Example

- Ambiguous grammar:
  \[
  \langle \text{exp} \rangle \ ::= \ 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle
  \]
- String with more then one parse:
  \[
  0 + 1 + 0 \\
  1 * 1 + 1
  \]
- Source of ambiguity: associativity and precedence

Disambiguating a Grammar

- Given ambiguous grammar \( G \), with start symbol \( S \), find a grammar \( G' \) with same start symbol, such that
  - language of \( G \) = language of \( G' \)
- Not always possible
- No algorithm in general

Steps to Grammar Disambiguation

- Identify the rules and a smallest use that display ambiguity
- Decide which parse to keep; why should others be thrown out?
- What syntactic restrictions on subexpressions are needed to throw out the bad (while keeping the good)?
- Add a new non-terminal and rules to describe this set of restricted subexpressions (called stratifying, or refactoring)
- Replace old rules to use new non-terminals
- Rinse and repeat

Two Major Sources of Ambiguity

- Lack of determination of operator precedence
- Lack of determination of operator assocoativity
- Not the only sources of ambiguity
How to Enforce Associativity

- Have at most one recursive call per production
- When two or more recursive calls would be natural, leave right-most one for right associativity, left-most one for left associativity

Example

- `<Sum> ::= 0 | 1 | <Sum> + <Sum> | ( <Sum> )`
- Becomes
  - `<Sum> ::= <Num> | <Num> + <Sum>`
  - `<Num> ::= 0 | 1 | ( <Sum> )`

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly)
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar

Example

- `<Sum> ::= 0 | 1 | <Sum> + <Sum> | ( <Sum> )`
- Becomes
  - `<Sum> ::= <Num> | <Num> + <Sum>`
  - `<Num> ::= 0 | 1 | ( <Sum> )`

Operator Precedence

- Operators of highest precedence evaluated first (bind more tightly)
- Precedence for infix binary operators given in following table
- Needs to be reflected in grammar

Primary Precedence in Grammar

- Higher precedence translates to longer derivation chain
- Example:
  - `<exp> ::= 0 | 1 | <exp> + <exp> | <exp> * <exp>`
  - Becomes
    - `<exp> ::= <mult_exp> | <exp> + <mult_exp>`
    - `<mult_exp> ::= <id> | <mult_exp> * <id>`
    - `<id> ::= 0 | 1`

Disambiguating a Grammar

- `<exp>::= 0|1| b<exp> | <exp>a`
- | `<exp>m<exp>`
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.

Disambiguating a Grammar

- `<exp>::= 0|1| b<exp> | <exp>a`
- | `<exp>m<exp>`
- Want a has higher precedence than b, which in turn has higher precedence than m, and such that m associates to the left.
- `<exp>::= <exp> m <not m> | <not m>`
- `<not m>::= b <not m> | <not b m>`
- `<not b m>::= <not b m>a | 0 | 1`
Disambiguating a Grammar – Take 2

- `<exp>::= 0|1 | b<exp> | <exp>a 
  | <exp>m<exp>`
- Want b has higher precedence than m, which in turn has higher precedence than a, and such that m associates to the right.

LR Parsing

- Read tokens left to right (L)
- Create a rightmost derivation (R)
- How is this possible?
- Start at the bottom (left) and work your way up
- Last step has only one non-terminal to be replaced so is right-most
- Working backwards, replace mixed strings by non-terminals
- Always proceed so that there are no non-terminals to the right of the string to be replaced

LR Parsing Tables

- Build a pair of tables, Action and Goto, from the grammar
  - This is the hardest part, we omit here
  - Rows labeled by states
  - For Action, columns labeled by terminals and “end-of-tokens” marker
  - (more generally strings of terminals of fixed length)
  - For Goto, columns labeled by non-terminals

Action and Goto Tables

- Given a state and the next input, Action table says either
  - **shift** and go to state n, or
  - **reduce** by production k (explained in a bit)
  - **accept** or **error**
- Given a state and a non-terminal, Goto table says
  - go to state m

