Syntax

Syntax

- Syntax defines what is grammatically valid in a programming language
  - Set of grammatical rules
  - E.g. in English, a sentence cannot begin with a period
  - Must be formal and exact or there will be ambiguity in a programming language
- We will study three levels of syntax
  - Lexical
    - Defines the rules for tokens: literals, identifiers, etc.
  - Concrete Syntax or just "Syntax"
    - Actual representation scheme down to every semicolon, i.e. every lexical token
  - Abstract Syntax – will cover in Semantics
    - Description of a program's information without worrying about specific details such as where the parentheses or semicolons go

BNF or Context Free Grammar

- BNF = Backus-Naur Form to specify a grammar
  - Equivalent to a context free grammar
- Set of **rewriting rules** (a rule that can be applied multiple times) also known as **production rules** defined on a set of **nonterminal symbols**, a set of **terminal symbols**, and a **start symbol**
  - Terminals, $\sum$: Basic alphabet from which programs are constructed. E.g., letters, digits, or keywords such as "int", "main", "{", "}"
  - Nonterminals, N : Identify grammatical categories
  - Start Symbol: One of the nonterminals which identifies the principal category. E.g., "Sentence" for english, "Program" for a programming language

Rewriting Rules

- Rewriting Rules, ρ
  - Written using the symbols → and |

    | is a separator for alternative definitions, i.e. "OR"

    → is used to define a rule, i.e. "IS"
  - Format
    - LHS → RHS1 | RHS2 | RHS3 | …
    - LHS is a single nonterminal
    - RHS is any sequence of terminals and nonterminals

Sample Grammars

- Grammar for subset of English
  - **Sentence → Noun Verb**
  - **Noun** → Jack | Jill
  - **Verb** → eats | bites
- Grammar for a digit
  - **Digit** → 0 | 1 | 2 | 3 | 4 | 5 | 6 |7 |8 |9
- Grammar for signed integers
  - **SignedInteger → Sign Integer**
  - **Sign** → + | -
  - **Integer → Digit | Digit Integer**
- Grammar for subset of Java
  - **Assignment → Variable** = **Expression**
  - **Expression → Variable | Variable** + **Variable | Variable – Variable**
  - **Variable** → X | Y

Derivation

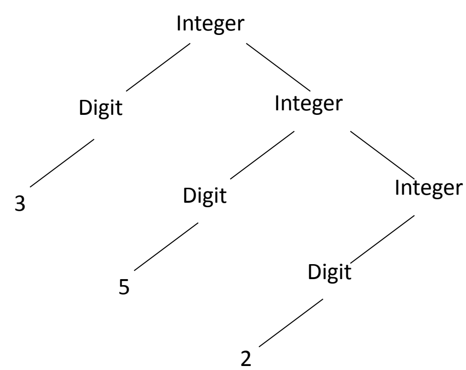
- Process of parsing data using a grammar
  - Apply rewrite rules to non-terminals on the RHS of an existing rule
  - To match, the derivation must terminate and be composed of terminals only
- Example
  - **Digit** → 0 | 1 | 2 | 3 | 4 | 5 | 6 |7 |8 |9
  - **Integer → Digit | Digit Integer**
  - Is 352 an Integer?

```
Integer → Digit Integer → 3 Integer →
3 Digit Integer →  3 5 Integer →
3 5 Digit → 3 5 2
```

