Parsing

- Transforming a Syntax Chart or EBNF description into a parser is straightforward:
  - for each non-terminal (Syntax Chart or EBNF production) we implement a method of the same name which will parse that construct.
- Consider grammar G4.1:
  - \(<S> ::= \langle A\rangle \mid \langle B\rangle\>
  - \(<A> ::= c\>
  - \(<B> ::= d\>

```c
void nS() {
    next_token symbol = yylex();
    if (symbol.sym == taSym) {
        nA();
        nB();
    }
    else {
        nB();
    }
}
```

G4.2:

```c
void nCompound() {
    while (have(taSym)) {
        nA();
    }
    else if (symbol.sym == taSym) {
        nB();
    }
    else {
        error("a..|b..|...|z.. expected");
    }
}
```

Mustbe & Have – Scanner Interfaces

- These two make the parser much easier to read:
  
  ```c
  if (have(taSym)) {
      ...
  } else if (have(tbSym)) {
      ...
  } else{
      mustbe(tzSym);
  }
  ```

- We can also store the current token in a static variable within the parser

Transformation Rule #1

- The EBNF rule:
  
  ```c
  <S> ::= a..|b..|...|z.. |
  ```

- Is transformed into a multi-way selection:
  ```c
  if (symbol.sym == taSym) {
      yylex();
  } else if (symbol.sym == tbSym) {
      yylex();
  } ...
  ```

- Note that the structure is repetitive — we always test against the symbol and then, if present, advance the scanner
  ```c
  – Introduce a new scanner interface
  – Have a scanner that does this
  ```

- Introduce a second that checks and if the symbol doesn’t match gives an error — mustbe(sym)

When to advance the lexer?

- The Jlex lexer is not very convenient — we need finer control over the advancement to the next token
  ```c
  – Write an alternative to yylex() — next_token which preserves the current token in a static variable within the Parser class
  ```

- This enables us to check it’s value repeatedly without losing the token!

- Code for have & mustbe:

  ```c
  boolean have(int s) {
      if (current_symbol == s) {
          error();
      } else { 
          return (false);
      }
  }
  ```

Transformation Rule #2

- Repetition maps to a while loop
- Let’s make G4.1 more complex

G4.2: \(<S> ::= \langle A\rangle^{*} \langle [ B ]^{*} \rangle\>

```c
void nS() {
    if (symbol.sym == taSym) {
        nA();
    } else if (symbol.sym == tbSym) {
        nB();
    } else {
        error(...);
    }
}
```

Using Syntax Charts

- Syntax charts are the same, but there is usually more in a single syntax chart than EBNF rule
- Eg: Compound Statement in Pascal:
  ```c
  void nCompound() {
      nStatement();
      while (have(SEMICOLON)) {
          nStatement();
      }
      mustbe(END);
  }
  ```

- Lucid
- Can transform back into a syntax chart
- So regular that we could build software that automatically transforms EBNF into Java Code
  ```c
  – Very easy to do!
  ```
Parsing (cont.)

• The transformation rules given so far only work for grammars in which a terminal is the first symbol in any alternative.
  - how do we cope with alternatives that start with a non–terminal?

• Consider G2.1:
  - \( <E> ::= <T> | <E> + <T> \)
  - \( <T> ::= <F> | <T> * <F> \)
  - \( <F> ::= \text{number} | ( <E> ) \)

• We can’t get started on this!
  - The reason is simple: we can’t determine which alternative to choose by looking at the next symbol

• We fix this problem by transforming the grammar into one that describes an identical language
  - There may be an infinite number of such grammars! See later!

Dealing with NonTerminals

• Choose the right path by re–writing (slightly)
  \[
  \begin{align*}
  &<E> ::= <T> \mid + <E> \\
  &<T> ::= <F> \mid * <T> \\
  &<F> ::= \text{number} \mid ( <E> ) \\
  \end{align*}
  \]

void nE()
{
  nT();
  while (have(tPLUS)) {
    nE();
  };
}

void nT()
{
  nF();
  while (have(tSTAR)) {
    nT();
  };
}

void nF()
{
  if (have(tLPAREN)) {
    nE();
    mustbe(tRPAREN);
  } else {
    nNumber();
  };
}