LR(i) Parsing Algorithm

- Based on push-down automata
- Uses states and transitions (as recorded in Action and Goto tables)
- Uses a stack containing states, terminals and non-terminals
LR(i) Parsing Algorithm

0. Insure token stream ends in special “end-of-tokens” symbol
1. Start in state 1 with an empty stack
2. Push state(1) onto stack
3. Look at next i tokens from token stream (toks) (don’t remove yet)
4. If top symbol on stack is state(n), look up action in Action table at (n, toks)
5. If action = shift m,
   a) Remove the top token from token stream and push it onto the stack
   b) Push state(m) onto stack
   c) Go to step 3
6. If action = reduce k where production k is E ::= u
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   b) If new top symbol on stack is state(m), look up new state p in Goto(m,E)
   c) Push E onto the stack, then push state(p) onto the stack
   d) Go to step 3
7. If action = accept
   • Stop parsing, return success
8. If action = error,
   • Stop parsing, return failure

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>

Example: <Sum> = 0 | 1 | (<Sum>)
           | <Sum> + <Sum>
Example: \( \text{<Sum>} = 0 | 1 | (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} \quad \Rightarrow \\
= \quad \star (0 + 1) + 0 \quad \text{shift}
\]

LR(i) Parsing Algorithm

5. If action = \textbf{shift} \(m\),
   a) Remove the top token from token stream and push it onto the stack
   b) Push \textit{state}(m) onto stack
   c) Go to step 3

Example: \( \text{<Sum>} = 0 | 1 | (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} \quad \Rightarrow \\
= \quad \star (0 + 1) + 0 \quad \text{shift}
\]

Example: \( \text{<Sum>} = 0 | 1 | (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} \quad \Rightarrow \\
= \quad (\star 0 + 1) + 0 \quad \text{reduce}
\]

LR(i) Parsing Algorithm

6. If action = \textbf{reduce} \(k\) where production \(k\) is \(E ::= u\)
   a) Remove \(2 \times \text{length}(u)\) symbols from stack (\(u\) and all the interleaved states)
   b) If new top symbol on stack is \textit{state}(\(m\)), look up new state \(p\) in \textit{Goto}(\(m,E\))
   c) Push \(E\) onto the stack, then push \textit{state}(\(p\)) onto the stack
   d) Go to step 3

Example: \( \text{<Sum>} = 0 | 1 | (<\text{Sum}>), \text{<Sum>} + \text{<Sum>} \)

\[
\text{<Sum>} \quad \Rightarrow \\
= \quad (\star 0 + 1) + 0 \quad \text{shift}
\]
Example: $<\text{Sum}> = 0 \mid 1 \mid ( <\text{Sum}> )$

<table>
<thead>
<tr>
<th>$&lt;\text{Sum}&gt; + &lt;\text{Sum}&gt;$</th>
<th>$&lt;\text{Sum}&gt; + &lt;\text{Sum}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;\text{Sum}&gt; =&gt;$</td>
<td>$&lt;\text{Sum}&gt; =&gt;$</td>
</tr>
<tr>
<td>$= ( &lt;\text{Sum}&gt; + 0 ) + 0$</td>
<td>$= ( &lt;\text{Sum}&gt; + 0 ) + 0$</td>
</tr>
<tr>
<td>$= ( &lt;\text{Sum}&gt; + 0 ) + 0$</td>
<td>$= ( &lt;\text{Sum}&gt; + 0 ) + 0$</td>
</tr>
<tr>
<td>$=&gt; ( 0 + 0 ) + 0$</td>
<td>$=&gt; ( 0 + 0 ) + 0$</td>
</tr>
<tr>
<td>$= ( 0 + 0 ) + 0$</td>
<td>$= ( 0 + 0 ) + 0$</td>
</tr>
<tr>
<td>$= * ( 0 + 0 ) + 0$</td>
<td>$= * ( 0 + 0 ) + 0$</td>
</tr>
</tbody>
</table>

