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?

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!

function do_nothing1(a:int,b:string) = do_nothing2(1+a)

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)
     \[
     \begin{align*}
     \text{stat} & \rightarrow \text{if expr then stat else stat} \\
     \text{expr} & \rightarrow \text{expr} + \text{expr} | \text{expr} * \text{expr} \\
     \text{id} & \rightarrow \text{id} \\
     \end{align*}
     \]
  2. a set of non-terminals and a set of terminals
     \[
     \begin{align*}
     \text{non-terminals} & \rightarrow \text{stat} \text{, expr} \\
     \text{terminals} & \rightarrow \text{id, +, *, (, ), if, then, else} \\
     \end{align*}
     \]
  3. lists are specified using recursion
     \[
     \begin{align*}
     \text{stat-list} & \rightarrow \text{begin stat-list end} \\
     \text{stat-list} & \rightarrow \text{stat | stat-list} \\
     \end{align*}
     \]
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 \( G=(T,N,P,S) \)
  \[
  T = \{ +, *, (, ), id \}, \\
  N = \{ E \}, \\
  P = \{ E \rightarrow E + E, E \rightarrow E \ast E, E \rightarrow (E), E \rightarrow id \}, \\
  S = E
  \]

• Written in BNF: \( E \rightarrow E + E \mid E \ast E \mid (E) \mid id \)

• 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 = \{ 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:
  \[
  E \Rightarrow E \ast E \\
  \Rightarrow id + E \\
  \Rightarrow id \ast (E) \\
  \Rightarrow id \ast (E + E) \\
  \Rightarrow id \ast (id + E) \\
  \Rightarrow id \ast (id + id)
  \]

• the intermediate forms always contain some non-terminal symbols

• leftmost derivation: at each step, leftmost non-terminal is replaced; e.g. \( E \Rightarrow E \ast E \Rightarrow id \ast E \Rightarrow id \ast id \)

• rightmost derivation: at each step, rightmost non-terminal is replaced; e.g. \( E \Rightarrow E \ast E \Rightarrow E \ast id \Rightarrow E \ast id \)

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 * id

Another leftmost derivation:

E => E + E => id + id * id
  => id + id * 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"

  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
  \[ A \rightarrow A \cdot \text{a} | \text{b} \]
  \[ A \rightarrow \text{b} A' \]
  \[ A' \rightarrow \text{a} A' | \epsilon \]
  (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
  \[ A \rightarrow \text{a} \cdot \text{b} | \text{c} \cdot \text{d} \]
  to
  \[ A \rightarrow \text{a} A' \]
  \[ A' \rightarrow \text{b} | \text{d} \]

Parsing

- parser : a program that, given a sentence, reconstructs a derivation for that sentence ---- if done successfully, it "recognize" 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: \[ S \rightarrow \text{c} \cdot \text{A} \cdot \text{d} \]
  \[ A \rightarrow \text{a} \cdot \text{b} | \text{a} \]
  \[ \text{input symbols: cad} \]
  decide which rule of \[ A \] to use here?
  decide to use 1st alternative of \[ A \]
  guessed wrong backtrack, and try 2nd one.
  \[ \text{Main algorithms : recursive descent, predictive parsing (details will be taught in the future)} \]

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 \( w \), "reduce" 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 \text{a} \text{A} \text{c} \text{B} \text{e} )</td>
<td>( \text{a} \text{b} \text{c} \text{d} \text{e} )</td>
<td>( S \rightarrow \text{b} )</td>
</tr>
<tr>
<td>( A \rightarrow \text{a} \text{B} \cdot \text{c} \cdot \text{d} )</td>
<td>( \text{a} \text{B} \text{c} \text{d} )</td>
<td>( A \rightarrow \text{Ab} )</td>
</tr>
<tr>
<td>( B \rightarrow \text{d} )</td>
<td>( \text{a} \text{B} \text{C} \text{d} )</td>
<td>( B \rightarrow \text{d} )</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.
  \[ \ldots abbcde \rightarrow abbcde \rightarrow aAbcde \rightarrow \text{stuck!} \]

- **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 \( \gamma \) is \( A \rightarrow \beta \) if \( S \Rightarrow^* \gamma \Rightarrow^* \alpha \beta \omega \) 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 \( S \):

  \[ S \Rightarrow^* \gamma_1 \Rightarrow^* \gamma_2 \Rightarrow^* \ldots \Rightarrow^* \gamma_{n-1} \Rightarrow^* \gamma_n \Rightarrow^* \omega \]

  (i.e., construct the rightmost derivation in reverse)

- **Example**:

  \[
  E \rightarrow E + E | E * E | ( E ) | a | b | c
  \]

  **right-sentential form**

  \[
  \begin{array}{ccc}
  \text{ambiguity} & \text{handle} & \text{reducing production} \\
  \text{a + b * c} & a & E \rightarrow a \\
  \text{E + b * c} & b & E \rightarrow b \\
  \text{E + E * c} & c & E \rightarrow c \\
  \text{E + E * E} & E * E & E \rightarrow E * E \\
  \text{E + E} & E + E & E \rightarrow E + E \\
  \end{array}
  \]

  **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**:

  \[
  E \rightarrow E + E | E * E | ( E ) | a | b | c
  \]

  **stack**

  \[
  \begin{array}{l}
  S \\
  \$ \\
  \$a \\
  \$E \\
  \$E+b \\
  \$E+c \\
  \$E+E \\
  \$E+E+c \\
  \$E+E*E \\
  \$E+E*E+c \\
  \$E
  \end{array}
  \]

