SSE2-PLDE_1

Contents:

- Language Processors, Compilation
- Lexical Analysis, Scanning
- Regular Expressions, Grammars, Formal Languages

Literature:

- Watt & Brown:
  - 1.3
  - 2.1 (not examples 2.2-2.10), 2.2 (not examples 2.12, 2.13)
  - 3.1
  - 4.1.1, 4.2.1-4.2.2, 4.5

Specification of Programming Languages

Syntax
Contextual constraints
Semantics

Program	::=	single-Command
Command	::=	single-Command
	|	Command **;** single-Command
Single-Command	::=	V-name **:=** Expression
	|	Identifier **(** Expression **)**
	|	**if** Expression **then** single-Command **else** single-Command
	|	**while** Expression **do** single-Command
	|	**let** Declaration **in** single-Command
	|	**begin** Command **end**
Expression	::=	…

n := d + 10 * n

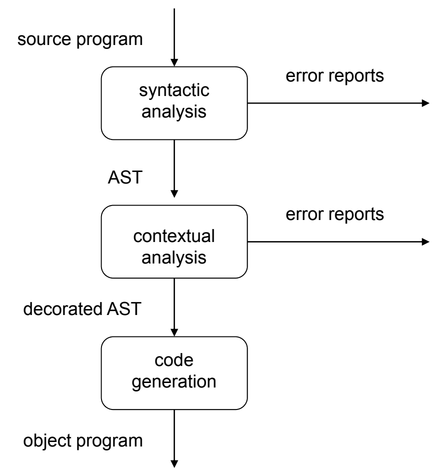
while b do begin n := 0; b := false end

let var y: Integer in y := y + 1

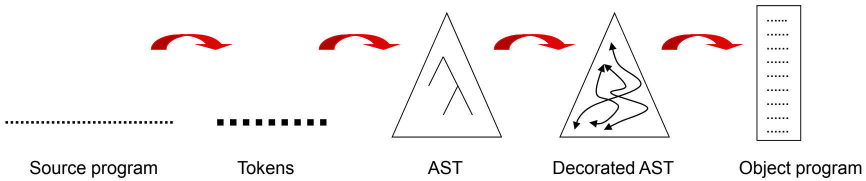
Mini Triangle

Program	::=	single-Command
Command	::=	single-Command
	|	Command **;** single-Command
Single-Command	::=	V-name **:=** Expression
	|	Identifier **(** Expression **)**
	|	**if** Expression **then** single-Command **else** single-Command
	|	**while** Expression **do** single-Command
	|	**let** Declaration **in** single-Command
	|	**begin** Command **end**
Expression	::=	primary-Expression
	|	Expression Operator primary-Expression
primary-Expression	::=	Integer-literal
	|	V-name
	|	Operator primary-Expression
	|	**(** Expression **)**
V-name	::=	Identifier
Declaration	::=	single-Declaration
	|	Declaration **;** single-Declaration
single-Declaration	::=	**const** Identifier **~** Expresion
	|	**var** Identifier **:** Type-denoter
Type-denoter	::=	Identifier
Operator	::=	**+** | **-** | ***** | **/** | **<** | **>** | **=** | ****
Identifier	::=	Letter | Identifier Letter | Identifier Digit
Integer-Literal	::=	Digit | Integer-Literal Digit
Commet	::=	**!** Graphic* eol