LR(i) Parsing Algorithm

6. If action = reduce $k$ where production $k$ is $E ::= u$
   
   a) Remove 2 * length(u) symbols from stack (u and all the interleaved states)
   
   b) If new top symbol on stack is state(m), look up new state $\rho$ in Goto(m,E)
   
   c) Push $E$ onto the stack, then push state($\rho$) onto the stack
   
   d) Go to step 3
Example: \(<Sum> = 0 \mid 1 \mid (<Sum>)\) | \(<Sum> + <Sum>\)

\(<Sum> \Rightarrow \)

\[= \langle Sum \rangle \cdot 0 \quad \text{shift} \]
\[= \langle Sum \rangle \cdot 0 \quad \text{reduce} \]
\[= (\langle Sum \rangle \cdot 0) + 0 \quad \text{shift} \]
\[= (\langle Sum \rangle + \langle Sum \rangle \cdot 0) + 0 \quad \text{reduce} \]
\[= (\langle Sum \rangle + 1 \cdot 0) + 0 \quad \text{reduce} \]
\[= (\langle Sum \rangle + 1 \cdot 0) + 0 \quad \text{shift} \]
\[= (0 \cdot 0 + 1) + 0 \quad \text{reduce} \]
\[= (0 \cdot 0 + 1) + 0 \quad \text{shift} \]
\[= \ast (0 + 1) + 0 \quad \text{shift} \]

LR(i) Parsing Algorithm

7. If action = accept
   ■ Stop parsing, return success
8. If action = error,
   ■ Stop parsing, return failure
Example

\[(0 + 1) + 0\]

Example

\[(0 + 1) + 0\]

Example

\[(0 + 1) + 0\]

Example

\[<\text{Sum}> (0 + 1) + 0\]

Example

\[<\text{Sum}> (0 + 1) + 0\]

Example

\[<\text{Sum}> (0 + 1) + 0\]
**Example**

```
Example: \( \text{<Sum>} = 0 \mid 1 \mid (<\text{Sum}>) \)
\| \text{<Sum>} + \text{<Sum>}
- 0 + 1 + 0 \quad \text{shift}
- \rightarrow 0 + 1 + 0 \quad \text{reduce}
- \rightarrow \text{<Sum>} + 1 + 0 \quad \text{shift}
- \rightarrow \text{<Sum>} + 1 + 0 \quad \text{shift}
- \rightarrow \text{<Sum>} + \text{<Sum>} + 0 \quad \text{reduce}
- \rightarrow \text{<Sum>} + \text{<Sum>} + 0
```

**Shift-Reduce Conflicts**

- **Problem**: can’t decide whether the action for a state and input character should be **shift** or **reduce**
- Caused by ambiguity in grammar
- Usually caused by lack of associativity or precedence information in grammar

**Example - cont**

- **Problem**: shift or reduce?
  - You can shift-shift-reduce-reduce or reduce-shift-shift-reduce
  - Shift first - right associative
  - Reduce first - left associative
Reduce - Reduce Conflicts

- **Problem**: can’t decide between two different rules to reduce by
- Again caused by ambiguity in grammar
- **Symptom**: RHS of one production suffix of another
- Requires examining grammar and rewriting it
- Harder to solve than shift-reduce errors

Example

- S ::= A | aB  A ::= abc  B ::= bc
  - abc  shift
  - a bc  shift
  - ab c  shift
  - abc
  - Problem: reduce by B ::= bc then by S ::= aB, or by A ::= abc then S::A?

Recursive Descent Parsing

- Recursive descent parsers are a class of parsers derived fairly directly from BNF grammars
- A recursive descent parser traces out a parse tree in top-down order, corresponding to a left-most derivation (LL - left-to-right scanning, leftmost derivation)

Recursive Descent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all phrases that the nonterminal can generate
- Each nonterminal in right-hand side of a rule corresponds to a recursive call to the associated subprogram

Recursive Descent Parsing

- Each subprogram must be able to decide how to begin parsing by looking at the left-most character in the string to be parsed
  - May do so directly, or indirectly by calling another parsing subprogram
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars
  - Sometimes can modify grammar to suit

Sample Grammar

```plaintext
<expr> ::= <term> | <term> + <expr>
        | <term> - <expr>

<term> ::= <factor> | <factor> * <term>
        | <factor> / <term>

<factor> ::= <id> | ( <expr> )
```
Tokens as OCaml Types

