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

Example

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

Predence in Grammar

- Higher precedence translates to longer derivation chain
- Example:
  \[
  \langle \text{exp} \rangle ::= 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
  \quad \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle
  \]
- Choice: + and * left assoc, * higher prec than +
- First Problem: * higher prec than +
  - Needs \langle \text{ntp} \rangle for all strings that can not be topmost parsed as a plus
  - Assume all other nonterminals mean want
- Ambiguous grammar:
  \[
  \langle \text{exp} \rangle ::= 0 \mid 1 \mid \langle \text{exp} \rangle + \langle \text{exp} \rangle \\
  \quad \mid \langle \text{exp} \rangle * \langle \text{exp} \rangle \\
  \]
- Example:
  \[
  0 + 1 + 0 \\
  1 * 1 + 1
  \]
- Choice: + and * left assoc, * higher prec than +
- Second Problem: * left assoc
  - Need \langle \text{ntpm} \rangle for all strings that can not be topmost parsed as a plus or a mult
  - Assume all other nonterminals mean want
- Ambiguous grammar:
  \[
  \langle \text{exp} \rangle ::= 0 \mid 1 \\
  \]
  \[
  \langle \text{ntp} \rangle ::= 0 \mid 1 \mid \langle \text{ntp} \rangle * \langle \text{ntpm} \rangle
  \]
- Example:
  \[
  0 + 1 + 0 \\
  1 * 1 + 1
  \]
**Predence in Grammar**

- Higher precedence translates to longer derivation chain
- Example:
  \[ \text{exp} ::= 0 | 1 | \text{exp} + \text{exp} \]
  
  \[ | \text{exp} \times \text{exp} \]
- Choice: + and * left assoc, * higher prec than +
- Third problem: + left assoc
  - Need \text{ntp} for all strings that can not be topmost parsed as a plus
  - Already have it
- \[ \text{exp} ::= \text{exp} + \text{ntp} | \text{ntp} \]
  
  \[ \text{ntp} ::= \text{ntpm} | \text{ntp} \times \text{ntpm} \]
  
  \[ \text{ntpm} ::= 0 | 1 \]

10/4/07

---

**Disambiguating a Grammar**

- \[ \text{exp} ::= 0|1| b\text{exp} | \text{exp}a \]
  
  | \[ \text{exp}m\text{exp} \]
- Want
  - \text{a} has higher precedence than \text{b},
  - which in turn has higher precedence than \text{m},
  - and such that \text{m} associates to the left.
- Higher precedence translates to longer derivation chain

11/8/19

---

**Disambiguating a Grammar – Take 2**

- \[ \text{exp} ::= 0|1| b\text{exp} | \text{exp}a \]
  
  | \[ \text{exp}m\text{exp} \]
- Want \text{b} has higher precedence than \text{m}, which in turn has higher precedence than \text{a}, and such that \text{m} associates to the right.
  - \text{b 0 m 1 a} \rightarrow \text{b 0 m 1 a}
  - \text{0 a m b 1 OK}
  - \text{b 0 m 1 m 0 a}
  - \text{b 0 m 0 a m 1}

11/8/19

---

**Predence in Grammar**

- Higher precedence translates to longer derivation chain
- Example:
  \[ \text{exp} ::= 0 | 1 | \text{exp} + \text{exp} \]
  
  \[ | \text{exp} \times \text{exp} \]
- Change names \text{ntpm} = \text{id},
  
  \[ \text{ntp} = \text{mult_exp} \]
- Becomes
  
  \[ \text{exp} ::= \text{mult_exp} | \text{exp} + \text{mult_exp} \]
  
  \[ \text{mult_exp} ::= \text{id} | \text{mult_exp} \times \text{id} \]

11/8/19

---

**Disambiguating a Grammar**

- \[ \text{exp} ::= 0|1| b\text{exp} | \text{exp}a \]
  
  | \[ \text{exp}m\text{exp} \]
- Want \text{a} has higher precedence than \text{b}, which in turn has higher precedence than \text{m}, and such that \text{m} associates to the left.
- Higher precedence translates to longer derivation chain
  
  \[ \text{exp} ::= \text{exp}m\text{not m} | \text{not m} \]
  
  \[ \text{not m} ::= b \text{not m} | \text{not b m} \]
  
  \[ \text{not b m} ::= \text{not b m}a | 0 | 1 \]

11/8/19
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.
- `<exp> ::= 
  <no a m> | <not m> m <no a> | <exp> a
  <no a> ::= <no a m> | <no a m> m <no a>
  <not m> ::= <no a m> | <exp> a
  <no a m> ::= b <no a m> | 0 | 1`
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 |
  Parenthesized_Expr_as_Factor of expr

Example - Lexer (exprlex.mll)

{ (*open Exprparse*) }

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)

%{ 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)**

```plaintext
factor:
   Id_token
   { Id_as_Factor $1 } |
   Left_parenthesis expr Right_parenthesis
   {Parenthesized_Expr_as_Factor $2 }
main:
   | expr EOL
   { $1 }
```

