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)

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

  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

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

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

**Example:** a context-free grammar \(G=(T,N,P,S)\)

\[
\begin{align*}
T &= \{ +, *, (, ), \text{id} \}, \\
N &= \{ \text{E} \}, \\
P &= \{ \text{E} \rightarrow \text{E} + \text{E}, \text{E} \rightarrow \text{E} * \text{E}, \text{E} \rightarrow ( \text{E} ), \text{E} \rightarrow \text{id} \}, \\
S &= \text{E}
\end{align*}
\]

- 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 \(L = \{ \text{id+id, id+(id*id), (id), id*id*id*id, \ldots \} \}\)

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:

\[
\begin{align*}
E & \rightarrow E * E \quad \text{--- "E derives E * E"} \\
& \rightarrow \text{id} * E \quad \text{--- "E derives \text{id}"} \\
& \rightarrow \text{id} * (\text{id} + \text{id}) \\
& \rightarrow \text{id} * (\text{id} + \text{id}) \\
\end{align*}
\]

- 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**.

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

another leftmost derivation:

\[
\begin{align*}
E & \rightarrow E + E \rightarrow id + E \\
   & \rightarrow id + id \times E \\
   & \rightarrow id + id \times id \\
\end{align*}
\]

Resolving Ambiguity

Solution #1: using "disambiguating rules" such as precedence ...

\[ E \rightarrow E + E \rightarrow E \times E \rightarrow id + id \times id \]

Solution #2: rewriting grammar to be unambiguous!

"dangling-else" stmt \( \rightarrow \) if expr then stmt

\[ \text{if expr then stmt else stmt} \]

How to parse the following?

\[ \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \]

How to rewrite?

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

Resolving Ambiguity (cont’d)

• solution: define "matched" and "unmatched" statements

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

Now how to parse the following?

\[ \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \]

Resolving Ambiguity (cont’d)

• Another ambiguous grammar

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

usual precedence: highest \( (E) \) \( - E \) \( id \)

lowest \( + \) \( \times / \)

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

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

try the leftmost derivation for \( id + id \times id \)

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

- **Elimination of Left Recursion** (useful for top-down parsing only)

  replace productions of the form with

  \[ A \rightarrow A x \mid y \]
  \[ A \rightarrow y A' \]
  \[ A' \rightarrow x A' \mid \epsilon \]

  (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

  \[ A \rightarrow x y \mid x z \]
  \[ A \rightarrow x A' \]
  \[ A' \rightarrow y \mid z \]

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: \[ S \rightarrow c \quad A \quad d \]
  \[ A \rightarrow ab \mid a \]
  input symbols: \[ \text{cad} \]

  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 (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 aAcBe )</td>
<td>( aAbcde )</td>
<td>( A \rightarrow b )</td>
</tr>
<tr>
<td>( A \rightarrow Ab</td>
<td>b )</td>
<td>( aAcde )</td>
</tr>
<tr>
<td>( B \rightarrow d )</td>
<td>( aAcBe )</td>
<td>( B \rightarrow d )</td>
</tr>
<tr>
<td>( S \rightarrow S )</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. For example:
  
  \[
  \text{abcde} \rightarrow \text{aAb} \\
  \text{cde} \rightarrow \text{aAAcde} \rightarrow \text{stuck!}
  \]

- **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** \( \gamma \) is a reducing production of the form \( A \rightarrow \beta \) if \( S \Rightarrow^* \alpha \omega \Rightarrow^* \alpha \beta \omega \) and \( \omega \) contains only terminals. For example, \( A \rightarrow Ab \) in \( \text{aAbcde} \).

**Handle Pruning**

- **Main idea**: start with terminal string \( w \) and "prune" handles by replacing them with l.h.s. of productions until we reach \( S \):
  
  \[
  S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \omega 
  \]

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

**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** | **Input** | **Action**
  ):-|---|---
  \( \$ \) | \( a+b*c\$ | **Shift**
  \( \$a \) | \( +b*c\$ | **Reduce**: \( E \rightarrow a \)
  \( \$E \) | \( b*c\$ | **Shift**
  \( \$E+E*b \) | \( c\$ | **Reduce**: \( E \rightarrow b \)
  \( \$E+E*c \) | \( c\$ | **Shift** (possible SR conflict)
  \( \$E+E*E \) | \( c\$ | **Shift**
  \( \$E+E*c \) | \$ | **Reduce**: \( E \rightarrow c \)
  \( \$E+E*E \) | \$ | **Reduce**: \( E \rightarrow E+E \)
  \( \$E \) | \$ | **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
  
  \[
  S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \ldots
  \]

  **Stack** | **Input** | **Action**
  ):-|---|---
  \( \$\text{id(id...)} \) | **Reduce or Shift?**

- **Reduce / Reduce (RR) conflicts**: choose between two reductions
  
  \[
  \text{stmt} \rightarrow \text{id (param)} \leftarrow \text{procedure call} \ a(i) \\
  \text{param} \rightarrow \text{id} \leftarrow \text{array subscript} \ a(i) \\
  E \rightarrow \text{id (E)} | \text{id} \leftarrow \text{id reduce to E or param?}
  \]

  **Stack** | **Input** | **Action**
  ):-|---|---
  \( \$\text{id(id...)} \) | **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 Program

- Parsing configurations: (stack, remaining input) written as
  \[(s_0 X_1 s_2 \ldots X_m s_m , a_1 a_2 a_3 \ldots a_n \text{$_\$}\)]

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

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

LR Parsing Driver Routine

Given the configuration:
\[(s_0 X_1 s_2 \ldots X_m s_m , a_1 a_2 a_3 \ldots a_n \text{$_\$}\)]

1. If ACTION\[s_m, a_i\] is "shift \(s\)", enter config
   \[(s_0 X_1 s_2 \ldots X_m s_m a_i , a_1 a_2 a_3 \ldots a_n \text{$_\$}\)]

2. If ACTION\[s_m, a_i\] is "reduce \(A \rightarrow \beta\)", enter config
   \[(s_0 X_1 s_2 \ldots X_m s_n a_1 a_2 \ldots a_n \text{$_\$}\)
   where \(r=|\beta|\), and \(s = \text{GOTO}[s_{m-r}, A]\)
   (here \(\beta\) should be \(X_{m-r+1} X_{m-r+2} \ldots X_m\))

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
  - \(gn\)  \# 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 a 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

structure A = Absyn
......
%%
(term EOF | ID of string ...)
&noterm exp | program ...
&pos int
&eop EOF
&noshift EOF
......
%%
grm (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.

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

- **BNF production** \( A \rightarrow \alpha \mid \beta \mid \ldots \mid \gamma \) is written as
  \[
  A : \alpha \quad \text{(action for } A \rightarrow \alpha) \\
  \beta \quad \text{(action for } A \rightarrow \beta) \\
  \ldots \\
  \gamma \quad \text{(action for } A \rightarrow \rho) 
  \]

- **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 \\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)
```

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 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 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

  ⎯ %prec UNARYMINUS
  %prec UNARYMINUS

  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)

Parser state:

<table>
<thead>
<tr>
<th>State</th>
<th>Program</th>
<th>Exp</th>
<th>LValue</th>
<th>.</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>exp</td>
<td>goto 2</td>
<td>goto 1</td>
<td>.</td>
<td>error</td>
</tr>
</tbody>
</table>

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”

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

Connecting Yacc and Lex

signature PARSE = sig val parse : string -> 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