Plan for Today

Recall Predictive Parsing

- when it works and when it doesn't
- necessary to remove left-recursion
- might have to left-factor

Error recovery for predictive parsers

Predictive parsing as a specific subclass of recursive descent parsing

- complexity comparisons with general parsing

Studying for the midterm

Predictive Parsing

Predictive parsing, such as recursive descent parsing, creates the parse tree TOP DOWN, starting at the start symbol.

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

This workes great for the below grammar:

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

because each production could be uniquely identified by looking ahead one token. Let's predictively build the parse tree for

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

When Predictive Parsing works, when it does not

What about our expression grammar:

E → E + T | E-T | T
T → T * F | F
F → (E) | ID | NUM

The E method cannot decide looking one token ahead whether to predict E+T, E-T, or T. Same problem for T.

Predictive parsing works for grammars where the first terminal symbol of each sub expression provides enough information to decide which production to use.

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)= ?

E → E + T | E-T | T
*T → T * F | F*
F → (E) | ID | NUM

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

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
- 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 $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$
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

$Z \rightarrow d \mid X\ Y\ Z$
$X \rightarrow a \mid Y$
$Y \rightarrow c \mid \varepsilon$

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

Another Transitive Closure algorithm:
keep doing STEP until nothing changes

STEP:
for each production $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$
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

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*

Multiple entries in the Predictive parse table: Ambiguity

An ambiguous grammar will lead to multiple entries in the parse table.

Our grammar IS ambiguous, e.g. **Z → d**
but also **Z→XYZ→YZ→d**

For grammars with no multiple entries in the table, we can use the table to produce one parse tree for each valid sentence. We call these grammars LL(1): Left to right parse, Left-most derivation, 1 symbol lookahead.

A recursive descent parser examines input left to right. The order it expands non-terminals is leftmost first, and it looks ahead 1 token.

Left recursion and Predictive parsing

What happens to the recursive descent parser if we have a left recursive production rule, e.g. **E → E+T|T**

E calls E calls E forever

To eliminate left recursion we rewrite the grammar:

from:	**to:**
E → E + T | E-T | T	**E →T E'**
T → T * F | F	**E' → + T E' | - T E' | ε**
F → (E) | ID | NUM	**T → F T'**
	T' → * T E' | ε
	F → (E) | ID | NUM

replacing left recursion $X \rightarrow X\gamma \mid \alpha$ (where α does not start with X) by right recursion, as X produces $\alpha\,\gamma^*$ that can be produced right recursively. Now we can augment the grammar (S→E$), compute nullable, FIRST and FOLLOW, and produce an LL(1) predictive parse table, see Section 3.13 in Basics of Compiler Design.

Left Factoring

Left recursion does not work for predictive parsing. Neither does a grammar that has a non-terminal with two productions that start with a common phrase, so we left factor the grammar:

$$S \rightarrow \alpha\beta_1 \quad S \rightarrow \alpha\beta_2 \xrightarrow{\textit{Left refactor}} S \rightarrow \alpha S' \quad S' \rightarrow \beta_1 \mid \beta_2$$

E.g.: if statement:

S → IF t THEN S ELSE S | IF t THEN S | o

becomes

S → IF t THEN S X | o

X→ ELSE S | ε

When building the predictive parse table, there will still be a multiple entries. WHY?

