Syntax Analysis

- Convert the list of tokens into a parse tree ("hierarchical" analysis)

- The syntactic structure is specified using context-free grammars
  [in lexical analysis, the lexical structure is specified using regular expressions]

- A parse tree (also called concrete syntax) is a graphic representation of a derivation that shows the hierarchical structure of the language

- Other secondary tasks: syntax error detection and recovery

Tokens ---> Parse Tree

Tokens:
FUNCTION
ID(do_nothing1)
LPAREN
ID(a)
COLON
ID(int)
COMMA
ID(b)
COLONID(string)
RPAREN
EQ
ID(do_nothing2)
LPAREN
INT
PLUS
ID(a)
RPAREN
The parse tree captures the syntactic structure!

Main Problems

- How to specify the syntactic structure of a programming language?
  by using Context-Free Grammars (CFG)

- How to parse? i.e., given a CFG and a stream of tokens, how to build its parse tree?
  1. bottom-up parsing
  2. top-down parsing

- How to make sure that the parser generates a unique parse tree? (the ambiguity problem)

- Given a CFG, how to build its parser quickly?
  using YACC ---- the parser generator

- How to detect, report, and recover syntax errors?

Grammars

- A grammar is a precise, understandable specification of programming language syntax (but not semantics!)

- Grammar is normally specified using Backus-Naur Form (BNF) ---
  1. a set of rewriting rules (also called productions)

  \[
  \begin{align*}
  \text{stmt} & \rightarrow \text{if expr then stmt else stmt} \\
  \text{expr} & \rightarrow \text{expr + expr} \mid \text{expr * expr} \\
  & \mid (\text{expr}) \mid \text{id}
  \end{align*}
  \]

  2. a set of non-terminals and a set of terminals

  non-terminals ---- stmt, expr
  terminals ---- if, then, else, +, *, (, ), id

  3. lists are specified using recursion

  \[
  \begin{align*}
  \text{stmt} & \rightarrow \text{begin} \text{stmt-list} \text{end} \\
  \text{stmt-list} & \rightarrow \text{stmt} \mid \text{stmt-list}
  \end{align*}
  \]
Context-Free Grammars (CFG)

- A context-free grammar is defined by the following \( (T,N,P,S) \):
  - \( T \) is vocabulary of terminals,
  - \( N \) is set of non-terminals,
  - \( P \) is set of productions (rewriting rules), and
  - \( S \) is the start symbol (also belong to \( N \)).

- Example: a context-free grammar \( G=(T,N,P,S) \)
  - \( T = \{ +, *, (, ), \text{id} \} \)
  - \( N = \{ E \} \)
  - \( P = \{ E \rightarrow E + E, E \rightarrow E * E, E \rightarrow ( E ), E \rightarrow \text{id} \} \)
  - \( S = E \)

- Written in BNF: \( E \rightarrow E + E | E * E | ( E ) | \text{id} \)

- All regular expressions can also be described using CFG

Context-Free Languages (CFL)

- Each context-free grammar \( G=(T,N,P,S) \) defines a context-free language \( L = L(G) \)

- The CFL \( L(G) \) contains all sentences of terminal symbols (from \( T \)) --- derived by repeated application of productions in \( P \), beginning at the start symbol \( S \).

- Example the above CFG denotes the language \( L = \{ \text{id+id, id+(id*id), (id), id*id+id*id, ............} \)"

- Every regular language must also be a CFG! (the reverse is not true)

Derivations

- derivation is repeated application of productions to yield a sentence from the start symbol:
  - \( E \Rightarrow E + E \)
  - \( \Rightarrow \text{id} * E \)
  - \( \Rightarrow \text{id} * (E) \)
  - \( \Rightarrow \text{id} * (E + E) \)
  - \( \Rightarrow \text{id} * (id + id) \)

- the intermediate forms always contain some non-terminal symbols

- leftmost derivation: at each step, leftmost non-terminal is replaced; e.g. \( E \Rightarrow E * E \Rightarrow \text{id} * E \Rightarrow \text{id} * \text{id} \)

- rightmost derivation: at each step, rightmost non-terminal is replaced; e.g. \( E \Rightarrow E * E \Rightarrow E * \text{id} \Rightarrow \text{id} * \text{id} \)

Parse Tree

- A parse tree is a graphical representation of a derivation that shows hierarchical structure of the language, independent of derivation order.