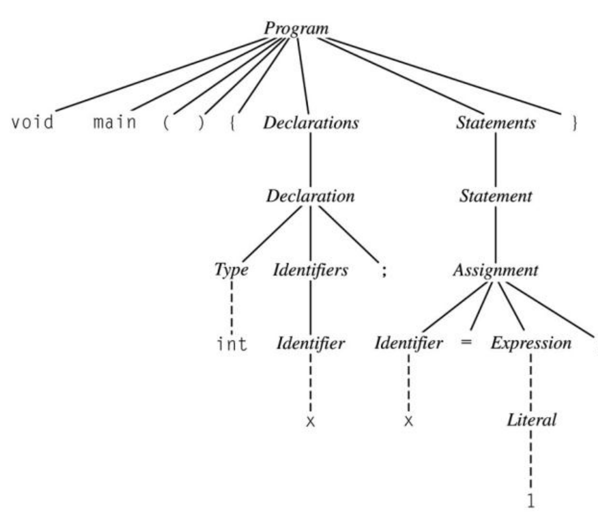
Intermediate formats are called **sentential forms**
This was called a Leftmost Derivation since we replaced the leftmost nonterminal symbol each time (could also do Rightmost)

Derivation and Parse Trees

- The derivation can be visualized as a parse tree



Parse Tree Sketch for Programs



BNF and Languages

- The **language** defined by a BNF grammar is the set of **all** strings that can be derived
  - Language can be infinite, e.g. case of integers
- A language is **ambiguous** if it permits a string to be parsed into two separate parse trees
  - Generally want to avoid ambiguous grammars
  - Example:
    - Expr → Integer | Expr + Expr | Expr * Expr | Expr - Expr
    - Parse: 3*4+1

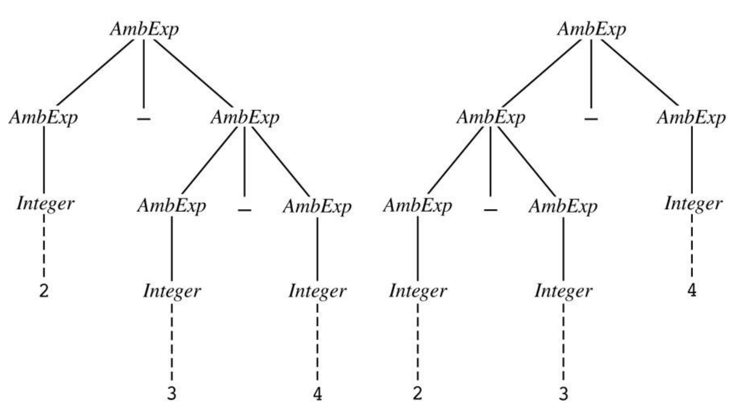
```
– Expr * Expr → Integer * Expr →
  3 * Expr → 3 * Expr+Expr → … 3 * 4 + 1
– Expr + Expr → Expr + Integer → Expr + 1
  Expr * Expr +1 → … 3 * 4 + 1
```

Ambiguity

- Example for

  AmbExp → Integer | AmbExp – AmbExp

  2-3-4



Ambiguous IF Statement

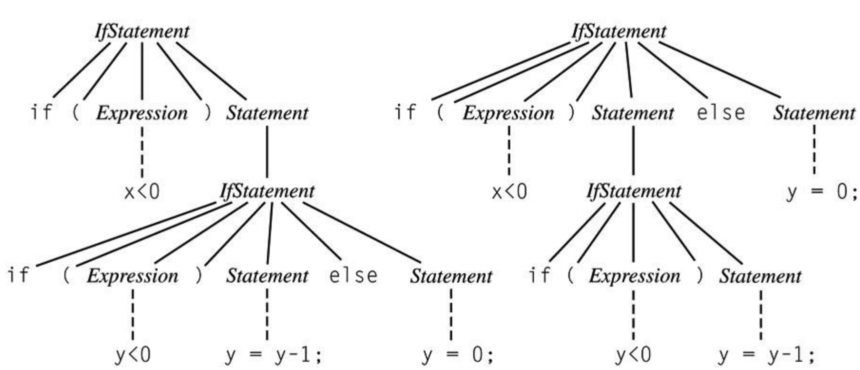
Dangling ELSE:

```
if (x<0)
if (y<0)  { y=y-1 }
else { y=0 };
```

Does the else go with the first or second if?

$$
\begin{aligned}
\mathit{IfStatement} &\rightarrow \text{if } (\ \mathit{Expression}\ )\ \mathit{Statement} \mid \\
&\ \text{if } (\ \mathit{Expression}\ )\ \mathit{Statement} \text{ else } \mathit{Statement} \\
\mathit{Statement} &\rightarrow \mathit{Assignment} \mid \mathit{IfStatement}
\end{aligned}
$$

Dangling Else Ambiguity



How to fix ambiguity?

