Top-Down Parsing

CIS*4650 (Winter 2020)

Parsing Algorithms

- Top-down parsing:
  - Parse input tokens by tracing out steps in a leftmost derivation
  - Predictive parsers: e.g., recursive descent parsing, LL(k)
  - Backtracking parsers: more powerful but much slower and may run into infinite loops
  - Relatively easy to write by hand

- Bottom-up parsing:
  - Parse input tokens by performing reductions to uncover the parsing steps in a rightmost derivation
  - E.g., LR(k), LALR(k)
  - Most parser generators use bottom-up parsing

Recursive Descent Parsing

- Basic idea:
  - Construct a parse tree from root and create the nodes of the parse tree in the depth-first order
  - Non-terminals become recursive functions
  - Productions become clauses in the non-terminal function
  - Clauses implement explicit calls to consume the expected tokens
  - Parse errors are generated if the expected tokens can't be consumed

e.g., STMT -> if EXPR then STMT else STMT
| while EXPR do STMT
| begin STMT_LIST end

Recursive Descent Parsing

```
enum token_t { IF, THEN, ELSE, WHILE, DO, BEGIN, END };

extern enum token_t getNextToken();

enum token_t token;

void advance() { token = getNextToken(); }

void match(enum token_t candidate) {
   if( candidate == token )
       advance();
   else
       error();
}

void error(void) { fprintf(stderr, “Error\n”); }
```

Recursive Descent Parsing

```
void STMT(void) {
    switch( token ) {
      case IF:
          match( IF );
          EXPR();
          match( THEN );
          STMT();
          match( ELSE );
          STMT();
          break;
      case WHILE:
          match( WHILE );
          EXPR();
          match( DO );
          STMT();
          break;
     case BEGIN:
         match( BEGIN );  STMT_LIST();  match( END );
         break;
     default:
         error();
   }
}
```

```
STMT -> if EXPR then STMT else STMT
      | while EXPR do STMT
      | begin STMT_LIST end
```

Recursive Descent Parsing

- Handling choices:

<if-stmt> -> if (<exp>) <stmt>
| if (<exp>) <stmt> else <stmt>

```
void if-stmt( void ) {
  match( IF );
  match( LPAREN );
  exp();
  match( RPAREN );
  stmt();
  if( token == ELSE ) {
     match( ELSE );
     stmt();
  }
}
```

In EBNF:
<if-stmt> -> if (<exp>) <stmt> [else <stmt>]

Recursive Descent Parsing

- Handling repetitions:

  <exp> -> <exp> <addop> <term> | <term>
  <addop> -> + | -

In EBNF:

<exp> -> <term> { <addop> <term> }
<addop> -> + | -

```
void exp( void ) {
  term();
  while( token == PLUS ||
         token == MINUS ) {
     match( token );
     term();
  }
}
```

Recursive Descent Parsing

- Not always easy to convert grammar in BNF to that in EBNF
- May be difficult to decide which case to use if they all start with non-terminals
- Need to handle A -> ε productions
- Early detect errors if possible: e.g., ")3 - 2)"

LL(1) Parsing

- LL(1): Left-to-right parse, Leftmost derivation, and 1 symbol lookahead
- Use an explicit stack rather than recursive calls

e.g.: S -> (S) S | ε

Steps	Parsing stack	Input	Action
1	$ S	( ) $	S -> (S) S
2	$ S) S (	( ) $	match
3	$ S) S	) $	S -> ε
4	$ S)	) $	match
5	$ S	$	S -> ε
6	$	$	accept

LL(1) Parsing

- Generation action:
  - Replace a non-terminal A at the top of the stack by a string α using the production A -> α
  - α is pushed in reverse order: e.g., S -> (S) S becomes $ S) S (
- Match action:
  - Match a token on top of the stack with the next input token
- Generations correspond to the leftmost derivation:

S => (S) S	S -> (S) S
=> ( ) S	S -> ε
=> ( )	S -> ε

LL(1) Parsing

- LL(1) Parsing table: M[N,T] where N is a non-terminal and T is a token.

e.g.: S -> (S) S | ε

M[N,T]	(	)	$
S	S -> (S) S	S -> ε	S -> ε

- LL(1) grammar: if the associated LL(1) parsing table has at most one production in each table entry.

