SSE2-PLDE_2

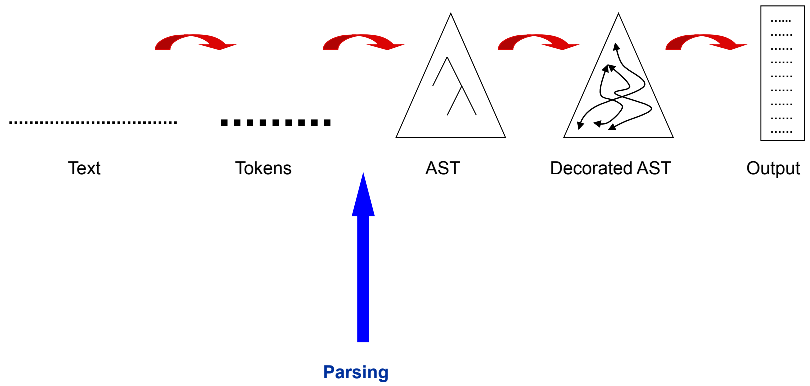
Contents:

- Syntactic Analysis, Parsing, Syntax Tree
- Grammar Transformations
- Recursive Descent
- Abstract Syntax Tree (AST)

Literature:

- Watt & Brown:
  - 4.2.3-4.2.4, 4.3.4(not examples 4.15, 4.16), 4.4

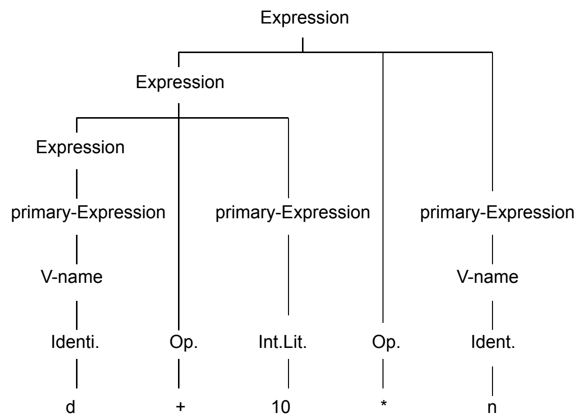
Translation



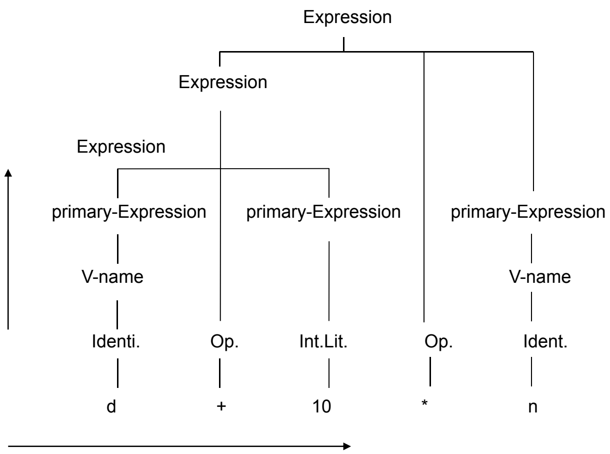
Mini Triangle

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

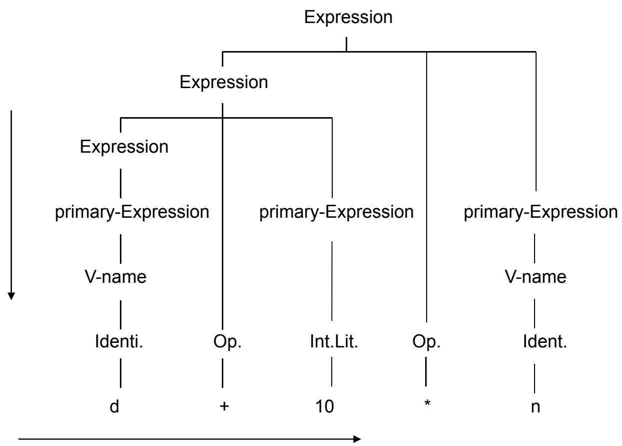
Syntax Tree



Bottom Up Parsing



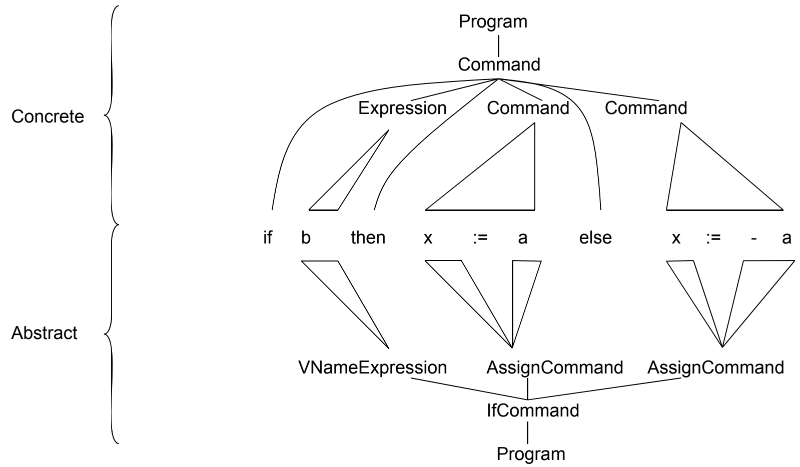
Top Down Parsing



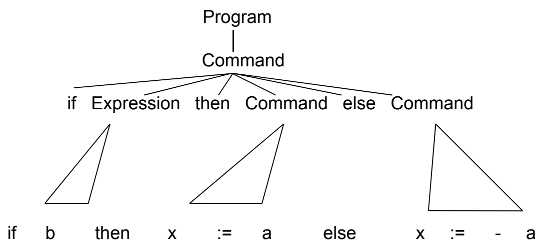
Mini Triangle: Abstract

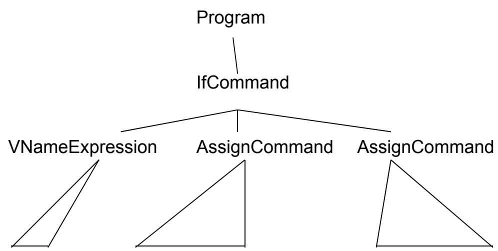
Program	::=	Command	Program
Command	::=	V-name **:=** Expression	AssignCommand
	|	Identifier **(** Expression **)**	CallCommand
	|	Command **;** Command	SequentialCommand
	|	**if** Expression **then** Command **else** Command	IfCommand
	|	**while** Expression **do** Command	WhileCommand
	|	**let** Declaration **in** Command	LetCommand
Expression	::=	Integer-literal	IntetegerExpression
	|	V-name	VNameExpression
	|	Operator Expression	UnaryExpression
	|	Expression Operator Expression	BinaryExpression
V-name	::=	Identifier	SimpleVname
Declaration	::=	**const** Identifier **~** Expresion	ConstDeclaration
