Parsing

Victor Eijkhout

Notes for CS 594 – Fall 2004

What is parsing?

- Check for correctness: is this a legal program
- Uncover meaning: convert to internal representation

Levels of parsing

- Check for illegal characters
- Build tokens (identifiers, numbers, operators &c) from characters (lexical analysis)
- Statements tokens (syntactical analysis)
- Semantical restrictions: define/use &c

```
my_array[ii] = 3+sin(1.0);
```

- Lexical analysis: '`my_array`', '`[`', '`ii`' &c.
- Syntactical: this is an assignment; lhs is something you can assign to, rhs is arithmetic expression
- Semantics: `my_array` is array, `ii` is integer, `sin` is defined function

Mixing of levels

In Fortran:

```
X = SOMENAME( Y+3 )
```

- Lexical analysis simple
- Syntax unclear: rhs can be function call or array element
- Solution: give lexer access to symbol table

Correctness

- Lexical analysis finds identifiers: `5ab` is illegal
- Syntactical analysis finds expressions: `array[ii)` is illegal
- In TEX?

Parsing by automaton

- Lexical analysis by Finite State Automaton
- Syntactical analysis by Pushdown Automaton
- In practice some mixing of levels

Terminology

- Language: a set of words

$$\{a^n | n \text{ is prime}\}$$

- Grammar: set of rules that produces a language
- Automaton: abstract device that can recognize a language
- Derivation: actual sequency of rules or transitions used to derive a string
- Parse tree: 2D way of writing derivation

to be precise

- Grammar:
  - Start symbol S
  - Terminal symbols $a, b, c, \ldots$ from the alphabet
  - Non-terminals $A, B, C, \ldots$, ultimately to be replaced
  - Rules $\alpha \rightarrow \beta$ where α, β strings of terminals and non-terminals
- Automaton:
  - Starting state
  - Accepting state
  - Work storage
  - Transition diagram

Types of languages

- Languages differ in types of grammar rules $\alpha \rightarrow \beta$
- Automata differ in amount of workspace
- Four levels Chomsky hierarchy; many other types

Type 0

- Regular languages

Type 0

- Regular languages
- Turing machine: infinite tape
- No restriction on grammar rules

Type 1

- Context-sensitive languages

Type 1

- Context-sensitive languages
- Linear-bounded automata
- No rules $\alpha \rightarrow \epsilon$

Type 1

- Context-sensitive languages
- Linear-bounded automata
- No rules $\alpha \to \epsilon$
- Normal form: $AB \to BA$, $AB \to A\beta$

Type 2

- Context-free languages

Type 2

- Context-free languages
- Push Down Automata
- Only rules $A \to \alpha$

Type 2

- Context-free languages
- Push Down Automata
- Only rules $A \rightarrow \alpha$
- Normal form: $A \rightarrow b\alpha$ or $A \rightarrow b$

Type 3

- Regular languages

Type 3

- Regular languages
- Finite State Automata
- Only rules $A \to bC$, $A \to b$

Lexical analysis

Function of a lexer

- Recognize identifiers, numbers
- Also side effects: store names of functions

Definition

Inductively, through regular expressions

- ϵ is the empty language
- 'a' denotes the language $\{a\}$ (a in alphabet)
- if α, β denote languages A, B, then
  - $\alpha\beta$ or $\alpha \cdot \beta$ denotes $\{xy|x \in A, y \in B\}$
  - $\alpha|\beta$ denotes the language $A \cup B$.
  - α^* denotes the language $\cup_{n\geq 0} A^n$.

Finite state automata

- Starting state S_0
- other states S_i; subset: accepting states
- input alphabet I; output alphabet O
- transition diagram $I \times S \to S$

Finite state automata

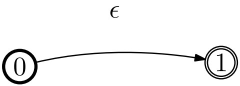
- Starting state S_0
- other states S_i; subset: accepting states
- input alphabet I; output alphabet O
- transition diagram $I \times S \to S$
- non-deterministic: $I \cup \{\epsilon\} \times S \to S$

Finite state automata

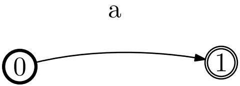
- Starting state S_0
- other states S_i; subset: accepting states
- input alphabet I; output alphabet O
- transition diagram $I \times S \to S$
- non-deterministic: $I \cup \{\epsilon\} \times S \to S$
- String is accepted if (any) sequence of transitions it causes leads to an accepting state

The NFA of a regular language

Automaton that accepts ϵ:

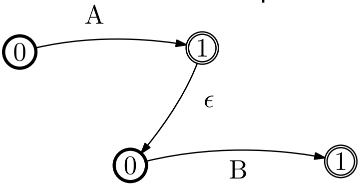


Automaton that accepts *a*:



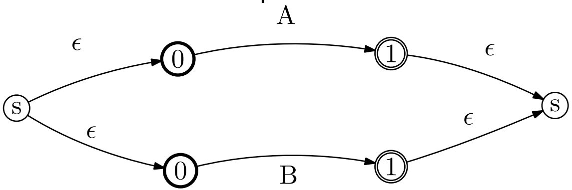
The NFA of a regular language

Automaton that accepts $A \cdot B$:



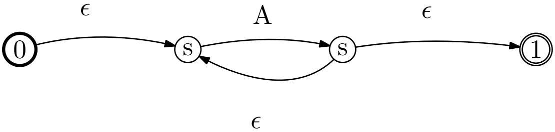
The NFA of a regular language

Automaton that accepts $A \cup B$:



The NFA of a regular language

Automaton that accepts A^*:

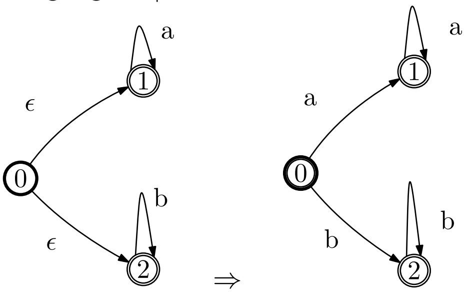


Characterization

- Any sufficiently long string $\alpha = uvw$
- then uv^nw also in the language

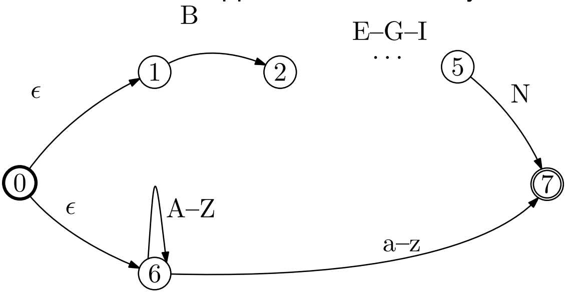
Example

Language $a^*|b^*$:

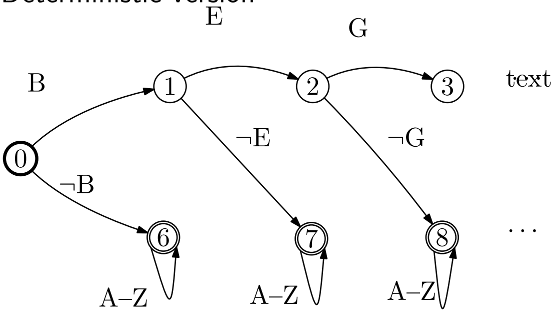


Example: keywords

A bit like what happens in lexical analysis:



Deterministic version



Converting NFA to DFA

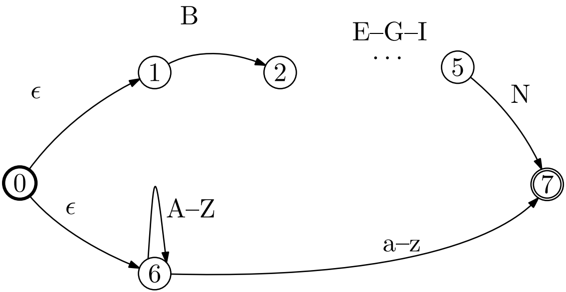
- Introduce new states

Converting NFA to DFA

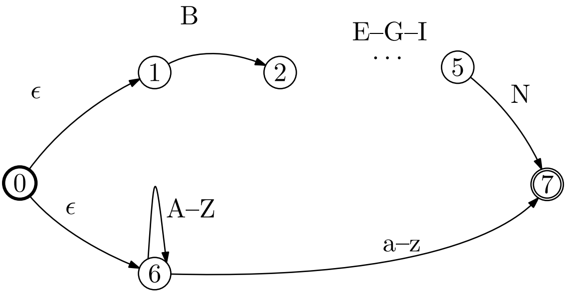
- Introduce new states
- new state is set of old states

Converting NFA to DFA

- Introduce new states
- new state is set of old states
- new states closed under ϵ-transitions

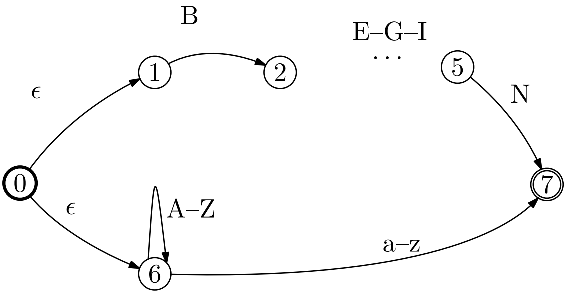


New $S_0 = \{0, 1, 6\}$



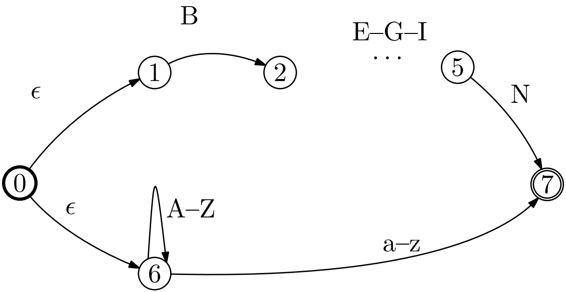
New $S_0 = \{0, 1, 6\}$

- $S_0 + \mathtt{B} \Rightarrow S_1 = \{2, 6, 7\}$,



New $S_0 = \{0, 1, 6\}$

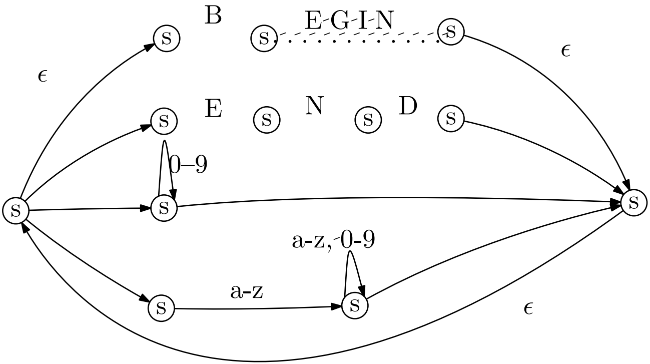
- $S_0 + \mathtt{B} \Rightarrow S_1 = \{2, 6, 7\}$,
- $S_0 + \neg\mathtt{B} \Rightarrow S_6 = \{6, 7\}$



New $S_0 = \{0, 1, 6\}$

- $S_0 + \mathtt{B} \Rightarrow S_1 = \{2, 6, 7\}$,
- $S_0 + \neg\mathtt{B} \Rightarrow S_6 = \{6, 7\}$
- $S_1 + \mathtt{E} \Rightarrow S_2 = \{3, 6, 7\}$, et cetera

NFA for lexical analysis



small problems

- Careful with the ϵ-transition back:

  ```
  printf("And then he said ""Boo!""");
  ```

  final state reached three times: only transition when maximum string recognized
- Not always:

  ```
  X = 4.E3
  IF (4.EQ.VAR) THEN
  ```

