Computer Languages

Parsing

February 2nd

Is it a legal sentence?

Given an input string that alleges to be a sentence infer a derivation (or conclude that no such exists).

Example

```
<A HREF="http://www.hh.se/CC-lab"> <li>CC-lab</A>
<A>  </A>(Computing and Communication)
<BR>
<A HREF="http://www.hh.se/IS-lab"><li>IS-lab</A>
<A> </A>(Intelligent Systems)
<BR>
<A></A>
<A HREF="http://www.hh.se/MI-lab"><li>MI-lab</A>
(Man and Information technology)
```

Is it a legal sentence?

Given an input string that alleges to be a sentence infer a derivation (or conclude that no such exists).

Example

```
<A HREF="http://www.hh.se/CC-lab"> <li>CC-lab</A>
<A>  </A>(Computing and Communication)
<BR>
<A HREF="http://www.hh.se/IS-lab"><li>IS-lab</A>
<A> </A>(Intelligent Systems)
<BR>
<A></A>
<A HREF="http://www.hh.se/MI-lab"><li>MI-lab</A>
(Man and Information technology)
```

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
  - it is the string!
- The root is known
  - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
  - it is the string!
- The root is known
  - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* | `TRUE` *true* & `ID` *y*

Build a parse tree

- The leaves are known
  - it is the string!
- The root is known
  - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* | `TRUE` *true* & `ID` *y*

Build a parse tree

- The leaves are known
    - it is the string!
- The root is known
    - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
    - it is the string!
- The root is known
    - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
    - it is the string!
- The root is known
    - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
  - it is the string!
- The root is known
  - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
    - it is the string!
- The root is known
    - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
    - it is the string!
- The root is known
    - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Discover a derivation

The string appears to the parser as a sequence of tokens, some of them with semantic values attached.

Example

`ID` *x* `|` `TRUE` *true* `&` `ID` *y*

Build a parse tree

- The leaves are known
  - it is the string!
- The root is known
  - it is the start symbol!

They have to be connected following a derivation!

Strategies

1. Top-Down
2. Bottom-UP

What productions should be applied?

Running example

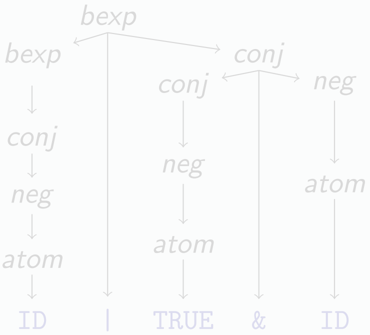
The grammar

$$\begin{array}{lcl}
bexp & \rightarrow & bexp \mid conj \\
 & \mid & conj \\
conj & \rightarrow & conj \ \& \ neg \\
 & \mid & neg \\
neg & \rightarrow & \neg \ atom \\
 & \mid & atom \\
atom & \rightarrow & \mathtt{TRUE} \\
 & \mid & \mathtt{FALSE} \\
 & \mid & \mathtt{ID} \\
 & \mid & (bexp)
\end{array}$$

The sentence

x | true & y

The parse tree



Running example

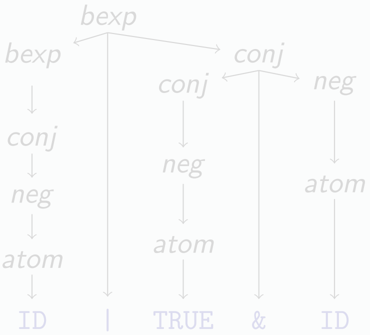
The grammar

bexp → *bexp* `|` *conj*
| *conj*
conj → *conj* `&` *neg*
| *neg*
neg → `¬` *atom*
| *atom*
atom → `TRUE`
| `FALSE`
| `ID`
| `(`*bexp*`)`

The sentence

x | *true* & *y*

The parse tree



Running example

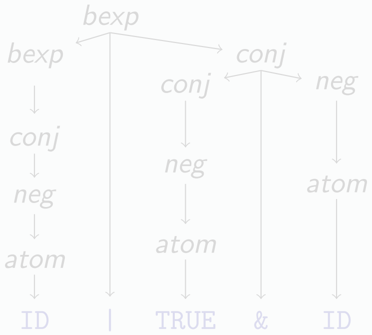
The grammar

$$\begin{array}{lcl}
\mathit{bexp} & \rightarrow & \mathit{bexp}\ \texttt{|}\ \mathit{conj} \\
 & | & \mathit{conj} \\
\mathit{conj} & \rightarrow & \mathit{conj}\ \texttt{\&}\ \mathit{neg} \\
 & | & \mathit{neg} \\
\mathit{neg} & \rightarrow & \neg\ \mathit{atom} \\
 & | & \mathit{atom} \\
\mathit{atom} & \rightarrow & \texttt{TRUE} \\
 & | & \texttt{FALSE} \\
 & | & \texttt{ID} \\
 & | & \texttt{(}\mathit{bexp}\texttt{)}
\end{array}$$

