

## 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:
  - $\langle S \rangle ::= a \langle A \rangle \mid b \langle B \rangle$
  - $\langle A \rangle ::= c$
  - $\langle B \rangle ::= d$

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

- $nA()$  and  $nB()$  are similar...
- Why is  $nS()$  poorly implemented?

## Transformation Rule #1

- The EBNF rule:  
 $\langle . \rangle ::= a. \mid b. \mid \dots \mid z.$
- Is transformed into a multi-way selection:

```
if (symbol == taSym) {
  ...yylex();...
} else if (symbol == tbSym) {
  ...yylex();...
  ...
} else if (symbol == tzSym) {
  ...yylex();...
} else {
  error("a..z expected!");
};
```

- Note that the structure is *repetitive* – we always test against the symbol and then, if present, advance the scanner
  - Introduce a new scanner interface – **have(sym)** that does this
  - Introduce a second that checks and if the symbol doesn't match gives an error – **mustbe(sym)**

## Mustbe & Have – Scanner Interfaces

- These two make the parser *much* easier to read:

```
if (have(taSym)) {
  ...
} else if (have(tbSym)) {
  ...
} else {
  mustbe(tzSym);
};
```

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

## When to advance the lexer?

- The Jflex lexer is not very convenient – we need finer control over the advancement to the next token
  - 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`:

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

void mustbe(int s) {
  if (current_symbol == s) {
    next_symbol();
  } else {
    error(...);
  }
};
```

## Transformation Rule #2

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

G4.2:  $\langle S \rangle ::= [ a A ]^* [ b B ]^*$

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

- IMPORTANT NOTE:  
• See how the code *exactly* reflects the grammar – 1:1 correspondence.

## 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:

```
void nCompound() {
  mustbe(tBEGIN);
  nStatement();
  while (have(tSEMICOLON)) {
    nStatement();
  };
  mustbe(tEND);
};
```
- Lucid
- Can transform *back* into a syntax chart
- So regular that we *could* build software that automatically transforms EBNF into Java Code
  - 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:
  - $\langle E \rangle ::= \langle T \rangle \mid \langle E \rangle + \langle T \rangle$
  - $\langle T \rangle ::= \langle F \rangle \mid \langle T \rangle * \langle F \rangle$
  - $\langle F \rangle ::= \langle \text{number} \rangle \mid ( \langle E \rangle )$
- 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)
    - $\langle E \rangle ::= \langle T \rangle [ + \langle E \rangle ]^*$
    - $\langle T \rangle ::= \langle F \rangle [ * \langle T \rangle ]^*$
    - $\langle F \rangle ::= \langle \text{number} \rangle \mid ( \langle E \rangle )$
- ```
void nE() {
  nT();
  while (have(+PLUS)) {
    nE();
  };
};

void nT() {
  nF();
  while (have(+STAR)) {
    nT();
  };
};

void nF() {
  if (have(+LPAREN)) {
    nE();
    mustbe(+RPAREN);
  } else {
    nNumber();
  }
};
```