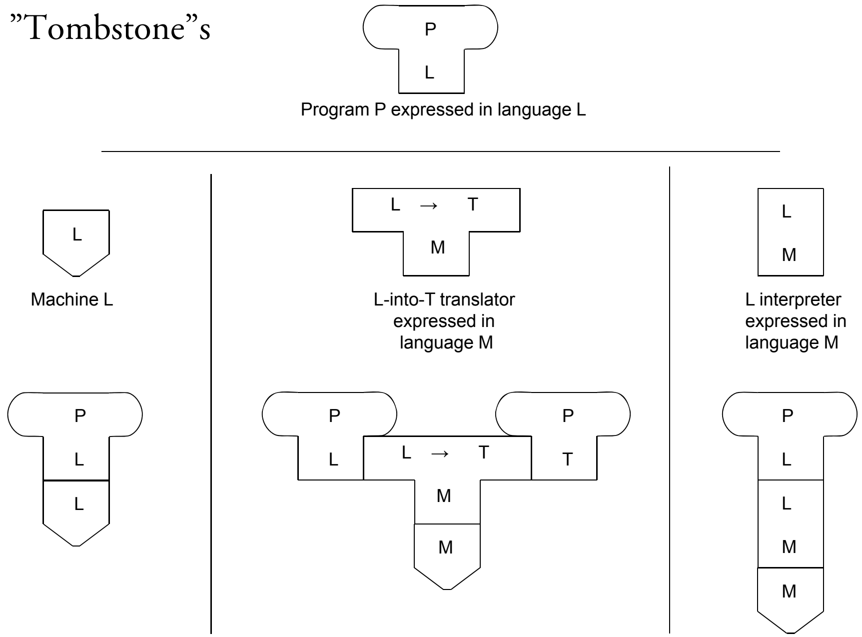
Translation



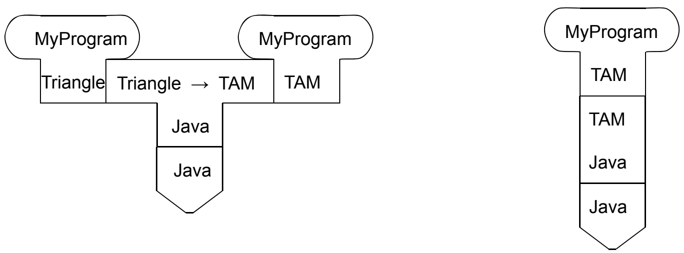
Translation



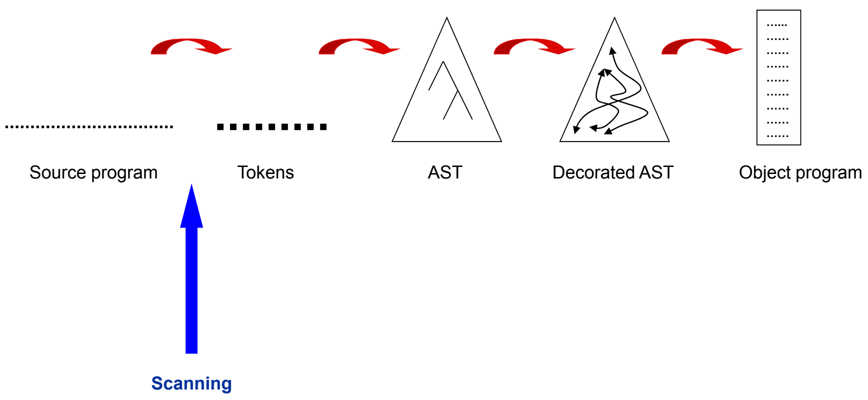
”Tombstone”s



Triangle



Translation



Tokens

let *var* *ident.* *colon* *ident.* *in* *ident.* *becomes* *ident.* *op.* *intlit.* *eot*

let var y: Integer in ! New Year
y := y + 1

let	*var*	*ident.*	*colon*	*ident.*	*in*	*ident.*	*becomes*	*ident.*	*op.*	*intlit.*	*eot*
let	var	y	:	Integer	in	y	:=	y	+	1	

Tokens

Identifier, integer, text (a | b | … | z) (a | b | … | z | 0 | 1 | … | 9)*

letter = (a | b | c | … | z)
digit = (0 | 1 | … | 9)
identifier = letter (letter | digit)*

Reserved words then | begin | end | else | …

Special symbols, operators := | : | = | <= | >= | <> | (|) | ; | , | + | - | …

Combinations (begin end)

Priority, ambiguity begincount := : =

Regular Expression (RE)

empty	ε
singleton	t
concatenation	X Y (or X • Y)
alternative	X | Y
iteration	X*
grouping	(X)

Tokens, Mini-Triangle

Token	::=	Indetifier | Integer_literal | Operator | ; | : | := | ~ | (|) | eot
Identifier	::=	Letter | Identifier Letter | Identifier Digit
Integer_Literal	::=	Digit | Integer-Literal Digit
Operator	::=	+ | - | * | / | < | > | = | \
Separator	::=	Comment | space | eol
Comment	::=	! Graphic* eol

