Parser Generation

- ***Main Problem**: given a grammar **G**, how to build a **top-down parser** or a **bottom-up parser** for it ?*
- ***parser** : a program that, given a sentence, reconstructs a derivation for that sentence ---- if done sucessfully, 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*

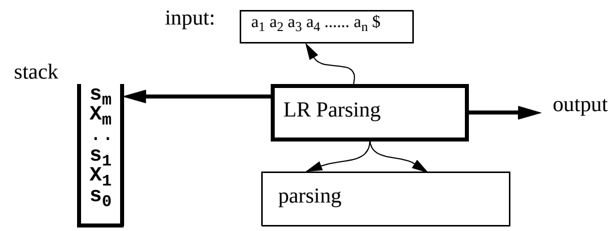
  shift-reduce, LR, SLR, LALR, operator precedence
- ***top-down parsers** --- construct the tree from root to leaves*

  recursive descent, predictive parsing, LL(1)

CS421 COMPILERS AND INTERPRETERS

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***
- ***LR parser** components: input, stack (strings of grammar symbols and **states**), driver routine, parsing tables.*



CS421 COMPILERS AND INTERPRETERS

LR Parsing

- *A sequence of new **state** symbols* $s_0, s_1, s_2, ..., s_m$ *----- each state sumarize the information contained in the stack below it.*
- *<u>Parsing configurations</u>**:** **(stack, remaining input)** written as*

  $(s_0 X_1 s_1 X_2 s_2 ... X_m s_m \ , \ a_i a_{i+1} a_{i+2} ... a_n \$)$

  next "move" is determined by s_m and a_i
- *<u>Parsing tables</u>**:*** **ACTION[s,a]** *and* **GOTO[s,X]**

  Table A **ACTION[s,a] --- s : *state*, a : *terminal***

  its entries **(1) shift** s_k **(2) reduce** $A \rightarrow \beta$ **(3) accept (4) error**

  Table G **GOTO[s,X] --- s : *state*, X : *non-terminal***
  *its entries are **states***

CS421 COMPILERS AND INTERPRETERS

Constructing LR Parser

How to construct the parsing table* ACTION *and* GOTO *?

- ***basic idea**: first construct DFA to recognize handles, then use DFA to construct the parsing tables ! different parsing table yield different LR parsers **SLR(1)**, **LR(1)**, or **LALR(1)***
- ***augmented grammar** for context-free grammar **G** = **G**(T,N,P,S) is defined as **G'** = **G'**(T, N ∪ { S'}, P ∪ { S' -> S}, S') ------ adding non-terminal S' and the production S' -> S, and S' is the new start symbol. When S' -> S is reduced, parser accepts.*
- ***LR(0) item** for productions of a context-free grammar G ----- is a production with **dot** at some position in the r.h.s.*

  For *A -> XYZ* , its items are *A -> .XYZ*   *A -> X.YZ*   *A -> XY.Z*   *A -> XYZ.*

  For *A* -> ε , its items are just *A -> .*

LR(0) items and LR(0) DFA

- *Informally, item* `A -> X.YZ` *means a string derivable from* `X` *has been seen, and one from* `YZ` *is expected.* ***LR(0) items*** *are used as state names for LR(0) DFA or LR(0) NFA that recognizes* ***viable prefixes.***
- ***Viable prefixes*** *of a CFG are prefixes of right-sentential forms with no symbols to right of the* ***handle****; we can always add terminals on right to form a right-sentential form.*
- ***Two way to construct the LR(0) DFA:***

  1. first construct LR(0) NFA and then convert it to a DFA !

  2. construct the LR(0) DFA directly !

- ***From LR(0) DFA to the Parsing Table*** -------- *transition table for the DFA is the* **`GOTO`** *table; the states of DFA are states of the parser.*

Example: LR(0) Items

CFG Grammar:

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

Augmented Grammar:

```
E' -> E
E  -> E + T | T
T  -> T * F | F
F  -> ( E ) | id
```

LR(0) terms:

```
E' -> . E          T -> . T * F       F -> ( E . )
E' -> E .          T -> T . * F       F -> ( E ) .
E -> . E + T       T -> T * . F       F -> . id
E -> E . + T       T -> T * F .       F -> id .
E -> E + . T       T -> . F
E -> E + T .       T -> F .
E -> . T           F -> . ( E )
E -> T .           F -> ( . E )
```


From LR(0) NFA to LR(0) DFA

- *Construct* ***LR(0) NFA*** *with all LR(0) items of G as states, connect states by moving the* ***dot****;* ***final states*** *are those with dots at the end.*

