Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Disambiguating a Grammar

- Idea: Each non-terminal represents all strings having some property
  - In fact each string of terminals and non-terminals represents all strings having some property (Remember reg exp semantics)
- Identify these properties (often in terms of things that can’t happen)
- Use these properties to inductively guarantee every string in language has a unique parse
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)

**Characterize each non-terminal by a language invariant**
- Replace old rules to use new non-terminals
- Rinse and repeat
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
  - Saw how last class
## Precedence Table - Sample

<table>
<thead>
<tr>
<th></th>
<th>Fortan</th>
<th>Pascal</th>
<th>C/C++</th>
<th>Ada</th>
<th>SML</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>highest</strong></td>
<td><strong>,</strong></td>
<td>*, /, div, mod</td>
<td>++, --</td>
<td><strong>,</strong></td>
<td>div, mod, /, *</td>
</tr>
<tr>
<td>* /, +, -</td>
<td>* /, +, -</td>
<td>* /, mod</td>
<td>mod</td>
<td>* /, mod</td>
<td></td>
</tr>
<tr>
<td>+, -</td>
<td>+, -</td>
<td>+, -</td>
<td>+, -</td>
<td>::</td>
<td></td>
</tr>
</tbody>
</table>
More Disambiguating Grammars

- \[ M ::= M \ast M \mid ( M ) \mid M ++ \mid 6 \]
- Ambiguous because of associativity of \(*\) and \(\ast\)
- Because of conflict between \(*\) and \(\ast\):
- \[ 6 \ast 6 ++ \quad 6 \ast 6 ++ \]

4/4/24
\[ M ::= M \ast M \mid (M) \mid M ++ \mid 6 \]

- How to disambiguate?
- Choose associativity for \( \ast \)
- Choose precedence between \( \ast \) and \( ++ \)
- Four possibilities
- Four different approaches
- Some easier than others
- Will do --- My choice, then yours if time
M ::= M * M | ( M ) | M ++ | 6

- Think about 6 * 6 ++ * 6 * 6 ++
- Let’s start with observations
- If * binds less tightly than ++, then no * can be the immediate subtree to a ++.
  - We would need a language for things that don’t parse as *
- If * binds more tightly than ++, then ...
- The right subtree to * can’t be a ++
- But the left can!
- Need different languages of the left and right
M ::= M * M | ( M ) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++  6 ++ * 6

M ::= = M++ | StarExp | (M) | 6

- What is StarExp
- It is everything that parses as a * and can’t parse as a ++
- But what is the associativity of *?
- I’ll choose left
M ::= M * M | ( M ) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++ * 6 ++ * 6
- * Left assoc

M ::= = M++ | StarExp | (M) | 6

StarExp ::= PossStar * NotStarNotPlusPlus

What is PossStar? It could it be a *, but it also doesn’t have to be.

Can it be ++? YES! It can be anything

It is M!
\[ M ::= M \ast M \mid (M) \mid M \text{ ++} \mid 6 \]

- * higher prec than ++
  - \[ 6 \ast 6 \text{ ++} \quad 6 \text{ ++} \ast 6 \]
- * Left assoc

\[ M ::= = M++ \mid \text{StarExp} \mid (M) \mid 6 \]

\[ \text{StarExp ::= } M \ast \text{NotStarNotPlusPlus} \]
M ::= M * M | (M) | M ++ | 6

* higher prec than ++
  * 6 * 6 ++ 6 ++ * 6

* Left assoc

M ::= M++ | StarExp | (M) | 6

StarExp ::= M * NotStarNotPlusPlus

But what is NotStarNotPlusPlus?

Well, the other two original rules: (M) | 6
M ::= M * M | (M) | M ++ | 6

- * higher prec than ++
  - 6 * 6 ++  6 ++ * 6

- * Left assoc

M ::= M++ | StarExp | (M) | 6

StarExp ::= M * NotStarNotPlusPlus

NotStarNotPlusPlus ::= (M) | 6

But we have (M) | 6 twice, and it’s the same language each time. Let’s have one
\[ M ::= M \ast M \mid (M) \mid M++ \mid 6 \]

