Overview

- Event Semantics
- FOPC
- Hierarchical Frames
- Frame Semantics
- Applying the Semantics
- Semantic Processing Formalism

Recap

- Introduced a simple parser
- Composed from meaning of parts
- Semantics: meaning of whole of words
- Parsing and Hierarchical Grouping
- Do interpretation on ASR output
- Semantic building formalism might be too complex to fit into ASR
- Need to map responses to semantics
- Need to interpret responses longer than single word
Splitting Semantics from Syntax

• Previous formalisms intermixed the semantic formalism with the syntactic rules.

\[
\text{Room} \leftarrow \text{living%livingroom room}%
\]

\[
\text{Command} \leftarrow \text{turn% State the Device in the Room}
\]

• Let's separate syntactic rule from semantic formalism.

\[
\text{Room} \leftarrow \text{living room}
\]

- 2nd line gives semantics for the syntactic rule.
- And the given semantics for the semantic rule.

\[
\text{Room} \leftarrow \text{Turing room}
\]

Recap: Terminal Substitution

• Mechanism used to build meaning of the sentence.

- Need to refer to semantics of State, Device, and Room.

- \[
\text{Command} \leftarrow \text{turn State the Device in the Room}
\]

- Semantics (and line) will always be a single thing, which will be the semantics of the variable on the left-side of the rule.

- Semantics (and line) will always be a single thing, which will be the semantics of the variable on the left-side of the rule.

• Syntactic rule with non-terminals on right-hand side.

- Need to refer to semantics of State, Device, and Room.

- Semantics (and line) will always be a single thing, which will be the semantics of the variable on the left-side of the rule.

- Semantics (and line) will always be a single thing, which will be the semantics of the variable on the left-side of the rule.

• Recap: Terminal Substitution
Use semantic representation of `turn(State,Device,Room)`

- Command <- `turn State the Device in the Room`
- Command <- `turn Device State in the Room`
- Command <- `turn Room Device State`

**Referring to the Semantics**

- Command <- `turn State the Device in the Room`
- Command <- `turn Device State in the Room`
- Command <- `turn Room Device State`

- Could just use non-terminal name in semantic rule
- Syntactic rule with non-terminals on right hand side

**Home Control Example**
Banking Example

• Use semantic representation of transfer(Amount,From,To)

  Command <- transfer Amount from Act to Act
  Account <- Det Type account
  Type <- savings

  Command <- transfer Amount to Act from Act
  Account <- Det Type account
  Type <- checking

  Command <- transfer from Act to Act Amount
  Account <- Det Type account
  Type <- line of credit
Aside: Variations in Notation

- Exact notation used for semantics does not matter
- We're using a notation that is convenient for Tcl’s regsub
- Would like advantage of its variable unification to make this easy
- In Prolog:
  - We using a notation that is convenient for Tcl’s regsub
  - Exact notation used for semantics does not matter

Duplicate Non-Terminals on Right-Hand Side

- Parallel syntactic and semantic rules
- Terminal Substitution
- Semantic building formalisms
  - Now can account for different transfer variations, and have them give the exact same semantics
  - Allow N. S to allow S to refer to semantics of N
  - Syntactic rule might have the same non-terminal twice
Example Parse & Semantic Interpretation

• Although parsing and semantics are separated in each rule, the parser will build semantics while it is applying parsing rules.

Parse: "tell me my savings account balance"

`Det <the> <of> <my> <savings> <account> <balance>`

Command `←` balance of Account `balance(<Account>)`

Account `←` Det Type `account(<Type>)`
Create parse list with word sequence as only member
Create semantics list with word sequence as only member
While parse list is not empty
Pull out first alternative from parse list, call it \( p \)
Pull out first alternative from semantics list, call it \( s \)
For each rule \( r \)
Let \( r.b \) be the body of \( r \)
For \( n = 1 \) to length of \( p \)
Let \( p_r \) be subsequence of \( p \) from \( n \) to \( n + \) length of \( r.b \) - 1
Let \( s_r \) be subsequence of \( s \) from \( n \) to \( n + \) length of \( r.b \) - 1
If \( r.b = p_r \) then
Create \( p' \) by rewriting \( p_r \) of \( p \) with rule's head
Create \( s' \) by rewriting \( s_r \) of \( s \) with appropriate semantics
If \( p' \) is start symbol
Halt with yes
Otherwise
Add \( p' \) to end of parse list
Add \( s' \) to end of semantics list
Otherwise
Halt with no

Augmenting the Bottom-Up Parser
• Old version: What do we need to change?
+ Actually, I modified old version to make it clearer when needs to be changed
Simplifying Computation of Semantics

Rewrite rules during initialization to simplify building the semantics.