**Example - Using Parser**

```plaintext
# #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**

```plaintext
# test "a + b";;
- : expr =
  Plus_Expr
  (Factor_as_Term (Id_as_Factor "a"),
   Term_as_Expr (Factor_as_Term
     (Id_as_Factor "b")))
```

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

```plaintext
<Sum> =>
| ( 0 + 1 ) + 0   shift
```

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

```plaintext
<Sum> =>
| ( ( 0 + 1 ) + 0  shift
```
Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & (0 + 1) + 0 & \text{reduce} \\
\Rightarrow & (0 + 1) + 0 & \text{shift} \\
\Rightarrow & (0 + 1) + 0 & \text{shift}
\end{align*}
11/8/19 31

Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & ( <\text{Sum}> \cdot 1 ) + 0 & \text{shift} \\
\Rightarrow & (0 + 1) + 0 & \text{reduce} \\
\Rightarrow & (0 + 1) + 0 & \text{reduce} \\
\Rightarrow & (0 + 1) + 0 & \text{shift} \\
\Rightarrow & (0 + 1) + 0 & \text{shift}
\end{align*}
11/8/19 32

Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift}
\end{align*}
11/8/19 33

Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift}
\end{align*}
11/8/19 34

Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift}
\end{align*}
11/8/19 35

Example: \(<\text{Sum}> = 0 \mid 1 \mid (<\text{Sum}>)
\mid <\text{Sum}> + <\text{Sum}>\)
\<\text{Sum}> \Rightarrow 
\begin{align*}
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift} \\
\Rightarrow & ( <\text{Sum}> + 1 ) + 0 & \text{shift}
\end{align*}
11/8/19 36
Example

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

Example

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

Example

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

Example

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

Example

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

Example

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

\[
\left( 0 + 1 \right) + 0
\]

Example

\[
\left( 0 + 1 \right) + 0
\]

Example

\[
\left( 0 + 1 \right) + 0
\]

Example

\[
\left( 0 + 1 \right) + 0
\]

Example

\[
\left( 0 + 1 \right) + 0
\]

Example

\[
\left( 0 + 1 \right) + 0
\]
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

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 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

Example

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

Example

\[
\langle \text{Sum} \rangle + \langle 0 \rangle + \langle 1 \rangle + 0
\]
**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

**Adding Synthesized Attributes**
- Add to each 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 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: \( <\text{Sum}> = 0 \ | \ 1 \ | \ (<\text{Sum}>), \) 
\[ <\text{Sum}> + <\text{Sum}> \]
- \( 0 + 1 + 0 \) shift
- \( 0 + 1 + 0 \) reduce
- \( <\text{Sum}> + 1 + 0 \) shift
- \( <\text{Sum}> + 1 + 0 \) shift
- \( <\text{Sum}> + 1 + 0 \) shift
- \( <\text{Sum}> + <\text{Sum}> + 0 \) reduce
- \( <\text{Sum}> + <\text{Sum}> + 0 \) reduce

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

\[ S ::= A \ | \ aB \quad A ::= abc \quad B ::= bc \]
- \( abc \) shift
- \( a \ bc \) shift
- \( ab \ c \) shift
- \( abc \) shift
- 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

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

\[
<\text{term}> ::= <\text{factor}> | <\text{factor}> * <\text{term}> \\
| <\text{factor}> / <\text{term}>
\]

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

Tokens as OCaml Types

- + - * / ( ) <id>
- Becomes an OCaml datatype

```
type token =
  Id_token of string
| Left_parenthesis | Right_parenthesis
| Times_token | Divide_token
| Plus_token | Minus_token
```

Parse Trees as Datatypes

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

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

Parse Trees as Datatypes

\[
<\text{term}> ::= <\text{factor}> | <\text{factor}> * <\text{term}> \\
| <\text{factor}> / <\text{term}>
\]

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

Parse Trees as Datatypes

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

```
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

```
<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 Plus Expression

```
<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 a Plus Expression

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

(match \text{expr} tokens_after_plus

with ( \text{expr}_\text{parse} , \text{tokens}_\text{after}_\text{expr}) ->
( Plus_Expr ( \text{term}_\text{parse} , \text{expr}_\text{parse} ),
\text{tokens}_\text{after}_\text{expr})
)

Building Plus Expression Parse Tree

Parsing a Minus Expression

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

| ( Minus_token :: tokens_after_minus) ->

(match \text{expr} tokens_after_minus

with ( \text{expr}_\text{parse} , \text{tokens}_\text{after}_\text{expr}) ->
( Minus_Expr ( \text{term}_\text{parse} , \text{expr}_\text{parse} ),
\text{tokens}_\text{after}_\text{expr})
)

Parsing an Expression as a Term

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

| _ -> (Term_as_Expr \text{term}_\text{parse} ,
\text{tokens}_\text{after}_\text{term}))

- Code for \text{term} is same except for replacing addition with multiplication and subtraction with division
Parsing Factor as Id

\[ \langle \text{factor} \rangle ::= \langle \text{id} \rangle \]

and factor tokens =
(match tokens
  with (Id_token id_name :: tokens_after_id) =
   (Id_as_Factor id_name, tokens_after_id)