  1. for each item $A \to \alpha . X \beta$

  $$\boxed{A \to \alpha . X\beta} \xrightarrow{X} \boxed{A \to \alpha X . \beta}$$

  2. for each pair $A \to \alpha . B \beta$, $B \to . \gamma$

  (expect to see a string derivable from γ)

  $$\boxed{A \to \alpha . B\beta} \xrightarrow{\varepsilon} \boxed{B \to . \gamma}$$

- *Convert NFA to DFA using* ***subset construction algorithm.***
- *The states of the resulting* ***LR(0) DFA*** --- $\mathbf{C} = \{\mathbf{I}_1, \mathbf{I}_2, ..., \mathbf{I}_n\}$ *are called* ***canonical LR(0) collection*** *for grammar G'*
- ***Disadvantage****: the NFA is often huge, and converting from NFA to DFA is tedious and time-consuming.*

Building LR(0) DFA Directly

- *Instead of building DFA from NFA,* ***we can build the LR(0) DFA directly****.*
- *Given a set of LR(0) items* **`I`** *,* **`CLOSURE(I)`** *is defined as*

```
repeat
  for each item A -> α.Bβ in I and
      each production B -> γ
           add B -> .γ to I, if it's not in I
until I does not change
```

- **`GOTO(I,X)`** *is defined as*

  `CLOSURE(`*all items* $A \to \alpha X . \beta$ *for each* $A \to \alpha . X \beta$ *in* **`I`**)

- ***Canonical LR(0) collection*** *is computed by the following procedure:*

```
I0 = CLOSURE({S' -> .S}) and C = {I0}
repeat
  for each I ∈ C and grammar symbol X
    T = GOTO(I,X); if T ≠ ∅ and T ∉ C then C = C ∪ { T };
until C does not change
```

Resulting `LR(0) DFA`***:*** `C` ***is the set of states;*** `GOTO` ***is the transition table***

Constructing SLR(1) Parsing Table

- *From the **LR(0) DFA**, we can construct the parsing table ----- **SLR(1) parsing table.** The parser based on SLR(1) parsing table is called **SLR(1) parser**. The **SLR(1) grammars** are those whose SLR(1) parsing table does **not** contain any conflicts.*
- ***Algorithm** --- use* **C = {I_0,...,I_n}, GOTO, FOLLOW:**

  1. **If** A -> a.aβ **is in I_i and GOTO(I_i,a) = I_j where** a **is a terminal, set ACTION[**i,a**] to "shift j".**
  2. **If** A -> α. **is in I_i, set ACTION[**i,a**] to "reduce** A **->** α**" for all terminal** a **in FOLLOW(**A**).**
  3. **If S' -> S. is in I_i, set ACTION[$_i$,$] to "accept"**
  4. **If GOTO(I_i,A) = I_j, set GOTO[i,A] = j**
  5. **set all other entries to "error"**
  6. **set initial state to be I_i with** S' -> .S

Limitation of SLR(1) Parser

- *Unfortunately, many **unambiguous** grammars are not SLR(1) gammars*

```
S -> L = R | R          L means "l-value"
L -> *R | id            R means "r-value"
R -> L                  * means "contents of"
```

Canonical LR(0) collection ---

```
I0 : S' -> .S          I3:  S -> R.          I6: S -> L=.R
     S -> .L=R                                   R -> .L
     S -> .R           I4:  L -> *.R             L -> .*R
     L -> .*R               R -> .L              L -> .id
     L -> .id               L -> .*R
     R -> .L                L -> .id         I7: L -> *R.

I1 : S' -> S.          I5:  L -> id.         I8: R -> L.

I2 : S -> L.=R                               I9: S -> L=R.
     R -> L.             FOLLOW(R) = {=,...}
```

state 2 has a shift/reduce conflict on **"="** *: shift 6 or reduce* R -> L

LR(1) Parsing

- *Conflict arises because **LR(0) states** do not encode enough left context --- in the previous example, reduction* R -> L *is wrong upon input* **=** *because "*R = ...*" **never** appears in right-sentential form.*
- ***Solution**: split LR(0) states by adding terminals to states, for example,* [A -> α. , a] *results in reduction only if next symbol is* a.
- ***An LR(1) term** is in the form of* [A -> α.β , a] *where* A -> αβ *is a production and* a *is a terminal or* **$**
- *To build **LR(1) parsing table** --- we first build **LR(1) DFA** --- then construct the parsing table using the same SLR(1) algorithm except*