- Parse trees have leaves labeled with terminals; interior nodes labeled with non-terminals.

  - example: \( E \Rightarrow* \text{id} * (id + id) \)

- Every parse tree has unique leftmost (or rightmost) derivation!
**Ambiguity**

- A language is **ambiguous** if a sentence has more than one parse tree, i.e., more than one leftmost (or rightmost) derivation

  **Example:** `id + id * id`

  - **Leftmost Derivation:**
    
    \[
    E \Rightarrow E + E \Rightarrow id + id + E \Rightarrow id + id + id
    \]

  - **Another Leftmost Derivation:**
    
    \[
    E \Rightarrow E + E \Rightarrow id + id + E \Rightarrow id + id + id
    \]

**Resolving Ambiguity**

- **Solution #1:** Using "disambiguating rules" such as precedence...
  - E.g., let `*` has higher priority over `+`

- **Solution #2:** Rewriting grammar to be unambiguous!
  - **Dangling-else**
    
    ```
    stmt -> if expr then stmt
    | if expr then stmt else stmt
    | .......
    ```

  - How to parse the following?
    
    ```
    if E1 then if E2 then S1 else S2
    ```

  - How to rewrite?
    
    * **Main Idea:** build "precedence" into grammar with extra non-terminals!
Other Grammar Transformations

- Elimination of Left Recursion (useful for top-down parsing only)
  replace productions of the form

  \[ A \to A \alpha_1 | \beta \]

  \[ A \to A \alpha_2 | \gamma \]

  with

  \[ A \to \beta A' \]

  \[ A' \to \alpha_1 A' | \epsilon \]

  (yields different parse trees but same language)

  see Appel pp 51-52 for the general algorithm

- Left Factoring --- find out the common prefixes (see Appel pp 53)
  change the production

  \[ A \to \alpha_1 | \alpha_2 \]

  to

  \[ A \to \alpha_1 A' \]

  \[ A' \to \alpha_2 | \beta \]

Parsing

- parser: a program that, given a sentence, reconstructs a derivation for that sentence ---- if done successfully, it “recognizes” the sentence

  - all parsers read their input left-to-right, but construct parse tree differently.

  - bottom-up parsers --- construct the tree from leaves to root

    shift-reduce, LR, SLR, LALR, operator precedence

  - top-down parsers --- construct the tree from root to leaves

    recursive descent, predictive parsing, LL(1)

  - parser generator --- given BNF for grammar, produce parser

    YACC --- a LALR(1) parser generator

Top-Down Parsing

- Construct parse tree by starting at the start symbol and “guessing” at derivation step. It often uses next input symbol to guide “guessing”.

  example: \[ S \to c \alpha d \]

  \[ \alpha \to a b | a \]

  input symbols: cad

  decide which rule of \( \alpha \) to use here?

  decide to use 1st alternative of \( \alpha \)

  guessed wrong backtrack, and try 2nd one.

  Main algorithms: recursive descent, predictive parsing (see the textbook for detail)

Bottom-Up Parsing

- Construct parse tree “bottom-up” --- from leaves to the root

  - Bottom-up parsing always constructs right-most derivation

  - Important parsing algorithms: shift-reduce, LR parsing, ...

