A clarification on terminology:

Recognizer: accepts or rejects strings in a language

Parser: recognizes and generates parse trees (imminent topic)

Assignment 3: building a recognizer for the Lake expression language

A general comment:

Many details in PL implementation (lots of space for ambiguity)

No one will suffer if they forget to dot an i

10 years from now:
Will you remember whether an octal has a leading \?
Will you remember what top-down vs. bottom-up parsing is?

How does parser recover from syntax error?

Three choices:

1. Insert a token
guess what token the user forgot

2. Delete a token
guess what extra token the user inserted

3. Replace a token
guess what token the user really meant

Can mix and match all 3 approaches
Common LL(1) strategy: skip all tokens not in follow set

```c
parseFact(Token nextToken)
{
    switch (nextToken)
    {
        case ID: break;
        case NUM: break;
        case LPAREN:
            parseExp(yylex());
            consume(RPAREN);
            break;
        default:
            error("Expected an ID, NUM or (\")
            scanTo(FOLLOW(Factor));
            break;
    }
}
```

Common strategy: insert token if looking for specific token

```c
parseFact(Token nextToken)
{
    switch (nextToken)
    {
        case ID: break;
        case NUM: break;
        case LPAREN:
            parseExp(yylex());
            consume_or_insert(RPAREN);
            break;
        default:
            error("Expected an ID, NUM or (\")
            scanTo(FOLLOW(Factor));
            break;
    }
}
```

Common strategy: substitute in special cases

```c
parseFact(Token nextToken)
{
    switch (nextToken)
    {
        case ID: break;
        case NUM: break;
        case LPAREN:
            parseExp(yylex());
            consume(RPAREN);
            break;
        case FOR: IF:
            error("Expected an ID")
            break;
        default:
            error("Expected an ID, NUM or (\")
            scanTo(FOLLOW(Factor));
            break;
    }
}
```
A warning on substitution strategy:

If original token was in the follow set (indicating omitted tokens),
insert strategy may be better

Substitution may introduce cascading errors (in next production)

Basically same approach to error recovery for LR(1)

Primary approach is skip tokens

Look for synchronizing tokens things like ); else

Selected insertion or substitution can be added

Java cup (and bison/yacc)

Compiler writer defines error recovery

Typically use error non-terminal

Pre-defined non terminal
Match only when error reported

typically error non-terminal

Example:

expr ::= LPAREN error RPAREN
When syntax error detected

Parser pops off states
    until state with error recovery found
Shifts on error non terminal
Discards tokens until synchronizing token found
Then resume normal parsing

\[
\text{factor} \rightarrow \text{( error )}
\]

\[
\text{+} \quad \text{factor} \quad \text{pop states}
\]

\[
\text{( a + + )}
\]

\[
\text{factor} \rightarrow \text{( error )}
\]

\[
\text{error} \quad \text{shift error}
\]

\[
\text{( a + + )}
\]
For other recovery strategies, add “wrong” productions:

To replace token
\[
\text{a}\_\text{expr} ::= \text{access}\ \text{ASSIGN}\ \text{expr}\ \text{SEMI};
\text{a}\_\text{expr} ::= \text{access}\ \text{EQ}\ \text{expr}\ \text{SEMI};
\]

To insert
\[
\text{access ::= access}\ \text{LBRACK}\ \text{expr}\ \text{RBRACK}
\text{access ::= access}\ \text{LBRACK}\ \text{RBRACK}
\]

Must add explicit messages for these

Strategies often fail to recognize “real” error consider:
\[
\text{fi (b)}
\begin{align*}
\text{x} & = 1; \\
\text{else} & \\
\text{x} & = 2;
\end{align*}
\]

Will report 2 errors:
\[
\text{missing ; after fi(b)}
\text{unexpected else}
\]

But: real problem is mis-spelled if

Some research in global error recovery:

Try various strategies in various combinations, including looking back

Compare results of different strategies, pick “best” solution (usually shortest fix)

Interesting theoretically, currently too inefficient in practice
So far we are working on a recognizer
Only silently accepts or reports an error message
Not very useful for compilation
Need to do work with each derivation step
Need to add parser actions

Every parser generator gives an option to perform actions on each reduction

java_cup is much like yacc or bison
Insert actions in the grammar as code

java_cup actions in {} :}

Cup actions anywhere in grammar
Unlike lex
Usually at the end

Any code can go in actions
Not usually return; would end parse!

expr ::= expr PLUS factor
{}: System.out.println("'x+f'"); ;}
printing “x+f” still not very useful
Want to get values from rhs elements

Can name and reference each piece
Use :id to name pieces
refer to as id in action

```java
expr ::= INT_CONST:l PLUS INT_CONST:r
{: System.out.println("sum = " + (l.intValue()+r.intValue())); ;}
```

Can use value of any rhs element
Token or non terminal

What is its type?