  **input**

  \[
  \begin{array}{l}
  a+b+c$ \\
  +b+c$ \\
  +b+c$ \\
  b+c$ \\
  b+c$ \\
  +E$ \\
  +E$ \\
  +E$ \\
  +E$ \\
  +E$
  \end{array}
  \]

  **action**

  \[
  \begin{array}{l}
  \text{shift} \\
  \text{shift} \\
  \text{shift} \\
  \text{reduce: } E \rightarrow b \\
  \text{reduce: } E \rightarrow E*E \\
  \text{reduce: } E \rightarrow E*E \\
  \text{reduce: } E \rightarrow E*E \\
  \text{reduce: } E \rightarrow E*E \\
  \text{accept}
  \end{array}
  \]

  **reduce or shift?**

  **error for syntax errors**

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

    \[
    S \rightarrow \text{if } E \text{ then } S | \text{if } E \text{ then } S \text{ else } S | \ldots \\
    \]

    **stack**

    \[
    \begin{array}{l}
    S \\
    \$ \text{if } E \text{ then } S \text{ else } \ldots \$
    \end{array}
    \]

    **input**

    \[
    \begin{array}{l}
    \text{if } E \text{ then } S \text{ else } \ldots \$
    \end{array}
    \]

    **action**

    \[
    \begin{array}{l}
    \text{reduce or shift?} \\
    \text{error for syntax errors}
    \end{array}
    \]

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

    **stmt**

    \[
    \begin{array}{l}
    \text{id (param)} \\
    \text{param } \rightarrow \text{id} \\
    \text{E } \rightarrow \text{id (E) } | \text{id}
    \end{array}
    \]

    **reduce or shift**

    **error for syntax errors**

    **reduce to $E$ or param ?**
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 Driver Routine

Given the configuration:
\[(s_0, s_1, s_2, \ldots, s_n, a_1 a_2 a_3 \ldots a_n)\]

1. If ACTION[s_m, a_i] is “shift s”, enter config
   \[(s_0, s_1, s_2, \ldots, s_n, a_1 a_2 a_3 \ldots a_n)\]

2. If ACTION[s_m, a_i] is “reduce A->\beta”, enter config
   \[(s_0, s_1, s_2, \ldots, s_n, a_1 a_2 a_3 \ldots a_n)\]
   where r=|\beta|, and s = GOTO[\beta, A]
   (here \beta should be X_m X_{m+1} X_{m+2} \ldots X_n)

3. If ACTION[s_m, a_i] is “accept”, parsing completes

4. If ACTION[s_m, a_i] is “error”, attempts error recovery.

---

Example: LR Parsing

Grammar:

1. \[S \rightarrow S ; S\]
2. \[S \rightarrow \text{id} := E\]
3. \[S \rightarrow \text{print} (L)\]
4. \[E \rightarrow \text{id}\]
5. \[E \rightarrow \text{num}\]
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
- g n -- go to state n
- rk -- reduce by rule k
- a -- accept and parsing completes
- _ -- 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

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

- Implementation of ML-Yacc is similar to implementation of Yacc

Implementation of Yacc:

Construct the LALR(1) parser table from the grammar specification
ML-Yacc Rules

• BNF production \( A \rightarrow \alpha | \beta | ... | \gamma \) is written as
  \[ A : \alpha \text{ (action for } A \rightarrow \alpha) \]
  \[ | \beta \text{ (action for } A \rightarrow \beta) \]
  \[ ... \]
  \[ | \gamma \text{ (action for } A \rightarrow \gamma) \]

• 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

```yacc
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 ";"); EXP

| EXP : NUM (NUM)
| 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 \textit{ACTION} will contain \textit{shift-reduce} or \textit{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 \textit{TIMES} has higher precedence than \textit{PLUS}; and also assume both \textit{TIMES} and \textit{PLUS} are left associative.
  (also read the examples on Appel pp73–74)

Precedence and Associativity

• To resolve conflicts in Yacc, you can define \textit{precedence} and \textit{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 \textit{shift} / \textit{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 terminal assoc. == left then reduce
    if terminal assoc. == right then shift
    if terminal assoc. == none then report error
  
  if the input terminal or the rule has no prec. then shift & report error
  ```

• On \textit{reduce} / \textit{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 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

- Must define UNARYMINUS as a new terminal! (Assuming unary minus has higher precedence than PLUS)

Exp : Exp MINUS Exp() %prec UNARYMINUS
| Exp TIMES exp |
| MINUS exp |

Only specifies the prec. of this rule == prec. of UNARYMINUS.

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)

state 0:
- program : . exp
- ID shift 13
- INT shift 12
- STRING shift 11
- PLUS shift 10
- MINUS shift 9
- IF shift 8

program goto 135
table GOTO

exp goto 2
lvalue goto 1
- error

Connecting Yacc and Lex

signature PARSE = sig
val parse : string \rightarrow unit end
structure Parse = PARSE =
    struct
    structure TigerLrVals = TigerLrValsFun(structure Token = LrParser.Token)
    structure Lex = ToyLexFun(structure Tokens = TigerLrVals.Tokens)
    structure TigerP =
        Join(structure ParserData = TigerLrVals.ParserData
        structure Lex = Lex
        structure LrParser = LrParser)
    fun parse filename =
        let
            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