Determining meaning for each phrase from its constituents
- Determining phrase meaning (hierarchical grouping 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
- Can decompose a sentence into hierarchical grouping

Towards a New Semantic Building Formalism

- Need a more powerful semantic building formalism
- Up to now, semantics were part of the grammar rules
- Speech recognizers 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
- Need a more powerful semantic building formalism
Overview

- Terminal Substitution
- Derivation and Parse Trees
  \[ \Rightarrow \]
  Grammar Formalism
  \[ \Rightarrow \]
  Parsing & Semantics
  \[ \Rightarrow \]
  Derivations and Parse Trees

Split Up Parsing & Semantics

- Separate recognition from interpretation
- Use parallel rules for semantic building formalism
- Derive meaning of whole from meaning of parts

- Syntact (Parsing):
  - Use a regular grammar
  - Derive hierarchical sequential grouping of words

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

But be able to work with any set of rules
- Algorithm should not be specific to the exact rules we write
- Need algorithms that will do parsing and semantics
Ensuring Regularness

Rules are ordered

Non-terminal can appear on left hand side 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

Since we need to write a parsing algorithm,

Let’s simplify the regular grammar notation

All rules are of form:

- For charity: tokens that start with lowercase letter
- For charity: tokens that start with uppercase letter
- For charity: tokens that start with a punctuation symbol

So far, almost identical to CSLU speech recognition grammar

Non-terminal → sequence of terminals and non-terminals

For clarity: tokens that start with lowercase letter

For clarity: tokens that start with uppercase letter

Start state

For clarity: tokens that start with a punctuation symbol

For clarity: tokens that start with lowercase letter
Transforming Grammar to New Notation

Old Grammar For Amounts

```
OneToNine <- one | ... | nine

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

OneToNineOpt <- OneToNine

Tys <- twenty OneToNineOpt
...

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

New Grammar For Amounts

```
OneToNine <- one | ... | nine

Tys <- twenty | ... | ninety

Tys <- OneToNine

OneToNineOpt <- OneToNine

OneToNineOpt <- ǫ

Tys <- twenty OneToNineOpt
...
```

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

```
```

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 | ... | nine

Tys <- twenty | ... | ninety

Tys <- OneToNine

OneToNineOpt <- OneToNine

OneToNineOpt <- ǫ

Tys <- twenty OneToNineOpt
...
```

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

```
```
Overview

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

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

- Does not fully capture order that rules were applied, but that doesn't really matter for semantic analysis.
- Captures how derivation grouped things into phrases.

Derivation

- We will directly use grammar, rather than convert to FSM.
Overview

Terminal Substitution

• Terminal Substitution

Parsing & Semantics

• Grammar Formalism

• Derivations and Parse Trees

⇒ Bottom-Up Parsing

Parsing Algorithm

We want an algorithm that will take a sequence of words and figure out its parse if it is in the grammar.

- Want algorithm that will work with any rule set
- Must be guaranteed to do the right thing
  - Complete: Find parse if it exists
  - Sound: Don't find a parse if one doesn't exist
- Can substitute any algorithm in

As long as it is complete and sound

We want an algorithm that will take a sequence of words and figure out its parse if it is in the grammar.
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

End while

If no parse, will keep looping

If we couldn’t rewrite the sequence, then we should halt

Will it always halt?

Halt with yes

Next while

Rewrite token sequence with rule

While token sequence is not start symbol

Will it always halt?

If no parse, will keep looping

String: tell me my savings account balance

Grammar:

- **Account:** Det Type account
- **Command:** Det Type account
- **Type:** savings
- **Type:** checking
- **Type:** line of credit

Det:

- **my**
- **the**

Defer:

- **the**
- **line of credit**
- **check**
- **account**
- **services**
- **savings**
- **checking**
- **debt**
- **balance**

Command:

- **what is the balance of account**
- **tell me the balance of account**
- **tell me balance of account**
- **tell me my account balance**
- **show me the balance of account**
- **show me account balance**
- **show me my account balance**
- **show me my balance**
- **show me my savings account balance**
- **show me my checking account balance**
- **show me my line of credit account balance**

**Start**
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?
\[ A \rightarrow aA \]
Add word sequence to list of alternatives

While list of alternatives is not empty
  Pull out first alternative from list
  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

Make it breath first

Could get stuck

Now stick
More inefficiencies

- Not leveraging locality in the parsing
- "an arrow" is always parsed as NP
- Even if we standardized order of applying rules, still inefficient

Inefficient

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

Not once for each different derivation
- Should just have to do this once.
- "an arrow" is always parsed as NP
Recap

Want to build semantic representations

• Semantic analysis requires
  - Parsing to determine hierarchical grouping
  - More complex ones need to be done as a separate process

• Simple semantic building formalism are built into speech recognizer's
  - Parse meaning to each group starting from leaves