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

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)

  ```
  stmt -> if expr then stmt else stmt
  expr -> expr + expr | expr * expr
        | ( expr ) | id
  ```

  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

  ```
  stmt_list -> begin stmt_list end
  stmt_list -> stmt | stmt_list
  ```

Tokens --> Parse Tree

Tokens:

```markdown
FUNCTION ID (do_nothing1) LPAREN ID(a) COLON ID(int) COMMA ID(b) COLON ID(string) RPAREN EQ ID(do_nothing2) LPAREN INT(1) PLUS ID(a) RPAREN
```

The parse tree captures the syntactic structure!

```
ParseTree: fundec
  \[\begin{array}{c}
  \text{FUNCTION} \\
  \text{LPAREN} \\
  \text{ID} (\text{do_nothing1}) \\
  \text{LPAREN} \\
  \text{ID} (\text{a}) \\
  \text{COLON} \\
  \text{ID} (\text{int}) \\
  \text{COMMA} \\
  \text{ID} (\text{b}) \\
  \text{COLON} \\
  \text{ID} (\text{string}) \\
  \text{RPAREN} \\
  \text{EQ} \\
  \text{ID} (\text{do_nothing2}) \\
  \text{LPAREN} \\
  \text{INT} (1) \\
  \text{PLUS} \\
  \text{ID} (\text{a}) \\
  \text{RPAREN}
\end{array}\]
```

```
function do_nothing1(a:int,b:string) = do_nothing2(1+a)
```
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 = { +, *, (, ), id },
  - N = { E },
  - P = { E -> E + E, E -> E * E, E -> ( E ), E -> id },
  - S = E

- Written in BNF:
  - E -> E + E | E * E | ( E ) | 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 =
  - L({ +, *, (, ), id }, { E }, { E -> E + E, E -> E * E, E -> ( E ), E -> id }, E)
  - It contains sentences such as 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 => E + E --- "E derives E + E"
  - => id * E --- "E derives id"
  - => id * (E) --- "E derives (E)"

- the intermediate forms always contain some non-terminal symbols

- leftmost derivation: at each step, leftmost non-terminal is replaced;
  - e.g. E => E * E => id * E => id * id

- rightmost derivation: at each step, rightmost non-terminal is replaced;
  - e.g. E => E * E => E * id => id * 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.

- 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: \( id + id \times id \)

a)  \[
E \Rightarrow E + E \\
  \Rightarrow id + E + E \\
  \Rightarrow id + id + E \\
  \Rightarrow id + id + id
\]

b)  \[
E \Rightarrow E \times E \\
  \Rightarrow id \times E + E \\
  \Rightarrow id + id \times E \\
  \Rightarrow id + id + id
\]

Another leftmost derivation:

- Solution #1: using “disambiguating rules” such as precedence...
  e.g. let \( * \) has higher priority over +
  (favor derivation (a))

- Solution #2: rewriting grammar to be unambiguous!
  “dangling-else”
  \[
  stmt \rightarrow if \ expr \ then \ stmt \\
  | \ if \ expr \ then \ stmt \ else \ stmt \\
  | \ . . . .
  \]

How to parse the following?

if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)

How to rewrite?

Main Idea: build “precedence” into grammar with extra non-terminals!

Resolving Ambiguity (cont’d)

- solution: define “matched” and “unmatched” statements

\[
\begin{align*}
\text{stmt} & \rightarrow \text{m-stmt} | \text{um-stmt} \\
\text{m-stmt} & \rightarrow \text{if expr then m-stmt else m-stmt} \\
\text{um-stmt} & \rightarrow \text{if expr then stmt} \\
& \quad | \text{if expr then m-stmt else um-stmt}
\end{align*}
\]

Now how to parse the following?

if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)

Resolving Ambiguity (cont’d)

- Another ambiguous grammar

\[
\begin{align*}
E & \rightarrow E + E | E - E | E \times E | E / E \\
& \quad | ( E ) | - E | \text{id}
\end{align*}
\]

usual precedence: highest
  \( - \) (unary minus) \\
  \( * \) / \\
  lowest \\
  \( + \) -