  $\Rightarrow$ look-ahead needed

lex

A tool for lexical analysis

- You write regular expressions, and *lex* reports if it finds any
- Three sections: definitions, rules, code

Example

```
%{
 int charcount=0,linecount=0;
%}

%%

. charcount++;
\n {linecount++; charcount++;}

%%
int main()
{
  yylex();
  printf("There were %d characters in %d lines\n",
         charcount,linecount);
  return 0;
}
```

Running lex

lex code gets translated to C:

```
lex -t count.l > count.c
cc -c -o count.o count.c
cc -o counter count.o -ll
```

Executable uses stdin/out, can be changed

Definitions section

- C code: between `%{ ... %}` copied to top of C file
- Definitions: '`letter [a-zA-Z]`' (like `#define`)
- State definitions (later)

Example 2

```
%{
 int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]

%%

{letter}+ {wordcount++; charcount+=yyleng;}
.          charcount++;
\n         {linecount++; charcount++;}
```

Rules section

- Input is matched by character
- Actions of longest match are taken, earliest if equal length
- Matched text is `char *yytext`, length `int yyleng`

Example 2'

```
{letter}+ {wordcount++; charcount+=yyleng;}
[ \t]     spacecount++;
.         charcount++;
\n        linecount++;
```

Example 3

```
[0-9]+              process_integer();
[0-9]+\.[0-9]*      |
\.[0-9]+            process_real();
```

Regular expressions

.	Match any character except newlines.
\n	A newline character.
\t	A tab character.
^	The beginning of the line.
$	The end of the line.
<expr>*	Zero or more occurrences of the expression.
<expr>+	One or more occurrences of the expression.
<expr>?	Zero or one occurrences of the expression.
(<expr1>|<expr2>)	One expression of another.
[<set>]	A set of characters or ranges, such as [,.:;] or [a-zA-Z].
[^<set>]	The complement of the set, for instance [^ \t].

Example: filtering comments

```
%%
"/*".*"*/"   ;
.            |
\n           ECHO;
```

works on

```
This text /* has a */ comment
in it
```

Does not work on

```
This text /* has */ a /* comment */ in it
```

Context

- Match in context
- Left context implemented through states:

  `<STATE>(some pattern) {...`

  State switching:

  `<STATE>(some pattern) {some action; BEGIN OTHERSTATE;}`

  Initial state is `INITIAL`, other states defined

  `%s MYSTATE`

  `%x MYSTATE`

Use of states

```
%x COMM

%%

.             |
\n            ECHO;
"/*"          BEGIN COMM;
<COMM>"*/"    BEGIN INITIAL;
<COMM>.       |
<COMM>\n      ;

%%
```

Context'

- Right context:

  `abc/de {some action}`
- context tokens not in `yytext`/`yyleng`

Example: text cleanup

Input:

```
This    text (all of it  )has occasional lapses , in
 punctuation(sometimes pretty bad) ,( sometimes not so).


(Ha! ) Is this : fun?Or what!
```

Solution with context more compact than without.
Define:

```
punct [,.;:!?]
text [a-zA-Z]
```

need for context

- Consider ‘`),`’ ‘`) ,`’ ‘`)a`’ ‘`) a`’
- Rules `")" " "+ {printf(")");}` depend on context

right context solution

```
")"" "+/{punct}         {printf(")");}
")"/{text}              {printf(") ");}
{text}+" "+/")"         {while (yytext[yyleng-1]==' ') yyleng--; ECHO;}

({punct}|{text}+)/"(" {ECHO; printf(" ");}
"("" "+/{text}          {while (yytext[yyleng-1]==' ') yyleng--; ECHO;}

{text}+" "+/{punct}     {while (yytext[yyleng-1]==' ') yyleng--; ECHO;}

^" "+                   ;
" "+                    {printf(" ");}
.                       {ECHO;}
\n/\n\n                 ;
\n                      {ECHO;}
```

left context solution

Use defined states:

```
punct [,.;:!?]
text [a-zA-Z]

%s OPEN
%s CLOSE
%s TEXT
%s PUNCT
```

left context solution, cont'd

```
" "+ ;

<INITIAL>"(" {ECHO; BEGIN OPEN;}
<TEXT>"("    |
<PUNCT>"("   {printf(" "); ECHO; BEGIN OPEN;}

")" {ECHO ; BEGIN CLOSE;}

<INITIAL>{text}+ |
<OPEN>{text}+    {ECHO; BEGIN TEXT;}
<CLOSE>{text}+   |
<TEXT>{text}+    |
<PUNCT>{text}+   {printf(" "); ECHO; BEGIN TEXT;}

{punct}+ {ECHO; BEGIN PUNCT;}

\n {ECHO; BEGIN INITIAL;}
```

Syntactical analysis

Function of syntactical analysis

- Recognize statements: loops, assignments &c
- Convert to internal representation: parse trees

- Semantics: define/use sequence &c

Grammars

- Backus Naur, or other formalism
- In LaTeX: `bnf.sty`

  ```
  \begin{bnf}
  Expr: number Tail.
  Tail: $\epsilon$ ; + number Tail; * number Tail
  \end{bnf}
  ```

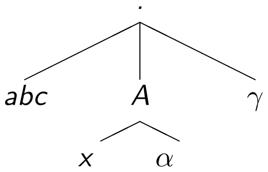
  Output:

  $$\begin{aligned} \mathit{Expr} &\longrightarrow \mathit{number\ Tail} \\ \mathit{Tail} &\longrightarrow \epsilon \;\mid\; + \mathit{number\ Tail} \\ &\mid\; * \mathit{number\ Tail} \end{aligned}$$

  (use my `bnf.env`)
- most language constructs are context-free

Concepts

- Grammar rules $A \rightarrow x\alpha$
- Derivations $abcA\gamma \Rightarrow abcx\alpha\gamma$
- Parse tree



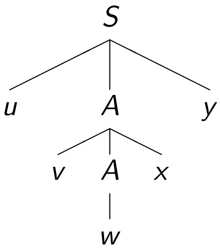
Context-free languages

Definition

- Grammatical: only rules $A \longrightarrow \alpha$
- From automata: pushdown automata

