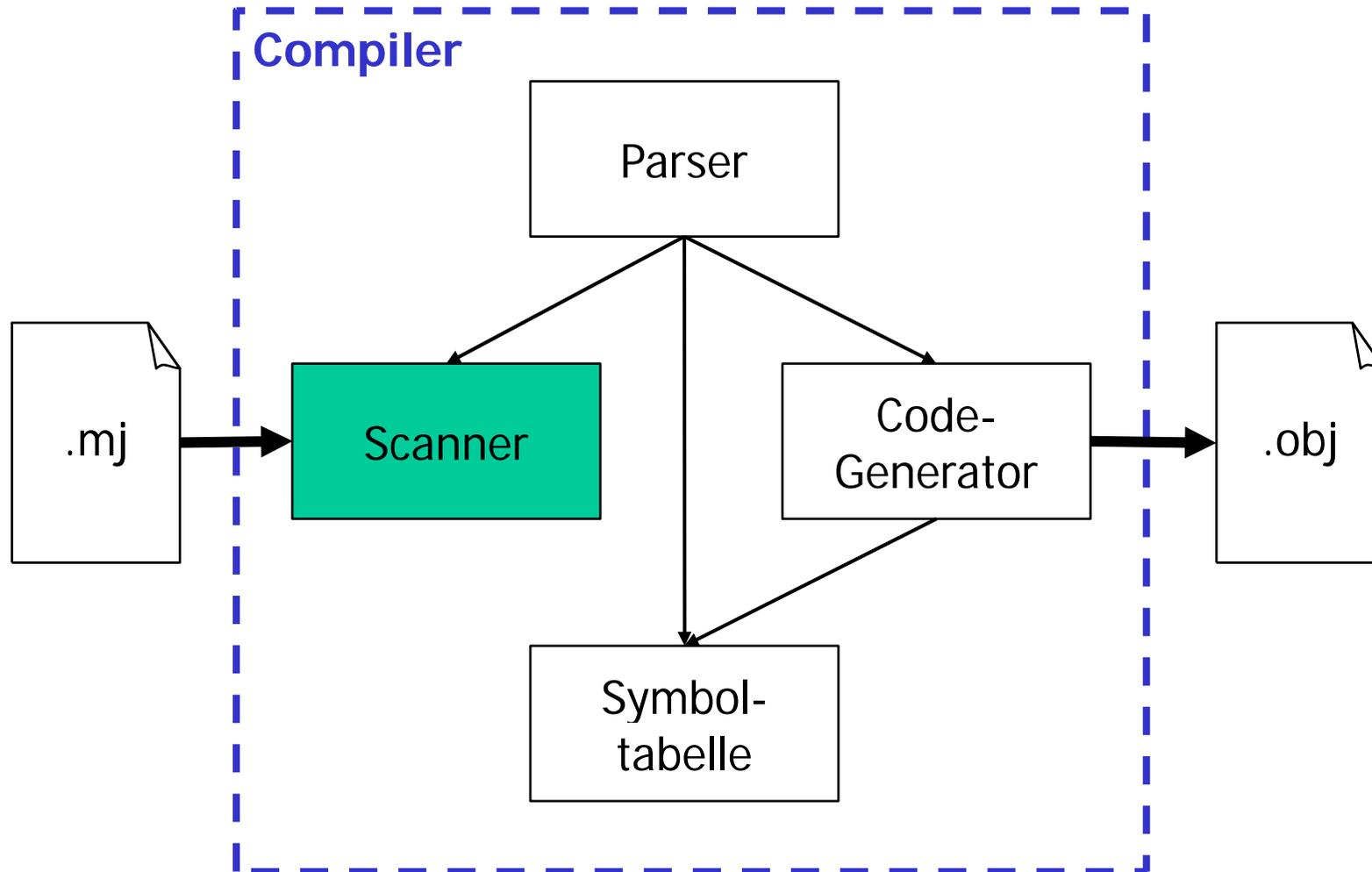


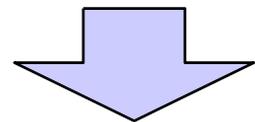
# Struktur des *MicroJava*-Compilers





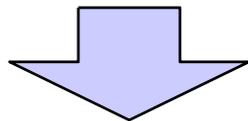
# Grammatik ohne Scanner

Expr = Term { "+" Term } .  
Term = Factor { "\*" Factor } .  
Factor = ident .



erlaube Kommentare an beliebiger Stelle

Expr = {Comment} Term {Comment}  
{ "+" {Comment} Term {Comment} } .  
Term = {Comment} Factor {Comment}  
{ "\*" {Comment} Factor {Comment} } .  
Factor = {Comment} ident {Comment} .



erlaube Whitespace an beliebiger Stelle

...