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" ));;

Parsing Factor as Parenthesized Expression

\[ \langle \text{factor} \rangle ::= (\langle \text{expr} \rangle) \]

| factor (Left_parenthesis :: tokens) =
  (match expr tokens
    with (expr_parse , tokens_after_expr) ->

( 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"];;
\[(a + b) * c - d\]

\[a + b * c - d\]

\[a + b * c - d\]

\[a + b) * c - d\]

---

**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 \((\text{token} \times \text{(unit -> token)})\) or \((\text{token} \times \text{(unit -> token option)})\) in place of a 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
  \[
  \text{FIRST}(y) = \{a | y =>* aw\} \cup \{\epsilon | \text{if } y =>* \epsilon\}
  \]
- For each pair of rules \[A ::= y\] and \[A ::= z\], require \[
  \text{FIRST}(y) \cap \text{FIRST}(z) = \{\}
  \]

Example

Grammar:
\[
\langle S \rangle ::= \langle A \rangle \ a \ \langle B \rangle \ b\\
\langle A \rangle ::= \langle A \rangle \ b \mid b\\
\langle B \rangle ::= a \ \langle B \rangle \mid a\\
\]

\[
\text{FIRST}(\langle A \rangle \ b) = \{b\}\\
\text{FIRST}(b) = \{b\}\\
\]
Rules for \(\langle A \rangle\) not pairwise disjoint.

Eliminating Left Recursion

- Rewrite grammar to shift left recursion to right recursion.
- Changes associativity.
- Given
  \[
  \langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle \text{ and}\\
  \langle \text{expr} \rangle ::= \langle \text{term} \rangle\\
  \]
- Add new non-terminal \(\langle e \rangle\) and replace above rules with
  \[
  \langle \text{expr} \rangle ::= \langle \text{term} \rangle <\langle e \rangle\\
  \langle e \rangle ::= + \langle \text{term} \rangle <\langle e \rangle \mid \epsilon\\
  \]


**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> ::= ε
- 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 <B1> ::= a <B1> | ε

You are delaying the decision point

**Programming Languages & Compilers**

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

**Programming Languages & Compilers**

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

**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:**

- $(B, m) \Downarrow \text{false}$
- $(B, m) \Downarrow \text{true}$
- $(B \& B’, m) \Downarrow \text{false}$
- $(B \& B’, m) \Downarrow \text{true}$
- $(B \lor B’, m) \Downarrow \text{false}$
- $(B \lor B’, m) \Downarrow \text{true}$
- $(\text{not } B, m) \Downarrow \text{false}$
- $(\text{not } B, m) \Downarrow \text{true}$

**Relations**

- $(E, m) \Downarrow U$ $(E’, m) \Downarrow V$ $U \sim V = b$
- $(E \sim E’, m) \Downarrow b$

- 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: \[(E, m) \downarrow V \quad (I::=E,m) \downarrow m[I <-- V]\]

Sequencing: \[(C,m) \downarrow m' \quad (C',m') \downarrow m'' \quad (C;C', m) \downarrow m''\]

If Then Else Command

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

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

While Command

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

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

Example: If Then Else Rule

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

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

11/8/19 133

### Example: Identifier(s)

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

11/8/19 134

### Example: Arith Relation

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

11/8/19 135

### Example: If Then Else Rule

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

11/8/19 136

### Example: Assignment

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

11/8/19 137

### Example: Arith Op

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

11/8/19 138
Example: Numerals

\[ 2 + 3 = 5 \]

\[ (2, \{x \mapsto 7\}) \downarrow 2 \quad (3, \{x \mapsto 7\}) \downarrow 3 \]

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

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

\[ (x > 5, \{x \mapsto 7\}) \downarrow \text{true} \]

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

Let in Command

\[ (E, m) \downarrow v \quad (C, m[I \mapsto v]) \downarrow m' \]

(\text{let } I = E \text{ in } C, m) \downarrow m''

\[ \text{Where } m''(y) = m'(y) \text{ for } y \neq I \text{ and} \]
\[ m''(I) = m(I) \text{ if } m(I) \text{ is defined,} \]
\[ \text{and } m''(I) \text{ is undefined otherwise} \]
Example

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

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

- 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

- \[\text{compute}\_\text{exp} (\text{Var}(v), m) = \text{look}\_\text{up}\ v \ m\]
- \[\text{compute}\_\text{exp} (\text{Int}(n), \_ ) = \text{Num} \ (n)\]
- ... \[\text{compute}\_\text{com}(\text{IfExp}(b,c1,c2),m) = \]
  - if \[\text{compute}\_\text{exp} (b,m) = \text{Bool}(\text{true})\]
  - then \[\text{compute}\_\text{com} (c1,m)\]
  - else \[\text{compute}\_\text{com} (c2,m)\]}
Natural Semantics Example

- $\text{compute_com}(\text{While}(b,c), m) =$
- \quad if $\text{compute_exp}(b, m) = \text{Bool}(\text{false})$
- \quad then $m$
- \quad else $\text{compute_com}($
- \quad \quad $\text{While}(b,c), \text{compute_com}(c,m))$

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