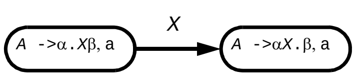
  2. *only if* [A -> α. , a] *is in* $\mathbf{I_i}$, *then set* **ACTION[i,a]** *to "reduce* A-> α*"*

- *To way to build **LR(1) DFA** ---- from NFA -> DFA or build DFA directly*

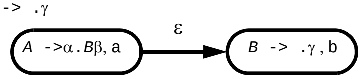
Building LR(1) DFA

- *Construct **LR(1) NFA** with all LR(1) items of G as states, connect states by moving the **dot**; then convert the NFA to DFA.*

1. ***for each item*** *[A -> α.X β , a]*



2. ***for each pair*** [A -> α.Bβ , a] B -> .γ ***and*** b ***in*** **FIRST**(β a).



- *Construct the LR(1) DFA directly (see the Dragon book)*
- *Given a set of LR(1) items* **I** , **CLOSURE(I)** *is now defined as*

```
repeat
  for each item [A -> α.Bβ , a] in I and
      each production B -> γ and each terminal b in FIRST(βa)
                add [B -> .γ, a] to I, if it's not in I
  until I does not change
```

Constructing LR(1) Parser

- ***Canonical LR(1) collection** is computed by the following procedure:*

```
I0 = CLOSURE([S' -> .S , $]) and C = {I0}
repeat
  for each I ∈ C and grammar symbol X
    T = GOTO(I,X); if T ≠ ∅ and T ∉ C then C = C ∪ { T };
until C does not change
```

Resulting `LR(1) DFA`***:*** `C` ***is the set of states;*** `GOTO` ***is the transition table***

- *From the **LR(1) DFA**, we can construct the parsing table ----- **LR(1) parsing table.** The parser based on LR(1) parsing table is called **LR(1) parser**. The **LR(1) grammars** are those whose LR(1) parsing table does **not** contain any conflicts (no duplicate entries).*

- ***Example:***

```
S' -> S
S -> C C
C -> c C | d
```


LALR(1) Parsing

- ***Bad News:** LR(1) parsing tables are too big; for PASCAL, SLR tables has about 100 states, LR table has about 1000 states.*

- ***LALR** (LookAhead-LR) parsing tables have same number of states as SLR, but use lookahead for reductions. The **LALR(1) DFA** can be constructed from the **LR(1) DFA.***

- ***LALR(1) states** can be constructed from LR(1) states by merging states with same **core**, or same LR(0) items, and **union** their lookahead sets.*

```
Merging  I8: C -> cC., c/d           I9: C -> cC., $
into a new state     I89:  C -> cC., c/d/$

Merging  I3: C -> c.C, c/d           I6: C -> c.C, $
             C -> .cC, c/d               C -> .cC, $
             C -> .d, c/d                C -> .d, $
into a new state     I36:  C -> c.C, c/d/$
                           C -> .cC, c/d/$
                           C -> .d, c/d/$I
```


LALR(1) Parsing (cont'd)

- *From the **LALR(1) DFA**, we can construct the parsing table ----- **LALR(1) parsing table.** The parser based on LALR(1) parsing table is called **LALR(1) parser**. The **LALR(1) grammars** are those whose LALR(1) parsing table does **not** contain any conflicts (no duplicate entries).*

- *LALR(1) DFA and LALR(1) parsing table can be constructed without creating LR(1) DFA --- see Dragon book for detailed algorithm.*

- ***LALR parser** makes same number of moves as LR parser on **correct input.***

- ***On incorrect input,** LALR parser may make erroneous reductions, but will signal "error" before shifting input, i.e., merging states makes reduce determination "less accurate", but has no effect on shift actions.*

Summary: LR Parser

- *Relation of three LR parsers:* ***LR(1) > LALR(1) > SLR(1)***

- *Most programming language constructs are **LALR(1)**. The **LR(1)** is unnecessary in practice, but the **SLR(1)** is not enough.*

- *YACC is an LALR(1) Parser Generator.*

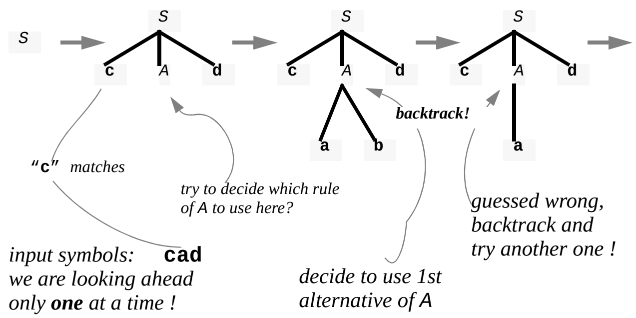
- *When parsing ambiguious grammars using LR parsers, the parsing table will contain multiple entries. We can specify the **precedence** and **associativity** for terminals and productions to resolve the conflicts. YACC uses this trick.*