- `+ - * / ( ) <id>`
- Becomes an OCaml datatype
  
  ```ocaml
type token =
  | Id_token of string
  | Left_parenthesis | Right_parenthesis
  | Times_token | Divide_token
  | Plus_token | Minus_token
  ```

Parse Trees as Datatypes

```ocaml
<expr> ::= <term> | <term> + <expr> | <term> - <expr>

type expr =
  | Term_as_Expr of term
  | Plus_Expr of (term * expr)
  | Minus_Expr of (term * expr)
```

Parse Trees as Datatypes

```ocaml
<term> ::= <factor> | <factor> * <term> | <factor> / <term>

and term =
  | Factor_as_Term of factor
  | Mult_Term of (factor * term)
  | Div_Term of (factor * term)
```

Parse Trees as Datatypes

```ocaml
<factor> ::= <id> | ( <expr> )

and factor =
  | Id_as_Factor of string
  | Parenthesized_Expr_as_Factor of expr
```

Parsing Lists of Tokens

- Will create three mutually recursive functions:
  - `expr : token list -> (expr * token list)`
  - `term : token list -> (term * token list)`
  - `factor : token list -> (factor * token list)`
- Each parses what it can and gives back parse and remaining tokens

Parsing an Expression

```ocaml
<expr> ::= <term> [( + | - ) <expr> ]

let rec expr tokens =
  (match term tokens
   with ( term_parse , tokens_after_term ) ->
   (match tokens_after_term
    with( Plus_token :: tokens_after_plus ) ->
```
Parsing an Expression

\[
<\text{expr}> ::= <\text{term}> [( + | - ) <\text{expr}> ] \\
\]

let rec expr tokens =
  (match term tokens
   with ( \text{term}_\text{parse}, \text{tokens}_\text{after}_\text{term} ) ->
   (match \text{tokens}_\text{after}_\text{term}
    with ( \text{Plus}_\text{token} :: \text{tokens}_\text{after}_\text{plus} ) ->

Parsing a Plus Expression

\[
<\text{expr}> ::= <\text{term}> [( + | - ) <\text{expr}> ] \\
\]

let rec expr tokens =
  (match term tokens
   with ( \text{term}_\text{parse}, \text{tokens}_\text{after}_\text{term} ) ->
   (match \text{tokens}_\text{after}_\text{term}
    with ( \text{Plus}_\text{token} :: \text{tokens}_\text{after}_\text{plus} ) ->

Parsing a Plus Expression

\[
<\text{expr}> ::= <\text{term}> [( + | - ) <\text{expr}> ] \\
\]

let rec expr tokens =
  (match term tokens
   with ( \text{term}_\text{parse}, \text{tokens}_\text{after}_\text{term} ) ->
   (match \text{tokens}_\text{after}_\text{term}
    with ( \text{Plus}_\text{token} :: \text{tokens}_\text{after}_\text{plus} ) ->

Parsing a Plus Expression

\[
<\text{expr}> ::= <\text{term}> + <\text{expr}> \\
\]

(let rec expr tokens =
  (match expr tokens_after_plus
   with ( expr_parse, tokens_after_expr ) ->
   ( Plus_Expr ( term_parse, expr_parse ),
     tokens_after_expr ))

Parsing a Plus Expression

\[
<\text{expr}> ::= <\text{term}> + <\text{expr}> \\
\]

(let rec expr tokens =
  (match expr tokens_after_plus
   with ( expr_parse, tokens_after_expr ) ->
   ( Plus_Expr ( term_parse, expr_parse ),
     tokens_after_expr ))
Building Plus Expression Parse Tree

\[
<\text{expr}> ::= <\text{term}> + <\text{expr}>
\]

(match \text{expr} tokens\text{after\_plus}
with ( expr\_parse, tokens\_after\_expr) ->
(Plus\_Expr (term\_parse, expr\_parse),
tokens\_after\_expr))

11/7/2017

Parsing a Minus Expression

\[
<\text{expr}> ::= <\text{term}> - <\text{expr}>
\]