  - shift-reduce parsing: given input string \( w \), “reduces” it to the start symbol! Main idea: look for substrings that match r.h.s of a production

Example:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>sentential form</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \to aA</td>
<td>aAbe</td>
<td>A \to b</td>
</tr>
<tr>
<td>S \to aAbe</td>
<td>aAbe</td>
<td>A \to Ab</td>
</tr>
<tr>
<td>A \to b</td>
<td>aAbe</td>
<td>B \to d</td>
</tr>
<tr>
<td>B \to d</td>
<td>aAbe</td>
<td>S \to aAbe</td>
</tr>
</tbody>
</table>
Handles

- **Handles** are substrings that can be replaced by l.h.s. of productions to lead to the start symbol.
- Not all possible replacements are handles --- some may not lead to the start symbol ... 
  abbcde → AAbcde → AAAbcde → stuck!
  this b is not a handle!
- Definition: if γ can be derived from S via right-most derivation, then γ is called a right-sentential form of the grammar G (with S as the start symbol). Similar definition for left-sentential form.
- handle of a right-sentential form γ = αΛω is A → β if
  
  \[ S \Rightarrow^* \alpha \Lambda \omega \Rightarrow^* \alpha \beta \omega \]
  and ω contains only terminals. E.g., A → Ab in aAbcde

Handle Pruning

- Main idea: start with terminal string w and “prune” handles by replacing them with l.h.s. of productions until we reach S:
  
  \[ S \Rightarrow^* \gamma_1 \Rightarrow^* \gamma_2 \Rightarrow^* \ldots \Rightarrow^* \gamma_n \Rightarrow^* \omega \]
  (i.e., construct the rightmost derivation in reverse)
- Example: 
  
  \[ E \rightarrow E + E | E \cdot E | ( E ) | a | b | c \]

  right-sentential form
  handle
  reducing production
  a
  E \rightarrow a
  b
  E \rightarrow b
  c
  E \rightarrow c
  E \cdot E
  E \rightarrow E \cdot E
  E + E
  E \rightarrow E + E

Key of Bottom-Up Parsing: Identifying Handles

Shift-Reduce Parsing

- Using a stack, shift input symbols onto the stack until a handle is found; reduce handle by replacing grammar symbols by l.h.s. of productions; accept for successful completion of parsing; error for syntax errors.
- Example: 
  
  \[ E \rightarrow E + E | E \cdot E | ( E ) | a | b | c \]

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a+b*c$</td>
<td>shift</td>
</tr>
<tr>
<td>$a</td>
<td>b*c$</td>
<td>shift</td>
</tr>
<tr>
<td>$E</td>
<td>b*c$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+b</td>
<td>c$</td>
<td>reduce: E → b</td>
</tr>
<tr>
<td>$E+c</td>
<td>$</td>
<td>reduce: E → $E</td>
</tr>
<tr>
<td>$E+E*</td>
<td>$</td>
<td>reduce: E → $E+E</td>
</tr>
<tr>
<td>$E+$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

  handle is always at the top!

Conflicts

- ambiguous grammars lead to parsing conflicts; conflicts can be fixed by rewriting the grammar, or making a decision during parsing
- shift/reduce (SR) conflicts: choose between reduce and shift actions
  
  \[ S \rightarrow \text{if } E \text{ then } S | \text{if } E \text{ then } S \text{ else } S | \ldots \]

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{if } E \text{ then } S</td>
<td>\text{else } \ldots$</td>
<td>reduce or shift?</td>
</tr>
</tbody>
</table>

- reduce/reduce (RR) conflicts: choose between two reductions
  
  stmt → id (param) --- procedure call a(l)
  param → id
  E → id (E) id --- array subscript a(l)

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{id(id )} \ldots$</td>
<td>reduce to E or param?</td>
<td></td>
</tr>
</tbody>
</table>
LR Parsing

today’s most commonly-used parsing techniques!

- **LR(k) parsing**: the “L” is for left-to-right scanning of the input; the “R” for constructing a rightmost derivation in reverse, and the “k” for the number of input symbols of lookahead used in making parsing decisions. (k=1)

- **LR parser** components: input, stack (strings of grammar symbols and states), driver routine, parsing tables.