- Build grammar from highest --> lowest precedence

\[
\begin{align*}
\text{element} & \rightarrow \{ \text{expr} \} | \text{id} \\
\text{primary} & \rightarrow - \text{primary} | \text{element} \\
\text{term} & \rightarrow \text{term} \cdot \text{primary} | \text{term} / \text{primary} \\
\text{expr} & \rightarrow \text{expr} + \text{term} | \text{expr} - \text{term} \\
\text{try the leftmost derivation for} & \quad - id + id \cdot id \\
\text{expr} & \Rightarrow \text{expr} + \text{term} \Rightarrow \text{term} + \text{term} \\
& \quad \Rightarrow \text{term} + \text{primary} \Rightarrow \text{element} + \text{term} \Rightarrow - id + \text{term} \\
& \quad \Rightarrow - id + \text{term} \cdot \text{primary} \Rightarrow \ldots \Rightarrow - id + id \cdot id
\end{align*}
\]
Other Grammar Transformations

- **Elimination of Left Recursion** (useful for top-down parsing only)
  replace productions of the form
  \[ A \rightarrow A \alpha | y \]
  with
  \[ A \rightarrow y A' \]
  \[ A' \rightarrow x A' | \epsilon \]
  (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 to
  \[ A \rightarrow \alpha y | \alpha z \]
  \[ A \rightarrow \alpha y | \alpha z \]

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: 
  \[ S \rightarrow c A d \]
  \[ A \rightarrow ab | a \]
  input symbols: 
  \[ cad \]
  \[ S \rightarrow c \]
  \[ A \rightarrow ab | a \]
  \[ S \rightarrow a \]
  \[ A \rightarrow ab | a \]
  decide which rule of \( A \) to use here?
  decide to use 1st alternative of \( A \)
  guessed wrong backtrack, and try 2nd one.

Bottom-Up Parsing

- Construct parse tree “bottom-up” --- from leaves to the root
- **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 \rightarrow aAcBe )</td>
<td>( abcd )</td>
<td>( A \rightarrow b )</td>
</tr>
<tr>
<td>( A \rightarrow Ab</td>
<td>b )</td>
<td>( aAcBe )</td>
</tr>
<tr>
<td>( B \rightarrow d )</td>
<td>( aAcBe )</td>
<td>( B \rightarrow d )</td>
</tr>
<tr>
<td>( S \rightarrow aAcBe )</td>
<td>( S \rightarrow aAcBe )</td>
<td></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 → aAAcde → stuck!
- **Definition**: if $\gamma$ can be derived from $S$ via right-most derivation, then $\gamma$ 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 $\gamma = A\omega$ is $A \rightarrow \beta$ if $S \Rightarrow^* A\omega = \Rightarrow^* A\beta \omega$ and $\omega$ contains only terminals. E.g., $A \rightarrow AB$ in abbcde.

Handle Pruning

- **Main idea**: start with terminal string $w$ and “prune” handles by replacing them with l.h.s. of productions until we reach $S$:
  $$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \omega$$
  (i.e., construct the rightmost derivation in reverse)
- **Example**:
  $$E \rightarrow E + E | E * E | (E) | a | b | c$$
  
  right-sentential form | handle | reducing production
  -----------------------|--------|-----------------------
  $a + b * c$ | $a$ | $E \rightarrow a$
  $b$ | $b$ | $E \rightarrow b$
  $E + E * c$ | $c$ | $E \rightarrow c$
  $E + E * E$ | $E * E$ | $E \rightarrow E * E$
  $E + E$ | $E + E$ | $E \rightarrow E + E$

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**:
  $$E \rightarrow E + E | E * E | (E) | a | b | c$$

<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>
</tbody>
</table>
| $E_a$ | $b * c$ | reduce: $E \rightarrow a$
| $E_b$ | $b * c$ | shift |
| $E_c$ | $c$ | reduce: $E \rightarrow b$
| $E_d$ | $c$ | shift (reduction in conflict)
| $E_e$ | $c$ | reduce: $E \rightarrow c$
| $E_f$ | $c$ | reduce: $E \rightarrow E * c$
| $E_g$ | $c$ | reduce: $E \rightarrow E * E$
| $E_h$ | $c$ | accept |

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>$$</td>
<td>$\text{if } E \text{ then } S \text{ else ...}$</td>
<td>reduce or shift?</td>
</tr>
</tbody>
</table>