And some additional remarks …

parser.java

```
public class Parser {

  private Scanner lexicalAnalyser;
  ...
  private Token currentToken;
  ...

  void acceptIt() {
    ...
    currentToken = lexicalAnalyser.scan();
  }

  ...

}
```

```
public Program parseProgram() {

   …
   currentToken = lexicalAnalyser.scan();

   try {
     Command cAST = parseCommand();
     …
     if (currentToken.kind != Token.EOT) {
       …
     }
   }
   …
}
```

scanner.java

```
public final class Scanner {
  private boolean isLetter(char c) {
    ...
  }

  private boolean isDigit(char c) {
    ...
  }

  private boolean isOperator(char c) {
    ...
  }

  ...

  private void takeIt() {
    ...
  }

  private void scanSeparator() {
    ...
  }

  private int scanToken() {
    ...
  }

  public Token scan () {
     ...
  }
}
```

token.java

```
public static final int

  // literals, identifiers, operators...
  INTLITERAL           = 0,
  …

  // reserved words
  ARRAY                = 4,
  …
  // punctuation...
  DOT                  = 21,
  …
  // brackets...
  LPAREN               = 27,
  …
  // special tokens...
  …

private static String[] tokenTable = new String[] {
  "<int>",
  …
  "}",
  "",
  "<error>"
};
```

```
final class Token extends Object {

  protected int kind;
  …

  public Token(int kind,
        String spelling,
        SourcePosition position) {
    …
  }
  …

…
}
```

token.java

```
// reserved words
 ARRAY                     = 4,
 BEGIN                     = 5,
 CONST                     = 6,
 DO                        = 7,
 ...
 VAR                       = 19,
 WHILE                     = 20,
```

```
if (kind == Token.IDENTIFIER) {
     int currentKind = firstReservedWord;
     boolean searching = true;

     while (searching) {
       int comparison = tokenTable[currentKind].compareTo(spelling);
       if (comparison == 0) {
         this.kind = currentKind;
         searching = false;
       } else if (comparison > 0 || currentKind == lastReservedWord) {
         this.kind = Token.IDENTIFIER;
         searching = false;
       } else {
         currentKind ++;
       }
     }
} else
     this.kind = kind;
```

```
ARRAY
BEGIN
CONST
DO
...
VAR
WHILE
```

scanner.java

```
public Token scan () {
    Token tok;
    SourcePosition pos;
    int kind;

    currentlyScanningToken = false;
    while (currentChar == '!'
           || currentChar == ' '
           || currentChar == '\n'
           || currentChar == '\r'
           || currentChar == '\t')
      scanSeparator();

    currentlyScanningToken = true;
    currentSpelling = new StringBuffer("");
    pos = new SourcePosition();
    pos.start = sourceFile.getCurrentLine();

    kind = scanToken();

    pos.finish = sourceFile.getCurrentLine();
    tok = new Token(kind, currentSpelling.toString(), pos);
    …
    return tok;
}
```

scanner.java

```
private int scanToken() {
    …
}
```

```
switch (currentChar) {

case 'a':  case 'b':  case 'c':  case 'd':  …
case 'A':  case 'B':  case 'C':  case 'D':  …
takeIt();
  while (isLetter(currentChar) || isDigit(currentChar))
    takeIt();
  return Token.IDENTIFIER;

case '0':  case '1':  case '2':  case '3':  case '4':
case '5':  case '6':  case '7':  case '8':  case '9':
  takeIt();
  while (isDigit(currentChar))
    takeIt();
  return Token.INTLITERAL;

case '+':  case '-':  case '*': case '/':  case '=':
case '<':  case '>':  case '\\':  case '&':  case '@':
case '%':  case '^':  case '?':
  takeIt();
  while (isOperator(currentChar))
    takeIt();
  return Token.OPERATOR;
```

