Topic 3: Parsing and Yaccing

COS 320

Compiling Techniques

Princeton University
Spring 2018

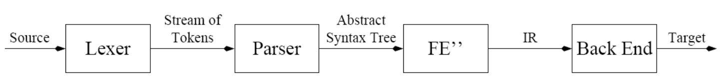
Prof. David August

Syntactical Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Syntactical Analysis

- Every programming language has a set of rules that describe syntax of well-formed programs in language.
- *Syntax Analysis* (Parsing) - Determine if source program satisfies these rules.
- Source program constructs may have recursive structure:

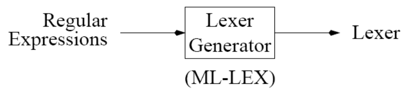
```
digits = [0-9]+
expr = {digits} | "(" {expr} "+" {expr} ")"
```

- Finite Automata cannot recognize recursive constructs. (A machine with N states cannot remember a parenthesis-nesting depth greater than N.)

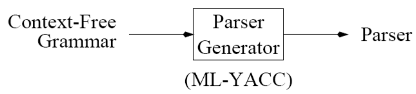
We need a more powerful formalism: *Context-Free Grammar*

Context-Free Grammar

Regular Expressions - describe lexical structure of tokens.



Context-Free Grammars - describe syntactic nature of programs.



Definitions

- *Language* - set of strings
- *String* - finite sequence of *symbols* taken from finite *alphabet*
  - Regular Expressions describe a language.
  - Context-Free Grammar also describes a language.

	Lexical Analysis	Syntax Analysis
language	set of tokens	set of source programs
string	token	source program
symbol	ASCII character	token

Context-Free Grammar

- Also known as BNF (Backus-Naur Form).
- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- Examples:
  - Matching parentheses
  - Nested comments

Context-Free Grammars

- Context-Free Grammars consist of a set of *productions*.

$$symbol \rightarrow symbol\ symbol\ ...\ symbol$$

- Symbol types:
  - *terminal* that corresponds to a token-type.
  - *non-terminal* that denotes a set of strings.
- Left-Hand Side (LHS) - *non-terminal*.
- Right-Hand Side (RHS) - *terminals* or *non-terminals*
- *Start Symbol* - A special *non-terminal*.
- Each production specifies how terminals and non-terminals may be combined to form a substring in language.
- Easy to specify recursion:

$$stmt \rightarrow IF\ exp\ THEN\ stmt\ ELSE\ stmt$$

Start Symbol

- String of token-types is in language described by grammar if it can be derived from *start symbol*
- Derivations:
  1. begin with start symbol
  2. while non-terminals exist, replace any non-terminal with RHS of production
- Multiple derivations exist for given sentence
  - Left-most derivation - replace left-most non-terminal in each step.
  - Right-most derivation - replace right-most non-terminal in each step.

Example

Non-Terminals:

```
stmt      : Statement
expr      : Expression
expr_list : Expression List
```

Terminals (tokens):

```
SEMI ";"
ID
ASSIGN ":="
LPAREN "("
RPAREN ")"
NUM
PLUS  "+"
PRINT "print"
COMMA ","
```

$stmt \rightarrow stmt; stmt$
$stmt \rightarrow ID := expr$
$stmt \rightarrow PRINT\ (\ expr\_list\ )$

$expr \rightarrow ID$
$expr \rightarrow NUM$
$expr \rightarrow expr + expr$
$expr \rightarrow (\ stmt,\ expr\ )$

$expr\_list \rightarrow expr$
$expr\_list \rightarrow expr\_list,\ expr$

Example: Leftmost Derivation

Show that expression can be derived from start symbol.

```
ID := NUM; PRINT(NUM)

a := 12; print(23)
```


Example: Rightmost Derivation

Show that expression can be derived from start symbol.

```
ID := NUM; PRINT(NUM)

a := 12; print(23)
```


Parse Trees

- *Parse Trees* - Graphical representation of derivation.
- Each internal node is labeled with a non-terminal.
- Each leaf node is labeled with a terminal.
- Parse Tree of the example using right-most derivation production:

Ambiguous Grammars

A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.

Non-Terminals:

```
expr : Expression
```

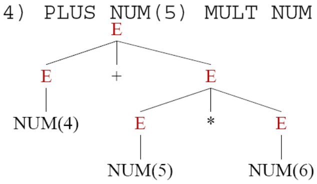
Terminals (tokens):

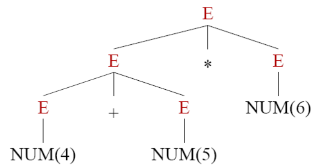
```
ID
NUM
PLUS   "+"
MULT   "*"
```

$expr \rightarrow ID$
$expr \rightarrow NUM$
$expr \rightarrow expr + expr$
$expr \rightarrow expr * expr$

Consider: $4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`





Ambiguous Grammars

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does $4 + 5 * 6$ equal 34 or 54?
- *Solution*: rewrite grammar to eliminate ambiguity.
  - If language doesn't have unambiguous grammar, then you have a bad programming language.
  - Operators have a relative *precedence*. We say some operands *bind tighter* than others. ("*" binds tighter than "+")
  - Operators with the same precedence must be resolved by *associativity*. Some operators have *left associativity*, others have *right associativity*.

Ambiguous Grammars

Non-Terminals:

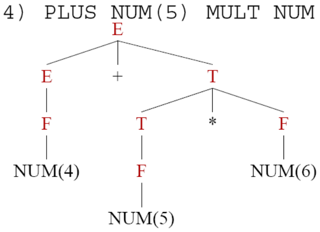
```
expr : Expression
term : Term (add)
fact : Factor (mult)
```

Terminals (tokens):

$expr \rightarrow expr + term$
$expr \rightarrow term$
$term \rightarrow term * fact$
$term \rightarrow fact$
$fact \rightarrow ID$
$fact \rightarrow NUM$

$4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



End-Of-File Marker

- Parse must also recognize the End-of-File (EOF).
- EOF marker in the grammar is "$"
- Introduce new start symbol and the production $E' \rightarrow E\$$

Grammars and Lexical Analysis

- Grammars can also describe token structure:

```
( a | b )* abb
```

$W \rightarrow aW$
$W \rightarrow bW$
$W \rightarrow aX$
$X \rightarrow bY$
$Y \rightarrow bZ$
$Z \rightarrow \epsilon$

- Can combine lexical analysis and syntax analysis into one module.
- Disadvantages:
  - Regular expression specification is more concise.
  - Separating phases increases compiler modularity.