LL(1) Parsing

- A non-LL(1) example:

  <stmt> -> <if-stmt> | other
  <if-stmt> -> if (<exp>) <stmt> <else-part>
  <else-part> -> else <stmt> | ε
  <exp> -> 0 | 1

M[N,T]	if	other	else	0	1	$
<stmt>	<stmt>-> <if-stmt>	<stmt> -> other				
<if-stmt>	<if-stmt> -> if (<exp>) <stmt> <else-part>					
<else-part>			<else-part> -> else <stmt> <else-part> -> ε			<else-part> -> ε
<exp>				<exp> -> 0	<exp> -> 1	

Left Recursion Removal

- What's the problem with left recursion?

  $A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots \mid A\ \alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

  e.g., <exp> -> <exp> + <term> | <exp> - <term> | <term>

- Removing left recursion:

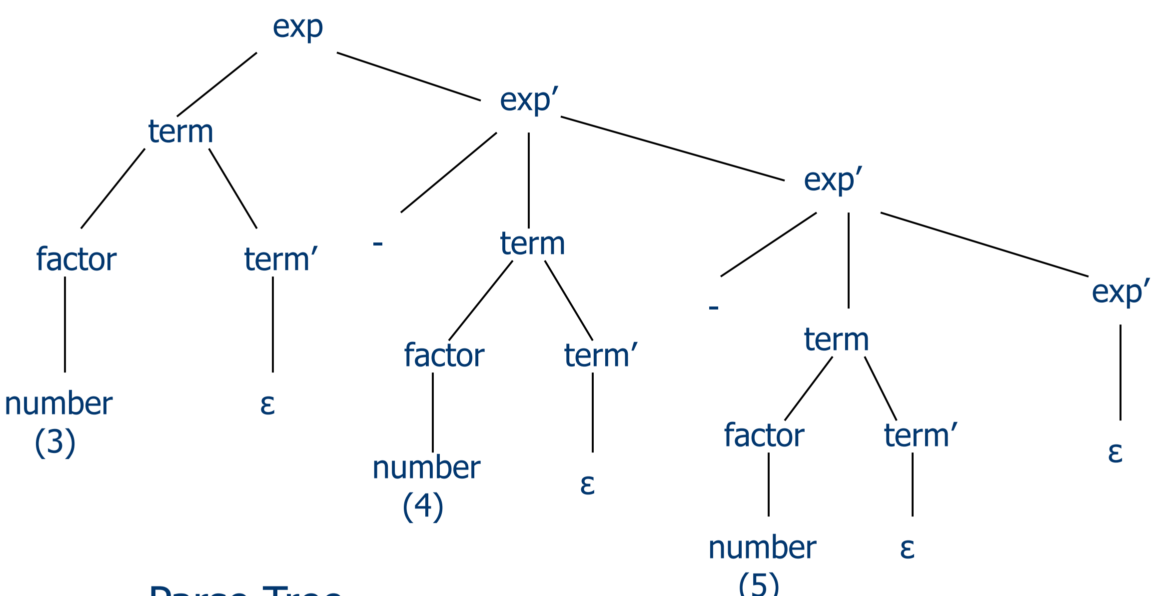
  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$
  $A' \rightarrow \alpha_1\ A' \mid \alpha_2\ A' \mid \ldots \mid \alpha_n\ A' \mid \varepsilon$

  e.g., <exp> -> <term> <exp'>
  <exp'> -> + <term> <exp'> | - <term> <exp'> | ε

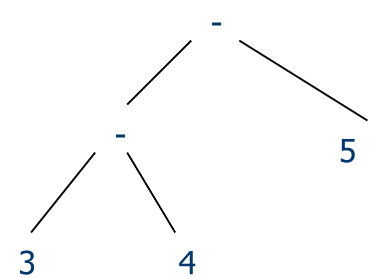
Left Recursion Removal

- How about left associativity?