- * higher prec than ++
  - \( 6 \ast 6 ++ \ 6++*6 \)
- * Left assoc
- \( M ::= M++ \mid \text{StarExp} \mid \text{NotStarNotPlusPlus} \)
- \( \text{StarExp ::=} M \ast \text{NotStarNotPlusPlus} \)
- \( \text{NotStarNotPlusPlus ::=} (M) \mid 6 \)
Parser Code

- `<grammar>.ml` defines one parsing function per entry point
- Parsing function takes a lexing function (lexer buffer to token) and a lexer buffer as arguments
- Returns semantic attribute of corresponding entry point
File format:

```ocaml
%{
    <header>
%
}

<declarations>

%%

<rules>

%%

<trailer>
```
Ocamlyacc <header>

- Contains arbitrary Ocaml code
- Typically used to give types and functions needed for the semantic actions of rules and to give specialized error recovery
- May be omitted
- <footer> similar. Possibly used to call parser
Ocamlyacc <declarations>

- %token symbol ... symbol
  - Declare given symbols as tokens
- %token <type> symbol ... symbol
  - Declare given symbols as token constructors, taking an argument of type <type>
- %start symbol ... symbol
  - Declare given symbols as entry points; functions of same names in <grammar>.ml
Ocamlyacc <declarations>

- `%type <type> symbol ... symbol

  Specify type of attributes for given symbols. Mandatory for start symbols

- `%left symbol ... symbol

- `%right symbol ... symbol

- `%nonassoc symbol ... symbol

  Associate precedence and associativity to given symbols. Same line, same precedence; earlier line, lower precedence (broader scope)
Ocamlyacc <rules>

- `nonterminal :`
  
  ```
  symbol ... symbol { semantic_action } \\
  ... \\
  symbol ... symbol { semantic_action }
  
  ;
  ```

- Semantic actions are arbitrary Ocaml expressions
- Must be of same type as declared (or inferred) for `nonterminal`
- Access semantic attributes (values) of symbols by position: $1$ for first symbol, $2$ to second ...
Example - Base types

(* File: expr.ml *)

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

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

and factor =
  Id_as_Factor of string
| Parenthesized_Expr_as_Factor of expr
Example - Lexer (exprlex.mll)

```latex
{ (*open Exprpparse*) } 
let numeric = ['0' - '9'] 
let letter = ['a' - 'z' 'A' - 'Z'] 
rule token = parse 
  | "+" {Plus_token} 
  | "-" {Minus_token} 
  | "+" {Times_token} 
  | "+" {Divide_token} 
  | "(" {Left_parenthesis} 
  | ")" {Right_parenthesis} 
  | letter (letter|numeric|"_"))* as id {Id_token id} 
  | [' ' '	' '
'] {token lexbuf} 
  | eof {EOL}
```
Example - Parser (exprparse.mly)

```ml
{% open Expr
%}
%token <string> Id_token
%token Left_parenthesis Right_parenthesis
%token Times_token Divide_token
%token Plus_token Minus_token
%token EOL
%start main
%type <expr> main
%%
```
Example - Parser (exprparse.mly)

expr:
  term
    { Term_as_Expr $1 }
  | term Plus_token expr
    { Plus_Expr ($1, $3) }
  | term Minus_token expr
    { Minus_Expr ($1, $3) }
Example - Parser (exprparse.mly)

term:

factor
    { Factor_as_Term $1 } |

factor Times_token term
    { Mult_Term ($1, $3) } |

factor Divide_token term
    { Div_Term ($1, $3) }
Example - Parser (exprparse.mly)

factor:
   Id_token
       { Id_as_Factor $1 }
    | Left_parenthesis expr Right_parenthesis
       {Parenthesized_Expr_as_Factor $2 }

main:
   | expr EOL
       { $1 }
Example - Using Parser

```
# use "expr.ml";;
...
# use "exprparse.ml";;
...
# use "exprlex.ml";;
...
# let test s =
   let lexbuf = Lexing.from_string (s^^"\n") in
   main token lexbuf;;
```
Example - Using Parser

```
# test "a + b";;
- : expr =

Plus_Expr
  (Factor_as_Term (Id_as_Factor "a"),
   Term_as_Expr (Factor_as_Term (Id_as_Factor "b")))
```
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
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>) \mid <\text{Sum}\> + <\text{Sum}\>\)