Context-Free Grammars and REs

- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- As a corollary, CFGs are strictly more powerful than DFAs and NFAs.
- The proof is in two parts:
  - Given a regular expression R , we can generate a CFG G such that L(R) == L(G).
  - We can define a grammar G for which there there is no FA F such that L(F) == L(G).

Context Free Grammars and REs

Base Cases:

- Symbol (a):
  $RE \rightarrow a$
- Epsilon (ϵ):
  $RE \rightarrow \epsilon$

Inductive Cases:

- Alternation ($M|N$):
  $RE \rightarrow M$
  $RE \rightarrow N$
- Concatenation ($M\ N$):
  $RE \rightarrow M\ N$
- Kleen closure ($M*$):
  $RE \rightarrow M\ RE$
  $RE \rightarrow \epsilon$

Context-Free Grammar with no RE/FA

$$S \rightarrow (S)$$
$$S \rightarrow \epsilon$$

- FAs have a FINITE number of states, N
- FA must "remember" number of "("s, to generate ")"s
- At or before $N+1$ "("s FA will revisit a state.
- That state represents two different counts of ")"s.
- Both counts must now be accepted.
- One count will be invalid.

Representations

- Regular, right-linear, finite-state grammars: FAs
- Context-free grammars: Push-Down Automata

Further Exploration

We have been talking about Context-Free Grammars.

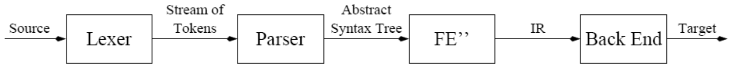
What is a **context-sensitive grammar?**

Parsing

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Outline

- Recursive Descent Parsing
- Shift-Reduce Parsing
- ML-Yacc
- Recursive Descent Parser Generation

Recursive Descent Parsing

- Recall discussion on Context-Free Grammars: symbols (terminal, non-terminal), productions, derivations, etc.
- Can parse many grammars using algorithm called *recursive descent* parsing.
  - A.K.A.: *predictive parsing*
  - A.K.A.: *top-down parsing*
  - A.K.A.: *LL(1)* - Left-to-right parse, Leftmost-derivation, 1-symbol lookahead.
- One recursive function for each non-terminal.
- Each production becomes clause in function.

Example

Grammar:
non-terminals: `S, L, E`

terminals: `IF`(*if*), `THEN`(*then*), `ELSE`(*else*), `BEGIN`(*begin*),
`PRINT`(*print*), `END`(*end*), `SEMI`(*;*), `NUM`, `EQ`(=)

S → *if* E *then* S *else* S
S → *begin* S L
S → *print* E
L → *end*
L → *;* S L
E → *num* = *num*

```
datatype token = EOF | IF | THEN | ELSE | BEGIN |
                 PRINT | END | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() = tok := getToken()
fun eat(t) = if (!tok = t) then advance() else error()

fun S() = case !tok of
            IF    => (eat(IF); E(); eat(THEN); S();
                      eat(ELSE); S())
            BEGIN => (eat(BEGIN); S(); L())
            PRINT => (eat(PRINT); E())
and L() = case !tok of
            END   => (eat(END))
            SEMI  => (eat(SEMI); S(); L())
and E() =             (eat(NUM); eat(EQ); eat(NUM))
```


Another Example

Grammar:

A → S EOF
S → *id* := E
S → *print* (L)

E → *id*
E → *num*
L → E
L → L, E

```
fun A() =               (S(); eat(EOF))
and S() = case !tok of
            ID      => (eat(ID); eat(ASSIGN); E())
            PRINT   => (eat(PRINT); eat(LPAREN);
                        L(); eat(RPAREN))
and E() = case !tok of
            ID      => (eat(ID))
            NUM     => (eat(NUM))
and L() = case !tok of
            ID      => (?????)
            NUM     => (?????)
```


The Problem

- If !tok = ID, parser cannot determine which production to use:
  L → E (E could be ID)
  L → L, E (L could be ID)
- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.
- Can write predictive parser by eliminating *left recursion*.

L → E
L → L, E

⟹

L → E M
M → , E M
M → ϵ

```
and L() = case !tok of
            ID      => (E(); M())
            NUM     => (E(); M())
and M() = case !tok of
            COMMA    => (eat(COMMA); E(); M())
            RPAREN   => ()
```

Another Option: Shift-Reduce Parsing

- Given next input token, predictive parser must predict which production to use.
- *Shift-reduce parsing* delays decision until it has seen input token corresponding to entire RHS of production.
  - A.K.A.: *bottom-up parsing*
  - A.K.A.: *LR(k)* - Left-to-right parse, Rightmost derivation, k-token lookahead
- Shift-reduce parsing can parse more grammars than predictive parsing.
- Parser has *stack*.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production (X → ABC); pop off RHS (C, B, A); push LHS (X).
- Stack is initially empty.
- Parser points to beginning of input stream.
- If $ is shifted, then input stream has been parsed successfully.

Shift-Reduce Parsing

How does parser know when to shift or reduce?

- DFA: applied to stack contents, not input stream
- Each state corresponds to contents of stack at some point in time.
- Edges labelled with terms/non-terms that can appear on stack.