<exp> -> <term> <exp'>
<exp'> -> + <term> <exp'> | - <term> <exp'> | ε
<term> -> <factor> <term'>
<term'> -> * <factor> <term'> | / <factor> <term'> | ε
<factor> -> (<exp>) | number | id



Parse Tree



Syntax Tree

Left Factoring

- What's the problem with a common prefix of symbols?

  A -> α β | α γ

  e.g., <stmts> -> <stmt> ; <stmts> | <stmt>
  e.g., <if-stmt> -> if (<exp>) <stmt>
  | if (<exp>) <stmt> else <stmt>

- Left factoring the common prefix:

  A -> α A′ A′ -> β | γ

  e.g., <stmts> -> <stmt> <stmts'>
  <stmts'> -> ; <stmts> | ε
  e.g., <if-stmt> -> if (<exp>) <stmt> <else-part>
  <else-part> -> else <stmt> | ε

Left Factoring

- Left factoring vs. left recursion removal:

  e.g., <exp> -> <term> + <exp> | <term>

  Left factoring:

  <exp> -> <term> <exp’>
  <exp’> -> + <exp> | ε

  Substituting <term> <exp’> for <exp>:

  <exp> -> <term> <exp’>
  <exp’> -> + <term> <exp’> | ε

- Both left factoring and left recursion removal may obscure the associativity

FIRST and FOLLOW Sets

- Notation
  - α --- string of terminals or non-terminals
  - A --- single terminal or non-terminal
- FIRST(α) --- set of terminals that can start any string derived from α
  - If $\alpha \Rightarrow^* \varepsilon$, α is nullable and $\varepsilon \in$ FIRST(α)
- FOLLOW(A) --- set of terminals that can immediately follow A in some phrase form

Computing FIRST Sets

for all non-terminals A **do** FIRST(A) = {}

for all terminals A **do** FIRST(A) = {A}

repeat
  for each production A -> X_1 X_2 … X_n **do**
    k = 1
    while k <= n **do**
      add FIRST(X_k) - {ε} to FIRST(A)
      if $\varepsilon \notin$ FIRST(X_k) **then** break;
      k = k + 1
    if k > n **then** add ε to FIRST(A)
until no changes to any FIRST(A)

Computing FIRST Sets

- Given the following grammar:

<exp> -> <exp> <addop> <term> | <term>
<addop> -> + | -
<term> -> <term> <mulop> <factor> | <factor>
<mulop> -> * | /
<factor> -> (<exp>) | number

- We get the following FIRST sets:

FIRST(exp) = { (, number }
FIRST(term) = { (, number }
FIRST(factor) = { (, number }
FIRST(addop) = { +, - }
FIRST(mulop) = { *, / }

Computing FIRST Sets

Production	Pass 1	Pass 2	Pass 3
<exp> -> <exp> <addop> <term>			
<exp> -> <term>			FIRST(exp) = {(, number}
<addop> -> +	FIRST(addop) = {+}		
<addop> -> -	FIRST(addop) = {+, -}		
<term> -> <term> <mulop> <factor>			
<term> -> <factor>		FIRST(term) = {(, number}	
<mulop> -> *	FIRST(mulop) = {*}		
<mulop> -> /	FIRST(mulop) = {*, /}		
<factor> -> (<exp>)	FIRST(factor) = {(}		
<factor> -> number	FIRST(factor) = {(, number}		

Computing FOLLOW Sets

FOLLOW(start-symbol) = {\$}

for all non-terminals A $\neq$ start-symbol **do** FOLLOW(A) = {}

repeat
   for each production A -> X_1 X_2 … X_n **do**
      for each X_i that is a non-terminal **do**
         add FIRST(X_{i+1} X_{i+2} … X_n) - {ε} to FOLLOW(X_i)
         /* note that if i = n, then X_{i+1} X_{i+2} … X_n = ε */
         if ε $\in$ FIRST(X_{i+1} X_{i+2} … X_n) **then** add FOLLOW(A) to FOLLOW(X_i)
until no changes to any FOLLOW(A)