- Use explicit grammar without ambiguity
  - E.g., add an “ENDIF” for every “IF”
- One problem with end markers is that they tend to bunch up. In Pascal you say

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
```

- With end markers this becomes

```
if A = B then …
else if A = C then …
else if A = D then …
else if A = E then …
else ...;
end; end; end; end;
```

Ambiguity

- Fixing Ambiguity
  - Java makes a separate category for if-else vs. if:

    IfThenStatement → If (**Expr**) **Statement**

    IfThenElseStatement → If (**Expr**) **StatementNoShortIf** else **Statement**

    StatementNoShortIf contains everything except **IfThenStatement**, so the else always goes with the **IfThenElse** statement not the **IfThenStatement**
- In general, we add new grammar rules that enforce precedence

Precedence Example

- Ambiguous
  - Expr → Identifier | Integer | Expr + Expr | Expr * Expr | Expr – Expr

- Unambiguous
  - Expr → Term | Expr + Term | Expr - Term
  - Term → Factor | Term * Factor
  - Factor → Integer | Identifier

- Parse: 3*4+1
  - `Expr + Term → Term + Term → Term * Factor + Term → Integer * Factor + Term → 3 * Factor + Term → 3 * Integer + Term → 3 * 4 + Term → 3 * 4 + Factor → 3 * 4 + Integer → 3 * 4 + 1`

- What has precedence, + or *?

Alternative to BNF

- The use of **regular expressions** is a common alternate way to express a language

	Regular Expression	**Meaning**
	`x`	A character (stands for itself)
	`"xyz"`	A literal string (stands for itself)
	`M	N`	M or N
Kleene Star	`M N`	M followed by N (concatenation)
	`M*`	Zero or more occurrences of M
	`M+`	One or more occurrences of M
	`M?`	Zero or one occurrence of M
	`[a-zA-Z]`	Any alphabetic character
	`[0-9]`	Any digit
	`.`	Any single character
	ε	The empty string

Regex to EBNF

- Sometimes the following variations on "standard" regular expressions are used:
  - { M } means zero or more occurrences of M
  - (M | N) means one of M or N must be chosen
  - [M] means M is optional

Use "{" to mean the literal { not the regex {

Regular Expressions

- Numerical literals in Pascal may be generated by the following:

$$\begin{array}{lll} \textit{digit} & \longrightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textit{unsigned\_integer} & \longrightarrow & \textit{digit digit} \, * \\ \textit{unsigned\_number} & \longrightarrow & \textit{unsigned\_integer} \; ((\; \texttt{.} \; \textit{unsigned\_integer}) \mid \epsilon) \\ & & (((\texttt{e} \mid \texttt{E}) (\texttt{+} \mid \texttt{-} \mid \epsilon) \; \textit{unsigned\_integer}) \mid \epsilon) \end{array}$$

RegEx Examples

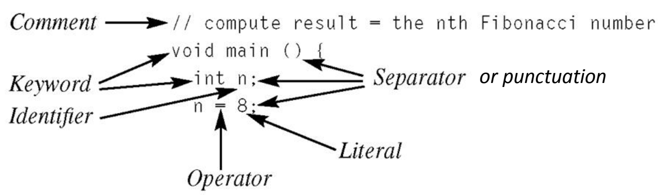
- Booleans
  - “true” | “false”
- Integers
  - (0-9)+
- Identifiers
  - (a-zA-Z)(a-zA-Z0-9)*
- Comments (letters/space only)
  - “//”(a-zA-Z)*(“\r” | “\n” | “\r\n”)
- Simple Expressions
  - Expr → Term ((+|-) Term)*
  - Term → Factor ((* | /) Factor) *
- Regular expressions seem pretty powerful
  - Can you write one for the language $a^n b^n$? (i.e. n a’s followed by n b’s)

Regular Expressions != Context Free Grammar

- Regular expressions express a subset of context free grammars
  - Regular Expressions ←→ Regular Languages ←→ Language of a Deterministic Finite State Automaton
  - Context Free Grammars ←→ Context Free Languages ←→ Language of a Pushdown Automata

Lexical Analysis

- **Lexicon** of a programming language – set of all nonterminals from which programs are written
- Nonterminals – referred to as **tokens**
  - Each token is described by its **type** (e.g. identifier, expression) and its **value** (the string it represents)
  - Skipping whitespace or comments



Categories of Lexical Tokens

- Identifiers
- Literals