Example

Grammar:
1 A → S EOF
2 S → *(* L *)*
3 S → *id* = *num*
4 L → L*;* S
5 L → S

Input:
$(a = 4; b = 5) \rightarrow (ID_a = NUM_4; ID_b = NUM_5)$

0

```
 input: ( ID = NUM ; ID = NUM )
        |
 stack:
action: shift
```

Example

1
```
 input: ( ID = NUM ; ID = NUM )
          |
 stack: (
action: shift
```

2
```
 input: ( ID = NUM ; ID = NUM )
             |
 stack: ( ID
action: shift
```

3
```
 input: ( ID = NUM ; ID = NUM )
               |
 stack: ( ID =
action: shift
```

4
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( ID = NUM
action: reduce 3
```


Example

5
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( S
action: reduce 5
```

6
```
 input: ( ID = NUM ; ID = NUM )
                   |
 stack: ( L
action: shift
```

7
```
 input: ( ID = NUM ; ID = NUM )
                     |
 stack: ( L ;
action: shift
```

8
```
 input: ( ID = NUM ; ID = NUM )
                        |
 stack: ( L; ID
action: shift
```


Example

9
```
 input: ( ID = NUM ; ID = NUM )
                          |
 stack: ( L ; ID =
action: shift
```

10
```
 input: ( ID = NUM ; ID = NUM )
                              |
 stack: ( L ; ID = NUM
action: reduce 3
```

11
```
 input: ( ID = NUM ; ID = NUM )
                              |
 stack: ( L ; S
action: reduce 4
```

12
```
 input: ( ID = NUM ; ID = NUM )
                              |
 stack: ( L
action: shift
```

Example

```
        input:  ( ID = NUM ; ID = NUM )
                                       |
13      stack:  ( L )
       action:  reduce 2
------------------------------------------
        input:  ( ID = NUM ; ID = NUM )
                                       |
14      stack:  S
       action:  ACCEPT
```


The Dangling Else Problem

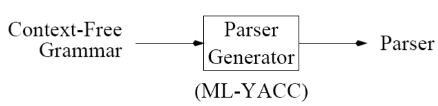
- Valid Program: if a then if b then S1 else S2

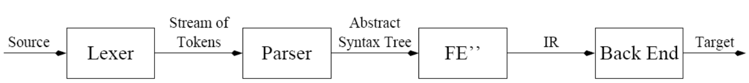
  1 S → *if* E *then* S *else* S
  2 S → *if* E *then* S
  3 S → OTHER

- 2 interpretations: if a then [if b then S1 else S2]
  if a then [if b then S1] else S2
- Want first behavoir, but parse will report *shift-reduce conflict* when S1 is on top stack.
- Eliminate Ambiguity by modifying grammar (matched/unmatched):

  1 S → M
  2 S → U
  3 M → *if* E *then* M *else* M
  4 M → OTHER
  5 U → *if* E *then* S
  6 U → *if* E *then* M *else* U

ML-YACC (Yet Another Compiler-Compiler)





- Input to **ml-yacc** is a context-free grammar specification.
- Output from **ml-yacc** is a shift-reduce parser in ML.

Context-Free Grammar Specification

- CFG specification consists of 3 parts:

```
User Declarations
%%
ML-YACC Definitions
%%
Rules
```

- **User Declarlarions:** define various values that are available to *rules* section.
- **ML-YACC Definitions:** declare terminal and non-terminal symbols; declare precedences for terminals that help resolve shift-reduce conflicts.
- **Rules:** specify productions of grammar and *semantic actions* associated with productions.

ML-YACC Declarations

- Need to specify type associated with positions of tokens in input file

```
%pos int
```

- Need to specify terminal and non-terminal symbols (no symbols can be in both lists)

```
%term IF | THEN | ELSE |...
%nonterm prog | stmt | expr |...
```

- Optionally specify end-of-parse symbol - terminals which may follow start symbol

```
%eop EOF
```

- Optionally specify start symbol - otherwise, LHS non-terminal of first rule is taken as start symbol

```
%start prog
```


Attribute Grammar

- ML-YACC employs *attribute grammar* scheme
  - Each terminal or non-terminal symbol may have associated attribute/value.
  - When parser reduces using production $A \rightarrow \alpha$, semantic action associated with production is exectued in order to compute value for A based on the values of symbols in α.
  - Parser returns value associated with start symbol. (If no attribute, `()` is returned.)
- Can specify *types* of atttributes associtated with symbols.

```
%term     ID of string | NUM of int | IF | THEN | ...
%nonterm  prgm | stmt | expr of int | ...
```

Rules

$symbol_0$: $symbol_1$ $symbol_2$ ... $symbol_n$ (*semantic_action*)

- Semantic action typically builds piece of AST corresponding to derived string
- Can access attribute/value of RHS symbol X using X<n>, where n specifies a particular occurrence of X on RHS.

```
%term    PLUS | MINUS | NUM of int | ...
%nonterm exp of int | ...

exp: exp PLUS exp     (exp1 + exp2)
|    exp MINUS exp    (exp1 - exp2)
|    NUM              (NUM)
```

- Type of value computed by semantic action must match type of value associated with LHS non-terminal.

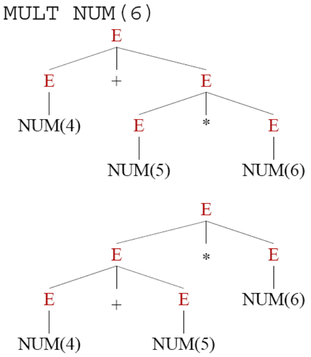
Example

```
%%

%term    ID | NUM | PLUS | MINUS | MULT | DIV | EOF
%nonterm expr
%pos int
%start expr
%eop EOF
%verbose

%%

expr : ID                ()
     | NUM               ()
     | expr PLUS expr    ()
     | expr MINUS expr   ()
     | expr MULT expr    ()
     | expr DIV expr     ()
```


ML-YACC and Ambiguous Grammars

- A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.
- Consider: $4 + 5 * 6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`

$$expr \rightarrow ID$$
$$expr \rightarrow NUM$$
$$expr \rightarrow expr + expr$$
$$expr \rightarrow expr * expr$$



- We perfer to bind “*” tighter than “+”.