	|	**var** Identifier **:** Type-denoter	VarDeclaration
	|	Declaration **;** Declaration	SequentialDeclaration
Type-denoter	::=	Identifier	SimpleTypeDenoter

(Abstract) Syntax Tree



Recursive Descent (RD) & Abstract Syntax Tree (AST)





Notation: Extended BNF (EBNF)

BNF:

A ::= α | ß | …

or

A → α | ß | …

where α and ß are strings over nonterminal and terminal symbols

EBNF:

A ::= X

or

A → X

where X is an extended regular expression over nonterminal and terminal symbols

Mini Triangle: Recursive Descent

Program	::=	Command
Command	::=	single-Command (**;** single-Command)*
Single-Command	::=	Identifier (**:=** Expression | **(** 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 (Operator primary-Expression)*
primary-Expression	::=	Integer-literal
	|	V-name
	|	Operator primary-Expression
	|	**(** Expression **)**
Declaration	::=	single-Declaration (**;** single-Declaration)*
single-Declaration	::=	**const** Identifier **~** Expresion
	|	**var** Identifier **:** Type-denoter
Type-denoter	::=	Identifier

RD: Command

Command ::= single-Command (; single-Command)*

```
parseCommand() {
   parseSingleCommand();
   while (currentToken.kind == Token.SEMICOLON) {
     acceptIt();
     parseSingleCommand();
   }
}
```

