### 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

- 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 ID LPAREN tyfields RPAREN EQ exp
stmt

tyf
expr
ID LPAREN exp RPAREN
ID COLON ID
exp
PLUS
ID
```

---

### 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
```

## 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 \(L = \{ id+id, id+( id*id ), ( id ), id*id*id*id, ............ \}\)

- 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:

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

- Summary: \(E \Rightarrow id * (id + id)\)
  
  - **derivates in 0 or more steps**

- 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: $id + id \times id$

Another leftmost derivation:

- $E \Rightarrow E + E \Rightarrow id + E \times E$
- $E \Rightarrow E + E \Rightarrow id + id \times id$

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” stmt -> 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

- stmt -> m-stmt | u-stmt
- m-stmt -> if expr then m-stmt else m-stmt
| ......
- u-stmt -> if expr then stmt
| if expr then m-stmt else u-stmt

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
| * /
| lowest

- Build grammar from highest ---> lowest precedence

$element \Rightarrow ( expr ) | id$
$primary \Rightarrow - primary | element$
$term \Rightarrow term + primary | term / primary | primary$
$expr \Rightarrow expr + term | expr - term | term$

Try the leftmost derivation for...

$- id + id \times id$
$expr \Rightarrow expr + term \Rightarrow term + term \Rightarrow primary + term$
| - primary + term \Rightarrow - element + term \Rightarrow - id + term$
| - id + term + primary \Rightarrow ... \Rightarrow - id + id \times id$
Other Grammar Transformations

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

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

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: \[ S \rightarrow c \ A \ d \]
  \[ A \rightarrow a b \ | \ a \]

  input symbols: \[ \text{cad} \]

  decide which rule of \[ A \] to use here?

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 \rightarrow aAbCd</td>
<td>aAbCd</td>
<td>A \rightarrow b</td>
</tr>
<tr>
<td>A \rightarrow Ab</td>
<td>aAbCd</td>
<td>A \rightarrow Ab</td>
</tr>
<tr>
<td>B \rightarrow d</td>
<td>aAbCd</td>
<td>B \rightarrow d</td>
</tr>
<tr>
<td>S \rightarrow aAbCd</td>
<td>S</td>
<td>S \rightarrow aAbCd</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!
- This b is not a handle!
- 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 \( \gamma = \alpha A \omega \) is \( A \rightarrow \beta \) if \( S \Rightarrow^* \alpha A \omega \Rightarrow^* \alpha \beta \omega \)
  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 \( S \):
  \[
  S \Rightarrow^* \gamma_1 \Rightarrow^* \gamma_2 \Rightarrow^* \ldots \Rightarrow^* \gamma_{n-1} \Rightarrow^* \omega
  \]
  (i.e., construct the rightmost derivation in reverse)
- Example: \( E \rightarrow E + E \mid E \* E \mid ( E ) \mid a \mid b \mid c \)

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: \( E \rightarrow E + E \mid E \* E \mid ( E ) \mid a \mid b \mid c \)

<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 \rightarrow b )</td>
</tr>
<tr>
<td>$E+c$</td>
<td>c</td>
<td>reduce: ( E \rightarrow c )</td>
</tr>
<tr>
<td>$E+c$</td>
<td>c</td>
<td>shift</td>
</tr>
<tr>
<td>$E+c$</td>
<td>$</td>
<td>reduce: ( E \rightarrow E+c )</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td>reduce: ( E \rightarrow E+c )</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
  \[
  S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \ldots....
  \]

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

- Reduce/reduce (RR) conflicts: choose between two reductions
  \[
  \text{stmt} \rightarrow \text{id (param)} \quad \text{--- procedure call} \quad a(1)
  \]
  \[
  \text{param} \rightarrow \text{id} \quad \text{--- array subscript} \quad a(1)
  \]

<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 ( \text{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

LR Parsing Driver Routine

Given the configuration:

\[(s_0X_1s_1X_2s_2...X_mS, \ a_1a_1a_1a_1...a_nS)\]

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

\[(s_0X_1s_1X_2s_2...X_mS[a]S, \ a_1a_1a_1a_1...a_nS)\]

(2) If ACTION\( [s_n, a_i] \) is “reduce A->", enter config

\[(s_0X_1s_1X_2s_2...X_{m-r}s_{n-r}A\beta, \ a_1a_1a_1a_1...a_nS)\]

where \( r=|\beta| \), and \( s = GOTO(s_{n-r}, A) \)
(here \( \beta \) should be \( X_{p-r}X_{p-r}...X_p \))

(3) If ACTION\( [s_n, a_i] \) is “accept”, parsing completes

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

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:

\[\begin{align*}
\text{sn} & \quad \text{shift and put state n on the stack} \\
\text{gn} & \quad \text{go to state n} \\
\text{rk} & \quad \text{reduce by rule k} \\
\text{a} & \quad \text{accept by rule k} \\
\text{error} & \quad \text{error}
\end{align*}\]

• 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 ()

user's ML declarations
Yacc declarations
rule-lists
can call the above 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
  
  $A : \alpha$(action for $A \rightarrow \alpha$)
  
  $| \beta$(action for $A \rightarrow \beta$)
  
  $| \ldots$(action for $A \rightarrow \ldots$)
  
  $| \gamma$(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 | 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
  
  lowest prec.  
  \%
  \%
  \%
  \%
  \%
  \%
  \%
  %left OR
  %left AND
  %noassoc EQ NEQ GT LT GE LE
  %left PLUS MINUS
  %left TIMES DIV
  highest prec.

- Defining precedence for rules using %prec

  %left PLUS MINUS%left TIMES DIV
  
  Exp : Exp MINUS Exp
   | Exp TIMES exp
   | MINUS exp
   | EXP exp
  
  Must define UNARYMINUS as a new terminal!

  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)

  state 0:
  
  program : . exp
  
  table ACTION
  
  ID   shift 13
  IF   shift 8
  program goto 135
  table GOTO
  
  exp    goto 2
  lvalue goto 1
  .     error

---

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

%%

%header (functor TigerLexFun(structure Tokens : Tiger_TOKENS));

%%

```

Connecting Yacc and Lex

```plaintext

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.Token)
    structure TigerP =
      Join(structure ParserData = TigerLrVals.ParserData
            structure Lex = LrParser)
    fun parse filename =
      let val _ = (ErrorMsg.reset(); ErrorMsg.fileName := filename)
        val file = open_in filename
        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

end
```

---