Pumping lemma

- For every language there is an n such that
  - strings longer than n can be written $uvwxy$
  - and for all k: uv^kwx^ky also in the language
- Proof:



- Non-{context-free} language: $\{a^nb^nc^n\}$

Deterministic and non-deterministic

- No equivalence
- Deterministic: $L_c = \{\alpha c \alpha^R | c \notin \alpha\}$
- Non-deterministic: $L = \{\alpha\alpha^R\}$

Algebra of languages

- Expressions $\mathbf{x}$ and $\mathbf{y}$ denote languages, then
  - union: $\mathbf{x}+\mathbf{y}=\mathbf{x}\cup\mathbf{y}$
  - concatenation: $\mathbf{xy}=\{w=xy|x\in\mathbf{x},y\in\mathbf{y}\}$
  - repetition: $\mathbf{x}^*=\{w=x^n|x\in\mathbf{x},n\geq 0\}$

Algebra: solving equations

- Equation: $\mathbf{x} = \mathbf{a} + \mathbf{x}\mathbf{b}$
- Interpretation: $\mathbf{x} = \mathbf{a} \cup \{w = xb | x \in \mathbf{x}, b \in \mathbf{b}\}$
- Solving:
  - first of all $\mathbf{x} \supset \mathbf{a}$
  - then also $\mathbf{x} \supset \mathbf{a} \cdot \mathbf{b}$
  - continuing: $\mathbf{x} \supset \mathbf{abb}, \ldots$
- verify: $\mathbf{x} = \mathbf{ab}^*$

Algebra: solving equations

- Equation: $\mathbf{x} = \mathbf{a} + \mathbf{xb}$
- Interpretation: $\mathbf{x} = \mathbf{a} \cup \{w = xb | x \in \mathbf{x}, b \in \mathbf{b}\}$
- Solving:
  - first of all $\mathbf{x} \supset \mathbf{a}$
  - then also $\mathbf{x} \supset \mathbf{a} \cdot \mathbf{b}$
  - continuing: $\mathbf{x} \supset \mathbf{abb}, \ldots$
- verify: $\mathbf{x} = \mathbf{ab}^*$
- Numerically: $x = a/(1 - b)$

Derive normal form

- Normal form: $A \rightarrow a\alpha$
- Write grammar of context-free language as $\mathbf{x}^t = \mathbf{x}^t\mathbf{A} + \mathbf{f}^t$, where $\mathbf{x}$ non-terminals, $\mathbf{f}$ rhs that are of normal form, $\mathbf{x}^t\mathbf{A}$ describes normal form rhs

Derive normal form

- Normal form: $A \rightarrow a\alpha$
- Write grammar of context-free language as $\mathbf{x}^t = \mathbf{x}^t\mathbf{A} + \mathbf{f}^t$, where $\mathbf{x}$ non-terminals, $\mathbf{f}$ rhs that are of normal form, $\mathbf{x}^t\mathbf{A}$ describes normal form rhs
- Example:

$$\begin{aligned} S &\rightarrow aSb|XY|c \\ X &\rightarrow YXc|b \\ Y &\rightarrow XS \end{aligned} \qquad [S, X, Y] = [S, X, Y] \begin{bmatrix} \phi & \phi & \phi \\ Y & \phi & S \\ \phi & Xc & \phi \end{bmatrix} + [aSb + c, b, \phi]$$

Derive normal form

- Normal form: $A \rightarrow a\alpha$
- Write grammar of context-free language as $\mathbf{x}^t = \mathbf{x}^t\mathbf{A} + \mathbf{f}^t$, where $\mathbf{x}$ non-terminals, $\mathbf{f}$ rhs that are of normal form, $\mathbf{x}^t\mathbf{A}$ describes normal form rhs
- Solution:

$$\mathbf{x}^t = \mathbf{f}^t\mathbf{A}^*$$

Derive normal form

- Normal form: $A \rightarrow a\alpha$
- Write grammar of context-free language as $\mathbf{x}^t = \mathbf{x}^t\mathbf{A} + \mathbf{f}^t$, where $\mathbf{x}$ non-terminals, $\mathbf{f}$ rhs that are of normal form, $\mathbf{x}^t\mathbf{A}$ describes normal form rhs
- Solution:

$$\mathbf{x}^t = \mathbf{f}^t\mathbf{A}^*$$

- Needed: more explicit expression for $\mathbf{A}^*$.

- Note $\mathbf{A}^* = \lambda + \mathbf{A}\mathbf{A}^*$
- then normal form:

$$\mathbf{x}^t = \mathbf{f}^t + \mathbf{f}^t\mathbf{A}\mathbf{A}^* = \mathbf{f}^t + \mathbf{f}^t\mathbf{B}$$

where $\mathbf{B} = \mathbf{A}\mathbf{A}^*$.

- Note $\mathbf{A}^* = \lambda + \mathbf{A}\mathbf{A}^*$
- then normal form:

$$\mathbf{x}^t = \mathbf{f}^t + \mathbf{f}^t\mathbf{A}\mathbf{A}^* = \mathbf{f}^t + \mathbf{f}^t\mathbf{B}$$

  where $\mathbf{B} = \mathbf{A}\mathbf{A}^*$.
- $\mathbf{B}$:

$$\mathbf{B} = \mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{B}$$

  not necessarily normal form

- Note $\mathbf{A}^* = \lambda + \mathbf{A}\mathbf{A}^*$
- then normal form:

$$\mathbf{x}^t = \mathbf{f}^t + \mathbf{f}^t\mathbf{A}\mathbf{A}^* = \mathbf{f}^t + \mathbf{f}^t\mathbf{B}$$

  where $\mathbf{B} = \mathbf{A}\mathbf{A}^*$.
- $\mathbf{B}$:

$$\mathbf{B} = \mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{B}$$

  not necessarily normal form
- Elements of $\mathbf{A}$ that start with a nonterminal can only start with nonterminals in $\mathbf{x}$. Hence substitute a rule from equation above.