\(<\text{Sum}\> \Rightarrow

= \bullet (0 + 1) + 0 \quad \text{shift}
Example: \(<\text{Sum}> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\) \\
\langle\text{Sum}\rangle \Rightarrow \\
= (0 + 1) + 0 \quad \text{shift} \\
= (0 + 1) + 0 \quad \text{shift}
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)\) \mid <\text{Sum}> + <\text{Sum}>\)

\(<\text{Sum}> \Rightarrow\)

\[\Rightarrow (0 \circ + 1) + 0\]  \hspace{1cm} \text{reduce}

\[= (\bigcirc 0 + 1) + 0\]  \hspace{1cm} \text{shift}

\[= \bigcirc (0 + 1) + 0\]  \hspace{1cm} \text{shift}
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (\langle\text{Sum}\rangle) \mid \langle\text{Sum}\rangle + \langle\text{Sum}\rangle\)

\[\langle\text{Sum}\rangle \Rightarrow\]

\[= (\langle\text{Sum}\rangle \bullet + 1) + 0 \quad \text{shift}\]
\[=> (0 \bullet + 1) + 0 \quad \text{reduce}\]
\[= (\bullet 0 + 1) + 0 \quad \text{shift}\]
\[= \bullet (0 + 1) + 0 \quad \text{shift}\]
Example: $\langle\text{Sum}\rangle = 0 \mid 1 \mid (\langle\text{Sum}\rangle)$

$\langle\text{Sum}\rangle \Rightarrow$

$$
= (\langle\text{Sum}\rangle + \pm 1) + 0 \quad \text{shift}
= (\langle\text{Sum}\rangle \pm 1) + 0 \quad \text{shift}
=> (0 \pm 1) + 0 \quad \text{reduce}
= (\pm 0 + 1) + 0 \quad \text{shift}
= \pm (0 + 1) + 0 \quad \text{shift}
$$
Example: $<\text{Sum}> = 0 \mid 1 \mid ( <\text{Sum}> ) \mid <\text{Sum}> + <\text{Sum}>$

$<\text{Sum}> \Rightarrow$

$=> ( <\text{Sum}> + 1 ) + 0 \quad \text{reduce}$
$= ( <\text{Sum}> + 1 ) + 0 \quad \text{shift}$
$= ( <\text{Sum}> + 1 ) + 0 \quad \text{shift}$
$=> ( 0 + 1 ) + 0 \quad \text{reduce}$
$= ( 0 + 1 ) + 0 \quad \text{shift}$
$= ( 0 + 1 ) + 0 \quad \text{shift}$
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>) \mid <\text{Sum}> + <\text{Sum}>\)

\(<\text{Sum}> \Rightarrow \)

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

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

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

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

\[
=> ( 0 + 1 ) + 0 \quad \text{reduce}
\]

\[
= ( 0 + 1 ) + 0 \quad \text{shift}
\]

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

\(<\text{Sum}\> \Rightarrow \)

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

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

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

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

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

\[
\Rightarrow \ ( \ 0 + 1 ) + 0 \quad \text{reduce}
\]

\[
= \ ( \ 0 + 1 ) + 0 \quad \text{shift}
\]

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

\[
=> ( <\text{Sum}\> ) \bullet + 0 \quad \text{reduce}
\]
\[
= ( <\text{Sum}\> \bullet ) + 0 \quad \text{shift}
\]
\[
=> ( <\text{Sum}\> + <\text{Sum}\> \bullet ) + 0 \quad \text{reduce}
\]
\[
=> ( <\text{Sum}\> + 1 \bullet ) + 0 \quad \text{reduce}
\]
\[
= ( <\text{Sum}\> + \bullet 1 ) + 0 \quad \text{shift}
\]
\[
= ( <\text{Sum}\> \bullet + 1 ) + 0 \quad \text{shift}
\]
\[
=> ( 0 \bullet + 1 ) + 0 \quad \text{reduce}
\]
\[
= ( \bullet 0 + 1 ) + 0 \quad \text{shift}
\]
\[
= \bullet ( 0 + 1 ) + 0 \quad \text{shift}
\]
Example: $\langle \text{Sum} \rangle = 0 \mid 1 \mid (\langle \text{Sum} \rangle) \mid \langle \text{Sum} \rangle + \langle \text{Sum} \rangle$