\( | (\text{Minus\_token} :: \text{tokens\_after\_minus}) ->
(\text{match} \text{expr} \text{tokens\_after\_minus}
with (expr\_parse, tokens\_after\_expr) ->
(Minus\_Expr (term\_parse, expr\_parse),
tokens\_after\_expr)) \)

11/7/2017

Parsing a Minus Expression

\[
<\text{expr}> ::= <\text{term}> - <\text{expr}>
\]

\( | (\text{Minus\_token} :: \text{tokens\_after\_minus}) ->
(\text{match} \text{expr} \text{tokens\_after\_minus}
with (expr\_parse, tokens\_after\_expr) ->
(Minus\_Expr (term\_parse, expr\_parse),
tokens\_after\_expr)) \)

11/7/2017

Parsing an Expression as a Term

\[
<\text{expr}> ::= <\text{term}>
\]

\( | _ -> (\text{Term\_as\_Expr} \text{term\_parse},
tokens\_after\_term)) \)

- Code for term is same except for replacing addition with multiplication and subtraction with division

11/7/2017

Parsing Factor as Id

\[
<\text{factor}> ::= <\text{id}>
\]

and factor tokens =
(match tokens
with (Id\_token id\_name :: tokens\_after\_id) =
( Id\_as\_Factor id\_name, tokens\_after\_id)

11/7/2017

Parsing Factor as Parenthesized Expression

\[
<\text{factor}> ::= ( <\text{expr}>)
\]

\( | \text{factor} (\text{Left\_parenthesis} :: \text{tokens}) =
(\text{match} \text{expr} \text{tokens}
with (expr\_parse, tokens\_after\_expr) ->

11/7/2017
Parsing Factor as Parenthesized Expression

<factor> ::= ( <expr> )

(match tokens_after_expr
with Right_parenthesis :: tokens_after_rparen ->
(Parenthesized_Expr_as_Factor expr_parse, tokens_after_rparen)

Error Cases

- What if no matching right parenthesis?
  | _ -> raise (Failure "No matching rparen") ))
- What if no leading id or left parenthesis?
  | _ -> raise (Failure "No id or lparen" ));;

( a + b ) * c - d

expr [Left_parenthesis; Id_token "a";
Plus_token; Id_token "b";
Right_parenthesis; Times_token;
Id_token "c"; Minus_token;
Id_token "d"];;

a + b * c - d

# expr [Id_token "a"; Plus_token; Id_token "b";
Times_token; Id_token "c"; Minus_token;
Id_token "d"];;
- : expr * token list =
(Plus_Expr
 (Mult_Term
 (Parenthesized_Expr_as_Factor
 (Plus_Expr
 (Factor_as_Term (Id_as_Factor "a"),
 Term_as_Expr (Factor_as_Term (Id_as_Factor "b"))),
 Factor_as_Term (Id_as_Factor "c")),
 Term_as_Expr (Factor_as_Term (Id_as_Factor "d"))),
[])

11/7/2017
\[ a + b \ast c - d \]

\[
\begin{align*}
&\langle \text{expr}\rangle \\
&\langle \text{term}\rangle + \langle \text{expr}\rangle \\
&\langle \text{factor}\rangle \times \langle \text{term}\rangle - \langle \text{expr}\rangle \\
&\langle \text{id}\rangle \times \langle \text{factor}\rangle \times \langle \text{term}\rangle - \langle \text{expr}\rangle \\
&\langle \text{id}\rangle \times \langle \text{factor}\rangle \times \langle \text{factor}\rangle \\
&\text{a} \times \langle \text{id}\rangle \times \langle \text{id}\rangle \\
&\text{b} \times \langle \text{factor}\rangle \times \langle \text{term}\rangle \\
&\text{c} \times \langle \text{term}\rangle \\
&\text{d}
\end{align*}
\]

\[ ( a + b \ast c - d ) \]

```
# expr [Left_parenthesis; Id_token "a"; Plus_token; Id_token "b"; Times_token; Id_token "c"; Minus_token; Id_token "d"];;

Exception: Failure "No matching rparen".
Can't parse because it was expecting a right parenthesis but it got to the end without finding one
```