# Struktur des *MicroJava*-Compilers



- Package `ssw.mj`
  - `Scanner.java` + `Token.java`: Übung 2
  - `Parser.java`: Übung 3 (und alle weiteren Übungen)
  - Hilfsklassen für Fehlermeldungen
- Package `ssw.mj.symtab`
  - Verwaltung der Symboltabelle: Übung 4
- Package `ssw.mj.codegen`
  - Code-Generator: Übung 5 und Übung 6
- Unterschiede zur Vorlesung
  - **Keine statischen Felder:** Vereinfacht die JUnit-Testfälle
  - **Enumerationen:** Typsicherheit bei Konstanten
- Diese Struktur darf nicht verändert werden
  - Keine zusätzlichen Klassen nötig
  - Bereits vollständig vorgegebene Klassen (= Klassen ohne TODO-Kommentar) nicht verändern

# Fehlermeldungen



- Klasse **Errors** sammelt alle Fehlermeldungen  
`void error(int line, int col, Message msg, Object... msgParams);`
- Fehlermeldungen sind in Enumeration `Errors.Message` definiert
  - Der `error`-Methode wird die Meldung übergeben
  - Manche Fehlermeldungen benötigen Parameter
    - Zusätzliche Parameter der `error`-Methode
  - Meldungstexte sind in der Enumeration definiert
- Hilfsmethode im Scanner (später auch im Parser)  
`void error(Token t, Message msg, Object... msgParams);`
  - Übernimmt die Fehlerposition aus dem angegebenen Token

# Klasse Scanner + Token



```
class Scanner {  
    public Scanner(Reader r);  
    public Token next();  
}
```

```
class Token {  
    Kind      kind;    // z.B. Kind.ident, Kind.assign, ...  
    int       line;    // Zeilenposition  
    int       col;     // Spaltenposition  
    int       val;     // numerischer Wert für number und charConst  
    String    str;     // Name von ident  
}
```

- Scanner wird (ab der 3. Übung) vom Parser aufgerufen
  - Jeder Aufruf von next() liefert das nächste Token
  - Scanner wartet, bis er aufgerufen wird

# Aufgaben des Scanners



- Erkennen von Terminalsymbolen
- Überlesen unbedeutender Zeichen (Blanks, Tabs, Zeilenumbrüche, ...)
- Überlesen von Kommentaren
- Erkennen von:
  - Namen
  - Schlüsselwörtern
  - Zahlen
  - Zeichenkonstanten
- Bildung von Terminalklassen (ident, number, ...)  
int, char, null, chr, ord, len sind **keine** Schlüsselwörter,  
nur vordeklarierte Namen (→ Erkennung als ident)
- Erkennen des Dateiendes
- lexikalische Fehler melden (Zahlenformat, ungültige Zeichen, ...)
- Einstellen der Token-Attribute (Symbolart, Position, Wert, ...)



# Scanner.next() (1)

```
public Token next() {  
    while (Character.isWhitespace(ch)) {  
        nextCh();    // skip white space  
    }  
}
```

```
Token t = new Token(none, line, col);
```

```
switch (ch) {  
    //----- identifier or keyword  
    case 'a': case 'b': ... case 'z':  
    case 'A': case 'B': ... case 'Z':  
        readName(t);    // distinguish between identifier and keyword  
        break;  
    //----- number  
    case '0': ... case '9':  
        readNumber(t);  
        break;  
    . . .  
}
```



