Plan for Today

Top-down Predictive Parsing

Logistics

- Office hours today 3-4.
- PA2a due Wednesday Feb 18th. Start working with groups ASAP.

Predictive Parsing

Predictive parsing, such as recursive descent parsing, creates the parse tree TOP DOWN, starting at the start symbol, and doing a LEFT-MOST derivation.

For each non-terminal N there is a function recognizing the strings that can be produced by N, with one (case) clause for each production. Consider:

```
start      -> stmts EOF
stmts      -> ε | stmt stmts
stmt       -> ifStmt |  whileStmt | ID = NUM
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```

can each production clause be uniquely identified by looking ahead one token? Let's predictively build the parse tree for

if t { while b { x = 6 }} $

Example Predictive Parser: Recursive Descent

```
start      -> stmts EOF
stmts      -> ε | stmt stmts
stmt       -> ifStmt |  whileStmt
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```

```
void start() { switch(m_lookahead) {
    case IF, WHILE, EOF: stmts(); match(Token.Tag.EOF); break;
    default:    throw new ParseException(…);
}}
void stmts() { switch(m_lookahead) {
    case IF,WHILE:  stmt(); stmts(); break;
    case EOF:       break;
    default:        throw new ParseException(…);
}}
void stmt() { switch(m_lookahead) {
    case IF:    ifStmt();break;
    case WHILE: whileStmt(); break;
    default:    throw new ParseException(…);
}}
void ifStmt() {switch(m_lookahead) {
    case IF: match(id); match(OPENBRACE);
             stmts(); match(CLOSEBRACE); break;
    default: throw new ParseException(…);
}}
```


Recursive Descent Parsing

Each non-terminal becomes a function
that mimics the RHSs of the productions associated with it
and choses a particular RHS:
an alternative based on a look-ahead symbol
and throws an exception if no alternative applies

When does this work?

First

Given a phrase γ of terminals and non-terminals (a rhs of a production), FIRST(γ) is the set of all terminals that can begin a string derived from γ.

FIRST(T*F) = ?
FIRST(F)= ?

FIRST(XYZ) = FIRST(X) ?

NO! X could produce ε and then FIRST(Y) comes into play

we must keep track of which non terminals are NULLABLE

FIRST example

```
start      -> stmts EOF
stmts      -> ε | stmt stmts
stmt       -> ifStmt |  whileStmt | ID = NUM
ifStmt     -> IF id { stmts }
whileStmt  -> WHILE id { stmts }
```


Follow

It also turns out to be useful to determine which terminals can directly follow a non terminal X (to decide parsing X is finished).

terminal t is in FOLLOW(X) if there is any derivation containing Xt.

This can occur if the derivation contains XYZt and Y and Z are nullable

FIRST and FOLLOW sets

NULLABLE
- X is a nonterminal
- nullable(X) is true if X can derive the empty string

FIRST
- FIRST(z) = {z}, where z is a terminal
- FIRST(X) = union of all FIRST(rhs_i), where X is a nonterminal and X -> rhs_i is a production
- FIRST(rhs_i) = union all of FIRST(sym) on rhs up to and including first nonnullable

FOLLOW(Y), only relevant when Y is a nonterminal
- look for Y in rhs of rules (lhs -> rhs) and union all FIRST sets for symbols after Y up to and including first nonnullable
- if all symbols after Y are nullable then also union in FOLLOW(lhs)

Constructive Definition of nullable, first and follow

for each terminal t, FIRST(t)={t}

Another Transitive Closure algorithm:
keep doing STEP until nothing changes

STEP:
for each production $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$
0: if Y_1 to Y_k nullable (or k = 0) nullable(X) = true
for each i from 1 to k, each j from i+1 to k
1: if $Y_1 \ldots Y_{i-1}$ nullable (or i=1) FIRST(X) += FIRST(Y_i) //+: union
2: if $Y_{i+1} \ldots Y_k$ nullable (or i=k) FOLLOW(Y_i) += FOLLOW(X)
3: if $Y_{i+1} \ldots Y_{j-1}$ nullable (or i+1=j) FOLLOW(Y_i) += FIRST(Y_j)

We can compute nullable, then FIRST, and then FOLLOW

Class Exercise

Compute nullable, FIRST and FOLLOW for

Z → d | X Y Z
X → a | Y
Y → c | ε

Constructing the Predictive Parser Table

A predictive parse table has a row for each non-terminal X, and a column for each input token t. Entries table[X,t] contain productions:

```
for each X -> gamma
    for each t in FIRST(gamma)
        table[X,t] = X->gamma
    if gamma is nullable
        for each t in FOLLOW(X)
            table[X,t] = X->gamma
```

Compute the predictive parse table for
Z → d | X Y Z
X → a | Y
Y → c | ε

	a	*c*	*d*
X	*X→a* *X→Y*	*X→Y*	*X→Y*
Y	*Y→ε*	*Y→ε* *Y→c*	*Y→ε*
Z	*Z→XYZ*	*Z→XYZ*	*Z→XYZ* *Z→d*