- ***Other Issues** in parser implementation: 1. compact representation of parsing table 2. error recovery and diagnosis.*

Top-Down Parsing

- ***Starting from the start symbol*** *and "guessing" which production to use next step. It often uses* ***next input token*** *to guide "guessing".*

example: `S -> c A d`
`A -> ab | a`



Top-Down Parsing (cont'd)

- *Typical implementation is to write a* ***recursive procedure*** *for each* ***non-terminal*** *(according to the r.h.s. of each grammar rule)*

Grammar:

```
E  -> T E'

E' -> + T E'
    | ε

T  -> F T'

T' -> * F T'
    | ε

F  -> id
    | ( E )
```

`advance` *sets* `c` *to next input token*
`err` *reports error message*

```
fun e() = (t(); eprime())

and eprime() = if (c = "+")
  then (advance(); t(); eprime())

and t() = (f(); tprime())

and tprime() = if (c = "*")
  then (advance(); f(); tprime())

and f() =
  (if (c = id) then advance()
   else if (c = "(") then
     (advance(); e();
      if (c=")") then advance() else err())
     else err()
```


Recursive Descent Parsing

- *The previously referred top-down parsing method is often called* ***recursive descent parsing*** *!*

- ***Main challenges:***

1. **back-tracking** is messy, difficult and inefficient
(solution: use input "lookahead" to help make the right choice)

2. **more alternatives** --- even if we use one lookahead input char, there are still > 1 rules to choose --- `A -> ab | a`
(**solution**: rewrite the grammar by **left-factoring**)

3. **left-recursion** might cause infinite loop
what is the procedure for `E -> E + E` ?
(**solution**: rewrite the grammar by **eliminating left-recursions**)

4. **error handling ---** errors detected "far away" from actual source.

Algorithm: Recursive Descent

- ***Parsing Algorithm (using 1-symbol lookahead in the input)***

1. Given a set of grammar rules for a non-terminal ***A***

$$A \; \texttt{->} \; \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

we choose proper alternative by looking at first symbol it derives ---- the next input symbol decides which α_i we use

2. for `A -> ε`, it is taken when none of the others are selected

- ***Algorithm****: constructing a recursive descent parser for grammar* ***G***

1. transform grammer ***G*** *to* ***G'*** *by removing left-recursions and do the left-factoring.*

2. write a (recursive) procedure for each non-terminal in ***G'***

Left Recursion Elimination

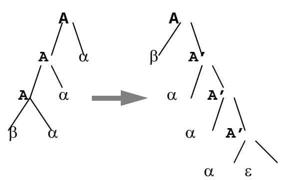
- *Elimination of **Left Recursion** (useful for top-down parsing only)*

replace productions of the form $A \text{ -> } A\ \alpha \mid \beta$

with $A \text{ -> } \beta A'$

$A' \text{ -> } \alpha\ A' \mid \varepsilon$

(yields different parse trees but same language)




```
example:
  E -> E + T | T
  T -> T * F | F
  F -> ( E ) | id
become

  E  -> T E'
  E' -> + T E' | ε
  T  -> F T'
  T' -> * F T' | ε
  F  -> ( E ) | id
```

Important: read Appel pp 51 - 53 for details

Left Factoring

- *Some grammars are unsuitable for recursive descent, even if there is no left recursion*

*"**dangling-else**"*
```
stmt -> if expr then stmt
      | if expr then stmt else stmt
      | ......
```

input symbol **`if`** *does not uniquely determine alternative.*

- ***Left Factoring** --- factor out the common prefixes (**see AHU pp 178**)*

change the production $A \text{ -> } x\ y \mid x\ z$

to $A \text{ -> } x\ A'$

$A' \text{ -> } y \mid z$

thus
```
stmt -> if expr then stmt S'
  S' -> else stmt | ε
```


Predictive Parsing

- ***Predictive parsing** is just **table-driven** recursive descent; it contains:*

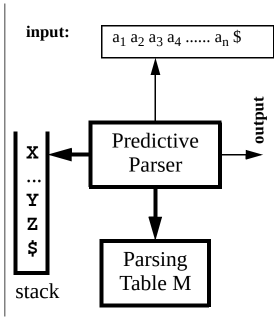
A **parsing stack** --- contains terminals and non-terminals

A **parsing table** : a 2-dimensional table **`M[X,a]`** where **`X`** is non-terminal, **`a`** is terminal, and table entries are <u>grammar productions</u> or <u>error indicators</u> .

