Syntax

- Need precise rules
  - For developers to write programs
  - For programs that write programs
  - For compilers to parse programs
- Example, precise syntax rule for digit:

$$digit \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

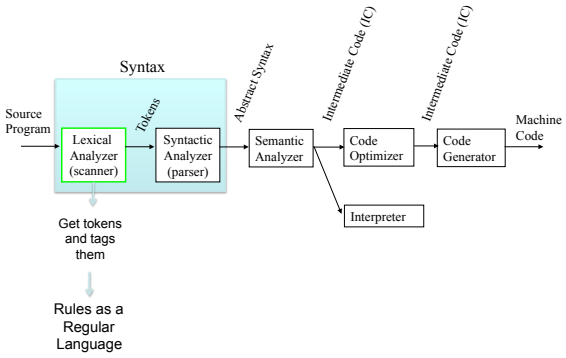
Syntax rules

- Character
- Empty string
- Operations
  - Concatenation
  - Alternation
  - Kleene closure – repetition
  - Recursion

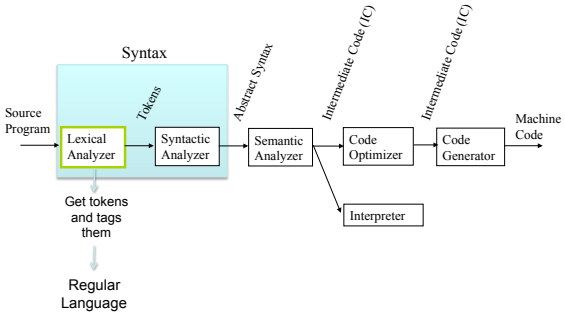
Regular Language

Recognized through regular expressions

Compilers and Interpreters



Lexer



Lexer

- Aims to tokenize an input stream
  - A token is a logically cohesive sequence of characters representing a single symbol.

```
// a first program
// with 2 comments
int main ( ) {
    char c;
    int i;
    c = 'h';
    i = c + 3;
} // main
```

```
Keyword     int
Keyword     main
Punctuation (
Punctuation )
Punctuation {
Keyword     char
Identifier  c
Punctuation ;
```

Regular Language

binaryDigit → 0
binaryDigit → 1

Or binaryDigit → 0 | 1

- For program/er to generate program
  - Scan from left to right
  - Choose alternatives
  - Choose number of repetitions

Regular Expressions

Regular Expression	Meaning
x	A character
“xwz”	A string
m | n	m or n
m n	m followed by n
m+	One or more occurrences of m
m*	Zero or more occurrences of m
m?	Zero or one occurrences of m
[0-9]	Any digit
[a-zA-Z]	Any letter

Regular Expressions

- Integer [0-9]+
- Identifier [a-zA-Z][a-zA-Z0-9]*
- Boolean “true” | “false”
- Real Number [0-9]+”.”[0-9]+

Regular Expressions

Give a regular expression that accepts email adresses (se@unl.edu) : strings made out of letters, period ('.'), and 1 '@' symbol.

But how about

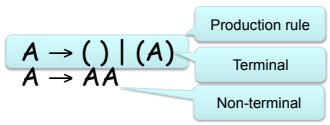
Give a regular expression that accepts up to *n* matching braces? (for every ‘{‘ there is a ‘}’)

n = 1, 2, 3, …

Syntax rules

- Character
- Empty string
- Operations
  - Concatenation
  - Alternation
  - Kleene closure – repetition
  - Recursion

Context Free Grammar



Regular expressions + *nested constructs*

Context-free Grammar
(also known as BNF)

- Set of *productions*: *P*
  - *terminal* symbols: *T* *(tokens, alphabet)*
  - *nonterminal* symbols: *N* *(category)*
  - *start* symbol: $S \in N$
- A *production* has the form

  $A \rightarrow \omega$

  where $A \in N$ and $\omega \in (N \cup T)^*$

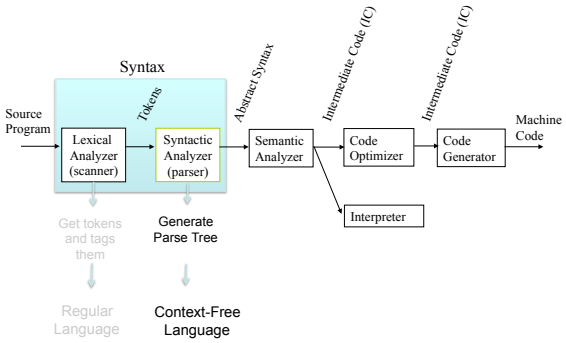
  and $\rightarrow$ replaces nonterminal with righthand

Does a string meet a grammar?