scanner.java

```
private int scanToken() {
    …
}
```

```
case '\'':
  takeIt();
  takeIt(); // the quoted character
  if (currentChar == '\'') {
    takeIt();
    return Token.CHARLITERAL;
  } else
    return Token.ERROR;

case '.':
  takeIt();
  return Token.DOT;

case ':':
  takeIt();
  if (currentChar == '=') {
    takeIt();
    return Token.BECOMES;
  } else
    return Token.COLON;

case ';':
  takeIt();
  return Token.SEMICOLON;

case ',':
  takeIt();
  return Token.COMMA;

case '~':
  takeIt();
  return Token.IS;

case '(':
  takeIt();
  return Token.LPAREN;

…

case SourceFile.EOT:
  return Token.EOT;

default:
  takeIt();
  return Token.ERROR;
 }
}
```

scanner.java

```
private boolean isDigit(char c) {
  return (c >= '0' && c <= '9');
}

private void takeIt() {
  if (currentlyScanningToken)
    currentSpelling.append(currentChar);
  currentChar = sourceFile.getSource();
}
```

```
private void scanSeparator() {
    switch (currentChar) {
    case '!':
      {
        takeIt();
        while ((currentChar != SourceFile.EOL) && (currentChar != SourceFile.EOT))
          takeIt();
        if (currentChar == SourceFile.EOL)
          takeIt();
      }
      break;

    case ' ': case '\n': case '\r': case '\t':
      takeIt();
      break;
    }
  }
```

Regular Expression (RE)

Notation	Meaning	Explanation
empty	ε	Represents the RE the"empty string" (nothing)
singleton	t	Represents the RE any single symbol t in Σ
concatenation	X Y (or X • Y)	Represents any RE of X concatenated by any RE of Y
alternation	X | Y	Represents any RE of X or any RE of Y
iteration	X*	Represents zero or more iterations of any different RE of X
grouping	(X)	Represents any RE of X (just seen as a group)

- Any RE X is based on an alphabet Σ and defines a language L(X) consisting of strings of symbols from Σ
- Σ* means the set of all strings, i.e. sequences of symbols, from Σ, including the empty string
- The meaning of an RE X is a subset of Σ* (L(X) denotes this subset)
- L(X) = { … } i.e. a set (Ø is the empty set)

(a | b | … | z) (a | b | … | z | 0 | 1 | … | 9)*

Extended Regular Expression (RE)

Extended RE Notation	Meaning
X^+	$X^* - \{ \varepsilon \}$
X^n	$X^{(n-1)} X$,n>0
X^0	$\{ \varepsilon \}$
$X^?$	$(\varepsilon \mid X)$
Ø	{ }

Notation for Regular Definition:

a = “regular expression”
b = … a …

- a Regular Definition is a sequence of named regular expressions of the form a = “regular expression”
- the name “a” defined for a regular expression can be used to define other named regular expressions
- in b = … a … the name “a” is used in order to define the name “b”
- no circularity is allowed

(a | b | … | z) (a | b | … | z | 0 | 1 | … | 9)*

letter = (a | b | c | … | z)
digit = (0 | 1 | … | 9)
identifier = letter (letter | digit)*

Regular Expression, Grammar & Language

- Regular expressions and grammars are notations
- Each defines a language
- A language is a set of strings.
- Extended regular expressions define the same languages as regular expressions
- Some languages defined by context free grammars cannot be defined by regular expressions

Some examples

b* a	defines L = { $b^n a$ | n>=0 }
S ::= bS | a	also defines L
T ::= Sa S ::= Sb | **ε**	also defines L
S::= aSb | c	defines { $a^n c b^n$ | n>=0 }

Notation

Alphabet	N and Σ, where A, B, …€ N a, b, …€ Σ α, β, …€ (N U Σ)* (but the book also uses X, Y, …€ (N U Σ)* and N, M ... € N)
Regular Expression (RE)	Operators include | • * ( ) ε and sometimes even ? + Ø means empty set of strings
Extended Regular Expression	Additional operators are included
Regular Definitions	A sequence of non-circular definitions is included
Context Free Grammar (CFG)	Productions have the form (N, (N U Σ)*), i.e. the righthandside is a string of symbols from N U Σ, and are denoted A→α The language of (CFG) grammar G is denoted L(G)
Extended CFG (ECFG)	Productions have the form (N, R(N U Σ)), i.e. the righthandside is a regular expression, and are denoted A→ α The language of (ECFG) grammar G is denoted L(G)
Context-Sensitive Grammar (CSG)	Productions have the form ((N U Σ)*, (N U Σ)*), i.e. lefthandside and righthandside are strings of symbols from N U Σ, and are denoted β →α

