Plan for Today

Context Free Grammars

- Create derivation and parse tree for some examples.
- Use syntax-directed translation of parse tree to evaluate examples.

Top-down Predictive Parsing

Logistics

- Office hours Friday 3-4.
- PA2a due Wednesday Feb 18th. Partners posted on Canvas just now due to button pressing issue.

Derivation, Parse Tree, and Interpretation for Example

Grammar

Stm --> id := Exp
Exp --> num
Exp --> (Stm, Exp)

Parse Tree

String

x := (y := 42, 7)

How about here?

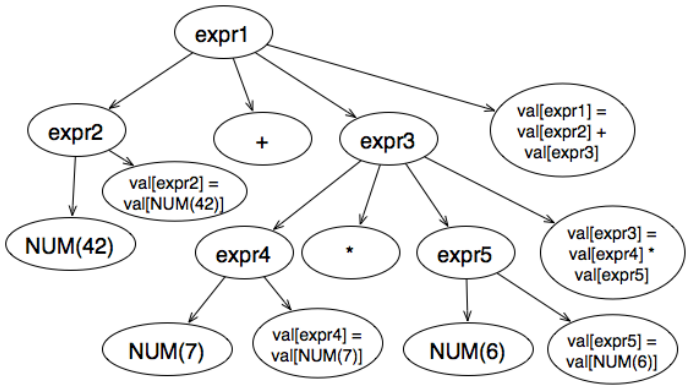
Grammar

(1) exp --> exp * exp
(2) exp --> exp + exp
(3) exp --> NUM

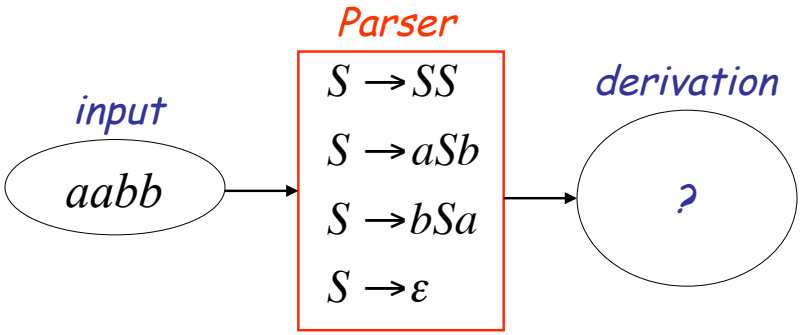
String

3 + 4 + 5

Recall Semantic Rules from Monday's Example



Example:



Exhaustive Search

$$S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon$$

Phase 1:

$S \Rightarrow SS$

$S \Rightarrow aSb$

$S \Rightarrow bSa$

$S \Rightarrow \varepsilon$

Find derivation of $aabb$

All possible derivations of length 1

$S \Rightarrow SS$

$S \Rightarrow aSb$

~~$S \Rightarrow bSa$~~

~~$S \Rightarrow \varepsilon$~~

$aabb$

Phase 2 $S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon$

Phase 1

$S \Rightarrow SS$

$S \Rightarrow SS \Rightarrow SSS$

$S \Rightarrow SS \Rightarrow aSbS$

$S \Rightarrow SS \Rightarrow bSaS$

$S \Rightarrow SS \Rightarrow S$

$S \Rightarrow aSb$

$S \Rightarrow aSb \Rightarrow aSSb$

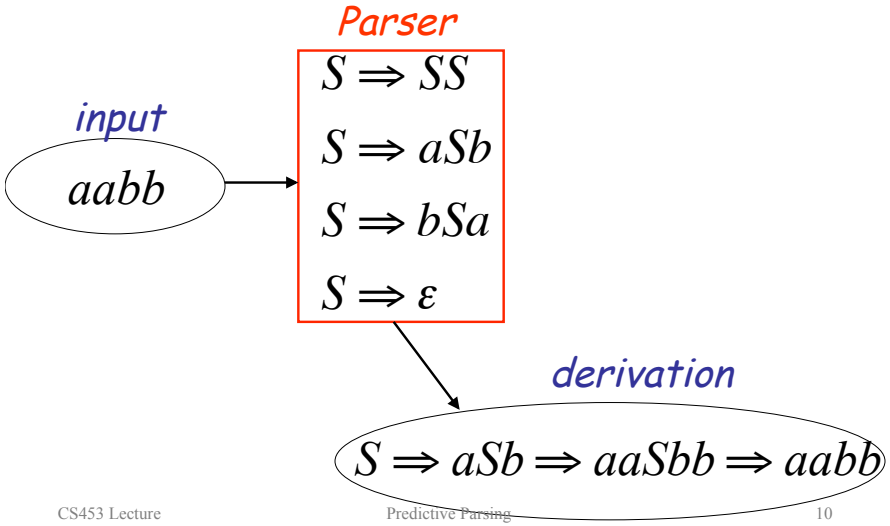
$S \Rightarrow aSb \Rightarrow aaSbb$

$S \Rightarrow aSb \Rightarrow abSab$

$S \Rightarrow aSb \Rightarrow ab$

$aabb$

Final result of exhaustive search (top-down parsing)



For general context-free grammars:

The exhaustive search approach is extremely costly: $O(|P|^{|w|})$

There exists a parsing algorithm that parses a string w in time $|w|^3$ for any CFG (Earley parser)

For LL(1) grammars, a simple type of CFGs that we will meet soon, we can use Predictive parsing and parse in time $|w|$

Context-Free Grammars

Grammar $G = (V, T, S, P)$

Nonterminals — Terminals — Start symbol

Productions of the form:

$A \rightarrow x$

Nonterminal — String of symbols, Nonterminals and terminals

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 → Y_1 Y_2 … 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*

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.

One more time

Balanced parentheses grammar 1:

S → (S) | SS | ε

1. **Augment the grammar with EOF/$**

2. **Construct Nullable, First and Follow**

3. **Build the predictive parse table, what happens?**

One more time, but this time with feeling …

Balanced parentheses grammar 2:

S → (S)S | ε

1. Augment the grammar with EOF/$

2. Construct Nullable, First and Follow

3. Build the predictive parse table

4. Using the predictive parse table, construct the parse tree for

( ) (( )) $

and

( ) ( ) ( ) $