$\langle \text{Sum} \rangle \Rightarrow$

= $\langle \text{Sum} \rangle \bullet + 0$ shift

$\Rightarrow (\langle \text{Sum} \rangle) \bullet + 0$ reduce

= $(\langle \text{Sum} \rangle \bullet) + 0$ shift

$\Rightarrow (\langle \text{Sum} \rangle + \langle \text{Sum} \rangle \bullet) + 0$ reduce

= $(\langle \text{Sum} \rangle + 1 \bullet) + 0$ reduce

$\Rightarrow (\langle \text{Sum} \rangle + \bullet 1) + 0$ shift

= $(\langle \text{Sum} \rangle \bullet + 1) + 0$ shift

$\Rightarrow (0 \bullet + 1) + 0$ reduce

= $(\bullet 0 + 1) + 0$ shift

= $\bullet (0 + 1) + 0$ shift
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)
\mid \text{<Sum> + <Sum>}

\text{<Sum> => }

= \ <\text{Sum}\> + 0
\quad \text{shift}

= \ <\text{Sum}\> 0 + 0
\quad \text{shift}

= \ (\ <\text{Sum}\> 0) + 0
\quad \text{shift}

= \ (\ <\text{Sum}\> + <\text{Sum}\> 0) + 0
\quad \text{reduce}

= \ (\ <\text{Sum}\> + 1 0) + 0
\quad \text{reduce}

= \ (\ <\text{Sum}\> + 0 1) + 0
\quad \text{shift}

= \ (\ <\text{Sum}\> 0 + 1) + 0
\quad \text{shift}

= \ (0 0 + 1) + 0
\quad \text{reduce}

= \ (0 0 + 1) + 0
\quad \text{shift}

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

$<\text{Sum}> \Rightarrow$

$\Rightarrow <\text{Sum}> + 0 \quad \text{reduce}$

$= <\text{Sum}> + \cdot 0 \quad \text{shift}$

$= <\text{Sum}> \cdot + 0 \quad \text{shift}$

$\Rightarrow ( <\text{Sum}> ) \cdot + 0 \quad \text{reduce}$

$= ( <\text{Sum}> \cdot ) + 0 \quad \text{shift}$

$\Rightarrow ( <\text{Sum}> + <\text{Sum}> \cdot ) + 0 \quad \text{reduce}$

$\Rightarrow ( <\text{Sum}> + 1 \cdot ) + 0 \quad \text{reduce}$

$= ( <\text{Sum}> + \cdot 1 ) + 0 \quad \text{shift}$

$= ( <\text{Sum}> \cdot + 1 ) + 0 \quad \text{shift}$

$\Rightarrow ( 0 \cdot + 1 ) + 0 \quad \text{reduce}$

$= ( \cdot 0 + 1 ) + 0 \quad \text{shift}$

$= \cdot ( 0 + 1 ) + 0 \quad \text{shift}$
Example: \(<\text{Sum}\> = 0 \mid 1 \mid (<\text{Sum}\>)\) 
\mid \text{<Sum> + <Sum>}

\[
\begin{align*}
\text{<Sum>} & \Rightarrow \text{<Sum>} + \text{<Sum>} \Rightarrow \text{reduce} \\
& \Rightarrow \text{<Sum>} + 0 \Rightarrow \text{reduce} \\
& = \text{<Sum>} + 0 \Rightarrow \text{shift} \\
& = \text{<Sum>} + 0 \Rightarrow \text{shift} \\
& \Rightarrow (\text{<Sum>}) + 0 \Rightarrow \text{reduce} \\
& = (\text{<Sum>} + 0) + 0 \Rightarrow \text{shift} \\
& \Rightarrow (\text{<Sum>} + \text{<Sum>} \Rightarrow \text{reduce} \\
& \Rightarrow (\text{<Sum>} + 1 \Rightarrow \text{reduce} \\
& = (\text{<Sum>} + 1 \Rightarrow \text{shift} \\
& = (\text{<Sum>} + 1 \Rightarrow \text{shift} \\
& \Rightarrow (0 + 1) + 0 \Rightarrow \text{reduce} \\
& = (0 + 1) + 0 \Rightarrow \text{shift} \\
& = (0 + 1) + 0 \Rightarrow \text{shift}
\end{align*}
\]
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)