LR Parsing Program

```
| s_m | X_m | ... | s_1 | X_1 | s_0 |
```

```
Parsing Table (action+goto)
```

LR Parsing Driver Routine

Given the configuration:

\( s_0 X_1 s_1 X_2 s_2 ... X_n s_n, a_1a_2a_3 ... a_nS \)

1. **If ACTION\[s_n, a_i\] is “shift s”, enter config**

   \( s_0 X_1 s_1 X_2 s_2 ... X_n s_n a_1 a_2a_3 ... a_n S \)

2. **If ACTION\[s_n, a_i\] is “reduce a\(\rightarrow\)β”, enter config**

   \( s_0 X_1 s_1 X_2 s_2 ... X_{n-r} s_{n-r} \beta S, a_1 a_2a_3 ... a_{n-r} S \)

   where \( r = |\beta|, \) and \( s = GOTO\[s_{n-r}, \beta] \)

   (here \( \beta \) should be \( X_{n-r}, X_{n-r-1}, ... X_0 \))

3. **If ACTION\[s_n, a_i\] is “accept”, parsing completes**

4. **If ACTION\[s_n, a_i\] is “error”, attempts error recovery.**

LR Parsing (cont’d)

- A sequence of new state symbols \( s_0, s_1, s_2, ..., s_m \) ----- each state summarizes the information contained in the stack below it.

- **Parsing configurations**: (stack, remaining input) written as

   \( (s_0 X_1 s_2 s_2 ... X_n s_m, a_1 a_2 a_3 ... a_n S) \)

   next “move” is determined by \( s_m \) and \( a_i \)

- **Parsing tables**: ACTION\[s, a\] and GOTO\[s, X\]

   **Table A**: ACTION\[s, a\] --- s : state, a : terminal

   its entries
   - (1) shift \( s_k \)
   - (2) reduce \( A \rightarrow \beta \)
   - (3) accept
   - (4) error

   **Table G**: GOTO\[s, X\] --- s : state, X : non-terminal

   its entries are states

Example: LR Parsing

- **Grammar**:

  1. \( S \rightarrow S ; S \)
  2. \( S \rightarrow id := E \)
  3. \( S \rightarrow print (L) \)
  4. \( E \rightarrow id \)
  5. \( E \rightarrow num \)
  6. \( E \rightarrow E + E \)
  7. \( E \rightarrow (S, E) \)
  8. \( L \rightarrow E \)
  9. \( L \rightarrow L , E \)

- **Tables**:

  sn -- shift and put state n on the stack

  gn -- go to state n

  rk -- reduce by rule k

  a -- accept by rule k

  _ -- error

- Details see figure 3.18 and 3.19 in Appel pp.56-57
Summary: LR Parsing

- **LR Parsing is doing reverse right-most derivation !!!**
- If a grammar is ambiguous, some entries in its parsing table (ACTION) contain multiple actions: "shift-reduce" or "reduce-reduce" conflicts.
- **Two ways to resolve conflicts** ---- (1) rewrite the grammar to be unambiguous
  (2) making a decision in the parsing table (retaining only one action!)
- **LR(k) parsing:** parsing moves determined by state and next k input symbols; k = 0, 1 are most common.
- A grammar is an **LR(k) grammar,** if each entry in its LR(k)-parsing table is uniquely defined.
- How to build **LR parsing table?** ---- three famous varieties: SLR, LR(1), LALR(1)  (detailed algorithms will be taught later !)

Yacc

- **Yacc** is a program generator -------- it takes grammar specification as input, and produces an **LALR(1) parser** written in C.
- **Implementation of Yacc:**
  - Construct the LALR(1) parser table from the grammar specification

ML-Yacc

- **ML-Yacc** is like Yacc -------- it takes grammar specification as input, and produces a **LALR(1) parser** written in Standard ML.
- **Implementation of ML-Yacc** is similar to implementation of Yacc

ML-Yacc Specification

```ml
structure A = Absyn

grammar

program : exp ()
exp : id ()
```