Parsing strategies

Top-down parsing

- Start with S on the stack, replace by appropriate rule, guided by input
- Example: expression 2∗5+3, which is produced by the grammar

$$\begin{aligned} \mathit{Expr} &\longrightarrow \ \mathit{number}\ \mathit{Tail} \\ \mathit{Tail} &\longrightarrow \ \epsilon \ \mid \ + \mathit{number}\ \mathit{Tail} \mid \ * \mathit{number}\ \mathit{Tail} \end{aligned}$$

initial queue:	$2*5+3$	
start symbol on stack:		Expr
replace		number Tail
match	$*5+3$	Tail
replace		* number Tail
match	$5+3$	number Tail
match	$+3$	Tail
replace		$+$ number Tail
match	3	number Tail
match	ϵ	Tail
match		

$$E \Rightarrow n\,T \Rightarrow n*n\,T \Rightarrow n*n+n\,T \Rightarrow n*n+n$$

$LL(1)$

Equivalent grammar:

$$Expr \longrightarrow \ number \mid \ number + Expr \mid \ number \text{ * } Expr$$

assuming one more token look-ahead:

initial queue:	$2 * 5 + 3$	
start symbol on stack:		Expr
replace		number * Expr
match	$5 + 3$	Tail
replace		number + Expr
match	3	Expr
replace	3	number
match	ϵ	

$LL(2)$

LL is recursive descent

Finding of proper rule:

```
define FindIn(Sym,NonTerm)
  for all expansions of NonTerm:
    if leftmost symbol == Sym
      then found
    else if leftmost symbol is nonterminal
      then FindIn(Sym,that leftmost symbol)
FindIn(symbol,S);
```

Problems with $LL(k)$

- Some grammars are not $LL(k)$ for any k:
  if `A<B` and `A<B>` both legal
- Infinite loop:

$$Expr \longrightarrow number \mid Expr + number \mid Expr * number$$

Bottom-up: Shift-reduce

- Recognize productions from terminals
- Example: expression $2 * 5 + 3$ produced by

$$E \longrightarrow \ \ number \mid \ \ E + E \mid \ \ E \ * \ E$$

	stack	queue
initial state:		$2 * 5 + 3$
shift	2	*5+3
reduce	E	*5+3
shift	E*	5+3
shift	E*5	+3
reduce	E*E	+3
reduce	E	+3
shift, shift, reduce	E+E	
reduce	E	

$$E \Rightarrow E + E \Rightarrow E + 3 \Rightarrow E * E + 3 \Rightarrow E * 5 + 3 \Rightarrow 2 * 5 + 3$$

$LR(0)$

Where to start reducing?

- 'Greedy' reducing is not always best
- Grammar:

$$S \longrightarrow aAcBe$$
$$A \longrightarrow bA \mid b$$
$$B \longrightarrow d$$

  and string abbcde.
- Derivation 1:

$$\texttt{abbcde} \Leftarrow \texttt{abAcde} \Leftarrow \texttt{aAcde} \Leftarrow \texttt{aAcBe} \Leftarrow \texttt{S}.$$

- Derivation 2:

$$\texttt{abbcde} \Leftarrow \texttt{aAbcde} \Leftarrow aAAcde \Leftarrow ?$$

Handle

If $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$ *is a right-most derivation, then* $A \rightarrow \beta$ *at the position after* α *is a handle of* $\alpha A w$.

Question: how to find handles

Operator-precedence grammars

- Operator grammar: 'expr-op-expr'
- Formally: never two consecutive non-terminals, and no rules $A \rightarrow \epsilon$.
- Declare precedences (and associativity)

	number	$+$	$\times$
number		$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\lessdot$	$\gtrdot$	$\gtrdot$

- Annotate expression: $5 + 2 * 3$ becomes $\lessdot 5 \gtrdot + \lessdot 2 \gtrdot * \lessdot 3 \gtrdot$
- Reducing: $E + E * E$
- Insert precedences: $\lessdot + \lessdot * \gtrdot$
- Scan forward to closing, back to open: $\lessdot E * E \gtrdot$ is handle
- Reduce: $E + E$
- $\Rightarrow$ precendences correctly observed
- (note: no global scanning; still shift-reduce like)

Definition of *LR* parser

An LR parser has the following components

- Stack and input queue; stack will also contain states
- Actions 'shift', 'reduce', 'accept', 'error'
- Functions `Action` and `Goto`
  - With input symbol a and state on top of the stack s:
  - If $\texttt{Action}(a, s)$ is 'shift', then a and a new state $s' = \texttt{Goto}(a, s)$ are pushed on the stack.
  - If $\texttt{Action}(a, s)$ is 'reduce $A \rightarrow \beta$' where $|\beta| = r$, then $2r$ symbols are popped from the stack, a new state $s' = \texttt{Goto}(a, s'')$ is computed based on the newly exposed state on the top of the stack, and A and s' are pushed. The input symbol a stays in the queue.

More powerful than simple shift/reduce; states much more complicated

motivating example

- Grammar

  $$E \longrightarrow \; E+E \mid \; E * E$$

  input string $1+2*3+4$.
- Define precedences: $\mathbf{op}(+)=1, \mathbf{op}(\times)=2$
- Define states as; initially state 0
- Transitions: push operator precedence, do not change state for numbers
- Shift/reduce strategy: reduce if precedence of input lower than of stack top