\[ a + b ) \ast c - d ( \]

\[
\begin{align*}
&\text{expr [Id_token "a"; Plus_token; Id_token "b"; Right_parenthesis; Times_token; Id_token "c"; Minus_token; Id_token "d"; Left_parenthesis];};
- : expr * token list = 
(Plus_Expr
 (Factor_as_Term (Id_as_Factor "a")),
 Term_as_Expr (Factor_as_Term (Id_as_Factor "b"))),
 [Right_parenthesis; Times_token; Id_token "c"; Minus_token; Id_token "d"; Left_parenthesis])
\]

**Parsing Whole String**

- Q: How to guarantee whole string parses?
  - A: Check returned tokens empty

```
let parse tokens =
match expr tokens
with (expr_parse, []) -> expr_parse
| _ -> raise (Failure "No parse"));;
```

- Fixes <expr> as start symbol

**Streams in Place of Lists**

- More realistically, we don't want to create the entire list of tokens before we can start parsing
- We want to generate one token at a time and use it to make one step in parsing
- Can use (token * (unit -> token)) or (token * (unit -> token option)) in place of token list

**Problems for Recursive-Descent Parsing**

- Left Recursion:
  - A ::= Aw
  - translates to a subroutine that loops forever
- Indirect Left Recursion:
  - A ::= Bw
  - B ::= Av
  - causes the same problem
Problems for Recursive-Descent Parsing

- Parser must always be able to choose the next action based only on the very next token.
- Pairwise Disjointedness Test: Can we always determine which rule (in the non-extended BNF) to choose based on just the first token.

Pairwise Disjointedness Test

- For each rule $A ::= y$
  
  Calculate $FIRST(y) = \{a | y \Rightarrow^* aw\} \cup \{\varepsilon | y \Rightarrow^* \varepsilon\}$
  
  - For each pair of rules $A ::= y$ and $A ::= z$, require $FIRST(y) \cap FIRST(z) = \{\}$

Example

Grammar:

$<S> ::= <A> a <B> b$
$<A> ::= <A> b | b$
$<B> ::= a <B> | a$

$FIRST(<A> b) = \{b\}$
$FIRST(b) = \{b\}$

Rules for $<A>$ not pairwise disjoint

Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion.
  - Changes associativity.
- Given $<expr> ::= <expr> + <term>$ and $<expr> ::= <term>$
  
  - Add new non-terminal $<e>$ and replace above rules with
    $<expr> ::= <term><e>$
    $<e> ::= + <term><e>$
    $<e> ::= - <term><e>$
    $<e> ::= \varepsilon$

Factoring Grammar

- Test too strong: Can’t handle $<expr> ::= <term>[ ( + | - ) <expr> ]$
  
  - Answer: Add new non-terminal and replace above rules by
    $<expr> ::= <term><e>$
    $<e> ::= + <term><e>$
    $<e> ::= - <term><e>$
    $<e> ::= \varepsilon$

- You are delaying the decision point.

Example

Both $<A>$ and $<B>$ have problems:

Transform grammar to:

$<S> ::= <A> a <B> b <S> ::= <A> a <B> b$
$<A> ::= <A> b | b$
$<A> ::= b<A1>$
$<B> ::= a <B> | a$
$<A1> ::= b<A1> | \varepsilon$
$<B> ::= a<B1>$
$<B1> ::= a<B1> | \varepsilon$
Semantics

- Expresses the meaning of syntax
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of describing meaning of executing a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics

Dynamic Semantics

- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the structure of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from axioms and inference rules
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (post-condition) of the state (the values of the program variables) after the execution of program, assuming another property (pre-condition) of the state before execution
- Written:
  - {Precondition} Program {Postcondition}
- Source of idea of loop invariant
Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Natural Semantics

- Aka Structural Operational Semantics, aka “Big Step Semantics”
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like
  $$(C, m) \downarrow m'$$
  or
  $$(E, m) \downarrow v$$