A → ( ) | (*A*)
A → *AA*

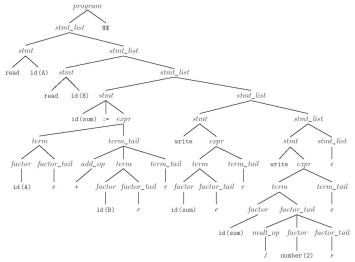
- Examples strings generated by grammar
  - ( )
  - ( )( )
  - (( ))( )
- Corresponding derivations
  - **A -> ( )**
  - **A -> AA -> ( )A -> ( )( )**
  - **A -> AA -> (A)A -> (A)( ) -> (( ))( )**

Compilers and Interpreters



Parser

- Recognizes a language defined by a CFG
- Builds a syntax tree from tokens



Derivations

Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Derive unsigned integer 352

  Integer ⇒ Integer Digit
  ⇒ Integer *2*
  ⇒ Integer Digit *2*
  ⇒ Integer *5 2*
  ⇒ Digit *5 2*
  ⇒ *3 5 2*

Each step is the application of a production rule.

Algorithm to Derive

- Start with S
- Repeat
  - Replace a non terminal with terminals (or non terminals) using productions
- Stop when string consists of only terminals

Leftmost Derivation: replace the left most non terminal at each step.

Derive unsigned integer 352
Integer ⇒ Integer Digit
⇒ Integer Digit Digit
⇒ Digit Digit Digit
⇒ 3 Digit Digit
⇒ 3 5 Digit
⇒ *3 5 2*

Rightmost Derivation: replace the right most non terminal at each step.

Derive unsigned integer 352
Integer ⇒ Integer Digit
⇒ Integer *2*
⇒ Integer Digit *2*
⇒ Integer *5 2*
⇒ Digit *5 2*
⇒ *3 5 2*

Grammar for "my little english"

<sentence> ::= <subject> <predicate>
<subject> ::= <article> <noun>
<predicate>::= <verb> <article> <noun>
<verb> ::= ran | ate
<article> ::= the
<noun> ::= boy | girl | cake

Alternative metalanguage notation

Derive: "the boy ate the cake"

<sentence> ⇒ <subject> <predicate> First rule
⇒ <article> <noun> <predicate> Second rule
⇒ the <noun> <predicate> Fifth rule
... ⇒ the boy ate the cake

Also from <sentence> you can derive
⇒ the cake ate the boy
Syntax does not imply correct semantics

Grammar for a PL

STMT → **while (EXPR)** STMT | **id (EXPR) ;**
EXPR → EXPR + EXPR | EXPR - EXPR | EXPR < EXPR | **(EXPR)** | **id**

Which strings are not from the language?

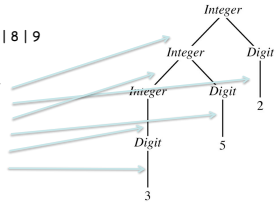
1. id (id) ;
2. id ((((id)))) ;
3. while (id < id) id (id) ;
4. while (while (id)) id (id) ;
5. while (id) while (id) while (id) id (id) ;

Parse Trees (derivation trees)

Graphical hierarchical representation of derivation
- Internal nodes correspond to steps in derivation
- Child nodes represent a right-hand side of a production
- Leaf nodes represent a symbol of derived string

Grammar
Integer → Digit | Integer Digit
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Derive unsigned integer 352
Integer ⇒ Integer Digit
⇒ Integer *2*
⇒ Integer Digit *2*
⇒ Integer *5 2*
⇒ Digit *5 2*
⇒ *3 5 2*



Grammar for Arithmetic Expression

Language of arithmetic expressions with
1-digit integers, addition, and subtraction

Expr → Expr + Term | Expr - Term | Term
Term → 0 | ... | 9 | (Expr)

Generate a Valid Arithmetic Expression

Expr → Expr + Term | Expr - Term | Term
Term → 0 | ... | 9 | (Expr)

Start with base rule (or select one if there are many)
While there are Non-terminals on RHS of derivation
Find rule with LHS matching non-terminal on RHS of derivation
Replace non-terminal in derivation with rule's RHS
End-While