ML-YACC and Ambigous Grammars

- Similarly Consider: $4+5+6$, `NUM(4) PLUS NUM(5) PLUS NUM(6)`
- We perfer to bind left "+" first.
- ML-YACC will report *shift-reduce* conflicts when parsing strings.
  - $4+5*6$, `NUM(4) PLUS NUM(5) MULT NUM(6)`
    - At some point, E + E will be on top of stack, "*" will be the current token-type in stream.
    - Parser can reduce by rule E → E + E, or shift. Prefer *shift*.
  - $4+5+6$, `NUM(4) PLUS NUM(5) PLUS NUM(6)`
    - At some point, E + E will be on top of stack, "+" will be the current token-type in stream.
    - Parser can reduce by rule E → E + E, or shift. Prefer *reduce*.

Directives

Three Solutions:

1. Let YACC complain, but demonstrate that its choice (to shift) was correct.
2. Rewrite grammar to eliminate ambiguity.
3. Keep grammar, but add *precedence directives* which enable conflicts to be resolved. Use `%left, %right, %nonassoc`
   - For this grammar:
     ```
     %left PLUS MINUS
     %left MULT DIV
     ```
   - PLUS, MINUS are left associative, bind equally tightly
   - MULT, DIV are left associative, bind equally tightly
   - MULT, DIV bind tighter than PLUS, MINUS

Directives

- Given directives, ML-YACC assigns precedence to each *terminal* and *rule*
  - Precedence of terminal based on order in which associativity specified
  - Precedence of rule is the precedence of right-most terminal. For example, precedence(E → E + E) = precedence(PLUS).
- Given shift-reduce conflict, ML-YACC performs the following:
  1. Find precedence of rule to be reduced, terminal to be shifted.
  2. prec(terminal) > prec(rule) ⇒ shift.
  3. prec(rule) > prec(terminal) ⇒ reduce.
  4. prec(terminal) = prec(rule), then:
     - assoc(terminal) = left ⇒ reduce.
     - assoc(terminal) = right ⇒ shift.
     - assoc(terminal) = nonassoc ⇒ report as error.

Precedence Examples

1

```
  input: 4 + 5 * 6
               |
  stack: 4 + 5
 action: prec(*) > prec(+) -> shift
```

2

```
  input: 4 * 5 + 6
               |
  stack: 4 * 5
 action: prec(*) > prec(+) -> reduce
```

3

```
  input: 4 + 5 + 6
               |
  stack: 4 + 5
 action: assoc(+) = left -> reduce
```


Default Behavior

What if directives not specified?

- shift-reduce: report error, *shift* by default.
- reduce-reduce: report error, reduce by rule that occurs first.

What to do:

- shift-reduce: acceptable in well defined cases (dangling else).
- reduce-reduce: unacceptable. Rewrite grammar.

Direct Rule Precedence Specification

Can assign *specific* precedence to rule, rather than precedence of last terminal.

- Use the %prec directive.
- Commonly used for the *unary minus* problem.

```
%left PLUS MINUS
%left MULT DIV
```

- Consider $-4 * 6$, `MINUS NUM(4) MULT NUM(6)`
- We perfer to bind left unary minus ("-") tighter. Here, precedence of MINUS is lower than MULT, so we get $-(4 * 6)$, not $(-4) * 6$.
- Solution:

```
%left PLUS MINUS
%left MULT DIV
%left UMINUS

exp : MINUS expr %prec UMINUS ()
    | expr PLUS expr ()...
```

Syntax vs. Semantics

Consider language with two classes of expressions

- *Arithmetic* expressions (ae)

```
ae : ae PLUS ae  ()
   | ID          ()
```

- *Boolean* expressions (be)

```
be : be AND be  ()
   | be OR be   ()
   | be EQ be   ()
   | ID         ()
```

- Consider: `a := b`, ID(a) ASSIGN ID(b):
  - Reduce-reduce conflict - parser can't choose between be → ID or ae → ID.
  - For now ae and be should be aliased - let semantic analysis (next phase) determine that `a & b + c` is a type error.
  - Type checking cannot be done easily in context free grammars.

Recursive Descent/Predictive/LL(1) Parser Generation

Grammar:

A → S EOF
S → *id* := E
S → *print* (L)

E → *id*
E → *num*
L → E
L → L, E

```
fun A() =              (S(); eat(EOF))
and S() = case !tok of
            ID     => (eat(ID); eat(ASSIGN); E())
            PRINT  => (eat(PRINT); eat(LPAREN);
                        L(); eat(RPAREN))
and E() = case !tok of
            ID     => (eat(ID))
            NUM    => (eat(NUM))
and L() = case !tok of
            ID     => (?????)
            NUM    => (?????)
```


Problem

- Based on current function and next token-type in input stream, parser must predict which production to use.
- If !tok = ID, parser cannot determine which production to use:
  L → E (E could be ID)
  L → L, E (L could be ID)
- Predictive parsing only works for grammars where first terminal symbol of each subexpression provides enough information to choose which production to use.

Formal Techniques

Can use formal techniques to determine whether or not a predictive parser can be built for a particular grammar.

- Let γ be a string of terminal and non-terminal symbols
- Need to compute 3 values:
  1. For each γ corresponding to RHS of production, must determine if γ can derive empty string (ϵ) $\Rightarrow$ **nullable**.
  2. For each γ corresponding to RHS of production, must determine set of all terminal symbols that can begin any string derived from $\gamma \Rightarrow$ **first(γ)**.
  3. For each non-terminal X in grammar, must determine set of all terminal symbols that can immediately follow X in a derivation $\Rightarrow$ **follow(X)**.

  Computation of Nullable:

  – γ is nullable if every symbol S $\in \gamma$ is nullable.

  – Check if every S can derive ϵ.