The sentence

x | *true* & *y*

The parse tree



Running example

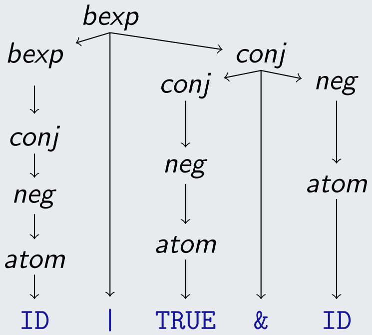
The grammar

bexp → *bexp* `|` *conj*
| *conj*
conj → *conj* `&` *neg*
| *neg*
neg → ¬ *atom*
| *atom*
atom → `TRUE`
| `FALSE`
| `ID`
| `(`*bexp*`)`

The sentence

x | *true* & *y*

The parse tree



Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & ID
bexp | *neg* & ID
bexp | *atom* & ID
bexp | TRUE & ID
conj | TRUE & ID
neg | TRUE & ID
atom | TRUE & ID
ID | TRUE & ID

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation ...

bexp

bexp | *conj*

bexp | *conj* & *neg*

bexp | *conj* & *atom*

bexp | *conj* & `ID`

bexp | *neg* & `ID`

bexp | *atom* & `ID`

bexp | `TRUE` & `ID`

conj | `TRUE` & `ID`

neg | `TRUE` & `ID`

atom | `TRUE` & `ID`

`ID` | `TRUE` & `ID`

...could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & ID
bexp | *neg* & ID
bexp | *atom* & ID
bexp | TRUE & ID
conj | TRUE & ID
neg | TRUE & ID
atom | TRUE & ID
ID | TRUE & ID

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation …

bexp

bexp | *conj*

bexp | *conj* & *neg*

bexp | *conj* & *atom*

bexp | *conj* & `ID`

bexp | *neg* & `ID`

bexp | *atom* & `ID`

bexp | `TRUE` & `ID`

conj | `TRUE` & `ID`

neg | `TRUE` & `ID`

atom | `TRUE` & `ID`

`ID` | `TRUE` & `ID`

… could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp

bexp | *conj*

bexp | *conj* & *neg*

bexp | *conj* & *atom*

bexp | *conj* & `ID`

bexp | *neg* & `ID`

bexp | *atom* & `ID`

bexp | `TRUE` & `ID`

conj | `TRUE` & `ID`

neg | `TRUE` & `ID`

atom | `TRUE` & `ID`

`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation …

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

… could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation …

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & ID
bexp | *neg* & ID
bexp | *atom* & ID
bexp | TRUE & ID
conj | TRUE & ID
neg | TRUE & ID
atom | TRUE & ID
ID | TRUE & ID

… could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation …

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

… could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation …

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & ID
bexp | *neg* & ID
bexp | *atom* & ID
bexp | TRUE & ID
conj | TRUE & ID
neg | TRUE & ID
atom | TRUE & ID
ID | TRUE & ID

… could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation . . .

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

. . . could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
   - taking in one more token from the input or
   - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Bottom-Up

The rightmost derivation ...

bexp
bexp | *conj*
bexp | *conj* & *neg*
bexp | *conj* & *atom*
bexp | *conj* & `ID`
bexp | *neg* & `ID`
bexp | *atom* & `ID`
bexp | `TRUE` & `ID`
conj | `TRUE` & `ID`
neg | `TRUE` & `ID`
atom | `TRUE` & `ID`
`ID` | `TRUE` & `ID`

...could be discovered in reverse order when inspecting the input from left to right!

For doing so we say that

1. we build a frontier in the parse tree by either
    - taking in one more token from the input or
    - reducing to a nonterminal using some rule from the grammar.
2. For doing this we need handles that tell us what can be reduced!

Handles

Example

On getting the first token `ID`

- There is one rule *atom*⟶`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*⟶*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*⟶*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp* `|` *conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp* `|` *conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*⟶`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*⟶*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*⟶*bexp* `|` *conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Handles

Example

On getting the first token `ID`

- There is one rule *atom*→`ID`

so building the frontier proceeds to reduce `ID` to *atom*.

