Sample Grammar

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

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

<factor> ::= <id> | ( <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

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

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

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

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

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

and factor = 
    Id_as_Factor of string 
  | Parenthesized_Expr_as_Factor of expr
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 a Plus Expression**

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

(match \(\text{expr} \\text{tokens}_{\text{after} +}\) 
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**

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

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

**Parsing a Minus Expression**

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

| ( Minus_token :: \(\text{tokens}_{\text{after} -}\) ) -> 
( match \(\text{expr} \\text{tokens}_{\text{after} -}\) -> 
( Plus_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

<factor> ::= <id>

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")

(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 * token list =
  (Minus_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"))));
   [])
(a + b) * c - d

```
expr
  term
    - expr
    factor
      * term
      factor
        ( expr )
        factor
          term
            + expr
            id
            factor
              id
              id
        a
        b
        c
        d
```

a + b * c - d

```
expr
  term
    + expr
    factor
      * term
      factor
        id
        id
        id
        id
        factor
          factor
            id
            id
            id
            id
        a
        b
        c
        d
```

(a + b * c - d)

```
expr
  term
    + expr
    factor
      * term
      term
        id
        factor
          term
            min
            id
            factor
              id
              id
              factor
                right_parenthesis
                times_token
                id
                minus_token
                id
                id
        a
        b
```

