CSE 401 - Compilers
Section 3

1/31/2013
12:30 - MEB 238
1:30 - EE 037

A CFG for Natural Languages

```
S   ::= NP VP                   // Sentence
NP  ::= DET NP | N | NP PP      // Noun Phrase
VP  ::= V NP | VP PP            // Verb Phrase
PP  ::= P NP                    // Prepositional Phrase

N   ::= boy | girl | house | UFO | radar | telescope
      | alien | government conspiracy            // Noun
V   ::= sees | meets | leaves | knows | greets   // Verb
DET ::= a | an | the              // Determiner
P   ::= to | for | with | in      // Preposition
```

"The girl sees the boy in the telescope."

More Natural Language Ambiguities

I cleaned the dishes in my pajamas.
I cleaned the dishes in the sink.

They cooked the beans in the pot on the stove with handles.
She knows you like the back of her hand.

Visiting relatives can be boring.

http://www.cs.washington.edu/education/courses/cse517/13wi/slides/cse517wi13-Parsing.pdf

Abstract Syntax Trees

A program, without the extra syntax for the parser (parenthesis, semicolons, ...)

Programs for people who don't care about parsing

Representation in Java:

- Simple classes represent AST nodes
- Inheritance used for Statements, Expressions, and Types

AST Representation in the Project

```
public abstract class Statement extends ASTNode { ... }
public abstract class Exp extends ASTNode { ... }

public class Assign extends Statement {
  public Identifier i;
  public Exp e;

  public Assign(Identifier ai, Exp ae, int ln) {
    super(ln);
    i=ai; e=ae;
  }
  ...
}
```

Project Part 2: Parser and Abstract Syntax

Due 2/13

- Construct parser via CUP that builds an AST
- No semantic analysis / type checking yet
- Build a pretty-print visitor
  - A comment-stripping auto-formatter
  - Mostly for practice implementing a visitor

Token Declarations in CUP

```
/* operators: */
terminal PLUS, BECOMES;

/* delimiters: */
terminal LPAREN, RPAREN, SEMICOLON;

/* tokens with values: */
terminal String IDENTIFIER;
```

Nonterminals in CUP

Declared similar to terminals:

```
nonterminal List<Statement> Program;
nonterminal Statement Statement;
...
```

Use provided classes AST classes in AST/

CUP Parsing Rules

```
AssignStatement ::=
  Identifier:id BECOMES Expression:expr SEMICOLON
  {: RESULT = new Assign(id, expr, idleft); :}
;
```

Semantic actions executed when rule reduced

`id` will refer to result of recursive parse

`idleft` is the line number of `id` (legacy magic)

Line Numbers in AST Nodes

```
10 int x;
11 x = "asdf";
```

Parse AST
Run type-checking visitor
Type error found - print a message:
- "attempt to assign string to int on line 11"
- Need to store line numbers in AST nodes

Precedence Declarations in CUP

Can specify precedence and associativity of operators:

```
   precedence left PLUS;
```

Best to read the CUP documentation for more details

Operating on ASTs

Object-Oriented approach:

- Each node knows how to do things:

```
public class While extends Statement {
   public typeCheck(...) { ... }
   public optimize(...) { ... }
   public generateX86(...) { ... }
   public prettyPrint(...) { ... }
}
```

Object-Oriented Approach to Operating on ASTs

Easy to add new kinds of nodes:

- Define a new class for the new node
- Extend `Statement`, `Expression`, or `ASTNode`
- Define abstract methods (`typeCheck`, `generateX86`, `prettyPrint`, etc.)
- All in one file
- But ASTs don't tend to change much over time

Harder to add new kinds of operations:

- Adding `generateARM`, new optimizations, etc.
- Have to touch a lot of files
- We want to add a lot of operations over time
- It would be nice if all of the code for one operation was in one place

Visitor Pattern

- Lets us put all of the code for one operation together in one class
  - Create a class extending Visitor
  - Implement methods defined in the Visitor interface

```
visit(If n) {...}
visit(While n) {...}
...
```

- Cleaner and more efficient than:

```
visit(ASTNode n) {
   if (n instanceof While) { ... }
   else if (n instanceof If) { ... }
   ...
```

Visitor Pattern

- To implement:
  - Every node has a method

    `accept(Visitor v) { v.visit(this); }`
    - First, usual dynamic dispatch on v
      - Which visitor class to select a method from
    - Second dispatch on type of this
      - Which method to call from that visitor class
- So inside a visitor method, calling `node.expr.accept (this)` does what you want
- Calling `this.visit(node.expr)` will dispatch on the compile-time (not run-time) type of node.expr, but `visit(Exp)` is not defined (compiler error)!

An Example TypeCheckVisitor

```
public class TypeCheckVisitor implements Visitor {
    public void visit(If n) {
        // visit the conditional expression first
        n.e.accept(this);
        assert(n.e.getType().equals(BOOL_TYPE));
        ...
    }
    ...
}
```

Flexible - visitors controls the descent down the AST

Modular - all of the code for one operation in one place

Wouldn't need the `accept` method if we had multimethods

Building an LR(0) Parser

Add start state to grammar
Build the DFA (Closure(S), Transition(I, X))
Build the LR(0) Parse Table (action and goto)

```
S ::= A$
A ::= aA
A ::= b
```

Building an SLR Parser

FIRST
FOLLOW
nullable

```
S ::= E$
E ::= 1E
E ::= 1
```

Questions?

Homework, Project, Lecture, ...