  Includes Integers, true, false, floats, chars
- Keywords
  bool char else false float if int main true while
- Operators
  = || && == != < <= > >= + - * / % ! [ ]
- Punctuation
  ; . { } ( )

Issues to consider: Ignoring comments, role of whitespace, distinguising the < operator from <=, distinguishing identifiers from keywords like “if”

A Simple Lexical Syntax for a Small C-Like Language

Primary → Identifier ["["Expression"]"] | Literal | "("Expression")"
| Type "("Expression")"

Identifier → Letter (Letter | Digit)*
Letter → a | b | … | z | A | B | … Z
Digit → 0 | 1 | 2 | … | 9
Literal → Integer | Boolean | Float | Char
Integer → Digit (Digit)*
Boolean → true | false
Float → Integer . Integer
Char → ‘ ASCIICHAR ‘

Scanning

- Recall scanner is responsible for
  - tokenizing source
  - removing comments
  - (often) dealing with *pragmas* (i.e., significant comments)
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages

Scanning

- Suppose we are building an ad-hoc (hand-written) scanner for Pascal:
  - We read the characters one at a time with look-ahead
- If it is one of the one-character tokens `{ ( ) [ ] < > , ; = + - etc }` we announce that token
- If it is a ., we look at the next character
  - If that is a dot, we announce ..
  - Otherwise, we announce . and reuse the look-ahead

Scanning

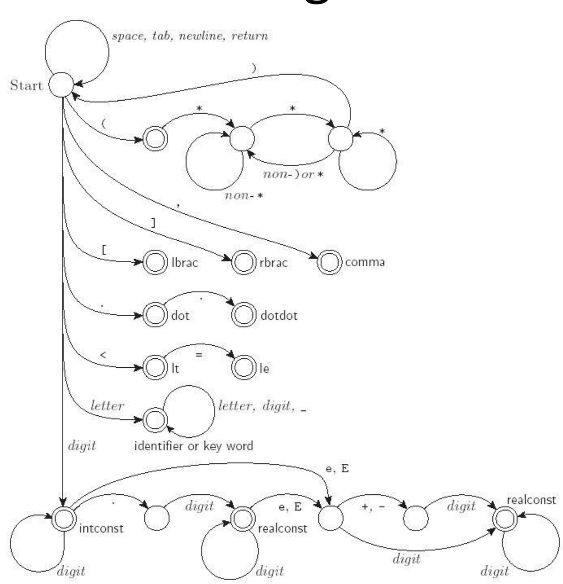
- If it is a <, we look at the next character
  - if that is a = we announce <=
  - otherwise, we announce < and reuse the look-ahead, etc.
- If it is a letter, we keep reading letters and digits and maybe underscores until we can't anymore
  - then we check to see if it is a reserved word

Scanning

- If it is a digit, we keep reading until we find a non-digit
  - if that is not a . we announce an integer
  - otherwise, we keep looking for a real number
  - if the character after the . is not a digit we announce an integer and reuse the . and the look-ahead

Scanning

- Pictorial representation of a Pascal scanner as a finite automaton



Scanning

- This is a deterministic finite automaton (DFA)
  - Lex, scangen, etc. build these things automatically from a set of regular expressions
  - Specifically, they construct a machine that accepts the language
    `identifier | int const`
    `| real const | comment | symbol |`
    `...`
  - This is the **Lexical Syntax** for the programming language

Scanning

- We run the machine over and over to get one token after another
  - Nearly universal rule:
    - always take the longest possible token from the input thus foobar is foobar and never f or foo or foob
    - more to the point, `3.14159` is a real const and never `3`, `.`, and `14159`
- Regular expressions "generate" a regular language; DFAs "recognize" it

Scanning

- Scanners tend to be built three ways
  - ad-hoc
  - semi-mechanical pure DFA
    (usually realized as nested case statements)
  - table-driven DFA
- Ad-hoc generally yields the fastest, most compact code by doing lots of special-purpose things, though good automatically-generated scanners come very close

Scanning

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use perl, awk, sed
- Table-driven DFA is what lex and scangen produce based on an input grammar
  - lex (flex) in the form of C code
  - scangen in the form of numeric tables and a separate driver (for details see Figure 2.11)

Scanning

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - In Pascal, for example, when you have a 3 and you a see a dot
    - do you proceed (in hopes of getting 3.14)? or
    - do you stop (in fear of getting 3..5)?

Scanning

- In messier cases, you may not be able to get by with any fixed amount of look-ahead. In Fortran, for example, we have