Computing FOLLOW Sets

Production	Pass 1	Pass 2
<exp> -> <exp> <addop> <term>	FOLLOW(exp) = {$, +, - } FOLLOW(addop) = {(, number} FOLLOW(term) = {$, +, - }	FOLLOW(term) = {$, +, -, *, /,) }
<exp> -> <term>		
<term> -> <term> <mulop> <factor>	FOLLOW(term) = {$, +, -, *, / } FOLLOW(mulop) = {(, number} FOLLOW(factor) = {$, +, -, *, / }	FOLLOW(factor) = {$, +, -, *, /,) }
<term> -> <factor>		
<factor> -> (<exp>)	FOLLOW(exp) = {$, +, -,) }	

Another Example

Production	Pass 1	Pass 2
X -> a	FIRST(X) = {a}	
X -> Y		FIRST(X) = {ε, a, c}
Y -> ε	FIRST(Y) = {ε}	
Y -> c	FIRST(Y) = {ε, c}	
Z -> d	FIRST(Z) = {d}	
Z -> XYZ	FIRST(Z) = {a, d} FOLLOW(X) = {a, c, d} FOLLOW(Y) = {a, c, d}	FIRST(Z) = {a, c, d}

LL(1) Parsing Table

- Construction Rule:
  - for each a in FIRST(α), enter A -> α in M[A, a].
  - if ε is in FIRST(α), enter A -> α in M[A,f] for each f in FOLLOW(A).

	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 -> d Z -> XYZ

Error Recovery

- Report the first error and tell where the error has occurred
- Report error(s) and allow parsing to continue
  - deletion: skip tokens until reaching one in FIRST or FOLLOW sets
  - worst case: consume all input tokens
- Error repair
  - insert/replace/delete/switch tokens based on minimal distance
  - dangerous (possible infinite loop if repair generates another error)
  - expensive (usually limited to simple cases)

Error Recovery

- Panic mode for recursive descent parsers:
  - Skip input tokens in order to resume parsing
  - may consume all input tokens, but always terminate

```
void scanto( synchset ) {
   while( token ∉ synchset ∪ {$} )
      advance();
}

void checkinput( firstset, followset ) {
   if( token ∉ firstset ) {
      error();
      scanto( firstset ∪ followset );
   }
}
```

Error Recovery

- Panic mode (continued):

```
void exp( synchset ) {
   checkinput( { (, number }, synchset );
   if( token ∉ synchset ) {
      term( synchset );
      while( token == + || token == - ) {
         match( token );
         term( synchset );
      }
      checkinput( synchset, { (, number } );
   }
}
```

```
void factor( synchset ) {
   checkinput( { (, number }, synchset );
   if( token ∉ synchset ) {
      switch( token ) {
         case (:
            match( ( );
            exp( { ) } );
            match( ) );
            break;
         case number:
            match( number );
            break;
         default:  error();
      }
      checkinput( synchset, { (, number } );
   }
}
```

Error Recovery in LL(1)

M[N,T]	(	number	)	+	-	*	/	$
exp	<exp> -> <term> <exp’>	<exp> -> <term> <exp’>	pop	scan	scan	scan	scan	pop
exp’	scan	scan	<exp’> -> ε	<exp’> -> <addop> <term> <exp’>	<exp’> -> <addop> <term> <exp’>	scan	scan	<exp’>-> ε
addop	pop	pop	scan	<addop> -> +	<addop>-> -	scan	scan	pop
term	<term> -> <factor> <term’>	<term> -> <factor> <term’>	pop	pop	pop	scan	scan	pop
term’	scan	scan	<term’>-> ε	<term’> -> ε	<term’> -> ε	<term’>-> <mulop> <factor> <term’>	<term’>-> <mulop> <factor> <term’>	<term’>-> ε
mulop	pop	pop	scan	scan	scan	<mulop>-> *	<mulop>-> /	pop
factor	<factor> -> (<exp>)	<factor> -> number	pop	pop	pop	pop	pop	pop

Error Recovery in LL(1)

- Enter "pop":
  if token == $ or token in FOLLOW(A)
- Enter "scan":
  if token != $ and
    token not in FIRST(A) or FOLLOW(A)
- Enter normal entries:
  if token != $ and token in FIRST(A)