Computation of First

- If T is a terminal symbol, then first(T) = $\{T\}$.
- If X is a non-terminal and $X \rightarrow Y_1Y_2Y_3...Y_n$, then

$$\begin{array}{l}\text{first}(Y_1) \in \text{first}(X)\\ \text{first}(Y_2) \in \text{first}(X), \text{ if } Y_1 \text{ is nullable}\\ \text{first}(Y_3) \in \text{first}(X), \text{ if } Y_1, Y_2 \text{ is nullable}\\ \vdots\\ \text{first}(Y_n) \in \text{first}(X), \text{ if } Y_1, Y_2, \ldots\ Y_{n-1} \text{ is nullable}\end{array}$$

- Let $\gamma = S_1S_2...S_k$. Then,

$$\text{first}(\gamma) = \begin{cases}\text{first}(S_1)\\ \text{first}(S_2), \text{if } S_1 \text{ is nullable}\\ \text{first}(S_3), \text{if } S_1, S_2 \text{ is nullable}\\ \vdots\\ \text{first}(S_k), \text{if } S_1, S_2, ..., S_{k-1} \text{ is nullable}\end{cases}$$

Computation of Follow

Let X, Y be non-terminals; γ, γ_1, and γ_2 be strings of terminals and non-terminals

- if grammar includes production: $X \rightarrow \gamma Y$
  $\Rightarrow$ follow(X) $\in$ follow (Y).
- if grammar includes production: $X \rightarrow \gamma_1 Y \gamma_2$
  $\Rightarrow$ first (γ_2) $\in$ follow (Y)
  $\Rightarrow$ follow (X) $\in$ follow (Y), if γ_2 is nullable.

Perform *iterative* technique in order to compute nullable, first, and follow sets for each non-terminal in grammar.

Building a Predictive Parser

$Z \to XYZ$ $\quad$ $Y \to$ c $\quad$ $X \to$ a

$Z \to$ d $\quad$ $Y \to \epsilon$ $\quad$ $X \to$ b Y e

Initial:

	nullable	first	follow
Z	no		
Y	no		
X	no		

Examine each production in grammar, modifying nullable and adding to first and follow sets, until no more changes can be made.

Iteration 1:

	nullable	first	follow
Z	no		
Y	no		
X	no		

Building a Predictive Parser

$Z \to XYZ$ $\quad$ $Y \to$ c $\quad$ $X \to$ a

$Z \to$ d $\quad$ $Y \to \epsilon$ $\quad$ $X \to$ b Y e

Iteration 1:

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Iteration 2:

	nullable	first	follow
Z	no		
Y	yes		
X	no		

Iteration 3:

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

No Changes

Predictive Parsing Table

	nullable	first	follow
Z	no	d,a,b	
Y	yes	c	e,d,a,b
X	no	a,b	c,d,a,b

Build *predictive parsing table* from nullable, first, and follow sets.

	a	b	c	d	e
Z	$Z \to XYZ$	$Z \to XYZ$		$Z \to d$	
Y	$Y \to \epsilon$	$Y \to \epsilon$	$Y \to c$	$Y \to \epsilon$	$Y \to \epsilon$
X	$X \to a$	$X \to bYe$			

- Enter $S \to \gamma$ in row S, column T: for each $T \in$ first(γ).
- If γ is nullable, enter $S \to \gamma$ in row S, column T: for each $T \in$ follow(S).
- Entry in row S, column T tells parser which clause to execute if current function is $S()$ and next token-type is T
- Blank entries are syntax errors.

Predictive Parsing Table

If the predictive parsing table contains *no* duplicate entries, can build predictive parser for grammar.

- Grammar is LL(1) (left-to-right parse, left-most derivation, 1 symbol lookahead).
- Grammar is LL(k) if its LL(k) predictive parsing table has no duplicate entries.
  - Rows correspond to non-terminals, columns correspond to every possible sequence of k terminals.
  - The first(γ) = set of all k-length terminal sequences that can begin any string derived from γ.
  - LL(k) paring tables can be too large.
  - Ambiguous grammars are not LL(k), $\forall$ k.

Example

$S' \rightarrow S\$$ $S \rightarrow$ IF E THEN A ELSE A $T \rightarrow$ NUM
$S \rightarrow E$ $E \rightarrow E + T$ $A \rightarrow$ ID = NUM
$S \rightarrow$ IF E THEN A $E \rightarrow T$

Iteration 1:

	nullable	first	follow
S'	no		
S	no	IF	$
E	no		$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Iteration 2:

	nullable	first	follow
S'	no	IF	
S	no	IF	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Example

$S' \rightarrow S\$$ $S \rightarrow$ IF E THEN A ELSE A $T \rightarrow$ NUM
$S \rightarrow E$ $E \rightarrow E + T$ $A \rightarrow$ ID = NUM
$S \rightarrow$ IF E THEN A $E \rightarrow T$

Iteration 3:

	nullable	first	follow
S'	no	IF	
S	no	IF, NUM	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

Iteration 4:

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	$
E	no	NUM	$, THEN, +
T	no	NUM	$, THEN, +
A	no	ID	$, ELSE

No futher changes

Predictive Parsing Table

	nullable	first	follow
S'	no	IF, NUM	
S	no	IF, NUM	\$
E	no	NUM	\$, THEN, +
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

Build *predictive parsing table* from nullable, first, and follow sets.

	IF	THEN	ELSE	+	NUM	ID	=	\$
S'	$S' \rightarrow S$				$S' \rightarrow S$			
S	$S \rightarrow$ IF E THEN A $S \rightarrow$ IF E THEN A ELSE A				$S \rightarrow E$			
E					$E \rightarrow E + T$ $E \rightarrow T$			
T					$T \rightarrow$ NUM			
A						$A \rightarrow$ ID = NUM		

Table has duplicate entries $\Rightarrow$ grammar is not LL(1)!

Problems

1. $E \rightarrow E + T$
   $E \rightarrow T$

- first(E+T) = first(T)
- When in function `E()`, if next token is NUM, parser will get stuck.
- Grammar is *left-recursive* - left-recursive grammars cannot be LL(1).
- Solution: rewrite grammar so that it is *right-recursive*.
  $E \rightarrow TE'$
  $E' \rightarrow \epsilon$
  $E' \rightarrow +TE'$
- In general, $X \rightarrow X\gamma$, $X \rightarrow \alpha$ derives strings of form $\alpha\gamma^*$ (α doesn't start with X).
  These two productions can be rewritten as follows:
  $X \rightarrow \alpha X'$
  $X' \rightarrow \epsilon$
  $X' \rightarrow \gamma X'$

Problems

2. $S \rightarrow$ IF E THEN A
   $S \rightarrow$ IF E THEN A ELSE A

- Two productions begin with same symbol.
- first(IF E THEN A) = first(IF E THEN A ELSE A)
- Solution: use *left-factoring*
  $S \rightarrow$ IF E THEN A V
  $V \rightarrow \epsilon$
  $V \rightarrow$ ELSE A

Example

Show that modified grammar is LL(1).