\(<\text{Sum}> \bullet \Rightarrow <\text{Sum}> + <\text{Sum}> \bullet \quad \text{reduce}
\Rightarrow <\text{Sum}> + 0 \bullet \quad \text{reduce}
= <\text{Sum}> + \bullet 0 \quad \text{shift}
= <\text{Sum}> \bullet + 0 \quad \text{shift}
=> ( <\text{Sum}> ) \bullet + 0 \quad \text{reduce}
= ( <\text{Sum}> \bullet ) + 0 \quad \text{shift}
=> ( <\text{Sum}> + <\text{Sum}> \bullet ) + 0 \quad \text{reduce}
=> ( <\text{Sum}> + 1 \bullet ) + 0 \quad \text{reduce}
= ( <\text{Sum}> + \bullet 1 ) + 0 \quad \text{shift}
= ( <\text{Sum}> \bullet + 1 ) + 0 \quad \text{shift}
=> ( 0 \bullet + 1 ) + 0 \quad \text{reduce}
= ( \bullet 0 + 1 ) + 0 \quad \text{shift}
= \bullet ( 0 + 1 ) + 0 \quad \text{shift}
Example

\[(0 + 1) + 0\]
Example

\[(0 + 1) + 0\]
Example

\[(0 + 1) + 0\]
Example

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

\[
(0 + 1) + 0
\]
Example

\[
\langle \text{Sum} \rangle
\]

(0 + 1) + 0
Example

\[ (\langle \text{Sum} \rangle 0 + \langle \text{Sum} \rangle 1 ) + 0 \]
Example

\[(0 + 1) + 0\]
Example

\[
\left( \begin{array}{c}
\text{<Sum>}
\end{array} \right) + 1 + 0
\]

\[
\left( \begin{array}{c}
0 + 1
\end{array} \right) + 0
\]
Example

\[
\begin{align*}
\langle \text{Sum}\rangle &+ \langle \text{Sum}\rangle \\
\langle \text{Sum}\rangle + 0 &+ 1
\end{align*}
\]
Example

\[
\langle \text{Sum} \rangle + \langle \text{Sum} \rangle + \langle \text{Sum} \rangle = \langle \text{Sum} \rangle
\]

\[
(0 + 1) + 0 = 1
\]
Example

\[
\begin{align*}
\langle \text{Sum} \rangle & \langle \text{Sum} \rangle \\
\langle \text{Sum} \rangle & \langle \text{Sum} \rangle \\
( & + \\
0 & + 1 \\
+ & 0
\end{align*}
\]
Example

\[
\begin{align*}
\text{Example} & \quad \text{Example} \\
\langle \text{Sum} \rangle & \quad \langle \text{Sum} \rangle \\
\langle \text{Sum} \rangle & \quad \langle \text{Sum} \rangle \\
\langle \text{Sum} \rangle & \quad \langle \text{Sum} \rangle \\
(0 & + 1) + 0
\end{align*}
\]
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 $(n, toks)$
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 \textbf{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
LR(i) Parsing Algorithm

7. If action = **accept**
   - Stop parsing, return success

8. If action = **error**,  
   - Stop parsing, return failure
Adding Synthesized Attributes

- Add to each \textbf{reduce} a rule for calculating the new synthesized attribute from the component attributes.
- Add to each non-terminal pushed onto the stack, the attribute calculated for it.
- When performing a \textbf{reduce},
  - gather the recorded attributes from each non-terminal popped from stack
  - Compute new attribute for non-terminal pushed onto stack
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: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)\)

\[
\begin{align*}
\bullet 0 + 1 + 0 & \quad \text{shift} \\
\rightarrow 0 \bullet + 1 + 0 & \quad \text{reduce} \\
\rightarrow <\text{Sum}> \bullet + 1 + 0 & \quad \text{shift} \\
\rightarrow <\text{Sum}> + \bullet 1 + 0 & \quad \text{shift} \\
\rightarrow <\text{Sum}> + 1 \bullet + 0 & \quad \text{reduce} \\
\rightarrow <\text{Sum}> + <\text{Sum}> \bullet + 0
\end{align*}
\]
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

- $S ::= A \mid aB$
- $A ::= abc$
- $B ::= bc$

- $abc$ shift
- $abc$ shift
- $a \quad bc$ shift
- $ab \quad c$ shift
- $abc$ shift

