My, what a lot of typos in assignment 3!
Your patience is appreciated

Will endeavor to not repeat this

But: typos, bugs will happen
Please contact me if something seems fishy
Will advise if updating; keep eye on email

Today:
Assignment 4 overview

Review of major topics so far
  Concepts
  Raise our heads out of syntactic details
Assignment 4:

Extend the Lake recognizer to accept *programs*- declarations and functions
Not *quite* full Lake language
Mostly limited in types

Will extend types in assignment 5

Functions defined with *statements*…

Support all basic control-flow statements:

*if*, *while*, *for*, *do-while*, *switch*  
(case, *default*)  
*continue*, *break*, *return*

Support *blocks*: { *statements* }

Support simple declarations:

*One* variable per declaration at this time
Simple base types
  *int*, *unsigned int*, *float*, …
No *struct*, *enum*, *union*
Simple variables / pointer variables only
Support functions:

ANSI-style functions only
   All parameters named
   No variable parameter lists

No function prototype declarations
   Just definitions

Will need delimited lists (as opposed to just sequences) in grammar

2 kinds:

Terminated elements
   stmt; stmt;

Separated elements
   param, param

Lists can be minimum 1 or 0
Both factors control structure

Semicolon terminated, possibly empty
   list1 ::= list1 elem SEMI;
   list1 ::= ;

Comma separated, non-empty
   list2 ::= list2 COMMA elem;
   list2 ::= elem;
### SEMI terminated, non-empty is similar

\[
\text{list3 ::= list3 elem SEMI;}
\]
\[
\text{list3 ::= elem SEMI;}
\]

### Last combo (comma separated, possibly empty) is harder

\[
\text{list4 ::= list4a;}
\]
\[
\text{list4 ::= ;}
\]
\[
\text{list4a ::= list4a COMMA elem ;}
\]
\[
\text{list4a ::= elem;}
\]

---

### NOTE: Intuitive specification of conditionals will produce two s/r conflicts with full grammar

\[
\text{if (b1) if (b2) f(x); else g(x);}
\]

Which if does else bind? (Dangling else problem)

\[
\text{stmt ::= SEMI;}
\]
\[
\text{stmt ::= expr SEMI;}
\]
\[
\text{stmt ::= IF LPAREN expr RPAREN stmt ;}
\]
\[
\text{stmt ::= IF LPAREN expr RPAREN stmt ELSE stmt;}
\]
\[
\text{expr ::= ID;}
\]

---

### Three options to solve the s/r:

**Use precedence**

Easy and reasonable

**Ignore it**

Java cup resolves in favor of shifting

shifting is the canonical solution

use -expect 2 option with java_cup

**Rewrite grammar**

By far messiest approach

Introduces right recursion
Need to add extra non terminals

```
stmt ::= matched_else;
stmt ::= unmatched_else;
other_stmt ::= expr SEMI;
other_stmt ::= SEMI;
expr ::= ID;
```

```
matched_else ::= IF LPAREN expr RPAREN matched_else ELSE matched_else;
matched_else ::= other_stmt;
unmatched_else ::= IF LPAREN expr RPAREN stmt;
unmatched_else ::= IF LPAREN expr RPAREN matched_else ELSE unmatched_else;
```

(Indirect) right recursion:

```
expr ::= term expr_tail;
expr_tail ::= ;
expr_tail ::= PLUS expr;
```

Necessary for LL(1) parsers
Problem for LR(1) parsers
complicates the actions
Consider:

\[ \begin{align*}
\text{lvalue} &::= \text{ID} \text{lval2}; \\
\text{lval2} &::= \text{LBRACK} \text{exp RBRACK lval2}; \\
\text{lval2} &::= \text{DOT ID lval2}; \\
\text{lval2} &::= ;
\end{align*} \]

(A grammar of struct and array access)

Easy way to express grammar

But what kind of value for lval2?

Whenever possible:

Every production should match to a complete, independent program piece

Simplifies definition of compilation procedures

Rules out right-recursive grammars

Previous grammar can be rewritten as

\[ \begin{align*}
\text{lval} &::= \text{ID}; \\
\text{lval} &::= \text{lval LBRACK exp RBRACK}; \\
\text{lval} &::= \text{lval DOT ID};
\end{align*} \]

Each production has clear value
So, what to do about two s/r conflicts related to conditionals?

Our advice: ignore them (use –expect 2)

Our solution will

cc reportedly has three s/r conflicts in its parser

For extra credit on assn 4
define error productions
The more the merrier, at least 20

Ideally:
Use productions with error
Use your own productions
Don’t forget to add call to Lake.error
And probably need to add error message
Don’t worry about LakePos for now

Topics so far:

Regexp
Finite automata
Lexers
Tokens
Jlex (automated lexer builders)
Grammars
Derivations
Parsers:
LL(1) (aka top-down aka predictive)
LR(k) (aka bottom-up aka shift-reduce)
Jcup (automated parser generators)
Error recovery
Parse Trees: concrete and abstract
Symbol tables
Regular expressions describe regular languages
A stands for itself
\epsilon represents empty
(can be given as nothing)
AB concatenation A then B
A/B alternative A or B
A+ one or more A
A* zero or more A
() grouping
. anything
[] any element inside
[^ ] anything not in [^ ]
\" escape special symbols