$S' \to S\$$	$V \to$ ELSE A	$T \to$ NUM
$S \to E$	$E \to TE'$	$A \to$ ID = NUM
$S \to$ IF E THEN A V	$E' \to \epsilon$	
$V \to \epsilon$	$E' \to +TE$	

Example

Show that the grammar is LL(1).

Example

Show that modified grammar is LL(1). Build predictive parsing table.

	nullable	first	follow
S'	no	IF,NUM	
S	no	IF,NUM	\$
V	yes	ELSE	\$
E	no	NUM	\$, THEN
E'	yes	+	\$, THEN
T	no	NUM	\$, THEN, +
A	no	ID	\$, ELSE

	IF	THEN	ELSE	+	NUM	ID	=	\$
S'	$S' \to S$				$S' \to S$			
S	$S \to$ IF E THEN A V				$S \to E$			
V			$V \to$ ELSE A					$V \to \epsilon$
E					$E \to TE'$			
E'		$E' \to \epsilon$		$E' \to +TE'$				$E' \to \epsilon$
T					$T \to$ NUM			
A						$A \to$ ID = NUM		

Table does not have duplicate entries $\Rightarrow$ modified grammar is LL(1)!

Outline

- LR(0)
- SLR
- LR(1)
- LALR(1)

Shift-Reduce, Bottom Up, LR(1) Parsing

- Shift-reduce parsing can parse more grammars than predictive parsing.
- *Shift-reduce parsing* has stack and input.
- Based on stack contents and next input token, one of two action performed:
  1. *Shift* - push next input token onto top of stack.
  2. *Reduce* - choose production (X $\rightarrow$ ABC); pop off RHS (C, B, A); push LHS (X).
- If $ is shifted, then input stream has been parsed successfully.

LR(k)

Can generalize to case where parser makes decision based on stack contents and next k tokens. LR(k):

- Left-to-right parse
- right-most derivation
- k-symbol lookahead

LR(k) parsing, $k > 1$, rarely used in compilation:

- DFA too large: need transition for every sequence of k terminals.
- Most programming languages can be described by LR(1) grammars.

Shift Reduce Parsing DFA

Parser uses DFA to make shift/reduce decisions:

- Each state corresponds to contents of stack at some point in time.
- Edges labeled with terminals/non-terminals.

Rather than scanning entire stack to determine current DFA state, parser can remember state reached for each stack element.

- Transition table for LR(1) or LR(0) DFA:

	Terminals $(T_1, T_2, \ldots, T_n)$	Non-Terminals $(N_1, N_2, \ldots, N_n)$
1	*actions*	*actions*
2	sn $\longrightarrow$ shift n	gz $\longrightarrow$ goto z
3	rk $\longrightarrow$ reduce k	
⋮	a $\longrightarrow$ accept	
n	$\longrightarrow$ error	

Parsing Algorithm

Look up DFA state on top of stack, next terminal in input:

- shift(n):
  - Advance input by one.
  - Push input token on stack with n (the new state).
- reduce(k):
  - Pop stack as many times as number of symbols on RHS of rule k.
  - Let X be LHS of rule k
  - In state now on top of stack, look up X to get goto(z)
  - Push X on stack with z (the new state).
- accept $\longrightarrow$ stop, report success.
- error $\longrightarrow$ stop, report syntax error.

To understand LR(k) parsing, first focus on LR(0) parser construction using an example.

LR(0) Parsing

1 $S' \longrightarrow S$ \$
2 $S \longrightarrow (L)$
3 $S \longrightarrow$ x
4 $L \longrightarrow S$
5 $L \longrightarrow L, S$

Initially, stack empty, input contains 'S' string followed by a '\$':

1
$S' \longrightarrow .S\$$
$S \longrightarrow .(L)$
$S \longrightarrow .\text{x}$

- Combination of production and '.' called LR(0) *item*.
- '.' specifies parser position.
- Three items represent *closure* of: $S' \longrightarrow .S$ \$
- Closure adds more items to a set when dot exists to left of a non-terminal.

LR(0) States

LR(0) Parsing

1 $S' \rightarrow S\,\$$	3 $S \rightarrow$ x	5 $L \rightarrow L, S$
2 $S \rightarrow (\,L\,)$	4 $L \rightarrow S$	

LR(0) states:

DFA Table Entry Computation

To compute transition table from state diagram perform the following:

- $^{i}\boxed{S' \rightarrow S.\$} \Rightarrow$ table[i, \$] = a.
- $^{i}\boxed{} \xrightarrow{T} {}^{j}\boxed{}$, Terminal $T \Rightarrow$ table[i, T] = sj.
- $^{i}\boxed{} \xrightarrow{N} {}^{j}\boxed{}$, Non-Terminal $N \Rightarrow$ table[i, N] = gj.
- $^{i}\boxed{A \rightarrow \gamma.} \Rightarrow$ table[i, T] = rk, for all terminals T.

