Syntax Analysis

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

source program → lexical analyzer → parser → parse tree → abstract syntax

• 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 → begin stmt-list end
     stmt-list → stmt | stmt-list

Tokens --> Parse Tree

Tokens:

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!
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 $G=(T,N,P,S)$
  
  $T = \{ +, *, (, ), id \}$,
  $N = \{ E \}$,
  $P = \{ E \rightarrow E + E, E \rightarrow E * E, E \rightarrow ( E ), E \rightarrow id \}$,
  $S = E$

- Written in BNF: $E \rightarrow E + E | E * E | ( E ) | id$

- 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 $L = L(\{ +, *, (, ), id \}, \{ E \}, \{ E \rightarrow E + E, E \rightarrow E * E, E \rightarrow ( E ), E \rightarrow id \})$

  - 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:

  $E \rightarrow E * E$  --- "$E$ derives $E * E$"
  $E \rightarrow id * E$  --- "$E$ derives $id * E$"
  $E \rightarrow id * (E)$  --- "$E$ derives $id * (E)$"
  $E \rightarrow id * (id + id)$  --- "$E$ derives $id * (id + id)$"

- the intermediate forms always contain some non-terminal symbols

- leftmost derivation: at each step, leftmost non-terminal is replaced; e.g. $E \rightarrow E * E \rightarrow id * E \rightarrow id * id$

- rightmost derivation: at each step, rightmost non-terminal is replaced; e.g. $E \rightarrow E * E \rightarrow E * id \rightarrow id * id$

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.

  example: $E \Rightarrow* id * (id + id)$

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

(a) 

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

(b) 

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

Resolving Ambiguity

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" 

\[
\text{stmt} \rightarrow \text{if expr then stmt} \\
\rightarrow \text{if expr then stmt else stmt} \\
\rightarrow \ldots....
\]

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

\[
E \rightarrow E + E | E - E | E \times E | E / E \\
| ( E ) | - E | id
\]

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

Build grammar from highest \( \rightarrow\) lowest precedence

\[
\begin{align*}
\text{element} & \rightarrow ( \text{expr} ) | \text{id} \\
\text{primary} & \rightarrow - \text{primary} | \text{element} \\
\text{term} & \rightarrow \text{term} \times \text{primary} | \text{term} / \text{primary} | \text{primary} \\
\text{expr} & \rightarrow \text{expr} + \text{term} | \text{expr} - \text{term} | \text{term}
\end{align*}
\]

Try the leftmost derivation for \( - id + id \times id \)

\[
\begin{align*}
\text{expr} & \Rightarrow \text{expr} + \text{term} \Rightarrow \text{term} + \text{expr} \Rightarrow \text{term} + \text{primary} + \text{term} \\
& \Rightarrow \text{element} + \text{term} \Rightarrow - \text{primary} + \text{term} \\
& \Rightarrow \text{term} + \text{primary} \Rightarrow \ldots \Rightarrow - id + id \times id
\end{align*}
\]
Other Grammar Transformations

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

  (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
  
  \[ A \rightarrow x \ y \ | \ x \ z \]
  \[ A \rightarrow x \ A' \]
  \[ A' \rightarrow y \ | \ z \]

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

  parse tree:
  
  S → S → A → A → c A d → ab A → ab → ab a

  decide which rule of A to use here?

  decide to use 1st alternative of A

  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 → aAcd</td>
<td>abcd</td>
<td>A → b</td>
</tr>
<tr>
<td>A → Ab</td>
<td>aAcde</td>
<td>A → Ab</td>
</tr>
<tr>
<td>B → d</td>
<td>aAcBe</td>
<td>B → d</td>
</tr>
<tr>
<td>S → S → aAcBe</td>
<td></td>
<td>S → S → aAcBe</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. \[ abbcde \rightarrow aAbcde \rightarrow aAAcde \rightarrow \text{stuck!} \]
  
  - **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 \( \gamma = \alpha \beta \omega \) is \( A \rightarrow \beta \) if \( S \Rightarrow^* \alpha \Lambda \omega \Rightarrow^* \alpha \beta \omega \) and \( \omega \) contains only terminals. E.g., \( A \rightarrow Ab \) 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 \( S \):
  
  \[
  S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \omega
  \]
  
  (i.e., construct the rightmost derivation in reverse)

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

  Handle Pruning:

  - **right-sentential form**  \( a + b * c \)
  - **handle** \( E \rightarrow a \)
  - **reducing production** \( E \rightarrow a \)

  - **ambiguity** \( a + b * c \)
  - **handle** \( b \rightarrow b \)
  - **reducing production** \( E \rightarrow b \)

  - **ambiguity** \( E + E * E \)
  - **handle** \( c \rightarrow c \)
  - **reducing production** \( E \rightarrow c \)

  - **ambiguity** \( E + E + E \)
  - **handle** \( E \rightarrow E + E \)
  - **reducing production** \( E \rightarrow E + E \)

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 $E \rightarrow a$</td>
</tr>
<tr>
<td>$E$</td>
<td>$b*c$</td>
<td>shift $E \rightarrow b$</td>
</tr>
<tr>
<td>$E+$</td>
<td>$b*c$</td>
<td>shift $E \rightarrow b$</td>
</tr>
<tr>
<td>$E+$</td>
<td>$c$</td>
<td>reduce $E \rightarrow E+c$</td>
</tr>
<tr>
<td>$E+$</td>
<td>$c$</td>
<td>reduce $E \rightarrow E+c$</td>
</tr>
<tr>
<td>$E$</td>
<td>$c$</td>
<td>accept $E$</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