Context Free Grammar

- A context free grammar G has the form G = (N, Σ, P, S) where N is the nonterminal symbol alphabet, Σ is the terminal symbol alphabet, P is a set of productions of the form (N, (N U Σ)*), and S € N is the start symbol.
- L(G) denotes the language generated by G, i.e. a subset of Σ* (strings or sentences of L(G))
- A string x in Σ* is in L(G) if and only if x can be derived from the start symbol S, denoted S =>* x
- If ...A... in (N U Σ)* and A→ α is in P, then ...α... also in (N U Σ)* can be derived from ...A..., denoted ...A...=>...α...
- =>* is the transitive closure of =>, i.e. =>* is => repeated zero or more times
- The start symbol S can be derived from itself, i.e. S =>* S
- If ...A... in (N U Σ)* can be derived from S, i.e. S =>*...A..., and A→ α is in P, then ... α... can be derived from S, i.e. S =>*...A...=>...α..., or S =>*...α...

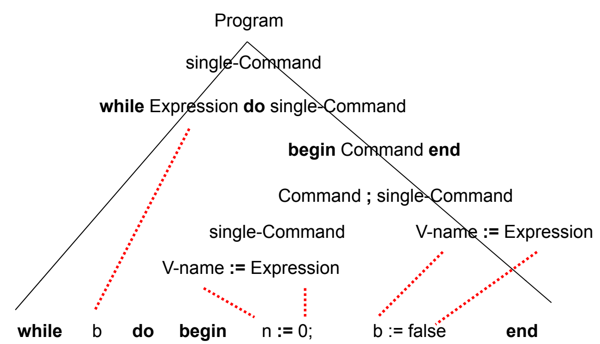
G = ({S}, {a, b, c}, {S::= aSb | c }, S)

L(G) = { $a^n c b^n$ | n>=0 }

S

Context Free Grammar

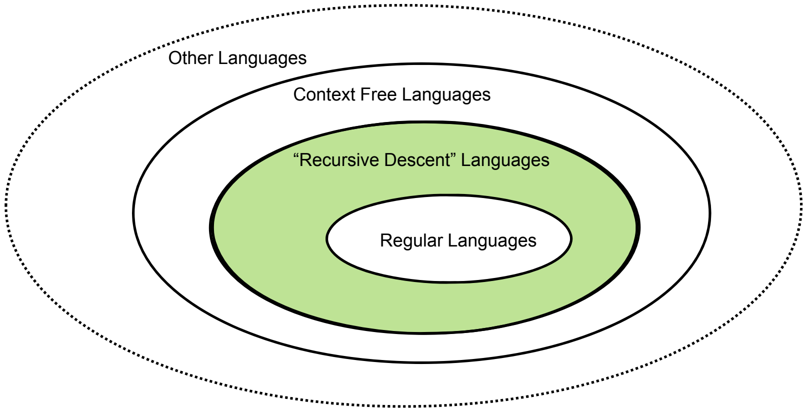
Program ::=		single-Command
Command ::=		single-Command
	|	Command **;** single-Command
Single-Command	::=	V-name **:=** Expression
	|	Identifier **(** Expression **)**
	|	**if** Expression **then** single-Command **else** single-Command
	|	**while** Expression **do** single-Command
	|	**let** Declaration **in** single-Command
	|	**begin** Command **end**
Expression ::=	…	



Languages

Given Σ,

- any Regular language over Σ is also a Recursive Descent language over Σ
- any Recursive Descent language over Σ is also a Context Free language over Σ
- any Context Free language over Σ is also a ... language over Σ



Languages

{ b }* a	defines a Regular Language
$a^n c b^n$, n≥0	is not a Regular Languages, but is a "Recursive Descent" Language
ww^R , w € Σ*	is not a "Recursive Descent" Language but is a Context Free language (w^R means w in reverse order)
ww , w € Σ*	is not a Context Free language