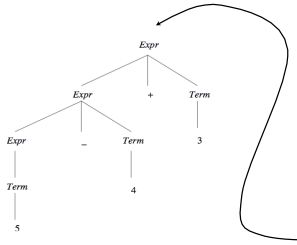
Infinite possibilities

Expr → Expr + Term
Expr → Expr - Term + Term
Expr → Expr - Term + 3
Expr → Expr - 4 + 3
Expr → Term - 4 + 3
Expr → 5 - 4 + 3

Top down

Parse string: 5-4+3

Expr → Expr + Term | Expr - Term | Term
Term → 0 | ... | 9 | (Expr)



Bottom up

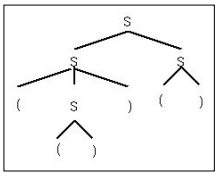
. 5 – 4 + 3	shift
5 . – 4 + 3	reduce
Term . – 4 + 3	reduce
Expr . – 4 + 3	shift
Expr – . 4 + 3	shift
Expr – 4 . + 3	reduce
Expr – Term . + 3	reduce
Expr . + 3	shift
Expr + . 3	shift
Expr + 3 .	shift
Expr + Term .	reduce
Expr .	

Derivations

- Derivations may not be unique
  S → SS | (S) | ()

  S ⇒ SS ⇒(S)S ⇒(())S ⇒(())()
  S ⇒ SS ⇒ S() ⇒(S)() ⇒(())()

- Different derivations still get the same parse tree



Order of derivations is lost in tree

Ambiguous Grammars

- A grammar is *ambiguous* if one of its strings has two or more different parse trees

<E> ::= <E> + <E>
<E> ::= <E> *<E>
<E> ::= <Id>

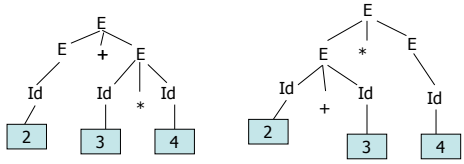
Try to construct a derivation with two different parse trees.

Ambiguous Grammars

<E> ::= <E> + <E>
<E> ::= <E> *<E>
<E> ::= <Id>

2 + 3 * 4

Problem: multiple program interpretations.



Precedence and associativity are not specified. Enforced through alternative means.

Extended BNF

- BNF
  - Recursion for iteration
  - Non-terminals for grouping
- EBNF: additional meta-characters rules to make grammars simpler and clearer
  - { } for a series of zero or more
  - ( ) for a list, must pick one
  - [ ] for an optional list; pick none or one

Expression -> Term { (+ | -) Term }

Extended BNF

Identifier: letter followed by 0 or more letters/digits

Extended BNF

I → L { L | D }
L → a | b |...
D → 0 | 1 |...

Regular BNF

I → L | L M
M → CM | C
C → L | D
L → a | b |...
D → 0 | 1 |...

EBNF to BNF

We can always rewrite an EBNF grammar as a BNF grammar

A -> x { y } z

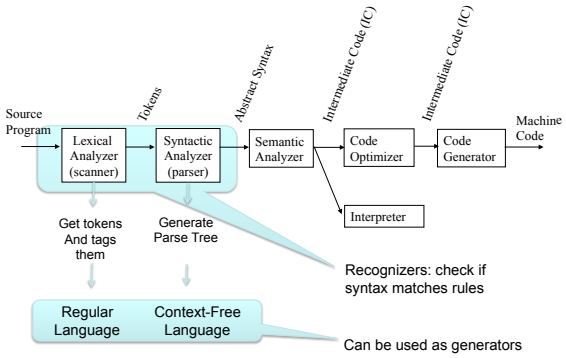
can be rewritten:

A -> x A' z
A' -> ε | y A'

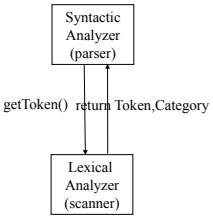
Grammar Sizes

Language	(pages)	Reference
Pascal	*5*	*Jensen & Wirth*
C	*6*	*Kernighan & Richie*
C++	*22*	*Stroustrup*
Java	*14*	*Gosling, et. Al.*

Compilers and Interpreters



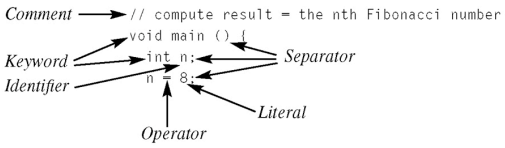
Implementing Syntax Checking



Implementing a Lexer

- Convert program into stream of tokens
- Token
  - String of nonblank characters
  - Defined by **lexicon** of a language
- Lexicon
  - Set of grammatical categories
    - Identifiers (variable names, function names…)
    - Literals (integers, reals…)
    - Operators (+,-,*,/,…..)
    - Separators (;,{,}…..)
    - Keywords (int, main, if, while…..)