- There is one rule *neg*→*atom*

so building the frontier proceeds to reduce *atom* to *neg*.

How long should we continue to reduce before taking in the next token?

We can make a decision by looking ahead in the input sequence!

Example

While working with `ID` we look ahead 1 token (without retrieving it!) and see it is `|`.

There is a rule *bexp*→*bexp*`|`*conj* so we will proceed reducing until we form the *bexp* to the left of the `|` before taking in the next token (`|`)

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*⟶*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$<bexp \longrightarrow bexp \mid conj \bullet>$

$<conj \longrightarrow conj \bullet \& neg>$

If the next token is a & we extend the frontier! Otherwise, we reduce to a bexp!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*⟶*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

< *bexp*⟶*bexp* | *conj*• >

< *conj*⟶*conj*•&*neg* >

If the next token is a & we extend the frontier! Otherwise, we reduce to a `bexp`!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*⟶*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$<bexp \longrightarrow bexp \mid conj \bullet >$

$<conj \longrightarrow conj \bullet \& neg>$

If the next token is a & we extend the frontier! Otherwise, we reduce to a bexp!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*→*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$< bexp \rightarrow bexp \mid conj \bullet >$

$< conj \rightarrow conj \bullet \& neg >$

If the next token is a & we extend the frontier! Otherwise, we reduce to a bexp!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*⟶*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$<bexp \longrightarrow bexp \,|\, conj \bullet >$

$<conj \longrightarrow conj \bullet \& neg>$

If the next token is a & we extend the frontier! Otherwise, we reduce to a bexp!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*⟶*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$< bexp \longrightarrow bexp \mid conj \bullet >$

$< conj \longrightarrow conj \bullet \& neg >$

If the next token is a & we extend the frontier! Otherwise, we reduce to a `bexp`!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is $\mathit{bexp} \mid \mathit{conj}$

- should we reduce *conj* with $\mathit{bexp} \rightarrow \mathit{conj}$?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$< \mathit{bexp} \rightarrow \mathit{bexp} \mid \mathit{conj} \bullet >$

$< \mathit{conj} \rightarrow \mathit{conj} \bullet \& \mathit{neg} >$

If the next token is a & we extend the frontier! Otherwise, we reduce to a `bexp`!

We use a bullet • to mark the state of the parser!

Potential handles

Example

When the frontier is *bexp* | *conj*

- should we reduce *conj* with *bexp*→*conj*?
- or should we *build a larger conj before reducing*?

It depends on the look ahead!

We describe potential handles using

1. rules of the grammar
2. the state of the parser
3. the look ahead

Example

With our frontier the potential handles are

$< bexp \rightarrow bexp \,|\, conj \bullet >$

$< conj \rightarrow conj \bullet \& neg >$

If the next token is a & we extend the frontier! Otherwise, we reduce to a `bexp`!

We use a bullet • to mark the state of the parser!

Observations

We always inspect the rightmost part of the frontier to find patterns in order to decide on new actions!

We can use a stack to store the frontier!

The operations on the stack are

- shift a new token from the input sequence
- reduce some items from the top of the stack to one non terminal according to some handle.

Observations

We always inspect the rightmost part of the frontier to find patterns in order to decide on new actions!

We can use a stack to store the frontier!

The operations on the stack are

- shift a new token from the input sequence
- reduce some items from the top of the stack to one non terminal according to some handle.

Observations

We always inspect the rightmost part of the frontier to find patterns in order to decide on new actions!

We can use a stack to store the frontier!

The operations on the stack are

- shift a new token from the input sequence
- reduce some items from the top of the stack to one non terminal according to some handle.

Observations

We always inspect the rightmost part of the frontier to find patterns in order to decide on new actions!

We can use a stack to store the frontier!

The operations on the stack are

- shift a new token from the input sequence
- reduce some items from the top of the stack to one non terminal according to some handle.

Observations

We always inspect the rightmost part of the frontier to find patterns in order to decide on new actions!

We can use a stack to store the frontier!

The operations on the stack are

- shift a new token from the input sequence
- reduce some items from the top of the stack to one non terminal according to some handle.

The number of potential handles is finite!

the number of rules
*
the lengths of the right hand sides.

We can use a finite automata to inspect the top of the stack looking for patterns!

The workings of the parser can be explained by saying what is to be done with a given stack and a lookahead!

All the knowledge is stored in two tables:

1. The Action table
2. The Goto table

The number of potential handles is finite!

the number of rules
*
the lengths of the right hand sides.