Simple Imperative Programming Language

- $I \in$ Identifiers
- $N \in$ Numerals
- $B ::= \text{true} | \text{false} | B \& B | B \lor B | \text{not} B |
| E < E | E = E$
- $E ::= N | I | E + E | E * E | E - E | - E$
- $C ::= \text{skip} | C ; C | I ::= E$
| if $B$ then $C$ else $C$ fi | while $B$ do $C$ od

Natural Semantics of Atomic Expressions

- Identifiers: $(I, m) \downarrow m(I)$
- Numerals are values: $(N, m) \downarrow N$
- Booleans: $(\text{true}, m) \downarrow \text{true}$
| $(\text{false}, m) \downarrow \text{false}$

Booleans:

$$(B, m) \downarrow \text{false} \quad (B, m) \downarrow \text{true} \quad (B', m) \downarrow b$$

$$(B \& B', m) \downarrow \text{false} \quad (B \& B', m) \downarrow b$$

$$(B \lor B', m) \downarrow \text{true} \quad (B \lor B', m) \downarrow b$$

Relations

$$(E, m) \downarrow U \quad (E', m) \downarrow V \quad U \sim V = b$$

$$(E \sim E', m) \downarrow b$$

- By $U \sim V = b$, we mean does (the meaning of) the relation $\sim$ hold on the meaning of $U$ and $V$
- May be specified by a mathematical expression/equation or rules matching $U$ and $V$
Arithmetic Expressions

\[(E, m) \downarrow U \quad (E', m) \downarrow V \quad U \text{ op } V = N\]

\[(E \text{ op } E', m) \downarrow N\]

where \(N\) is the specified value for \(U \text{ op } V\)

Commands

Skip: \((\text{skip}, m) \downarrow m\)

Assignment: \((E, m) \downarrow V\)

\[(I := E, m) \downarrow [I \leftarrow V]\]

Sequencing: \(\langle C, m \rangle \downarrow m'\)

\[(E \text{ op } E', m) \downarrow N\]

\[
\begin{align*}
\text{if } B \text{ then } C \text{ else } C' \text{ fi, } m & \downarrow m' \\
\text{if } B \text{ then } C \text{ else } C' \text{ fi, } m & \downarrow m''
\end{align*}
\]

Example: If Then Else Rule

\[
\begin{align*}
(x > 5, \{x \rightarrow 7\}) \downarrow ?
\end{align*}
\]

\[
\begin{align*}
(x > 5, \{x \rightarrow 7\}) \downarrow ?
\end{align*}
\]
Example: Arith Relation

\[
\begin{align*}
? > ? &= ? \\
(x,\{x>7\}) &\cup? \quad (5,\{x>7\}) \cup? \\
(x > 5, \{x -> 7\}) &\cup? \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]

Example: Identifier(s)

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x,\{x>7\}) &\cup 7 \quad (5,\{x>7\}) \cup 5 \\
(x > 5, \{x -> 7\}) &\cup \text{true} \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]

Example: Arith Relation

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x,\{x>7\}) &\cup 7 \quad (5,\{x>7\}) \cup 5 \\
(x > 5, \{x -> 7\}) &\cup \text{true} \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]

Example: If Then Else Rule

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x,\{x>7\}) &\cup 7 \quad (5,\{x>7\}) \cup 5 \\
(x > 5, \{x -> 7\}) &\cup \text{true} \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]

Example: Assignment

\[
\begin{align*}
7 > 5 &= \text{true} \\
(x,\{x>7\}) &\cup 7 \quad (5,\{x>7\}) \cup 5 \\
(x > 5, \{x -> 7\}) &\cup \text{true} \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]

Example: Arith Op

\[
\begin{align*}
? + ? &= ? \\
(2,\{x>7\}) &\cup 2 \quad (3,\{x>7\}) \cup 3 \\
(2+3, \{x -> 7\}) &\cup\ ? \\
\text{(if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x -> 7\}) &\cup? 
\end{align*}
\]
Example: Numerals

\[ 2 + 3 = 5 \]
\[ (2, \{x > 7\}) \cup 2 \quad (3, \{x > 7\}) \cup 3 \]
\[ 7 > 5 = \text{true} \]
\[ (x, \{x > 7\}) \cup 7 \quad (5, \{x > 7\}) \cup 5 \]
\[ (x > 5, \{x > 7\}) \cup \text{true} \quad \downarrow ? \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x > 7\}) \downarrow ? \]