	$1 + 2 * 3 + 4$	push symbol; highest precedence is 0
$1\ S_0$	$+2 * 3 + 4$	highest precedence now becomes 1
$1\ S_0 + S_1$	$2 * 3 + 4$	
$1\ S_0 + S_1\ 2\ S_1$	$*3 + 4$	highest precedence becoming 2
$1\ S_0 + S_1\ 2\ S_1 * S_2$	$3 + 4$	
$1\ S_0 + S_1\ 2\ S_1 * S_2\ 3\ S_2$	$+4$	reduce because $\mathbf{op}(+) < 2$
$1\ S_0 + S_1\ 6\ S_1$	$+4$	the highest exposed precedence is 1
$1\ S_0 + S_1\ 6\ S_1 + S_1$	4	
$1\ S_0 + S_1\ 6\ S_1 + S_1\ 4\ S_1$		at the end of the queue we reduce
$1\ S_0 + S_1\ 10\ S_1$		
11		

Parser states

item Grammar rule with location indicated.
From $\mathtt{A} \rightarrow \mathtt{B}\ \mathtt{C}$ items: $\mathtt{A} \rightarrow \bullet\mathtt{B}\ \mathtt{C}$, $\mathtt{A} \rightarrow \mathtt{B}\ \bullet\mathtt{C}$, $\mathtt{A} \rightarrow \mathtt{B}\ \mathtt{C}\bullet$
(stack is left of dot, queue right)

closure of an item The smallest set that

- Contains that item;
- If I in closure and $I = \mathtt{A} \rightarrow \alpha\ \bullet\mathtt{B}\ \beta$ with $\mathtt{B}$ nonterminal, then I contains all items $\mathtt{B} \rightarrow \bullet\gamma$.

state Set of items.

follow of $\mathtt{A}$: set of all terminals that can follow A's expansions

Motivation: valid items

- Recognized so far: $\alpha\beta_1$
- Consider item $\mathtt{A} \rightarrow \beta_1 {}_\bullet \beta_2$
- Item is called *valid*, if rightmost derivation

$$S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$$

- Case: $\beta_2 = \epsilon$, then $A \rightarrow \beta_1$ handle: reduce
- Case: $\beta_2 \neq \epsilon$, so shift β_2.

example of valid items

- String `E+T*` in grammar:

$$\begin{aligned} E &\longrightarrow E{+}T \mid T \\ T &\longrightarrow T{*}F \mid F \\ F &\longrightarrow (E) \mid id \end{aligned}$$

- Derivations

$$E \Rightarrow E + T \Rightarrow E + T * F$$

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * (E)$$

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * \mathrm{id}$$

give items $\mathtt{T} \rightarrow \mathtt{T*}_\bullet\mathtt{F}$ $\mathtt{F} \rightarrow {}_\bullet\mathtt{(E)}$ $\mathtt{F} \rightarrow {}_\bullet\mathtt{id}$

States and transitions

- New start symbol `S'`, added production $\mathtt{S}' \rightarrow \mathtt{S}$.
- Starting state is closure of $\mathtt{S}' \rightarrow {\bullet}\mathtt{S}$.
- Transition $d(s, \mathtt{X})$: the closure of

$$\{\mathtt{A} \rightarrow \alpha\ \mathtt{X}_\bullet\ \beta | \mathtt{A} \rightarrow \alpha\ {\bullet}\mathtt{X}\ \beta \text{ is in } s\}$$

- The initial state is the closure of $\mathtt{S}' \rightarrow {\bullet}\mathtt{S}$.

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta)\})$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta)\})$:

Grammar: $S \rightarrow (S)S \mid |\epsilon$

States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\mathit{cl})(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$

Grammar: $S \rightarrow (S)S \mid |\epsilon$

States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$
- $d(2, S) = 3$

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$
- $d(2, S) = 3$
- $d(2,'(') = 2$

Grammar: $S \rightarrow (S)S \mid \mid \epsilon$

States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .\mathtt{(S)S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow \mathtt{(.S)S}, \mathtt{S} \rightarrow .\mathtt{(S)S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow \mathtt{(S.)S}\}$
4. $\{\mathtt{S} \rightarrow \mathtt{(S).S}, \mathtt{S} \rightarrow .\mathtt{(S)S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow \mathtt{(S)S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$
- $d(2, S) = 3$
- $d(2,'(') = 2$
- $d(3,')') = 4$

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$
- $d(2, S) = 3$
- $d(2,'(') = 2$
- $d(3,')') = 4$
- $d(4, S) = 5$

Grammar: $S \rightarrow (S)S \mid |\epsilon$
States (after adding $\mathtt{S}' \rightarrow .\mathtt{S}$):

0. $\{\mathtt{S}' \rightarrow .\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
1. $\{\mathtt{S}' \rightarrow \mathtt{S}.\}$
2. $\{\mathtt{S} \rightarrow (.\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
3. $\{\mathtt{S} \rightarrow (\mathtt{S}.)\mathtt{S}\}$
4. $\{\mathtt{S} \rightarrow (\mathtt{S}).\mathtt{S}, \mathtt{S} \rightarrow .(\mathtt{S})\mathtt{S}, \mathtt{S} \rightarrow .\}$
5. $\{\mathtt{S} \rightarrow (\mathtt{S})\mathtt{S}.\}$

with transitions $(\{A \rightarrow \alpha_\bullet X\beta \in s \Rightarrow (\,cl)(A \rightarrow \alpha X_\bullet \beta))$:

- $d(0, S) = 1$
- $d(0,'(') = 2$
- $d(2, S) = 3$
- $d(2,'(') = 2$
- $d(3,')') = 4$
- $d(4, S) = 5$
- $d(4,'(') = 2$

Stack handling

Loop:

(1) **if** the current state contains $\mathtt{S}' \rightarrow \mathtt{S}\bullet$

accept the string

(2) **else if** the current state contains any other final item $\mathtt{A} \rightarrow \alpha\bullet$

pop all the tokens in α from the stack,

along with the corresponding states;

let s be the state left on top of the stack:

push $\mathtt{A}$, push $\mathtt{d}(s,\mathtt{A})$

(3) **else if** the current state contains any item $\mathtt{A} \rightarrow \alpha\ \bullet\mathtt{x}\ \beta$,

where x is the next input token

let s be the state on top of the stack: push $\mathtt{x}$, push $\mathtt{d}(s,\mathtt{x})$

else report failure

Ambiguity and conflicts

Shift/reduce conflict

- Grammar for $2+5*3$:

$$\langle\texttt{expr}\rangle \longrightarrow \langle\texttt{number}\rangle \mid \langle\texttt{expr}\rangle + \langle\texttt{expr}\rangle \mid \langle\texttt{expr}\rangle \times \langle\texttt{expr}\rangle$$