Dangling else problem: ambiguity

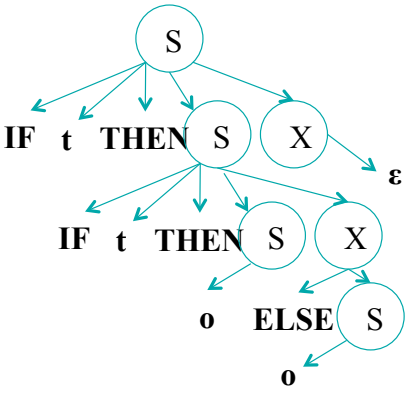
Given

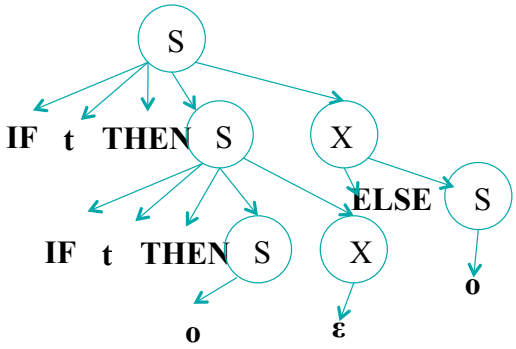
S → IF t THEN S X | o

X→ ELSE S | ε

construct two parse trees for

IF t THEN IF t THEN o ELSE o

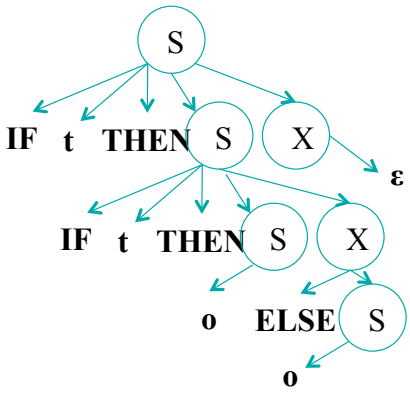




Which is the correct parse tree? (C, Java rules)

Dangling else disambiguation

The correct parse tree is:



We can get this parse tree by removing the X→ε rule in the multiple entry slot in the parse tree. See written homework 2.

General Error Recovery

Goals

- Provide program with a list of as many errors as possible
- Provide USEFUL error messages
  - appropriate line and position information
  - guidance for fixing the error
- Avoid infinite loops or recursion
- Add minimal overhead to the processing of correct programs

Approaches

- Stop after first error
  very simple, but unfriendly
- Panic mode
  skip tokens until a “synchronizing” token is encountere

Panic mode error recovery

The function for nonterminal X has one clause for each possible production rule for X. A clause includes a case for every character in the FIRST set for the rhs of the production, each character in the FOLLOW set if the rhs is nullable, and calls to match tokens and other nonterminals to process the rhs of the production.

For panic mode, skip tokens until a follow of the nonterminal encountered

```
// panic method for nonterminal N
panic_N( ) {
    print error;
    while ( scan() not in (FOLLOW(N) union {EOF}) ) {
    }
}
```

Example: simple assignment grammar

S → StmtList EOF
Stm → id ASSIGN float
StmList → Stm StmList | ε

What is nullable, FIRST,
FOLLOW for each nonterminal?

What is the predictive parser table?

Predictive parser with panic mode error recovery

```
// Float assignment grammar.
void S() { switch (m_lookahead) {
    case ID:
    case EOF:// the 2 characters in the FIRST(StmList EOF)
       try { StmList(); match(EOF); } catch { panic_S(); } break;
    default: panic_S(); break;
}}
void StmList() { switch (m_lookahead) {
    case ID:  // FIRST( Stm StmList ) = { ID }
        Stm(); StmList(); break;
    case EOF: // FOLLOW(StmList) = { EOF }
        break;
    default: panic_StmList(); break;
}}
void Stm() { switch (m_lookahead) {
    case ID:  try { match(ID); match(ASSIGN); match(FLOAT);
              } catch { panic_Stm(); } break;
    default: panic_Stm(); break;
}}
```

Predictive Parsing Complexity

LL(k) grammar classes

- Left-to-right scan
- Left-most derivation
- k tokens of lookahead

Comparing complexity

- $O(N^3)$ for general case context free grammars, where N is the number of tokens in the stream (Earley parsing algorithm)
- O(N) for predictive parsing

Requirements for LL(1), for all productions of nonterminal A

- None of the FIRST(rhs) for A production rules can overlap
- If nullable(A) then FOLLOW(A) must not overlap with FIRST(rhs) for any A→ rhs

Studying for the Midterm

Start Now

- Example midterms have been posted on the schedule.
- There is a set of concepts you are expected to know that have been posted on the Midterm in class link on the schedule.
- Thursday will be an interactive review session for the midterm. Example problems are already posted.

Practice Problems while studying!!

- The exam will be mostly multiple choice; however, you will have to work through problems by hand to select the correct answer.
- While doing examples, feel free to post examples and what answer you think it is on the midterm and final forum. Give each other feedback.