  - **Example**: \( S \rightarrow \text{if E then S else $\ldots$} \)

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

- **reduce/reduce (RR) conflicts**: choose between two reductions

  - **Example**: \( \text{stmt} \rightarrow \text{id (param)} \quad \text{--- procedure call} \quad \text{a(i)} \)

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

  - **Example**: \( \text{stmt} \rightarrow \text{id (param)} \quad \text{--- array subscript} \quad \text{a(i)} \)

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{id(id ...)}$</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

**Parsing Table (action+goto)**

<table>
<thead>
<tr>
<th>s</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>...</th>
<th>an</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>an</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LR Parsing Driver Routine

Given the configuration: 
\( (s_0X_1s_2X_3s_4...s_n, a_1a_1+1a_1+2...a_nS) \)

1. If \( \text{ACTION}[s,n, a] \) is "shift s", enter config
   \( (s_0X_1s_2X_3s_4...s_n a_1a_1+1a_1+2...a_nS) \)

2. If \( \text{ACTION}[s,n, a] \) is "reduce \( A \rightarrow \beta \)", enter config
   \( (s_0X_1s_2X_3s_4...s_{n-r}a\beta s_{n-r}A\beta, a_1a_1+1a_1+2...a_nS) \)
   where \( r=|\beta|, \) and \( s = \text{GOTO}[s_{n-r}, A] \)
   (here \( \beta \) should be \( X_{n-r}X_{n-r+1}...X_n \))

3. If \( \text{ACTION}[s,n, a] \) is "accept", parsing completes

4. If \( \text{ACTION}[s,n, a] \) 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 
\( (s_0X_1s_2s_3...X_ms_m, a_1a_1+1a_1+2...a_nS) \)
next “move” is determined by \( s_m \) and \( a_1 \)

• Parsing tables: \( \text{ACTION}[s, a] \) and \( \text{GOTO}[s, X] \)

Table A \( \text{ACTION}[s, a] \) --- \( s : \text{state}, a : \text{terminal} \)
its entries are (1) shift \( s \)k (2) reduce \( A \rightarrow \beta \) (3) accept (4) error

Table G \( \text{GOTO}[s, X] \) --- \( s : \text{state}, X : \text{non-terminal} \)
its entries are states

Example: LR Parsing

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

• 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
...

%%
%term EOF | ID of string ...
%nonterm exp | program ...
%pos int
%eop EOF
%noshift EOF
...

%% grammar (action)
program : exp ()
exp : id ()
```

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

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

- The terminals or tokens are defined by the Yacc declaration \%term
  
  \%term ID of string | NUM of int | PLUS | EOF | ...

- The non-terminals are defined by the Yacc declaration \%nonterm
  
  \%nonterm EXP of int | START of int

Example: calc.grm

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
%nonterm EXP of int | START of int
%verbose
%name Calc

%%

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

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:
    
    \[
    \begin{align*}
    \text{if input terminal prec.} & > \text{rule prec. then shift} \\
    \text{if input terminal prec.} & < \text{rule prec. then reduce} \\
    \text{if input terminal prec.} & = \text{rule prec. then} \\
    \end{align*}
    \]

    \[
    \begin{align*}
    \text{if terminal assoc.} & = \text{left then reduce} \\
    \text{if terminal assoc.} & = \text{right then shift} \\
    \text{if terminal assoc.} & = \text{none then report error} \\
    \end{align*}
    \]

  - On reduce / reduce conflict: report error & rule listed first is chosen

Yacc : Conflicts

- Yacc uses the LR parsing (i.e. LALR); if the grammar is ambiguous, the resulting parser table \texttt{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 \texttt{ID PLUS ID PLUS ID}

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

  we can specify \texttt{TATIES} has higher precedence than \texttt{PLUS}; and also assume both \texttt{TATIES} and \texttt{PLUS} are left associative.

  (also read the examples on Appel pp73-74)
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

  %prec UNARYMINUS

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

Tiger.Lex File “mumbo-jumbo”

You have to modify your “tiger.lex” file in assignment 2 by adding the following --- in order to generate the functor “TigerLexFun”

```plaintext
type svalue = Tokens.svalue
type pos = int
type ('a, 'b) token = ('a, 'b) Tokens.token
type lexresult = (svalue, pos) token

signature PARSE = sig val parse : string -> unit end
structure Parse : PARSE =
  struct
    structure TigerLrVals = TigerLrValsFun(structure Token = LrParser.Token)
    structure TigerP = Join(structure ParserData = TigerLrVals.ParserData
                              structure Lex = ToyLexFun(structure Tokens = TigerLrVals.Tokens)
                              structure LrParser = LrParser)
  end
fun parse filename =
  let
    val val = (ErrorMsg.reset(); ErrorMsg.fileName := filename)
    val file = open_in filename
    fun parseerror(s,p1,p2) = ErrorMsg.error p1 s
    val lexer = LrParser.Stream.streamify
      (Lex.makeLexer (fn _ => TextIO.input file))
    val (absyn, _) = TigerP.parse(30,lexer,parseerror, ()
      in close_in file
    absyn
    end
  handle LrParser.ParseError => raise ErrorMsg.Error
end
```