- Change semantic rules so it just refers to the position of non-terminal in
  the syntactic rule

Example: Command <- transfer Amount from Act:From to Act:To

\[\text{transfer(<Amount>,<From>,<To>)}\]

\[\text{Command <- transfer Amount from Act to Act}\]

Determining the Semantics

- What needs to be done in:
  - Create \( s' \) by rewriting \( s \) with appropriate semantics
  - Create \( \pi \) by rewriting \( s' \) with appropriate semantics

Example:

Create \( s' \) of \( s \) with appropriate semantics

When needs to be done in:

Create \( \pi \) of \( s' \) with appropriate semantics
We have already seen predicate argument structure used for semantics:

- Bank Account: turn (State, Device, Room)
- Frame type
  - Each argument has a specific meaning
  - Each takes a fixed number of arguments

A generalization of this are Frames:

- Frame type
  - A fixed number of slots, which can have values

Banking Application could have three frame types:

- withdrawal
- balance
- transfer

Frame Semantics

Overview:

- Event Semantics
- FOPC
- Hierarchical Frames
- Frame Semantics
- Applying the Semantics
- Semantic Processing Formalism
Notation for Frame Semantics

- Notation:
  - Frame name
  - Followed by list of slot & fillers
- Order of slots doesn't matter
- Followed by list of slot & fillers
- Frame name
- Example (using Tcl lists):
  - {withdraw account=savings amount=20}
  - {balance account=checking}
  - {transfer from=savings to=checking amount=20}

Frame Examples

Withdraw 20 dollars from my savings account
Give me 20 dollars from savings
20 dollars please from savings
Can you give me 20 dollars from my savings
Withdraw 20 dollars from my savings account
Check my checking balance
Can you tell me the balance in my checking account
Would you be so kind as to tell me my checking balance
Move 20 dollars from savings to checking
What is my checking balance
Can you transfer 20 dollars from savings to checking
20 dollars to checking from my savings account
Please move 20 dollars into checking out of savings
A Slightly Different Frame Representation

• Withdraws, transfers & deposits can be thought of as moves
  - From/to can be cash, savings, checking, line of credit:
    - deposit: \{move from=savings to=checking\}
    - withdraw: \{move from=savings to=cash\}
    - transfer: \{move from=savings to=checking\}

• What frame inventory is best?
  - What is easiest for semantic processing
  - What does the back-end expect?

Adding Frame Semantics to Grammar

```plaintext
We have standardized rules to ALWAYS have slot names

acct <- deposit savings accept
command <- transfer from acct:from to acct:to amount:amt
command <- transfer amount:amt from acct:from to acct:to
```

CSE560 Class 03: 2003 P. Heeman, 2003

CSE560 Class 03: 21

Adding Frame Semantics to Grammar

CSE560 Class 03: 22

Adding Frame Semantics to Grammar

CSE560 Class 03: 23

Adding Frame Semantics to Grammar

CSE560 Class 03: 24

Adding Frame Semantics to Grammar

CSE560 Class 03: 25

Adding Frame Semantics to Grammar

CSE560 Class 03: 26
Toward More Complex Representation

- Could be cumbersome, especially for other applications
  - Account type: checking John, loan 158, loan 160
  - Cannot do compositional meaning
  - Could be cumbersome, especially for other applications

+ travel domain: from can be city, airport, bus station, train station frame

Overview

- Semantic Processing Formalism
- Applying the Semantics
- Frame Semantics
- Hierarchical Frames
- Event Semantics
- FOPC
Grammar Rules

For hierarchic frames, need a way to delimit them.

DetOpt -> John's
DetOpt -> e
DetOpt -> the
DetOpt -> my

Act <- DetOpt savings ActOpt
{ ... } Use Tcl embedded lists

Note: some frame slots can be mandatory, some optional.

Hierarchical Frames

<table>
<thead>
<tr>
<th>From</th>
<th>Move Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>cash</td>
<td>to</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>To</th>
<th>Move Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>cash</td>
<td>from</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Account Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
</tr>
<tr>
<td>owner</td>
</tr>
<tr>
<td>amount</td>
</tr>
</tbody>
</table>

From/To slots can have either a simple value as filler or a frame.
- Could have a slot for account number
- Could have a slot for owner
- Would have a slot for type: checking, savings, loan, line of credit
- Might want account to be a frame
First Order Predicate Calculus (FOPC)

- Back end will figure out what X refers to, say account 1098838
- Back end needs to be sophisticated enough to handle FOPC
  - We just refer to it with the variable X and state what we know about it
  - We don't need to know how backend refers to John's checking account, we just refer to it with the variable X, and state what we know about it.