```
DO 5 I = 1,25  loop
DO 5 I = 1.25  assignment
```

- Here, we need to remember we were in a potentially final state, and save enough information that we can back up to it, if we get stuck later

Parsing – From lexical to concrete syntax

- Terminology:
  - context-free grammar (CFG)
  - symbols
    - terminals (tokens)
    - non-terminals
  - production
  - derivations (left-most and right-most - canonical)
  - parse trees
  - sentential form

Parsing

- By analogy to RE and DFAs, a context-free grammar (CFG) is a *generator* for a context-free language (CFL)
  - a parser is a language *recognizer*
- There is an infinite number of grammars for every context-free language
  - not all grammars are created equal, however

Parsing

- It turns out that for any CFG we can create a parser that runs in O(n^3) time
- There are two well-known parsing algorithms that permit this
  - Early's algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- O(n^3) time is clearly unacceptable for a parser in a compiler - too slow

Parsing

- Fortunately, there are large classes of grammars for which we can build parsers that run in linear time
  - The two most important classes are called **LL** and **LR**
- LL stands for 'Left-to-right, Leftmost derivation'.
- LR stands for 'Left-to-right, Rightmost derivation’

Parsing

- LL parsers are also called 'top-down', or 'predictive' parsers & LR parsers are also called 'bottom-up', or 'shift-reduce' parsers
- There are several important sub-classes of LR parsers
  - SLR
  - LALR
- We won't be going into detail on the differences between them

Parsing

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

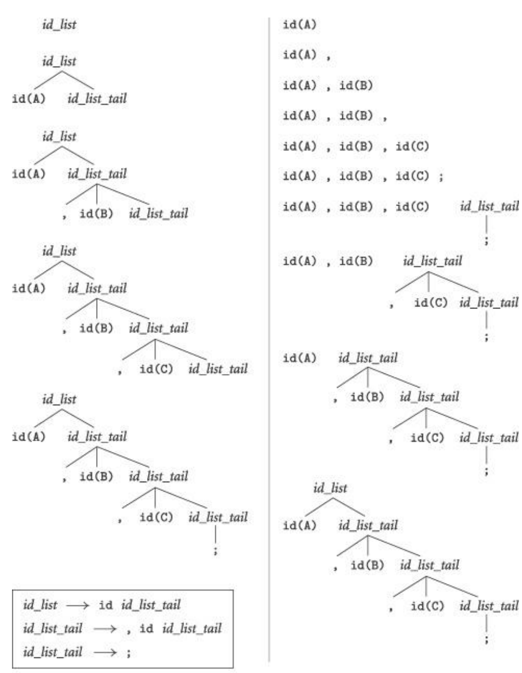
Parsing

- You commonly see LL or LR written with a number in parentheses after it
  - This number indicates how many tokens of look-ahead are required in order to parse
  - Almost all real compilers use one token of look-ahead
- This grammar is LL(1)
  - idlist → idlist id | id

LL vs. LR

Input string: A, B, C;

Token list:
A
,
B
,
C
;



LL Parsing

- Here is an LL(1) grammar for a calculator language (Fig 2.15):