We can use a finite automata to inspect the top of the stack looking for patterns!

The workings of the parser can be explained by saying what is to be done with a given stack and a lookahead!

All the knowledge is stored in two tables:

1. The Action table
2. The Goto table

The number of potential handles is finite!

the number of rules
*
the lengths of the right hand sides.

We can use a finite automata to inspect the top of the stack looking for patterns!

The workings of the parser can be explained by saying what is to be done with a given stack and a lookahead!

All the knowledge is stored in two tables:

1. The Action table
2. The Goto table

The number of potential handles is finite!

the number of rules

*

the lengths of the right hand sides.

We can use a finite automata to inspect the top of the stack looking for patterns!

The workings of the parser can be explained by saying what is to be done with a given stack and a lookahead!

All the knowledge is stored in two tables:

1. The Action table
2. The Goto table

The tables

1	*bexp*	→	*bexp* `|` *conj*
2		|	*conj*
3	*conj*	→	*conj* `&` *neg*
4		|	*neg*
5	*neg*	→	`¬` *atom*
6		|	*atom*
7	*atom*	→	`TRUE`
8		|	`FALSE`
9		|	`ID`
10		|	`(`*bexp*`)`

Actions

state	**eof**	`|`	`&`	`true`	`ID`	...
0				s 5		
1	acc					
2	r 2					
3	r 4					
4	r 6					
5	r 7					
...						

Goto

state	*bexp*	*conj*	*neg*	*atom*
0	1	2	3	4
...				

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

The tables

Actions

For every state and every lookahead there is an action that can be

- s i shift the terminal onto the stack and change to state i.
- r j reduce according to rule j. The state is the one in the stack before the pattern that is replaced by a nonterminal.
- acc accepting the sentence!
- err to report an error (whenever the table does not have one of the actions above!)

Goto

For every state and every non-terminal on the top of the stack, indicates to what state to change.

The parser generator

Reads the grammar and generates these tables and organizes a driver!

This is not always possible! If it is we say that the grammar is LR(1).

Shift or Reduce?

$$\begin{array}{lcl} \mathit{goal} & \longrightarrow & \mathit{stm} \\ \mathit{stm} & \longrightarrow & \texttt{if<exp>then } \mathit{stm} \texttt{ else } \mathit{stm} \\ & | & \texttt{if<exp>then } \mathit{stm} \\ & | & \texttt{<assign>} \end{array}$$

Example

`if<exp>then if<exp>then <assign>` • `else <assign>`

Shift or Reduce?

$$
\begin{array}{lcl}
\mathit{goal} & \rightarrow & \mathit{stm} \\
\mathit{stm} & \rightarrow & \texttt{if<exp>then } \mathit{stm} \texttt{ else } \mathit{stm} \\
 & | & \texttt{if<exp>then } \mathit{stm} \\
 & | & \texttt{<assign>}
\end{array}
$$

Example

`if<exp>then if<exp>then <assign> • else <assign>`

Shift or reduce?

Example

`if<exp>then if<exp>then <assign> • else <assign>`

Reduce

`if<exp>then` *stm* `• else <assign>`

Shift

`if<exp>then if<exp>then <assign> else •<assign>`

```
jacc if.jacc
WARNING: conflicts:  1 shift/reduce, 0 reduce/reduce
```

Shift or reduce?

Example

if<exp>then if<exp>then <assign> • else <assign>

Reduce

if<exp>then *stm* • else <assign>

Shift

if<exp>then if<exp>then <assign> else • <assign>

```
jacc if.jacc
WARNING: conflicts:  1 shift/reduce, 0 reduce/reduce
```

Shift or reduce?

Example

if<exp>then if<exp>then <assign> • else <assign>

Reduce

if<exp>then *stm* • else <assign>

Shift

if<exp>then if<exp>then <assign> else •<assign>

```
jacc if.jacc
WARNING: conflicts:  1 shift/reduce, 0 reduce/reduce
```

Shift or reduce?

Example

`if<exp>then if<exp>then <assign> • else <assign>`

Reduce

`if<exp>then` *stm* `• else <assign>`

Shift

`if<exp>then if<exp>then <assign> else •<assign>`

```
jacc if.jacc
WARNING: conflicts:  1 shift/reduce, 0 reduce/reduce
```

Reformulating the grammar

statement → `if<exp>then` *statement*
| `if<exp>then` *withElse* `else` *statement*
| `<assign>`
withElse → `if<exp>then` *withElse* `else` *withElse*
| `<assign>`

Example

`if<exp>then if<exp>then <assign> •` `else <assign>`
can only be followed by a shift!