Transition Table

	(	)	x	,	\$	S'	S	L
1								
2								
3								
4								
5								
6								
7								
8								
9								

No duplicate entries $\Rightarrow$ grammar is LR(0)

Using The Transition Table

$1\ S' \rightarrow S\,\$$ $\quad$ $3\ S \rightarrow \mathrm{x}$ $\quad$ $5\ L \rightarrow L, S$

$2\ S \rightarrow (L)$ $\quad$ $4\ L \rightarrow S$

	(	)	x	,	$	S'	S	L
1	s3		s2				g9	
2	r3	r3	r3	r3	r3			
3	s3		s2				g5	g4
4		s6		s7				
5	r4	r4	r4	r4	r4			
6	r2	r2	r2	r2	r2			
7	s3		s2				g8	
8	r5	r5	r5	r5	r5			
9					a			

```
STACK                  INPUT   ACTION
1              ( x , x ) $    shift 3
1 (3             x , x ) $    shift 2
1 (3 x2            , x ) $    reduce 3
1 (3 S5            , x ) $    reduce 4
1 (3 L4            , x ) $    shift 7
1 (3 L4 ,7           x ) $    shift 2
1 (3 L4 ,7 x2          ) $    reduce 4
1 (3 L4 ,7 S8          ) $    reduce 5
1 (3 L4                ) $    shift 6
1 (3 L4 )6               $    reduce 2
1 S9                     $    accept
```


Another Example

$1\ S' \rightarrow E\,\$$ $\quad$ $3\ E \rightarrow T$ $\quad$ $4\ T \rightarrow \mathrm{x}$

$2\ E \rightarrow T + E$

LR(0) states:

Another Example - SLR

Transition Table:

	+	x	$	S'	E	T
1		s3			g2	g4
2			a			
3	r4	r4	r4			
4	s5/r3	r3	r3			
5		s3			g6	g4
6	r2	r2	r2			

Duplicate entries $\Rightarrow$ grammar is NOT LR(0)
Can make grammar bottom-up parsable using more powerful parsing techniques: **SLR** (Simple LR)

- Use same LR(0) states.
- $\boxed{{}^{i}\,A \rightarrow \gamma.}$ $\Rightarrow$ table[i,T] = reduce(k), for all terminals $T \in$ follow(A)

Another Example – SLR

Transition Table:

	+	x	$	S'	E	T
1		s3			g2	g4
2			a			
3	r4	r4	r4			
4	s5/r3	r3	r3			
5		s3			g6	g4
6	r2	r2	r2			

Follow Set Computation:

	nullable	first	follow
S'	no	x	
E	no	x	$
T	no	x	+,$

$1\ S' \rightarrow E\ \$$ $3\ E \rightarrow T$

$2\ E \rightarrow T + E$ $4\ T \rightarrow$ x

SLR Transition Table:

	+	x	$	S'	E	T
1		s3			g2	g4
2			a			
3	r4		r4			
4	s5		r3			
5		s3			g6	g4
6			r2			

No duplicate entries ⇒ grammar is SLR.

Yet Another Example

Sometimes grammar can't be parsed using SLR techniques.

$1\ S' \rightarrow S\ \$$ $3\ S \rightarrow E$ $5\ V \rightarrow$ x

$2\ S \rightarrow V = E$ $4\ E \rightarrow V$ $6\ V \rightarrow *\ E$

This grammar is not SLR. Need more powerful parsing algorithm ⇒ LR(1)

LR(1) Parsing

- LR(1) item consists of two components: ($A \rightarrow \alpha.\beta$, x)
  1. Production
  2. Lookahead symbol (x)
- α is on top of stack, head of input is string derivable from β x.

LR(0) closure computation

- Initial: $A \rightarrow \alpha.X$
- Add all items $X \rightarrow .\gamma$
- Repeat closure computation

LR(1) closure computation

- Initial: $A \rightarrow \alpha.X\beta,$ z
- Add all items ($X \rightarrow .\gamma, \omega$) for each $\omega \in$ first(βz)
- Repeat closure computation

- shift, goto, accept table entries computed same way as LR(0)/SLR.
- reduce entries computed differently:

$^{i}\boxed{A \rightarrow \gamma.\ ,\ \text{z}}$ ⇒ table[i,z] = reduce(k)

Yet Another Example – LR(1)

1 $S' \to S\ \$$	3 $S \to E$	5 $V \to$ x
2 $S \to V = E$	4 $E \to V$	6 $V \to$ * E

LR(1) states:

Yet Another Example – LR(1)

	=	x	*	$	S'	S	L	V
1		s11	s12			g2	g10	g3
2				a				
3	s4			r4				
4		s7	s8				g5	g6
5				r2				
6				r4				
7				r5				
8		s7	s8				g9	g6
9				r6				
10				r3				
11	r5			r5				
12		s11	s12				g13	g14
13	r6			r6				
14	r4			r4				

No duplicate entries $\Rightarrow$ grammar is LR(1)

LALR(1)

- Problem with LR(1) parsers: tables too large!
  - Can make smaller table by merging states whose items are identical except for look-ahead sets $\Rightarrow$ LALR(1) (Look-Ahead LR(1)).
  - LALR(1) transition table may contain shift-reduce/reduce-reduce conflicts where LR(1) table has none.

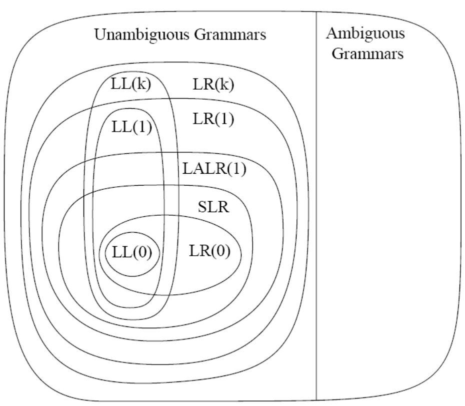
LALR(1)

Can make smaller table by merging states whose items are identical except for look-ahead sets ⇒ LALR(1) (Look-Ahead LR(1)).

	=	x	*	\$	S'	S	L	V
1		s11	s12			g2	g10	g3
2				a				
3	s4			r4				
4		s7	s8				g5	g6
5				r2				
6/14	r4			r4				
7/11	r5			r5				
8/12		s7/11	s8/12				g9/13	g6/14
9/13	r6			r6				
10				r3				

No conflicts ⇒ grammar is LALR(1).

Parsing Power



ML-YACC uses LALR(1) parsing because reasonable programming languages can be specified by an LALR(1) grammar. (Figure from MCI in ML.)

Parsing Error Recovery

Syntax Errors:

- A *Syntax Error* occurs when stream of tokens is an invalid string.
- In LL(k) or LR(k) parsing tables, blank entries refer to syntax errors.