<u>algorithm</u> `$` ***is end-of-file***, `S` ***is start symbol***

```
push($); push(S);

while top <> $ do (
   a <- the input char
   if top is a terminal or $ then
        (if top == a then
               pop(); advance()
         else err())
   else if M[top,a] is X->Y1Y2...Yk then
           (pop();
            push(Yk); ...; push(Y1))
        else err()
)
```



Constructing Predictive Parser

- *The key is to build the parse table* **`M[A,a]`**

```
for each production A -> α do
  for each a ∈ FIRST(α) do
        add A -> α to M[A,a]
  if ε ∈ FIRST(α) then
        for each b ∈ FOLLOW(A) do
            add A -> α to M[A,b]

rest of M is error
```

- **`FIRST`**(α) *is a set of terminals (plus* ε*) that **begin** strings derived from* α*, where* α *is any string of non-terminals and terminals.*

- **`FOLLOW`**(A) *is a set of terminals that can **follow** A in a sentential form, where A is any non-terminal*

First & Follow

- *To compute* **FIRST(X)** *for any grammar symbol* **X** *:*

FIRST(X) = **{**X**}**, *if* X *is a terminal*;
FIRST(X) = **FIRST**(X) $\cup$ **{**a**}**, *if* X -> aα ;
FIRST(X) = **FIRST**(X) $\cup$ **{**ε**}**, *if* X -> ε ; *and*
FIRST(X) = **FIRST**(X) $\cup$ **FIRST**($Y_1Y_2...Y_k$),
if X -> $Y_1Y_2...Y_k$.

repeat until nothing new is added to any **FIRST**

- **FIRST**($Y_1Y_2...Y_k$) **=** **FIRST**(Y_1)**-{**ε**}**
$\cup$ **FIRST**(Y_2)**-{**ε**}** *if* $\varepsilon \in$ **FIRST**(Y_1)
$\cup$ **FIRST**(Y_3)**-{**ε**}** *if* $\varepsilon \in$ **FIRST**(Y_1Y_2)
....
$\cup$ **FIRST**(Y_k)**-{**ε**}** *if* $\varepsilon \in$ **FIRST**($Y_1...Y_{k-1}$)
$\cup$ **{**ε**}** *if all* **FIRST**(Y_i)$_{(i=1,...,k)}$ *contain* ε

First & Follow (cont'd)

- *To compute* **FOLLOW(X)** *for any non-terminal* **X** *:*

FOLLOW(S) = **FOLLOW**(S) $\cup$ **{**\$**}**, *if* S *is start symbol*;

FOLLOW(B) = **FOLLOW**(B) $\cup$ **(FIRST**(β) **- {**ε**})**,
if A -> α B β *and* $\beta \neq \varepsilon$

FOLLOW(B) = **FOLLOW**(B) $\cup$ **FOLLOW**(A)
if A -> α B *or* A -> α B β *and* $\varepsilon \in$ **FIRST**(β)

- ***Example:***

	reason
FOLLOW(E) = {**\$**},	E *is start symbol*
= {**\$)**},	F -> **(** E **)**
FOLLOW(E')= **FOLLOW**(E)	E -> T E'
= {**\$)**},	

(Read Appel pp 47 - 53 for detailed examples)

Summary: LL(1) Grammars

- *A grammar is* ***LL(1)*** *if parsing table* **M[A,a]** *has no duplicate entries, which is equivalent to specifying that for each production*

$$A \text{ -> } \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

1. All **FIRST**(α_i) *are disjoint.*

2. At most one α_i *can derive* ε *; in that case,* **FOLLOW**(***A***) *must be disjoint from* **FIRST**(α_1) $\cup$ **FIRST**(α_2) $\cup$ ...... $\cup$ **FIRST**(α_n)

- *Left-recursion and ambiguity grammar lead to* ***multiple entries*** *in the parsing table. (try the* ***dangling-else*** *example)*

- *The* ***main difficulty*** *in using (top-down)* ***predicative parsing*** *is in rewriting a grammar into an LL(1) grammar. There is no general rule on how to resolve* ***multiple entries*** *in the parsing table.*