RD: Single-Command

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

```
parseSingleCommand(){
    switch (currentToken.kind) {

    case Token.IDENTIFIER:
      {
        parseIdentifier();
        if (currentToken.kind == Token.LPAREN) {
          acceptIt();
          parseExpression();
          accept(Token.RPAREN);

        } else {

          accept(Token.BECOMES);
          parseExpression();
        }
      }
      break;
    …
    }
}
```

RD: Single-Command

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

```
parseSingleCommand(){
    switch (currentToken.kind) {

    case Token.BEGIN:
      acceptIt();
      parseCommand();
      accept(Token.END);
      break;

    case Token.LET:
      {
        acceptIt();
        parseDeclaration();
        accept(Token.IN);
        parseSingleCommand();}
      break;
    …
    default:
      …
      break;
    }
}
```

```
case Token.IF:
      {
        acceptIt();
        parseExpression();
        accept(Token.THEN);
        parseSingleCommand();
        accept(Token.ELSE);
        parseSingleCommand();
      }
      break;

    case Token.WHILE:
      {
        acceptIt();
        parseExpression();
        accept(Token.DO);
        parseSingleCommand();
      }
      break;
```

Recursive Descent

followers[[X]] in context N::= …X Z is *starters*[[Z]]

If Z can derive **ε** then also include *followers*[[N]] in **any** context

- N ::= X

  private void parse*N* () {
  parse X
  }

- parse ? to be refined recursively:
  - parse ε — dummy statement
  - parse t — `accept(t);` — if t is already known `acceptit();`
  - parse N — `parse`*N*`();`
  - parse (X Y) — `{ parse X parse Y }` — X1 X2 … Xn similarly
  - parse (X | Y) — X1 | X2 |…| Xn similarly

    switch `(currentToken.kind) {`
    `cases in` *starters*[[X]]:
    parse X
    break;
    `cases in` *starters*[[Y]]:
    parse Y
    break;
    default: report a syntactic error
    }

    starters[[X]] and *starters*[[Y]] disjoint

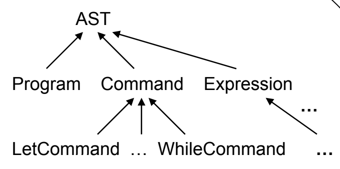
    If *Y* can derive **ε** then *starters*[[X]] **also** disjoint from *followers*[[X|Y]] in this context
  - parse (X*) — **while** `(currentToken.kind is in` *starters*[[X]])
    parse X

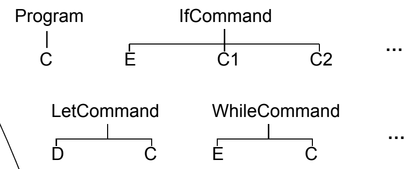
    starters[[X]] disjoint from *followers*[[X*]] in this context

AST

Program	::=	Command	Program
Command	::=	V-name **:=** Expression	AssignCommand
	|	Identifier **(** Expression **)**	CallCommand
	|	Command **;** Command	SequentialCommand
	|	**if** Expression **then** Command **else** Command	IfCommand
	|	**while** Expression **do** Command	WhileCommand
	|	**let** Declaration **in** Command	LetCommand
Expression	::=	Integer-literal	IntetegerExpression
	|	V-name	VNameExpression
	|	Operator Expression	UnaryExpression
	|	Expression Operator Expression	BinaryExpression

…





AST

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

```
public abstract class Command extends AST {…}

public class LetCommand extends Command {
  ...
  public Declaration D;
  public Command C;
}
public class IfCommand extends Command {
  ...
  public Expression E;
  public Command C1, C2;
}
public class WhileCommand extends Command {
  ...
  public Expression E;
  public Command C;
}
public abstract class Expression extends AST {…}

public abstract class Declaration extends AST {…}
```

```
Command parseCommand() {
    …
}

Command parseSingleCommand(){
    …
}

Expression parseExpression() {
    …
}

Declaration parseDeclaration() {
    …
}
```

AST

Command	::=	single-Command (**;** single-Command)*
Single-Command	::=	Identifier (**:=** Expression | **(** Expression **)**)
	|	**if** Expression **then** single-Command **else** single-Command
	|	**…**