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
```
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. Will use \((\text{token } \ast (\text{unit } \rightarrow \text{token}))\) or \((\text{token } \ast (\text{unit } \rightarrow \text{token option}))\) in place of \(\text{token list}\).

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

Parser must always be able to choose the next action based only on the very next token.

**Pairwise Disjointedness Test:**

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

**Example**

Grammar:
\[
\text{<S>} ::= \text{<A>} \ a \ <B> \ b \\
\text{<A>} ::= \text{<A>} \ b \ | \ b \\
\text{<B>} ::= a \ <B> \ | \ a
\]

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
  \[
  \text{<expr>} ::= \text{<expr>} + \text{<term>} \text{ and} \\
  \text{<expr>} ::= \text{<term>}
  \]
- Add new non-terminal \text{<e>} and replace above rules with
  \[
  \text{<expr>} ::= \text{<term><e>} \\
  \text{<e>} ::= + \text{<term><e>} \mid \varepsilon
  \]
Factoring Grammar

- Test too strong: Can’t handle
  \[ \text{expr} ::= \text{term} [ ( + | - ) \text{expr} ] \]
- Answer: Add new non-terminal and replace above rules by
  \[ \text{expr} ::= \text{term}\text{e} \]
  \[ \text{e} ::= + \text{term}\text{e} \]
  \[ \text{e} ::= - \text{term}\text{e} \]
  \[ \text{e} ::= \epsilon \]
- You are delaying the decision point

Example

Both \(<A>\) and \(<B>\) have problems:
Transform grammar to:

\[ \text{S} ::= \text{A} a \text{B} b \]
\[ \text{A} ::= \text{A} b | b \]
\[ \text{B} ::= a \text{B} | a \]

Transform:

\[ \text{S} ::= \text{A} a \text{B} b \]
\[ \text{A} ::= b\text{A1} \]
\[ \text{A1} ::= b\text{A1} | \epsilon \]
\[ \text{B} ::= a\text{B1} \]
\[ \text{B1} ::= a\text{B1} | \epsilon \]

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

Programming Languages & Compilers

III: Language Semantics

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics

Programming Languages & Compilers

Order of Evaluation

- Operational Semantics
- Lambda Calculus
- Axiomatic Semantics

Speciation to Implementation

- CS422
- CS426
- CS477

Speciation to Implementation
**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 $\mathcal{M}$ 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 “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

- $I \in \text{Identifiers}$
- $N \in \text{Numerals}$
- $B ::= \text{true} | \text{false} | B \& B | B \text{ or } B | \text{not } B$
  | $E < E | E = E$
- $E ::= N | I | E + E | E \ast E | E - E | - E$
- $C ::= \text{skip} | C;C | I ::= E$
  | 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:

<table>
<thead>
<tr>
<th>$B, m$ \Downarrow \text{false}</th>
<th>$B, m$ \Downarrow \text{true}</th>
<th>$(B', m)$ \Downarrow b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(B &amp; B', m)$ \Downarrow \text{false}</td>
<td>$(B &amp; B', m)$ \Downarrow b</td>
<td></td>
</tr>
<tr>
<td>$(B \text{ or } B', m)$ \Downarrow \text{true}</td>
<td>$(B \text{ or } B', m)$ \Downarrow b</td>
<td></td>
</tr>
<tr>
<td>$(\text{not } B, m)$ \Downarrow \text{false}</td>
<td>$(\text{not } B, m)$ \Downarrow \text{true}</td>
<td></td>
</tr>
</tbody>
</table>

Relations

<table>
<thead>
<tr>
<th>$(E, m) \Downarrow U$</th>
<th>$(E', m) \Downarrow V$</th>
<th>$U \sim V = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(E \sim E', m) \Downarrow b$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

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

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

Commands

Skip:
\[(\text{skip}, m) \Downarrow m\]

Assignment:
\[(E, m) \Downarrow V \quad (I := E, m) \Downarrow m[I \leftarrow V]\]

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

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

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

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

Example: If Then Else Rule

\[x \text{ > 5, } \{x \rightarrow 7\} \Downarrow ?\]

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

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

Example: Identifier(s)

7 > 5 = true
(x, (x -> 7)) \implies_7 (5, (x -> 7)) \implies_5
(x > 5, (x -> 7)) \implies_? (if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) \implies_? ?

Example: Arith Relation

7 > 5 = true
(x, (x -> 7)) \implies_7 (5, (x -> 7)) \implies_5
(x > 5, (x -> 7)) \implies_? (if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) \implies_? ?

Example: If Then Else Rule

7 > 5 = true
(x, (x -> 7)) \implies_7 (5, (x -> 7)) \implies_5
(x > 5, (x -> 7)) \implies_? (if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) \implies_? ?