Example: transfer 20 dollars from John's checking to my savings

transfer(20, X, Y)
\(\land\) type(X, checking) \(\land\) owner(X, john)
\(\land\) type(Y, savings)

- Variables can be quantified with \(\exists X\) and \(\forall X\)
- Allowing quantifiers on variables \(\exists X\) and \(\forall X\)
- When we do not know the name of the thing we want to refer to, we can use variables to refer to it
- Allowing variables in predicates \(X, Y, \ldots\)
- Allowing conjunctions of predicates \(\land\)

Predicate argument structure can be expanded into FOPC

E | V | E = V =
---|---|---
F | P | F ⊆ P
H | F | Hierarchical Frames
F | F | Frame Semantics
A | F | Applying the Semantics
S | F | Semantic Processing Formulation

Overview

CSE560 Class 03: 27

P. Heeman, 2003
Lambda expressions are ideal for capturing meaning of arbitrary phrases.

- \( \forall X \, \text{owner}(X, \text{john}) \) means we don't need a predicate \( \text{ownerOfJohn}(X) \)
- \( \forall X \, \text{owner}(X, \text{john}) \) means we don't need a predicate \( \text{ownedByJohn}(X) \)
- From a small set of predicates, we can build arbitrary predicates.

Lambda expressions are a way to build an arbitrary predicate.

We can capture this intuition: use of lambda expressions.

\( \forall X \, \text{owner}(X, \text{john}) \)

It really should be something closer to:

\( \text{owner}(X, \text{john}) \) seems weird as semantics of John's

Lambda Expressions

First Attempt at Mapping to FOPC

Semantics of "transfer fifty from savings to John's checking"

<table>
<thead>
<tr>
<th>Command</th>
<th>transfer Amt from Act:From to Act:To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act</td>
<td>type(X, savings) ( \land ) type(X, checking) ( \land ) owner(X, john)</td>
</tr>
</tbody>
</table>

Oops... Giving each rule its own variable also will not help.
Applying Lambda Expressions

Predicate symbol can be applied to value to produce predicate:

ownedByJohn applied to boat224 gives ownedByJohn(boat224)

Can do the same with lambda expressions:

\( \lambda X \) owner(X,john) applied to boat224 gives \( (\lambda X \) owner(X,john))(boat224) \)
which can be simplified to owner(boat224,john)

Use in Semantic Rules:

- When we want to apply a lambda expression, write:
  - apply(Expression,Value)

Example:

\( \lambda X \) owner(X,john)) applied to boat224 gives owner(boat224,john)

Predicate symbol can be applied to value to produce predicate:

Example Revisited

- Previous


- New

DetOpt <- john's owner(X,john) Act <- DetOpt savings ActOpt type(X,savings) Command <- transfer Amt from Act:From to Act:To transfer(Amt,X,Y) ∧ From ∧ To

Previous

DetOpt <- john's owner(X,john) Act <- DetOpt savings ActOpt type(X,savings) Command <- transfer Amt from Act:From to Act:To transfer(Amt,X,Y) ∧ From ∧ To
Worked Out Example

Transfer fifty from savings to John's checking

In building the semantics, anytime an `apply` is seen, the expression should be simplified to remove the `apply`.

Example Revisited Again

Previous

NEW

Previous
Event Semantics

- Have a constant that represents the event, say \( e \)
- Has a type (similar to slots of frame)
  \[ \text{type}(e) = \text{move} \]
- Has roles (similar to slots of frame)
  \[ \text{from}(e) = \text{savings} \]
  \[ \text{to}(e) = \text{checking} \]

\[ \exists E . \text{type}(E) = \text{move} \land \text{from}(E) = \text{savings} \land \text{to}(E) = \text{checking} \]

Depending on the richness of the domain, you might do something this powerful:

- Lambda expressions can be used for expressing partial meanings.

- Depending on the richness of the domain, you might do:
  \[ \exists E . \text{type}(E) = \text{move} \land \text{from}(E) = \text{savings} \land \text{to}(E) = \text{checking} \]

- Has a type (similar to frame type)
  \[ \text{type}(e) = \text{move} \]

- Has roles (similar to slots of frame)
  \[ \text{from}(e) = \text{savings} \]
  \[ \text{to}(e) = \text{checking} \]

- Have a constant that represents the event, say \( e \)
Recap

• Semantic building formalisms:
  - Event Semantics
  - PFOC
  - Hierarchical Frames
  - Frames
  - Predicate and arguments

Semantic formalisms:
- Parallel syntactic and semantic rules in Lambda Calculus
- Parallel syntactic and semantic rules
- Term and substitution
- Semantic building formalisms