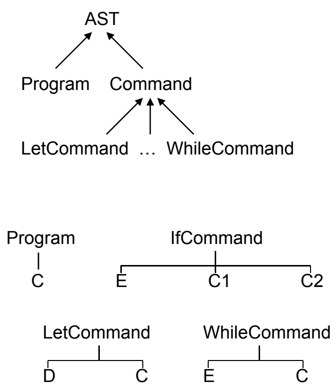
```
public abstract class Command extends AST {…}

public class SequentialCommand extends Command {
  ...
  public Command SC1, SC2;
}

public class IfCommand extends Command {
  ...
  public Expression E;
  public Command C1, C2;
}

...
```

```
Command parseCommand() {
    …
}

Command parseSingleCommand(){
    …
}

...
```

AST

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

```
public abstract class Program extends AST {
  ...
   public Command C;
}
public abstract class Command extends AST {…}
public class LetCommand extends Command {
  ...
  public Declaration D;
  public Command C;
}
public class IfCommand extends Command {
  ...
  public Expression E;
  public Command C1, C2;
}
public class WhileCommand extends Command {
  ...
  public Expression E;
  public Command C;
}
```



AST in Java

1. Each node in AST is an object of a subclass of AST: Include
   `public abstract class AST { … }`

2. For each nonterminal symbol, N, i.e. in N ::= X, include
   `public class "N" extends AST { … }`

3. For each righthandside X_i symbol N named "X_i" include a subclass of the lefthandside (n>1), i.e.

   ...
   N ::= X_i "X_i"
   ...
   `public class "Xi" extends "N" { … }`

4. For each nonterminal symbol, M_j, on a righthandside include a corresponding reference, i.e.
for production N ::= a_0 M_1 a_1 ... M_m a_m with name X_i include

```
public class "Xi" extends … {
   …
   public "Mj" Mj;
   …
}
```

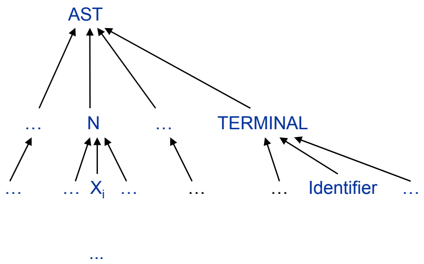
5. Include

```
public abstract class TERMINAL extends AST {
   public String Spelling
}
```

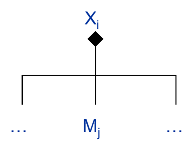
6. For each terminal symbol, i.e. `Identifier`, include
   `public class Identifier extends TERMINAL { … }`

AST in Java

```
public abstract class AST { … }

public class "N" extends AST { … }

public class "Xi" extends "N" { … }

public abstract class TERMINAL extends AST {
   public String Spelling
}

public class Identifier extends TERMINAL { … }
```




```
public class "Xi" extends … {
   …
   public "Mj" Mj;
   …
}
```



RD + AST

Single-Command ::= **let** Declaration **in** single-Command
| **begin** Command **end**

```
Command parseSingleCommand() {
   …
   switch (currentToken.kind) {

   case Token.BEGIN:
     acceptIt();
     commandAST = parseCommand();
     accept(Token.END);
     break;

   case Token.LET:
     {
       acceptIt();
       Declaration dAST = parseDeclaration();
       accept(Token.IN);
       Command cAST = parseSingleCommand();
       …
       commandAST = new LetCommand(dAST, cAST, …);
     }
     break;
   …
   return commandAST;
 }
```

RD + AST

Single-Command ::= **if** Expression **then** single-Command **else** single-Command
| **while** Expression **do** single-Command

```
     case Token.IF:
      {
        acceptIt();
        Expression eAST = parseExpression();
        accept(Token.THEN);
        Command c1AST = parseSingleCommand();
        accept(Token.ELSE);
        Command c2AST = parseSingleCommand();
        …
        commandAST = new IfCommand(eAST, c1AST, c2AST, …);
      }
      break;

    case Token.WHILE:
      {
        acceptIt();
        Expression eAST = parseExpression();
        accept(Token.DO);
        Command cAST = parseSingleCommand();
        …
        commandAST = new WhileCommand(eAST, cAST, …);
      }
      break;
```