- interpretations:

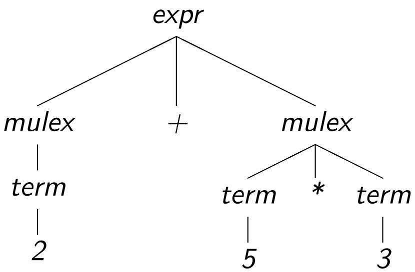
- Parse: reduce $2+5$ to <expr> + <expr>, then reduce to <expr>, or shift the minus?

solutions

- Reformulate the grammar as

$$\begin{array}{lcl} \langle\texttt{expr}\rangle & \longrightarrow & \langle\texttt{mulex}\rangle \mid \langle\texttt{mulex}\rangle + \langle\texttt{mulex}\rangle \\ \langle\texttt{mulex}\rangle & \longrightarrow & \langle\texttt{term}\rangle \mid \langle\texttt{term}\rangle \times \langle\texttt{term}\rangle \\ \langle\texttt{term}\rangle & \longrightarrow & \textit{number} \end{array}$$

- new parse:



- Introduce precedence of operators.
  Possibly more efficient if large number of operators.

'Dangling else'

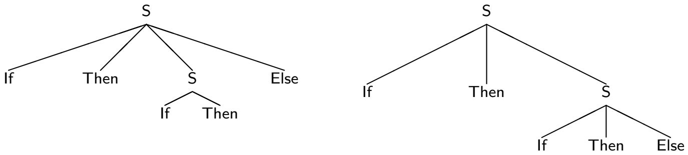
- Consider the grammar

$$\langle\texttt{statement}\rangle \longrightarrow \textit{if}\ \langle\texttt{clause}\rangle\ \textit{then}\ \langle\texttt{statement}\rangle \mid \textit{if}\ \langle\texttt{clause}\rangle\ \textit{then}\ \langle\texttt{statement}\rangle\ \textit{else}\ \langle\texttt{statement}\rangle$$

- string

$$\texttt{if}\ \texttt{c}_1\ \texttt{then}\ \texttt{if}\ \texttt{c}_2\ \texttt{then}\ \texttt{s}_1\ \texttt{else}\ \texttt{s}_2$$

- Interpretations:



Reduce/reduce conflict

- Grammar for `x y c`

$$\begin{array}{lcl} A & \longrightarrow & B\,c\,d \mid E\,c\,f \\ B & \longrightarrow & x\,y \\ E & \longrightarrow & x\,y \end{array}$$

- $LR(1)$ parser: shift `x y`,
  then reduceto `B` or `E`?
- $LR(2)$ parser: sees `d` or `f`
- An LL parser: ambiguity in the first 3 tokens
  $LL(4)$ parser can sees `d` or `f`.

- Grammar for $x\,y\,c^n\,\{d|f\}$:

$$\begin{array}{lcl} A & \longrightarrow & B\;C\;d \;\mid\; E\;C\;f \\ B & \longrightarrow & x\;y \\ E & \longrightarrow & x\;y \\ C & \longrightarrow & c \;\mid\; C\;c \end{array}$$

- confusing for any $LR(n)$ or $LL(n)$ parser with a fixed amount of look-ahead
- rewrite:

$$\begin{array}{lcl} A & \longrightarrow & BorE\;c\;d \;\mid\; BorE\;c\;f \\ BorE & \longrightarrow & x\;y \end{array}$$

  or (for an $LL(n)$ parser):

$$\begin{array}{lcl} A & \longrightarrow & BorE\;c\;tail \\ tail & \longrightarrow & d \;\mid\; f \\ BorE & \longrightarrow & x\;y \end{array}$$

yacc

yacc and *lex*

- *lex* produces tokens
- *yacc* analyzes sequences of tokens
- lexer returns on recognizing a token
- main program in *yacc* code

File structure

```
  ...definitions...
%%
  ...rules...
%%
  ...code...
```

- Default main calls `yyparse`

Example: *yacc* code header

File name `words.y`

```
%{

#include <stdlib.h>
#include <string.h>
  int yylex(void);
#include "words.h"
  int nwords=0;
#define MAXWORDS 100
  char *words[MAXWORDS];
%}

%token WORD

%%
```

include file

Generated by running *yacc*:

```
%% cat words.h
#define WORD 257
```

Example: *lex* code

```
%{

#include "words.h"
int find_word(char*);
extern int yylval;
%}

%%

[a-zA-Z]+ {yylval = find_word(yytext);
   return WORD;}
.          ;
\n         ;

%%
```

```
text : ;
       | text WORD  ; {
           if ($2<0) printf("new word\n");
           else printf("matched word %d\n",$2);
                     }
%%

int find_word(char *w)
{ int i;
  for (i=0; i<nwords; i++)
    if (strcmp(w,words[i])==0) return i;
  words[nwords++] = strdup(w); return -1;
}

int main(void)
{
  yyparse();
  printf("there were %d unique words\n",nwords);
}
```

Running *lex* and *yacc*

```
/* create and compile yacc C file */
yacc -d -t -o YACCFILE.c YACCFILE.y
cc -c -o YACCFILE.o YACCFILE.c

/* create and compile lex C file */
lex -t LEXFILE.l > LEXFILE.c
cc -c -o LEXFILE.o LEXFILE.c

/* link together */
cc YACCFILE.o LEXFILE.o -o YACCPROGRAM -ly -ll
```