- **reduce/reduce (RR) conflicts** : choose between two reductions
  $$\text{stmt} \rightarrow \text{id (param)}$$  --- procedure call $a(id)$
  $$\text{param} \rightarrow \text{id}$$  --- array subscript $a(id)$

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

```
sm Xm ...
s1 X1 s0
```

**Parsing Table (action+goto)**

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>s0</td>
<td>X0</td>
<td>sn</td>
</tr>
<tr>
<td>a2</td>
<td>s1</td>
<td>X1</td>
<td>gn</td>
</tr>
<tr>
<td>a3</td>
<td>s2</td>
<td>X2</td>
<td>rk</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>an</td>
<td>sm</td>
<td>Xm</td>
<td>ak</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td>ak</td>
</tr>
</tbody>
</table>

**LR Parsing Driver Routine**

**Given the configuration:**

\( (s_0X_1s_1X_2s_2...X_ms_m, a_{i+1}a_{i+2}...a_n\$) \)

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

\( (s_0X_1s_1X_2s_2...X_ms_m, a_{i+1}a_{i+2}...a_n\$) \)

2. **If** ACTION\([s_n, a_i]\) **is** “reduce A -> \(\beta\)”, **enter config**

\( (s_0X_1s_1X_2s_2...X_m-1s_m-1A, a_{i+1}a_{i+2}...a_n\$) \)

where \(r=|\beta|\), and \(s = GOTO\([s_{m-1}, A]\)\)

(Here \(\beta\) should be \(X_{m-1}X_{m-2}...X_0\))

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 \(s_0, s_1, s_2, ..., s_n\) ---- each state summarizes the information contained in the stack below it.

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

\( (s_0X_1s_1X_2s_2...X_ms_m, a_{i+1}a_{i+2}...a_n\$) \)

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

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

**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 and parsing completes
- \(e\) ---- 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.

ML-Yacc

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

ML-Yacc Specification

- 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
  
  \[
  A : \alpha \quad \text{(action for } A \rightarrow \alpha) \\
  | \beta \quad \text{(action for } A \rightarrow \beta) \\
  | \ldots \\
  | \gamma \quad \text{(action for } A \rightarrow \gamma) 
  \]

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

- The terminals or tokens are defined by the Yacc declaration
  
  \%

- The non-terminals are defined by the Yacc declaration
  
  \%

**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 | EOF | ...
%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 `ID PLUS ID PLUS ID`

```
E : E PLUS E ()
| E TIMES E ()
| ID ()
```

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:

```
if input terminal prec. > rule prec. then shift
if input terminal prec. < rule prec. then reduce
if input terminal prec. == rule prec. then if
  if terminal assoc. == left then reduce
  if terminal assoc. == right then shift
  if terminal assoc. == none then report error
else
  if the input terminal or the rule has no prec. then shift & 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 PLUS MINUS %left TIMES DIV
  %left UNARYMINUS

  Must define UNARYMINUS

  Assuming unary minus has higher precedence than

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

Parser Description (.desc file)

• The Yacc declaration %verbose will produce a verbose description of the generated parser (i.e., the "desc" file)
  
  1. A summary of errors found while generating the parser
  2. A detailed description of all errors
  3. The parsing engine --- describing the states and the parser table
     (see Example 3.1 on pp15-18 in Appel’s book)

Parser States:

<table>
<thead>
<tr>
<th>State</th>
<th>Program</th>
<th>Current States</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>exp</td>
<td>lvalue goto 1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parser Table:

<table>
<thead>
<tr>
<th>State</th>
<th>Program goto 115</th>
</tr>
</thead>
<tbody>
<tr>
<td>135</td>
<td>table GOTO</td>
</tr>
</tbody>
</table>

Parser Actions:

<table>
<thead>
<tr>
<th>State</th>
<th>Exp</th>
<th>GoTo 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>exp</td>
<td>goto 2</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>error</td>
</tr>
</tbody>
</table>

Connecting Yacc and Lex

signature PARSE = sig val parse : string -> unit end

structure Parse = PARSE =

structure TigerP = Join(structure ParserData = TigerLrVals.ParserData structure Lex = ToyLexFun)