Lexers are implemented as finite automata (FA)
Edge for each letter in alphabet
Some states are accepting
Fact: Regexps describe regular languages, finite automata (FA) recognize regular langs.
Consider these regexps:
if
[0-9]+ [a-z][a-z0-9]* 
*[/[^\n]+]*

Implemented as this finite automata:
NFA: nondeterministic finite automata

DFA: deterministic finite automata

Fact: Any regexp can be automatically converted to an NFA

Fact: Any NFA can be converted to a DFA

DFAs more efficient

Grammars describe context-free languages

PL syntax is context-free

*Productions* map non-terminals to sequence of terminals and non-terminals

\[ e \rightarrow e + e \]

\[ e \rightarrow e - e \]

\[ \text{exp} \rightarrow n \]

Every grammar has a start symbol

A sentence is a sequence of non-terminals

A *sentential form* is a sequence of terminals and non-terminals that can occur in a legal derivation

A derivation is a sequence of derivation steps

Each derivation step replaces a single non-terminal \( s \) with sentential form \( \gamma \), if \( s \rightarrow \gamma \) is in grammar

A sentence is in a language if there exists a derivation of it, given grammar
Example:
\[
\begin{align*}
\text{e} & \rightarrow \text{e} + \text{e} \\
\text{e} & \rightarrow \text{e} - \text{e} \\
\text{e} & \rightarrow \text{e} + 2 \\
\text{e} & \rightarrow 3 - 2 \\
\text{e} & \rightarrow 1 + 3 - 2
\end{align*}
\]

Note: leftmost derivation

Rightmost derivation of same sentence:
\[
\begin{align*}
\text{e} & \rightarrow \text{e} - \text{e} \\
\text{e} & \rightarrow \text{e} + \text{e} \\
\text{e} & \rightarrow \text{e} - \text{e} \\
\text{e} & \rightarrow \text{e} + 2 \\
\text{e} & \rightarrow 1 + 3 - 2
\end{align*}
\]

Note: multiple derivations do not imply ambiguous grammar

Recognizers decide whether a sentence is in a language defined by grammar

**Recognizers** decide whether a sentence is in a language defined by grammar

**Recognizers** decide whether a sentence is in a language defined by grammar

**Recognizers** decide whether a sentence is in a language defined by grammar

Parsers are recognizers that additionally generate parse trees

Recognizers/parsers reconstruct derivations of input sentences

LL parsing: reads tokens left-to-right, reconstructs leftmost derivation top-down

LR parsing: reads tokens left-to-right, reconstructs rightmost derivation bottom-up

**Simplest parsing is LL(1)**

Places limitations on grammar; no left recursion

Remove by left factoring; the following:

\[
\begin{align*}
\text{S} : & (\text{S}) | \text{a} \\
\text{L} : & \text{L,S} | \text{S}
\end{align*}
\]

becomes:

\[
\begin{align*}
\text{S} : & (\text{S}) | \text{a} \\
\text{L} : & \text{S L2} \\
\text{L2} : & ,\text{S L2} |
\end{align*}
\]
LL(1) grammar parsed by recursive descent or table-driven parser
also called top-down parsing

Recursive descent defined with big switch statement
One function per non-terminal
Every case is token

Need FIRST set for each token

LL(1) table driven parsers more efficient, defined via FIRST and FOLLOW sets

Let A be non-terminal in a grammar

FIRST(γ) is set of terminals that can begin sentential forms derivable from γ

FOLLOW(A) is set of terminals that can immediately follow A in sentential forms

LR(1) grammars less restrictive

Parsed by shift-reduce parsers

Shift pushes input tokens onto stack
Reduce matches TOS to RHS of prod
Pops off match
Pushes on lhs of production
2 kinds of conflicts for s/r parsers

Shift/reduce conflicts:
- Can shift on another token, or reduce top of stack
- Cannot tell which from next token
  - Both may be valid

Reduce/reduce conflicts
- Can reduce two productions

Parse trees: generated during parse, represent grammatical structure

Concrete parse trees are faithful representations of derivations
- Parse trees always rooted with start symbol
- Read from left to right, leaves are all tokens
- Each parent/children subtree is one production

Abstract parse trees eliminates “junk”
- Preferable for PL compilation

expr ::= add_expr;

add_expr ::= add_expr PLUS mult_expr;
add_expr ::= mult_expr;

mult_expr ::= mult_expr STAR constant;
mult_expr ::= constant;

constant ::= INT_CONST;
1 + 2 + 3 * 4 is implicitly \((1 + 2) + (3 * 4)\):

If more than one concrete parse tree valid for any sentence, grammar is ambiguous.

Lack of definition of precedence or associativity introduces most ambiguity.

Precedence / associativity controlled by grammar or rules.

Precedence rules list operators in precedence order.
Each rule indicates associativity for left, right, non-associative.

Grammars chain for precedence.
Use left recursion for left associative.
Right recursion for right associative.
Symbol tables
Store information about identifiers, e.g.:
  Name
  Type
  Source code position
We will use them for static type checking
Implemented as hash tables, with opns.:
  Insert
  Lookup
  Extend (for entering new scope)
  Retract (for leaving scope)