```
1. program      → stmt_list $$
2. stmt_list    → stmt stmt_list
3.              | ε
4. stmt         →   id := expr
5.              | read id
6.              | write expr
7. expr         → term term_tail
8. term_tail    → add_op term term_tail
9.              | ε
```

LL Parsing

- LL(1) grammar (continued)

```
10. term        →   factor fact_tail
11. fact_tail   → mult_op fact fact_tail
12.             | ε
13. factor      →   ( expr )
14.             | id
15.             | number
16. add_op      →   +
17.             | -
18. mult_op     → *
19.             | /
```

LL Parsing

- Example program

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

- First we extract tokens and find identifiers
- We start at the top and predict needed productions on the basis of the current left-most non-terminal in the tree and the current input token
  - Called **recursive descent**

Recursive Descent Parser

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

void program()
        if input_token = ID, READ, WRITE, $$
                stmt_list()
                match($$)
        else parse_error

void stmt_list()
        if input_token = ID, READ, WRITE
                stmt();
                stmt_list();
        if input_token = $$
                skip
        else parse_error
```

```
1.   program      → stmt_list $$
2.   stmt_list    → stmt stmt_list
3.                | ε
4.   Stmt         →  id := expr
5.                | read id
6.                | write expr
```

Recursive Descent Parser

```
void stmt()
        if input_token = ID
                match(id)
                match(:=)
                expr()
        if input_token = READ
                match(read)
                match(id)
        if input_token = WRITE
                match(write)
                expr()
        else parse_error

void expr()
        if input_token = ID, NUMBER, (
                term();
                term_tail()
        else parse_error
```

```
Stmt  →  id := expr
        | read id
        | write expr
expr → term term_tail
term_tail → add_op term term_tail
        | ε
term      →  factor fact_tail
factor    →  ( expr )
        | id
        | number
```

Recursive Descent Parser

```
void term_tail()
        if input_token = +, -
                add_op()
                term()
                term_tail()
        if input_token = ), ID, READ, WRITE, $$
                skip
        else parse_error

void term()
        if input_token = ID, NUMBER, (
                factor()
                factor_tail()
        else parse_error
```

```
term_tail → add_op term term_tail | ε
term      → factor fact_tail
fact_tail → mult_op fact fact_tail | ε
factor    → ( expr )
          | id
          | number
add_op    →  +  |  -
mult_op   → * | /
```