How should syntax errors be handled?

1. Report error, terminate compilation ⇒ not user friendly
2. Report error, *recover* from error, search for more errors ⇒ better

Error Recovery

***Error Recovery*: process of adjusting input stream so that parsing may resume after syntax error reported.**

- Deletion of token types from input stream
- Insertion of token types
- Substitution of token types

Two classes of recovery:

1. *Local Recovery*: adjust input at point where error was detected.
2. *Global Recovery*: adjust input *before* point where error was detected.

These may be applied to both LL and LR parsing techniques.

LL Local Error Recovery

Consider LL(1) parsing context:

$Z \rightarrow XYZ$	$Y \rightarrow$ c	$X \rightarrow$ a
$Z \rightarrow$ d	$Y \rightarrow \epsilon$	$X \rightarrow$ b Y e

	nullable	first	follow
Z	no	a,b,d	
Y	yes	c	a,b,d,e
X	no	a,b	a,b,c,d

	a	b	c	d	e
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$		$Z \rightarrow d$	
Y	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$	$Y \rightarrow c$	$Y \rightarrow \epsilon$	$Y \rightarrow \epsilon$
X	$X \rightarrow a$	$X \rightarrow bYe$			

LL Local Error Recovery

Local Recovery Technique: in function A(), delete token types from input stream until token type in follow(A) found ⇒ *synchronizing* token types.

```
datatype token = a | b | c | d | e;
val tok = ref(getToken());
fun advance() = tok := getToken();
fun eat(t) = if(!tok = t) then advance() else error();
...
and X() = case !tok of
      a => (eat(a))
    | b => (eat(b); Y(); eat(e))
    | c => (print "error!"; skipTo[a,b,c,d])
    | d => (print "error!"; skipTo[a,b,c,d])
    | e => (print "error!"; skipTo[a,b,c,d])

and skipTo(synchTokens) =
    if member(!tok, synchTokens) then ()
    else (eat(!tok); skipTo(synchTokens))
```

LR Local Error Recovery

Consider:

$1\ E \rightarrow$ ID $\quad 3\ E \rightarrow (\ E\ ) \quad 5\ ES \rightarrow ES\ ;\ E$

$2\ E \rightarrow E + E \quad 4\ ES \rightarrow E$

- Match a sequence of erroneous input tokens using the *error* token (a terminal).

  $6\ E \rightarrow$ (error) $\quad 7\ ES \rightarrow$ error ; E

- In general, follow *error* with synchronizing lookahead token.
  1. Pop stack (if necessary) until a state is reached in which the action for the *error* token is *shift*.
  2. Shift the *error* token.
  3. Discard input symbols (if necessary) until a state is reached that has a non-error action in the current state.
  4. Resume normal parsing.

Global Error Recovery

Consider LR(1) parsing:

```
let type a := intArray[10] of 0 in ... end
```

Local Recovery Techniques would:

1. report syntax error at ':='
2. substitute '=' for ':='
3. report syntax error at '['
4. delete token types from input stream, synchronizing on 'in'

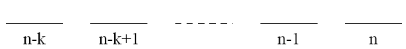
Global Recovery Techniques would substitute 'var' for 'type':

- Actual syntax error occurs *before* point where error was detected.
- ML-Yacc uses global error recovery technique ⇒ *Burke-Fisher*
- Other Yacc versions employ local recovery techniques.

Burke-Fisher

Suppose parser gets stuck at n^{th} token in input stream.

- Burke-Fisher repairer tries every *single-token-type* insertion, deletion, and substitution at all points between $(n-k)^{th}$ and n^{th} token.



- Best repair: one that allows parser to parse furthest past n^{th} token.
- If languages has N token types, then:

  total # of repairs = deletions + insertions + substitutions

  total # of repairs = $(k) + (k+1)N + (k)(N-1)$

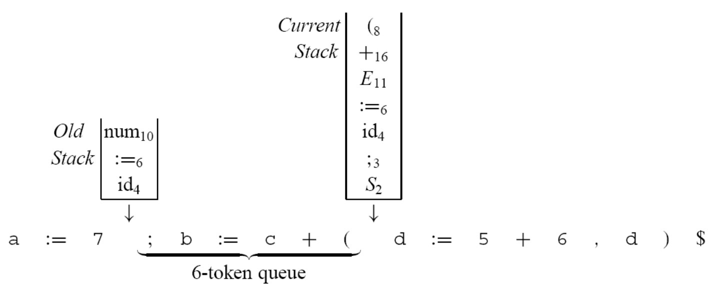
Burke-Fisher

In order to backup K tokens and reparse repaired input, 2 structures needed:

1. *k-length buffer/queue* - if parser currently processing n^{th} token, queue contains tokens $(n-k) \rightarrow (n-1)$. (ML-Yacc $k = 15$)
2. *old parse stack* - if parser currently processing n^{th} token, old stack represents stack state when parser was processing $(n-k)^{th}$ token.

- Whenever token shifted onto current stack, also put onto queue tail.
- Simultaneously, queue head removed, shifted onto old stack.
- Whenever token shifted onto either stack, appropriate reductions performed.

Burke-Fisher Example



- Semantic actions are only applied to old stack.
  - Not desirable if semantic actions affect lexical analysis.
  - Example: `typedef` in C.

(Figure from MCI/ML.)

Burke-Fisher

For each repair R that can be applied to token $(n-k) \rightarrow n$:

1. copy queue, copy n^{th} token
2. copy old parse stack
3. apply R to copy of queue or copy of n^{th} token
4. reparse queue copy (and copy of n^{th} token) from old stack copy
5. evaluate R

Choose best repair R, and apply.

Burke-Fisher in ML-YACC

Semantic Values

- Insertions need semantic values

```
%value ID {"bogus"}
%value INT {1}
%value STRING {"STRING")
```

Programmer-Specified Substitutions

- Some single token insertions and deletions are common.
- Some multiple token insertions and deletions are common.

```
%change EQ -> ASSIGN | SEMICOLON ELSE -> ELSE
        |   -> IN INT END
```