Implementing a Lexer




```
…
Identifier : Letter { Letter | Digit }
  Letter : a | b | … | z | A | B | … | Z
   Digit : 0 | 1 | … | 9
 Literal : Integer | Boolean | Float | Char
 Integer : Digit { Digit }
 Boolean : true | False
   Float : Integer . Integer
    Char : ' ASCII Char '
…
```

Implementing a Lexer

- Ad-hoc
  - Compact and easy to read
  - Hard to maintain / extend
- RE -> DFA (deterministic finite automata)
  - Tool support to translate directly from RE
  - Classes
    - Nested case statements
    - Table-driven

Implementing a Lexer (ad-hoc)

- Read characters one at a time with look-ahead
- Always get the longest token possible

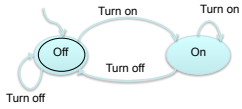
```
If it is { ( ) [ ] , ; = + - etc }
        return (separator, token)
If it is a letter
        peek ahead for more letters and digits until separator is found
If it is a reserve word
        return (reserved, token)
else
        return (identifier, token)
If it is a <
        peek ahead at next character
        if it is a =
                return (separator, <=)
        else
                reuse look-ahead
                return (separator, <)
...
```

Implementing a Lexer

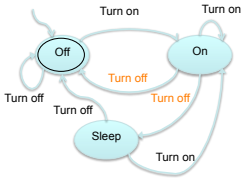
- Ad-hoc
  - Compact and easy to read
  - Hard to maintain / extend
- RE -> DFA (deterministic finite automata)
  - Tool support to translate directly from RE
  - Classes
    - Nested case statements
    - Table-driven

Implementing a Lexer (DFA)

DFA
Deterministic
Finite State Automaton



NDFA
Non Deterministic
Finite State Automaton

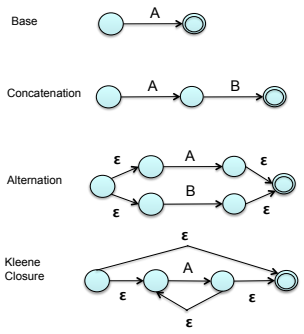


Implementing a Lexer (DFA)

Regular expression

NDFA

Invariants to allow integration:
- no transitions in
- one accepting state
- no transitions out of accepting state

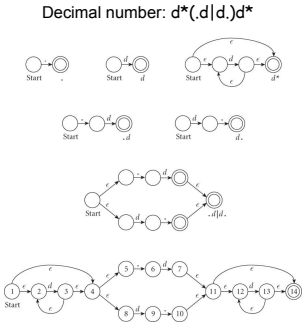


Implementing a Lexer (DFA)

Regular expression

NDFA

Decimal number: d*(.d|d.)d*



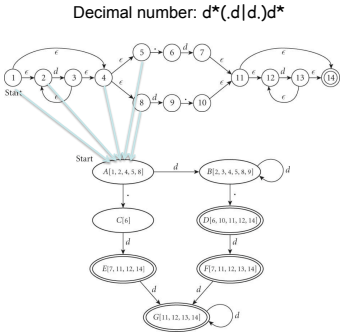
Implementing a Lexer (DFA)

Regular expression

NDFA

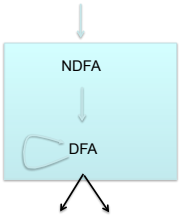
DFA

Decimal number: d*(.d|d.)d*



Implementing a Lexer (DFA)

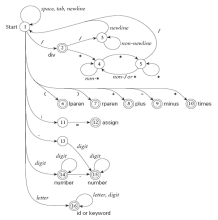
Regular expression



Switches and Loops

Table

```
...
Identifier : Letter { Letter | Digit }
  Letter : a | b | ... | z | A | B | ... | Z
    Digit : 0 | 1 | ... | 9
 Literal : Integer | Boolean | Float | Char
Integer : Digit { Digit }
Boolean : true | False
  Float : Integer . Integer
   Char : ' ASCII Char '
...
```