Problem: reduce by $B ::= bc$ then by $S ::= aB$, or by $A ::= abc$ then $S ::= A$?
Three Main Topics of the Course

I
New Programming Paradigm

II
Language Translation

III
Language Semantics
Programming Languages & Compilers

Order of Evaluation

I
New Programming Paradigm

II
Language Translation

III
Language Semantics

Specification to Implementation
III : Language Semantics

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics
Programming Languages & Compilers

Order of Evaluation

Operational Semantics
Lambda Calculus
Axiomatic Semantics

Specification to Implementation

CS422
CS426
CS477
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 (i.e., 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:
  \[
  \{\text{Precondition}\} \text{ Program } \{\text{Postcondition}\}
  \]

- Source of idea of \(loop\ invariant\)
Denotational Semantics

- Construct a function $M$ 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 “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

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

- $(B, m) \downarrow \text{false} \quad (B, m) \downarrow \text{true}$
- $(B \land B', m) \downarrow \text{false} \quad (B \land B', m) \downarrow \text{true}$
- $(B, m) \downarrow \text{false} \quad (B, m) \downarrow \text{true}$
- $(B, m) \downarrow \text{true} \quad (B, m) \downarrow \text{false}$
- $(B \lor B', m) \downarrow \text{true} \quad (B \lor B', m) \downarrow \text{false}$
- $(B, m) \downarrow \text{true} \quad (B, m) \downarrow \text{false}$
- $(\neg B, m) \downarrow \text{false} \quad (\neg B, m) \downarrow \text{true}$
- $(B \lor B', m) \downarrow \text{true} \quad (B \lor B', m) \downarrow \text{false}$
- $(B, m) \downarrow \text{true} \quad (B, m) \downarrow \text{false}$
- $(B, m) \downarrow \text{false} \quad (B, m) \downarrow \text{true}$
- $(B \land B', m) \downarrow \text{false} \quad (B \land B', m) \downarrow \text{true}$
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:

\[
\begin{align*}
(E, m) & \downarrow V \\
(I := E, m) & \downarrow m[I \leftarrow V ]
\end{align*}
\]

Sequencing:

\[
\begin{align*}
(C, m) & \downarrow m' \\
(C', m') & \downarrow m'' \\
(C; C', m) & \downarrow m''
\end{align*}
\]
If Then Else Command

\[
(B, m) \downarrow \text{true} \quad (C, m) \downarrow m' \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \downarrow m'
\]

\[
(B, m) \downarrow \text{false} \quad (C', m) \downarrow m' \\
\text{(if } B \text{ then } C \text{ else } C' \text{ fi, } m) \downarrow m'
\]
While Command

\[(B, m) \downarrow \text{false} \]

\[(\text{while } B \text{ do } C \text{ od, } m) \downarrow m\]

\[(B, m) \downarrow \text{true} \quad (C, m) \downarrow m' \quad (\text{while } B \text{ do } C \text{ od, } m') \downarrow m''\]

\[(\text{while } B \text{ do } C \text{ od, } m) \downarrow m''\]
Example: If Then Else Rule

(if x > 5 then y := 2 + 3 else y := 3 + 4 fi,
{x -> 7}) ↓ ?
Example: If Then Else Rule

\[(x > 5, \{x \rightarrow 7\}) \downarrow?\]

\[(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi, } \{x \rightarrow 7\}) \downarrow?\]
Example: Arith Relation

\[ \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \]
\[ \{x \rightarrow 7\} \downarrow ? \]
Example: Identifier(s)

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

7 > 5 = true

\( (x,\{x->7\}) \downarrow \begin{array}{c} 7 \\ (5,\{x->7\}) \downarrow 5 \end{array} \)

\( (x > 5, \{x -> 7\}) \downarrow \text{true} \)

\( \text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \)
\( \{x -> 7\} \) \downarrow ?
Example: If Then Else Rule

7 > 5 = true
(x, {x -> 7}) ⊑ 7 (5, {x -> 7}) ⊑ 5
(x > 5, {x -> 7}) ⊑ true
(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ⊑ ?

(y := 2 + 3, {x -> 7}) ⊑ ?