## Scanner.next() (2)

```
...
//----- simple tokens
case ';': t.kind = semicolon; nextCh(); break;
case EOF: t.kind = eof; /* no nextCh() */ break;
//----- compound tokens
case '=':
    nextCh();
    if (ch == '=') { t.kind = eql; nextCh(); }
    else { t.kind = assign; }
    break;
case '/':
    nextCh();
    if (ch == '*') { skipComment(t); t = next(); /* recursion */ }
    else { . . . } break;
default:
    error(t, INVALID_CHAR, ch); nextCh();
    break;
}
return t;
}
```

# Hilfsmethoden



- **void nextCh()**
  - Liest das nächste Eingabezeichen und speichert es im Feld ch
    - oder EOF beim Dateiende
  - Führt die Position in den Feldern line und col mit
    - Erkennt Zeilenumbrüche: LF oder CR LF
- **void readName(Token t)**
  - Liest einen Bezeichner
  - Erkennt Schlüsselwörter (HashMap String → Token.Kind)
- **void readNumber(Token t)**
  - Liest eine Zahlenkonstante
- **void readCharConst(Token t)**
  - Liest eine Zeichenkonstante
- **void skipComment(Token t)**
  - Überliest geschachtelte Kommentare
  - ch enthält anschließend das Zeichen nach dem Kommentar

# Zahlen-Konstanten



- Gültige Zahlen
  - Normale Zahlen: 123
    - Ein Token: *number*
  - Negative Zahlen: -123
    - Zwei Tokens: *minus* und *number*
  - Buchstaben: 123abc
    - Zwei Tokens: *number* und *ident*
  - *Achtung*: abc123
    - Ein Token: *ident*
  
- Fehlerhafte Zahlen
  - Zu große Zahlen: 2147483648      `error(t, BIG_NUM, str);`
  - Spezialfall: -2147483648      `error(t, BIG_NUM, str);`

# Zeichen-Konstanten



- Gültige Zeichen-Konstanten
  - Normale Zeichen: 'A'
  - Escape-Sequenzen: '\r', '\n', '\'' und '\\'
- Fehlerhafte Zeichen-Konstanten
  - Kein Zeichen: ''           error(t, EMPTY\_CHARCONST);
  - Fehlendes Ende: 'A       error(t, MISSING\_QUOTE);
  - Escape-Sequenzen: '\A'   error(t, UNDEFINED\_ESCAPE, ch);
  - Zeilenumbruch: '\n'      error(t, ILLEGAL\_LINE\_END);

# Kommentare



- Nur Block-Kommentare
  - Scanner ignoriert alles zwischen `/*` und `*/`
  - Kommentare können auch geschachtelt sein
    - `/* a /* b */ c */`
  - Methode `skipComment()` muss daher die Schachtelungstiefe mitführen
- Fehlerhafte Kommentare
  - Fehlendes Ende: `/* ohne */` `error(t, EOF_IN_COMMENT);`

# MicroJava



- eine einzige Quellcode-Datei
- Hauptmethode `void main()`: kein Rückgabewert, keine Parameter
- Typen: `int` (4 Byte), `char` (1 Byte, ASCII)
- globale und lokale Variablen, globale Konstanten
- eindimensionale Arrays
- Records (sehen aus wie innere Klassen)
- Parameterübergabe *call-by-value* (Objektparameter sind aber Referenzen)
- Ein-/Ausgabe mit Hilfe der *read*- und *print*-Anweisung
- eingebaute Methoden `ord()`, `chr()`, `len()` und Konstante `null`
- keine Packages und Importanweisungen, ...
- kein GC und auch kein `delete` (Objekte bleiben übrig – who cares 😊)
- nur `while`-Schleife
- keine Ausnahmebehandlung (*exception handling*)
- keine Zeiger

# UE 2: Lexikalische Analyse (*Scanner*)



- MJ-Angabe
  - `svn://ssw.jku.at/2008W/UB/k<MatrNr>/trunk/Compiler`
  - Compilerklassen (Token.java, Errors.java, Gerüst von Scanner.java, ...)
- MJ-Tests
  - `svn://ssw.jku.at/2008W/UB/k<MatrNr>/trunk/Tests`
  - JUnit-Testfälle
  - nächste Woche in der UE:
    - Testen mit JUnit
- Abgabe
  - siehe Abgabeanleitung auf Homepage!
  - elektronisch bis Mi, 29.10.2008, 20:15
    - alle zum Ausführen benötigten Dateien
  - auf Papier
    - nur Scanner.java