Implementing a Lexer
DFA with switches and loops

- Outer switch maps to DFA states
- Inner switch maps to transitions + return token
- Exceptions
  - Keywords
  - Handling errors

```
state = start
loop
    read char
    switch state
        case start:
            switch char
                case 'space', 'tab', '\n':
                    continue
                case '/':
                    peek next char
                    if next_char == '*'
                        state = in_comment
                        continue
                ...
        case in_comment:
        ...
```

Implementing a Lexer (DFA)

DFA to Table

- 2D data structure
  - Rows: chars
  - Columns: states
  - Cell specifies
    - Move state
    - Return Token
    - Announce error

State	Space/ tab	newline	/	*	(	...	Token
1	8	8	2	10	6		
2	-	-	3	4	-		div
3	3	18	3	3	3		
4	4	4	4	5	4		
5	4	4	18	5	4		lparen
6	-	-	-	-	-		
7	-	-	-	-	-		
8	8	8	-	-	-		space
…							

End of token, start over

Implementing a Lexer (DFA)

DFA to Table

- 2D data structure
  - Rows: chars
  - Columns: states
  - Cell specifies
    - Move state
    - Return Token
    - Announce error
- Used by generators
  - Lex: regular exp. to C

From Niemann's calculator

Section divider

Matched pattern (col 1)

Action for pattern (c statement/s or lex macro)

```
%
{ #include <stdlib.h>
  #include "calc3.h"
  #include "y.tab.h"
  void yyerror(char *);
  %}                                   Definitions

%%
[a-z]     {
              yylval.sIndex = *yytext - 'a';
              return VARIABLE;
          }
[0-9]+    {
              yylval.iValue = atoi(yytext);
              return INTEGER;
          }
[-()<>=+*/;{}.] {
              return *yytext;
          }
">="      return GE;                 Rules
"<="      return LE;
"=="      return EQ;
"!="      return NE;
[ \t\n]+  ;       /* ignore whitespace */ .
yyerror("Unknown character");

%%
int yywrap(void) { return 1; }
                                     Subroutines
```

Implementing a Parser

```
id_list  ⟶  id  id_list_tail
id_list_tail  ⟶  ,  id  id_list_tail
id_list_tail  ⟶  ;
```

Parse "A, B, C;"

LL
Top down
Predict-look ahead

id_list

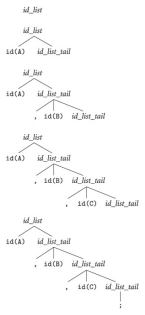
LR
Bottom up
Group and match

id(A)

id(A) ,

id(A) , id(B)

Implementing a Parser



```
id_list  ⟶  id  id_list_tail
id_list_tail  ⟶  ,  id  id_list_tail
id_list_tail  ⟶  ;
```

Parse "A, B, C;"

LL
Top down
Predict-look ahead

LR
Bottom up
Group and match

Implementing a Parser

- Recursive Descent of LL
- Table-driven of LR

Implementing a Parser (RD)

// Sample Calculator CFG
program → *stmt_list* $$
stmt_list → *stmt stmt_list* | ε
stmt → id := *expr* | read id | write expr
expr → *term term_tail*
term_tail → *add_op term term_tail* | ε
term → *factor fact_tail*
fact_tail → *mult_op factor fact_tail* | ε
factor → (*expr*) | id | number
add_op → + | -
mult_op → * | /

Sample Program

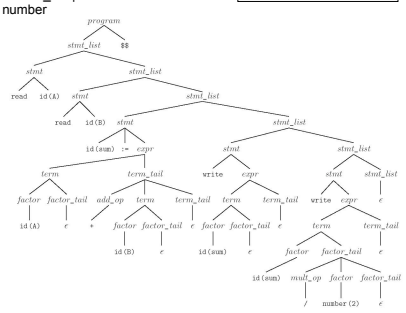
```
read A
read B
sum := A + B
write sum
write sum / 2
```

Usage
- Start at initial node
- Predict next production based on
  - Current left-most non-terminal in the tree
  - Current input token

// Sample Calculator CFG
program → *stmt_list* $$
stmt_list → *stmt stmt_list* | ε
stmt → id := *expr* | read id | write expr
expr → *term term_tail*
term_tail → *add_op term term_tail* | ε
term → *factor fact_tail*
fact_tail → *mult_op factor fact_tail* | ε
factor → (*expr*) | id | number
add_op → + | -
mult_op → * | /