Must declare types when declaring terminals and non terminals
```java
terminal Integer INT_CONST;
non terminal Integer expr;
```

For tokens:
Value is value assigned during lexing
Third argument to LakeSym()

What is value of non terminal?
Set by each production rule
Assign to RESULT (implicitly labels lhs)
```java
expr ::= expr:l PLUS expr:r
{: RESULT =new Integer (l.intValue()+r.intValue()); ;}
```
Can now build working calculator:

term Integer NUMBER;
term PLUS, MINUS, TIMES, DIVIDE;
term LPAREN, RPAREN;

non terminal Integer expr, term, factor;
non terminal calculation;

calculation ::= expr : e
   { System.out.println(e); ; };

expr ::= expr PLUS term : r
   { RESULT = new Integer
     (l.intValue()+r.intValue()); ; };

expr ::= expr MINUS term : r
   { RESULT = new Integer
     (l.intValue()-r.intValue()); ; };

expr ::= term : t
   { RESULT = t; ; };

term ::= term TIMES factor : r
   { RESULT = new Integer
     (l.intValue()*r.intValue()); ; };

term ::= term DIVIDE factor : r
   { RESULT = new Integer
     (l.intValue()/r.intValue()); ; };

term ::= factor : f
   { RESULT = f; ; };

factor ::= NUMBER:n
   { RESULT = n; ; };

factor ::= LPAREN expr: e RPAREN
   { RESULT = e; ; };

This was example of an *interpreter*

Behavior of source language defined in terms of another language

*Compiler* transforms source code into machine language; no mediation

Most actions are at end of production they need not be
Sometimes embedded actions useful

```latex
block ::= LBRACE (: startBlock(); :) declarations statements RBRACE ({ : endBlock(); :});
```

startBlock will run before any declarations reduced

What if two rules for block?
```
block ::= LBRACE (: startBlock(); :) declarations statements RBRACE ({ : endBlock(); :});
block ::= LBRACE RBRACE;
```

Parser reads token past LBRACE before calling startBlock

Can cause unexpected behavior if action affects next token more problems in yacc/lex
Each action is assumed distinct on parse stack

```
block ::= LBRACE { : startBlock(); : }
declarations statements RBRACE
    { : endBlock(); : };
block ::= LBRACE { : startBlock(); : }
declarations RBRACE
    { : endBlock(); : };
```

Introduces conflict

don’t know which action until RBRACE

---

Fix with common rule

```
block ::= LBRACE startblock
declarations statements RBRACE
    { : endBlock(); : };
block ::= LBRACE startblock
declarations RBRACE
    { : endBlock(); : };
startblock ::= { : startBlock(); : };
```

---

Many parse trees thus far

Most of them:
- have tokens on leaves
- have non terminals on interior nodes

Reading the leaves gives the entire program
- as a sequence of tokens

Called a concrete parse tree
Assume the grammar

```plaintext
expr ::= expr PLUS mult_expr;
expr ::= expr MINUS mult_expr;
expr ::= mult_expr;
mult_expr ::= mult_expr TIMES factor;
mult_expr ::= mult_expr DIV factor;
mult_expr ::= factor;
factor ::= ID;
factor ::= NUMBER;
factor ::= LPAREN expr RPAREN;
```

What is concrete parse tree for

```
a * (b - c)
```
Concrete parse trees are messy
Include specifics of language
Include specifics of grammar

Want the semantics of program
Leave behind all the clutter

Called an abstract parse tree
Occasionally used abstract trees

Abstract parse tree for a * ( b - c):

Obviously much simpler
And much easier to work with

Abstracted away grammar
Don’t care about mult_expr

Abstracted away language
Don’t care infix/prefix/RPN/…
Compilers use abstract parse trees

Assume abstract parse trees now
Unless specified otherwise

But remember
ambiguity is defined on concrete parse trees:
If distinct concrete parse trees exist for same sentence, grammar ambiguous

Create parse trees with actions
Type of non terminals is ParseTree:

non terminal ParseTree expr,stmt;
stmt ::= expr SEMI
{ : RESULT = e ; : };
expr ::= exprl PLUS term:r
{ : RESULT = new BinParseTree(l,r,
BinParseTree.Plus) ; : };