Yacc declarations

```ml
%%
%% %term EOF | ID of string ...
%% %nonterm exp | program ...
%% %pos int
%% %eop EOF
%% %noshift EOF
%%
%% grammar (action)
%% program : exp ()
%% exp : id ()
```

user's ML declarations

- **grammar** is specified as BNF production rules; **action** is a piece of ML program; when a grammar production rule is reduced during the parsing process, the corresponding **action** is executed.
ML-Yacc Rules

- **BNF production** \( A \rightarrow \alpha | \beta | \ldots | \gamma \) is written as:

  \[
  A : \alpha
  | \beta
  | \ldots
  | \gamma
  \]

  (action for \( A \rightarrow \alpha \))

  (action for \( A \rightarrow \beta \))

  (action for \( A \rightarrow \gamma \))

- The **start symbol** is l.h.s. of the first production or symbol \( S \) in the Yacc declaration.

  \%

  \%start S

- The **terminals or tokens** are defined by the Yacc declaration:

  \%

  \%term ID of string | NUM of int | PLUS | EOF | ...

- The **non-terminals** are defined by the Yacc declaration:

  \%

  \%nonterm EXP of int | START of int

Example: calc.grm

```plaintext
fun lookup "bogus" = 10000 | lookup s = 0
%
%eop EOF SEMI
%pos int
%left SUB PLUS
%left TIMES DIV
%term ID of string | NUM of int | PLUS | TIMES | PRINT | SEMI | EOF | DIV | SUB
%nonterm EXP of int | START of int
%verbose
%name Calc
%
%%

START : PRINT EXP (print EXP; print "\n"; EXP)
  | EXP (EXP)

EXP : NUM (NUM)
  | ID (lookup ID)
  | EXP PLUS EXP (EXP1+EXP2)
  | EXP TIMES EXP (EXP1*EXP2)
  | EXP DIV EXP (EXP1 \( div \) EXP2)
  | EXP SUB EXP (EXP1-EXP2)

Yacc : Conflicts

- Yacc uses the LR parsing (i.e. LALR); if the grammar is ambiguous, the resulting parser table **ACTION** will contain shift-reduce or reduce-reduce conflicts.

- In Yacc, you resolve conflicts by (1) rewriting the grammar to be unambiguous (2) declaring precedence and associativity for terminals and rules.

- Consider the following grammar and input:

  \[
  E : E PLUS E \ (\epsilon) \\
  | E TIMES E \ (\epsilon) \\
  | ID \ (\epsilon)
  \]

  we can specify **TIMES** has higher precedence than **PLUS**; and also assume both **TIMES** and **PLUS** are left associative.

  (also read the examples on Appel pp73-74)

Precedence and Associativity

- To resolve conflicts in Yacc, you can define **precedence** and **associativity** for each terminal. The precedence of each grammar rule is the precedence of its rightmost terminal in r.h.s of the rule.

- **On shift / reduce conflict:**

  ```plaintext
  if input terminal prec. > rule prec. then shift
  if input terminalprec. < rule prec. then reduce
  if input terminal prec. == rule prec. then:
    if terminal assoc. == left then reduce
    if terminal assoc. == right then shift
    if terminal assoc. == none then report error
  ```

- **On reduce / reduce conflict:** report error & rule listed first is chosen.
Defining Prec. and Assoc.

- Defining precedence and associativity for terminals
  \%
  \%left OR
  \%left AND
  \%noassoc EQ NEQ GT LT GE LE
  \%left PLUS MINUS
  \%left TIMES DIV

- Defining precedence for rules using %prec
  %left OR
  %left AND
  %noassoc EQ NEQ GT LT GE LE
  %left PLUS MINUS
  %left TIMES DIV
  %left UNARYMINUS
  %prec UNARYMINUS

   %prec UNARYMINUS

   Only specifies the prec.
   of this rule == prec. of UNARY-

   %prec UNARYMINUS

   Must define UNARYMINUS
   as a new terminal?

   Assuming unary minus has
   higher precedence than
   PLUS,

   exp : Exp MINUS Exp
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%
   \%