Example: Arith Op

\[ 2 + 3 = 5 \]
\[ (2, \{x > 7\}) \cup 2 \quad (3, \{x > 7\}) \cup 3 \]
\[ 7 > 5 = \text{true} \]
\[ (x, \{x > 7\}) \cup 7 \quad (5, \{x > 7\}) \cup 5 \]
\[ (x > 5, \{x > 7\}) \cup \text{true} \quad \downarrow ? \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x > 7\}) \downarrow ? \]

Example: Assignment

\[ 2 + 3 = 5 \]
\[ (2, \{x > 7\}) \cup 2 \quad (3, \{x > 7\}) \cup 3 \]
\[ 7 > 5 = \text{true} \]
\[ (x, \{x > 7\}) \cup 7 \quad (5, \{x > 7\}) \cup 5 \]
\[ (x > 5, \{x > 7\}) \cup \text{true} \quad \downarrow ? \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x > 7\}) \downarrow ? \]

Example: If Then Else Rule

\[ 2 + 3 = 5 \]
\[ (2, \{x > 7\}) \cup 2 \quad (3, \{x > 7\}) \cup 3 \]
\[ 7 > 5 = \text{true} \]
\[ (x, \{x > 7\}) \cup 7 \quad (5, \{x > 7\}) \cup 5 \]
\[ (x > 5, \{x > 7\}) \cup \text{true} \quad \downarrow ? \]
\[ (\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x > 7\}) \downarrow ? \]

Let in Command

\[ (E, m) \downarrow \{ C, m[I < -v] \} \downarrow m' \]
\[ \text{(let } I = E \text{ in } C, m) \downarrow m'' \]

Where \( m'' (y) = m' (y) \) for \( y \neq I \) and \( m'' (I) = m (I) \) if \( m(I) \) is defined, and \( m'' (I) \) is undefined otherwise

Example

\[ (x, \{x > 5\}) \cup 5 \quad (3, \{x > 5\}) \cup 3 \]
\[ (x+3, \{x > 5\}) \cup 8 \]
\[ (5, \{x > 17\}) \cup 5 \quad (x := x + 3, \{x > 5\}) \cup \{x > 8\} \]
\[ \text{(let } x = 5 \text{ in } (x := x + 3, \{x > 17\}) \downarrow ? \]
Example

\[(x, \{x > 5\}) \Downarrow 5\]
\[(3, \{x > 5\}) \Downarrow 3\]
\[(x + 3, \{x > 5\}) \Downarrow 8\]
\[(5, \{x > 17\}) \Downarrow 5\]
\[(x := x + 3, \{x > 5\}) \Downarrow \{x > 8\}\]
\[(\text{let } x = 5 \text{ in } (x := x + 3), \{x > 17\}) \Downarrow \{x > 17\}\]

Comment

- Simple Imperative Programming Language introduces variables *implicitly* through assignment
- The let-in command introduces scoped variables *explicitly*
- Clash of constructs apparent in awkward semantics

Interpretation Versus Compilation

- A *compiler* from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An *interpreter* of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
  - One procedure for each syntactic category (nonterminal)
    - eg one for expressions, another for commands
  - If Natural semantics used, tells how to compute final value from code
  - If Transition semantics used, tells how to compute next “state”
    - To get final value, put in a loop

Natural Semantics Example

- compute_exp (Var(v), m) = look_up v m
- compute_exp (Int(n), _) = Num (n)
- ...
- compute_com(IfExp(b,c1,c2),m) =
  if compute_exp (b,m) = Bool(true)
  then compute_com (c1,m)
  else compute_com (c2,m)
Natural Semantics Example

- compute_com(While(b,c), m) =
  if compute_exp (b,m) = Bool(false)
  then m
  else compute_com
  (While(b,c), compute_com(c,m))

- May fail to terminate - exceed stack limits
- Returns no useful information then