RD + AST

```
public class Parser {
 …
 private Token currentToken;
 …

 public Program parseProgram() {
   …
     currentToken = lexicalAnalyser.scan();
     …
     Command cAST = parseCommand();
     programAST = new Program(cAST, …);
     if (currentToken.kind != Token.EOT) {
       …
   }
   …
   return programAST;
 }
```

Constructing AST by Recursive Descent

- N
  - Each method *parseN*: parses an *N*-phrase and returns that phrase's AST
- N ::= X
  - The body of parse*N* returns parses subphrases of X and combines these into that phrase's AST
  - private AST_N parse*N* ( ) {

    AST_N itsAST;

    parse X, and at the same time constructing itsAST

    return itsAST;

    }
- AST_N is the abstract subclass of AST corresponding to N

Starter sets

starters [[ε]] = { }

starters [[t]] = { t } t is a terminal symbol

starters [[X Y]] = *starters* [[X]] U *starters* [[Y]] if X generates ε
starters [[X Y]] = *starters* [[X]] if X does not generate ε

starters [[X | Y]] = *starters* [[X]] U *starters* [[Y]]

starters [[X*]] = *starters* [[X]]

starters [[(X)]] = *starters* { X }

followers[[X]] in context N::= …X Z is *starters*[[Z]]

If Z can derive **ε** then also include *followers*[[N]] in **any** context

LL(1) Grammar: Conditions

If the grammar contains X | Y

starters [[X]] U *starters* [[Y]]
must be disjoint

and

If (say) *Y* can derive **ε** then *starters*[[*X*]]
must also be disjoint from *followers* of *X*|*Y* in this context

We want to choose between X or Y, therefore

If Y can derive **ε** then *followers*(X|Y) indicate the choice of Y

If Y can derive **ε** then X should not also be able to derive **ε**

If the grammar contains X*

starters [[X]]
must be disjoint from the set of tokens that can *followers* of X* in this context

We want to choose between to repeat the X or to finish X*, therefore

followers(X*) must be disjoint to *starters*[[X]]

Recursive Descent parsing is suitable only for LL(1) grammars

Grammar Transformations

Left factorization (RE)

S ::= if E then S else S | if E then S | id := E
S ::= if E then S { ε | else S } | id := E

Elimination of left recursion

SL ::= SL ; S | S
SL ::= S { ; S }*

Substitution of terminal symbols

S ::= N := E | N (A)
N ::= identifier

S ::= identifier := E | identifier (A)

Grammar Transformations

Left factorization (RE) X Y | X Z equals X (Y | Z)

… (XY) | (XZ) … → … XY … / … XZ … … X (Y|Z) … → … X Y … / … X Z …

Elimination of left recursion N ::= X | N Y equals N ::= X (Y)*

…N… →…NY… →…NYY… → … NY…YY… → XY…YY

…N… → …X(Y)*… →…XY…YY…

Substitution of terminal symbols N ::= X X may replace N

on any right hand side

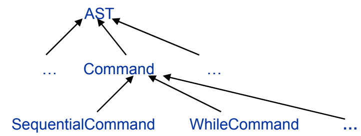
A::= …N… A::= …X…

…A… →…N… → …X… …A… → …X…

Command ::= Command ; Command
| while Expression do Command
| …

Command
SequentialCommand
WhileCommand
…

```
public abstract class Command extends AST {…}

public class SequentialCommand extends Command {
  ...
  public Command SC1, SC2;
}

public class WhileCommand extends Command {
  ...
  public Expression E;
  public Command C;
}
```



Command ::= single-Command (**;** single-Command)*
Single-Command ::= **while** Expression **do** single-Command
| …

Command
SequentialCommand
WhileCommand
…

```
Command parseCommand() {
  …
  commandAST = new SequentialCommand(…);
  …
  return commandAST;
}
```

```
Command parseSingleCommand(){
  …
  case Token.WHILE:
    { …
      commandAST = new WhileCommand(…);
    }
    break;
  …
  return commandAST;
}
```

Command ::= single-Command (**;** single-Command)*

```
Command parseCommand() {
   …
   commandAST = parseSingleCommand();
   while (currentToken.kind == Token.SEMICOLON) {
     acceptIt();
     Command c2AST = parseSingleCommand();
     …
     commandAST = new SequentialCommand(commandAST, c2AST, …);
   }
   return commandAST;
 }
```

RD + AST