Sample Program

```
read A
read B
sum := A + B
write sum
write sum / 2
```



CFG to Parser

```
procedure match(expected)
    if input_token = expected
        consume input_token
    else parse_error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match($$)
        otherwise parse_error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip       -- epsilon production
        otherwise parse_error

procedure stmt
    case input_token of
        id : match(id); match(:=); expr
        read : match(read); match(id)
        write : match(write); expr
        otherwise parse_error

procedure expr
    case input_token of
        id, number, ( : term; term_tail
        otherwise parse_error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise parse_error

procedure term
```

Recursive Descent Parser
- One subroutine per each non-terminal
- Each subroutine predicts expected token
  - *If prediction is correct*
    - *Consume token, process rest of production*
  - *Else*
    - *Report error*

program → *stmt_list* $$
stmt_list → *stmt stmt_list* | ε
stmt → id := *expr* | read id | write expr
expr → *term term_tail*
term_tail → *add_op term term_tail* | ε
term → *factor fact_tail*
fact_tail → *mult_op factor fact_tail* | ε
factor → (*expr*) | id | number
add_op → + | -
mult_op → * | /

CFG to Parser

```
procedure match(expected)
    if input_token = expected
        consume input_token
    else parse_error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match($$)
        otherwise parse_error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip       -- epsilon production
        otherwise parse_error

procedure stmt
    case input_token of
        id : match(id); match(:=); expr
        read : match(read); match(id)
        write : match(write); expr
        otherwise parse_error

procedure expr
    case input_token of
        id, number, ( : term; term_tail
        otherwise parse_error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise parse_error

procedure term
```

Sample Program

```
read A
read B
sum := A + B
write sum
write sum / 2
```

match
stmt_list
stmt_list
program

CFG to Parser

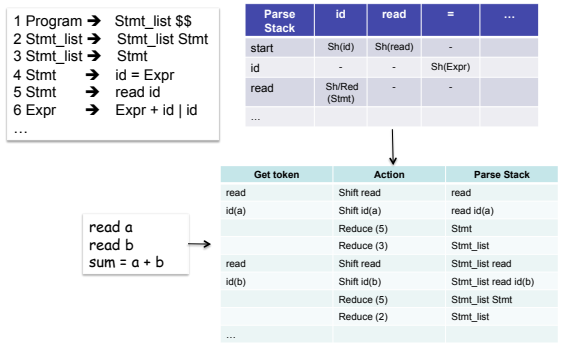
```
procedure match(expected)
    if input_token = expected
        consume input_token
    else parse_error

-- this is the start routine:
procedure program
    case input_token of
        id, read, write, $$ :
            stmt_list
            match($$)
        otherwise parse_error

procedure stmt_list
    case input_token of
        id, read, write : stmt; stmt_list
        $$ : skip       -- epsilon production
        otherwise parse_error

procedure stmt
    case input_token of
        id : match(id); match(:=); expr
        read : match(read); match(id)
        write : match(write); expr
        otherwise parse_error

procedure expr
    case input_token of
        id, number, ( : term; term_tail
        otherwise parse_error

procedure term_tail
    case input_token of
        +, - : add_op; term; term_tail
        ), id, read, write, $$ :
            skip       -- epsilon production
        otherwise parse_error

procedure term
```

program → *stmt_list* $$
stmt_list → *stmt stmt_list* | ε
stmt → id := *expr* | read id | write expr
expr → *term term_tail*
term_tail → *add_op term term_tail* | ε
term → *factor fact_tail*
fact_tail → *mult_op factor fact_tail* | ε
factor → (*expr*) | id | number
add_op → + | -
mult_op → * | /

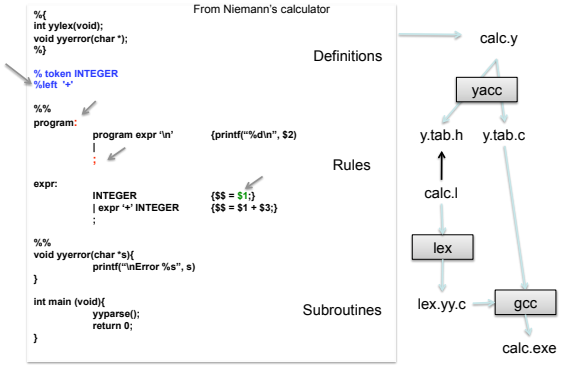
"Predictions"
How do you derive those terminals?

Production may yield a string beginning with those terminals

Implementing a Bottom Up Parser



Implementing a Bottom Up Parser



TODO

- Finish reading Ch1 and Ch2
- Practice
  - Play with Lex/Yacc
  - A good short reference by T. Niemann is in BB
- Start reading Ch3