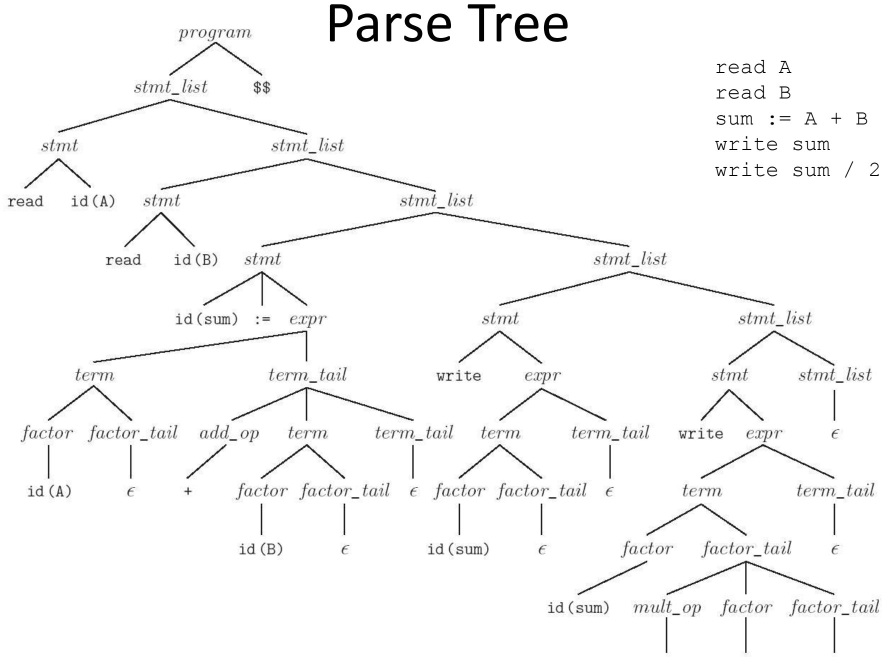
Recursive Descent Parser

```
void factor_tail()
        if input_token = *, /
                mult_op()
                factor()
                factor_tail()
        if input_token = +,-,), ID, READ, WRITE, $$
                skip
        else parse_error

void factor()
        if input_token = ID
                match(id)
        if input_token = NUMBER
                match(number)
        if input_token = (
                match (()
                expr()
                match())
        else parse_error
```

```
term_tail  → add_op term term_tail | ε
term       → factor fact_tail
fact_tail  → mult_op fact fact_tail | ε
factor     → ( expr )
           | id
           | number
add_op     → +  | -
mult_op    → * | /
```

```
void add_op()
        if input_token = +
                match(+)
        if input_token = -
                match(-)
        else parse_error

void mult_op()
        if input_token = *
                match(*)
        if input_token = /
                match(/)
        else parse_error
```

Parse Tree




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

LL Parsing

- Table-driven LL parsing: you have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are
  - (1) match a terminal
  - (2) predict a production
  - (3) announce a syntax error

LL Parsing

- LL(1) parse table for parsing for calculator language

Top-of-stack nonterminal	Current input token id	number	read	write	:=	(	)	+	-	*	/	$$
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

LL Parsing

- To keep track of the left-most non-terminal, you push the as-yet-unseen portions of productions onto a stack
  - for details see Figure 2.20
- The key thing to keep in mind is that the stack contains all the stuff you expect to see between now and the end of the program
  - what you *predict* you will see

LL Parsing

- Problems trying to make a grammar LL(1)
  - left recursion
    - example:

$$\mathit{id\_list} \rightarrow \mathtt{id} \mid \mathit{id\_list}\ ,\ \mathtt{id}$$

equivalently

$$\mathit{id\_list} \rightarrow \mathtt{id}\ \mathit{id\_list\_tail}$$

$$\mathit{id\_list\_tail} \rightarrow ,\ \mathit{id}\ \mathit{id\_list\_tail}$$

$$\mid\ \varepsilon$$

    - we can get rid of all left recursion mechanically in any grammar

LL Parsing

- Problems trying to make a grammar LL(1)
  - common prefixes: another thing that LL parsers can't handle
    - solved by "left-factoring"
    - example:

```
stmt → id := expr | id ( arg_list )
            equivalently
stmt → id id_stmt_tail
id_stmt_tail → := expr
             | ( arg_list)
```

    - we can eliminate left-factor mechanically

LL Parsing

- Note that eliminating left recursion and common prefixes does NOT make a grammar LL
  - there are infinitely many non-LL LANGUAGES, and the mechanical transformations work on them just fine
  - the few that arise in practice, however, can generally be handled with kludges

Bottom-Up and LR Parsing

- Skipping this part in the text
  - Almost always table-driven
- The algorithm to build predict sets is tedious (for a "real" sized grammar), but relatively simple
- It consists of three stages:
  - (1) compute FIRST sets for symbols
  - (2) compute FOLLOW sets for non-terminals (this requires computing FIRST sets for some *strings*)
  - (3) compute predict sets or table for all productions