**Example: Assignment**

\[
7 > 5 = \text{true} \quad (x, \{x > 7\}) \downarrow 7 \quad (5, \{x > 7\}) \downarrow 5 \quad (x > 5, \{x \to 7\}) \downarrow \text{true} \quad ((y := 2 + 3, \{x \to 7\}) \downarrow ? \quad (y := 3 + 4 \text{ fi}, \{x \to 7\}) \downarrow ?
\]
Example: Arith Op

\[ (x, \{x \mapsto 7\}) \cup (5, \{x \mapsto 7\}) \cup 5 \]

\[ (x > 5, \{x \mapsto 7\}) \cup 7 \]

\[ 7 > 5 = \text{true} \]

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

\[ (2, \{x \mapsto 7\}) \cup ? \]

\[ (2 + 3, \{x \mapsto 7\}) \cup ? \]

\[ (3 + 3, \{x \mapsto 7\}) \cup ? \]

\[ ? + ? = ? \]

\[ y := 2 + 3, \{x \mapsto 7\} \]

\[ \{x \mapsto 7\} \]

\[ (x > 7) \]

\[ \{x \mapsto 7\} \]

\[ ? \]

\[ \]
Example: Numerals

\[
\begin{align*}
2 + 3 &= 5 \\
(2, \{x \mapsto 7\}) &\downarrow 2 & (3, \{x \mapsto 7\}) &\downarrow 3 \\
7 &> 5 = \text{true} \\
(x, \{x \mapsto 7\}) &\downarrow 7 & (5, \{x \mapsto 7\}) &\downarrow 5 \\
(x > 5, \{x \mapsto 7\}) &\downarrow \text{true} \\
(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} & (y := 2 + 3, \{x \mapsto 7\}) & \downarrow ?
\end{align*}
\]
Example: Arith Op

2 + 3 = 5

(2, {x -> 7}) ↓ 2  (3, {x -> 7}) ↓ 3

7 > 5 = true

(2 + 3, {x -> 7}) ↓ 5

(x, {x -> 7}) ↓ 7  (5, {x -> 7}) ↓ 5

(x > 5, {x -> 7}) ↓ true

(y := 2 + 3, {x -> 7}) ↓ ?

(if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) ↓ ?
Example: Assignment

\[
\begin{align*}
2 + 3 &= 5 \\
(2, \{x \to 7\}) &\downarrow 2 \\
(3, \{x \to 7\}) &\downarrow 3 \\
7 &> 5 = \text{true} \\
(x, \{x \to 7\}) &\downarrow 7 \\
(5, \{x \to 7\}) &\downarrow 5 \\
(x > 5, \{x \to 7\}) &\downarrow \text{true} \\
(y := 2 + 3, \{x \to 7\}) &\downarrow \{x \to 7, y \to 5\} \\
\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi,} \\
\{x \to 7\} &\downarrow ?
\end{align*}
\]
Example: If Then Else Rule

$$2 + 3 = 5$$

$$(2, \{x \rightarrow 7\}) \Downarrow 2$$  $$(3, \{x \rightarrow 7\}) \Downarrow 3$$

$$7 > 5 = \text{true}$$

$$(x, \{x \rightarrow 7\}) \Downarrow 7$$  $$(5, \{x \rightarrow 7\}) \Downarrow 5$$

$$(x > 5, \{x \rightarrow 7\}) \Downarrow \text{true}$$

$$(\text{if } x > 5 \text{ then } y := 2 + 3 \text{ else } y := 3 + 4 \text{ fi}, \{x \rightarrow 7\}) \Downarrow \{x \rightarrow 7, y \rightarrow 5\}$$
Let in Command

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

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

\[(x, \{x \rightarrow 5\}) \downarrow 5 \quad (3, \{x \rightarrow 5\}) \downarrow 3\]

\[(x+3, \{x \rightarrow 5\}) \downarrow 8\]

\[(5, \{x \rightarrow 17\}) \downarrow 5 \quad (x := x+3, \{x \rightarrow 5\}) \downarrow \{x \rightarrow 8\}\]

(let \(x = 5\) in (x := x+3), \{x \rightarrow 17\}) \downarrow ?
Example

\[(x,\{x->5\}) \Downarrow 5 \quad (3,\{x->5\}) \Downarrow 3\]

\[(x+3,\{x->5\}) \Downarrow 8\]

\[(5,\{x->17\}) \Downarrow 5 \quad (x:=x+3,\{x->5\}) \Downarrow \{x->8\}\]

(\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