In `jacc` we can leave the conflict unresolved, in which case it solves it in favour of shift (the same as we achieved with the corrected grammar).

Reformulating the grammar

$$
\begin{array}{lcl}
\textit{statement} & \rightarrow & \texttt{if<exp>then } \textit{statement} \\
 & | & \texttt{if<exp>then } \textit{withElse} \texttt{ else } \textit{statement} \\
 & | & \texttt{<assign>} \\
\textit{withElse} & \rightarrow & \texttt{if<exp>then } \textit{withElse} \texttt{ else } \textit{withElse} \\
 & | & \texttt{<assign>}
\end{array}
$$

Example

`if<exp>then if<exp>then <assign>` • `else` `<assign>`
can only be followed by a shift!

In jacc we can leave the conflict unresolved, in which case it solves it in favour of shift (the same as we achieved with the corrected grammar).

Reformulating the grammar

$$\begin{array}{lcl} \textit{statement} & \rightarrow & \texttt{if<exp>then}\ \textit{statement} \\ & | & \texttt{if<exp>then}\ \textit{withElse}\ \texttt{else}\ \textit{statement} \\ & | & \texttt{<assign>} \\ \textit{withElse} & \rightarrow & \texttt{if<exp>then}\ \textit{withElse}\ \texttt{else}\ \textit{withElse} \\ & | & \texttt{<assign>} \end{array}$$

Example

`if<exp>then if<exp>then <assign> •` `else <assign>`
can only be followed by a shift!

In `jacc` we can leave the conflict unresolved, in which case it solves it in favour of shift (the same as we achieved with the corrected grammar).

Generating the push-down automaton

```
%token TRUE FALSE ID
%token '-' '&' '|' '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg : '-' atom
    | atom
    ;
atom : TRUE | FALSE
     | ID | '(' bexp ')';
%%
```

Trace the workings without having to write a lexer:

```
jacc -pt bexp.jacc -r file
```

Inspect the push down automaton to understand conflicts:

```
jacc -h bexp.jacc
```

generates an html version of the tables with

- hyperlinks to change state on shift and goto,
- and the back button for reductions!

Generating the push-down automaton

```
%token TRUE FALSE ID
%token '-' '&' '|' '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg : '-' atom
    | atom
    ;
atom : TRUE | FALSE
     | ID | '(' bexp ')';
%%
```

Trace the workings without having to write a lexer:

`jacc -pt bexp.jacc -r` *`file`*

Inspect the push down automaton to understand conflicts:

`jacc -h bexp.jacc`

generates an html version of the tables with

- hyperlinks to change state on shift and goto,
- and the back button for reductions!

Generating the push-down automaton

```
%token TRUE FALSE ID
%token '-' '&' '|' '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg : '-' atom
    | atom
    ;
atom : TRUE | FALSE
     | ID | '(' bexp ')';
%%
```

Trace the workings without having to write a lexer:

```
jacc -pt bexp.jacc -r file
```

Inspect the push down automaton to understand conflicts:

```
jacc -h bexp.jacc
```

generates an html version of the tables with

- hyperlinks to change state on shift and goto,
- and the back button for reductions!

Generating the push-down automaton

```
%token TRUE FALSE ID
%token '-' '&' '|' '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg : '-' atom
    | atom
    ;
atom : TRUE | FALSE
     | ID | '(' bexp ')';
%%
```

Trace the workings without having to write a lexer:

```
jacc -pt bexp.jacc -r file
```

Inspect the push down automaton to understand conflicts:

```
jacc -h bexp.jacc
```

generates an html version of the tables with

- hyperlinks to change state on shift and goto,
- and the back button for reductions!

Generating the push-down automaton

```
%token TRUE FALSE ID
%token '-' '&' '|' '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg : '-' atom
    | atom
    ;
atom : TRUE | FALSE
     | ID | '(' bexp ')';
%%
```

Trace the workings without having to write a lexer:

```
jacc -pt bexp.jacc -r file
```

Inspect the push down automaton to understand conflicts:

```
jacc -h bexp.jacc
```

generates an html version of the tables with

- hyperlinks to change state on shift and goto,
- and the back button for reductions!

PDA practice prob. $\Sigma = \{a, b\}$

Note Title 10/7/2004

$\{w \mid n_a(w) = 2 * n_b(w)\}$

do the PDA

strategy: - see b on input & push TWO b's on stack

- stack tracks extra's