Example: Assignment

7 > 5 = true
(x, (x -> 7)) \implies_7 (5, (x -> 7)) \implies_5
(x > 5, (x -> 7)) \implies_? (if x > 5 then y := 2 + 3 else y := 3 + 4 fi, {x -> 7}) \implies_? ?

Example: Arith Op

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

\[2 + 3 = 5\]
\[(2, \{x \mapsto 7\}) \downarrow 2\]
\[(3, \{x \mapsto 7\}) \downarrow 3\]
\[7 > 5 = \text{true}\]
\[(7, \{x \mapsto 7\}) \downarrow 5\]
\[7 > 5 = \text{true}\]
\[(2 + 3, \{x \mapsto 7\}) \downarrow 5\]
\[(y := 2 + 3, \{x \mapsto 7\}) \downarrow 5\]
\[(x > 5, \{x \mapsto 7\}) \downarrow \text{true}\]
\[(x > 5, \{x \mapsto 7\}) \downarrow \text{true}\]
\[(y := 2 + 3, \{x \mapsto 7\}) \downarrow 5\]
\[(x > 5, \{x \mapsto 7\}) \downarrow \text{true}\]
\[(y := 2 + 3, \{x \mapsto 7\}) \downarrow ?\]
Example

\[
\begin{align*}
(x, (x \mapsto 5)) & \Downarrow 5 & (3, (x \mapsto 5)) & \Downarrow 3 \\
(x + 3, (x \mapsto 5)) & \Downarrow 8 \\
(5, (x \mapsto 17)) & \Downarrow 5 & (x := x + 3, (x \mapsto 5)) & \Downarrow (x \mapsto 8) \\
(let \ x = 5 in (x := x + 3), (x \mapsto 17)) & \Downarrow \{x \mapsto 17\}
\end{align*}
\]

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

- \(\text{compute\_exp} \ (\text{Var}(v), m) = \text{look\_up} \ v \ m\)
- \(\text{compute\_exp} \ (\text{Int}(n), \_ ) = \text{Num} \ (n)\)
- ...
- \(\text{compute\_com}(\text{IfExp}(b, c_1, c_2), m) =\)
  - if \(\text{compute\_exp} \ (b, m) = \text{Bool}(\text{true})\)
  - then \(\text{compute\_com} \ (c_1, m)\)
  - else \(\text{compute\_com} \ (c_2, m)\)
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

Expression Semantics
- Form of operational semantics
- Describes how each program construct transforms machine state by transitions
- Rules look like
  \((C, m) \rightarrow (C', m')\) or \((C, m) \rightarrow m'\)  
  \(C, C'\) is code remaining to be executed
  \(m, m'\) represent the state/store/memory/environment
- Partial mapping from identifiers to values
- Sometimes \(m\) (or \(C\)) not needed
- Indicates exactly one step of computation

Expressions and Values
- \(C, C'\) used for commands; \(E, E'\) for expressions; \(U,V\) for values
- Special class of expressions designated as values
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
- Other possibilities exist

Evaluation Semantics
- Transitions successfully stops when \(E/C\) is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final meaning of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

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

Transitions for Expressions
- Numerals are values
- Boolean values = \{true, false\}
- Identifiers: \((I,m) \rightarrow (m(I), m)\)
Boolean Operations:

- Operators: (short-circuit)
  - \((\text{false} \& B, m) --> (\text{false}, m)\)
  - \((B, m) --> (B'', m)\)
  - \((\text{true} \& B, m) --> (B, m)\)
  - \((B \& B', m) --> (B'' \& B', m)\)
  - \((\text{false} \lor B, m) --> (B, m)\)
  - \((B \lor B', m) --> (B'' \lor B', m)\)
  - \((\text{true} \lor B, m) --> (B, m)\)
  - \((B, m) --> (B'', m)\)
  - \((\text{not true}, m) --> (\text{false}, m)\)
  - \((B, m) --> (\text{not } B', m)\)
  - \((\text{not false}, m) --> (\text{true}, m)\)
  - \((\text{not } B, m) --> (\text{not } B', m)\)

Relations

- \((E, m) --> (E'', m)\)
- \((E \sim E', m) --> (E'' \sim E', m)\)
- \((E, m) --> (E', m)\)
- \((V \sim E, m) --> (V \sim E', m)\)
- \((U \sim V, m) --> (\text{true}, m) \text{ or } (\text{false}, m)\) depending on whether \(U \sim V\) holds or not

Arithmetic Expressions

- \((E, m) --> (E'', m)\)
- \((E \circ E', m) --> (E'' \circ E', m)\)
- \((E, m) --> (E', m)\)
- \((V \circ E, m) --> (V \circ E', m)\)
- \((U \circ V, m) --> (N, m)\) where \(N\) is the specified value for \(U \circ V\)

Commands - in English

- \((\text{skip}, m) --> m\)
- \((E, m) --> (E', m)\)
- \((I := E, m) --> (I := E', m)\)
- \((I := V, m) --> m[I <-- V]\)
- \((C, m) --> (C'', m')\)
- \((C; C', m) --> (C''; C', m')\)
- \((C; C', m) --> (C'; m')\)

If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.
If Then Else Command

(if true then \( C \) else \( C' \) fi, \( m \)) \( \rightarrow \) (\( C, m \))

(if false then \( C \) else \( C' \) fi, \( m \)) \( \rightarrow \) (\( C', m \))

\( (B,m) \rightarrow (B',m) \)

(if \( B \) then \( C \) else \( C' \) fi, \( m \))  
\( \rightarrow \) (if \( B' \) then \( C \) else \( C' \) fi, \( m \))

While Command

(while \( B \) do \( C \) od, \( m \)) \( \rightarrow \)

(if \( B \) then \( C; \) while \( B \) do \( C \) od skip fi, \( m \))

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.