Make with suffix rules

```
# disable normal rules
.SUFFIXES:
.SUFFIXES: .l .y .o
# lex rules
.l.o :
        lex -t $*.l > $*.c
        cc -c $*.c -o $*.o
# yacc rules
.y.o :
        if [ ! -f $*.h ] ; then touch $*.h ; fi
        yacc -d -t -o $*.c $*.y
        cc -c -o $*.o $*.c ;
        rm $*.c
# link lines
lexprogram : $(LEXFILE).o
        cc $(LEXFILE).o -o $(LEXFILE) -ll
yaccprogram : $(YACCFILE).o $(LEXFILE).o
        cc $(YACCFILE).o $(LEXFILE).o -o $(YACCFILE) -ly -ll
```

yacc definitions section

- C code in between `%{ ... %}`
- Token definitions: the *lex* return tokens
- Associativity rules (later)

Tokens

- Definition: `%token FOO`
- In `.h` file: `#define FOO 257` (or so)
- *lex* code: `return FOO`

Returning values over the stack

- *lex* assigns to `yylval`
- value is put on top of stack
- if a *yacc* rule is matched: `$1, $2, $3` are assigned (as many as elements in rhs)
- replace stack top: assign to `$$`

Calculator example: *lex* code

```
%{
#include "calc1.h"
void yyerror(char*);
extern int yylval;
%}

%%
[ \t]+ ;
[0-9]+      {yylval = atoi(yytext);
             return INTEGER;}
[-+*/]      {return *yytext;}
"("         {return *yytext;}
")"         {return *yytext;}
\n          {return *yytext;}
.           {char msg[25];
             sprintf(msg,"%s <%s>","invalid character",yytext);
             yyerror(msg);}
```

Calculator example: *yacc* code

```
%{
int yylex(void);
#include "calc1.h"
%}

%token INTEGER

%%

program:
        line program
        | line
line:
        expr '\n'            { printf("%d\n",$1); }
        | 'n'
```

Calculator example: *yacc* code, cont'd

```
expr:
        expr '+' mulex      { $$ = $1 + $3; }
        | expr '-' mulex    { $$ = $1 - $3; }
        | mulex             { $$ = $1; }
mulex:
        mulex '*' term      { $$ = $1 * $3; }
        | mulex '/' term    { $$ = $1 / $3; }
        | term              { $$ = $1; }
term:
        '(' expr ')'        { $$ = $2; }
        | INTEGER           { $$ = $1; }
```

Calculator with variables

- Simple case: single letter variables
- more complicated: names
- Extra rule: assignments
- *lex* returns
  - `double` values
  - `int` index of variable

Multiple return types

- Declare possible return types:

```
%union {int ival; double dval;}
```

- Connect types to return tokens:

```
%token <ival> NAME
%token <dval> NUMBER
```

- The types of non-terminals need to be given:

```
%type <dval> expr
%type <dval> mulex
%type <dval> term
```

- In `.h` file will now have

```
#define name 258
#define NUMBER 259
typedef union {int ival; double dval;} YYSTYPE;
extern YYSTYPE yylval;
```

Multiple return types: *lex* code

```
[ \t]+ ;
(([0-9]+(\.[0-9]*)?)|([0-9]*\.[0-9]+)) {
                 yylval.dval = atof(yytext);
                 return DOUBLE;}
[-+*/=]         {return *yytext;}
"("             {return *yytext;}
")"             {return *yytext;}
[a-z]           {yylval.ivar = *yytext - 'a';
                 return NAME;} /* more later */
\n              {return *yytext;}
.               {char msg[25];
                 sprintf(msg,"%s <%s>","invalid character",yyte
                 yyerror(msg);}
```

Example: calculator with variables

Tokens are `double` numbers, or variables (`int` index in table)

```
%{
#define NVARS 100
char *vars[NVARS]; double vals[NVARS]; int nvars=0;
%}
%union { double dval; int ivar; }
%token <dval> DOUBLE
%token <ivar> NAME
%type <dval> expr
%type <dval> mulex
%type <dval> term
```

Symbol table handling

- *lex* parses variable names:

```
[a-z][a-z0-9]* {
               yylval.ivar = varindex(yytext);
               return NAME;}
```

- names are dynamically stored:

```
int varindex(char *var)
{
  int i;
  for (i=0; i<nvars; i++)
    if (strcmp(var,vars[i])==0) return i;
  vars[nvars] = strdup(var);
  return nvars++;
}
```

Arithmetic

Largely as before:

```
expr:
        expr '+' mulex        { $$ = $1 + $3; }
        | expr '-' mulex      { $$ = $1 - $3; }
        | mulex               { $$ = $1; }
mulex:
        mulex '*' term        { $$ = $1 * $3; }
        | mulex '/' term      { $$ = $1 / $3; }
        | term                { $$ = $1; }
term:
        '(' expr ')'          { $$ = $2; }
        | NAME                { $$ = vals[$1]; }
        | DOUBLE              { $$ = $1; }
```

Assignments

```
line:
        expr '\n'                { printf("%g\n",$1); }
      | NAME '=' expr '\n'       { vals[$1] = $3; }
```

Operator precedence and associativity

Increasing precedence order:

```
%left '+' '-'
%left '*' '/'
%right '^'
%%
expr:
        expr '+' expr ;
        expr '-' expr ;
        expr '*' expr ;
        expr '/' expr ;
        expr '^' expr ;
        number ;
```

Unary operators

Declare non-associative;
indicate presence in rule

```
%left '-' '+'
%nonassoc UMINUS
%
expression : expression '+' expression
    | expression '-' expression
    | '-' expression %prec UMINUS
```

Error handling

- Default: `yyerror` prints `syntax error`
- Better:
  lex code:

  ```
  \n     lineno++;
  ```

  yacc code:

  ```
  void yyerror(char *s)
  {
    printf("Parsing failed in line %d because of %s\n",lineno,s);
    return;
  }
  ```

- Your own error messages:

  ```
  expr : name '[' name ']'
          {if (!is_array($1) yyerror("array name expected");
  ```

Error recovery

- Use of `error` token:

```
foo : bar baz ;
    | error baz printf("Hope for the best\n");
```