Language Specification Overview

A programming language is usually specified in 3 stages:

1. Lexical specification of the words or tokens in the language. Typical languages have words which consist of *reserved words* like `for` and `while`, *constant literals* like `1234`, `0x1a2c`, `"quoted string"`, *operators* like `+`, `+=` and *punctuation symbols* like `(`, `;`.
2. Syntactic specification of how tokens combine to form phrases in the language. Typical phrases are *declarations*, *statements* and *expressions*.
3. Semantic restrictions on legal phrases which constitute programs, as well as specifying the overall meaning of legal programs in the language.

Lexical Analysis Example

Consider the following Pascal GCD program:

```
{ Compute GCD of integers read from input }
program gcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i <> j do
    if (i > j then i:= i - j
    else j := j -i;
  writeln(i);
end.
```

Lexical Analysis Example

Lexical analysis will take the character stream constituting the above program and transform it into the following word or *token*-stream.

`program`	`gcd`	`(`	`input`	`,`	`output`
`)`	`;`	`var`	`i`	`,`	`j`
`:`	`integer`	`;`	`begin`	`read`	`(`
`i`	`,`	`j`	`)`	`;`	`while`
`i`	`<>`	`j`	`do`	`if`	`i`
`>`	`j`	`then`	`i`	`:=`	`i`
`-`	`j`	`else`	`j`	`:=`	`j`
`-`	`i`	`;`	`writeln`	`(`	`i`
`)`	`;`	`end`	`.`		

Note the removal of whitespace and comments.

Regular Expressions

Regular expressions (regex) are the primary method for specifying the lexical aspects of a programming language.

- Need to define the tokens of a programming language formally.
- Regular expressions are used to specify the sequence of characters which constitute the tokens of a programming language.
- Regular expressions are also used in many tools like `grep` or programming languages like Perl for string pattern matching.

Regular Expressions Example

Natural numbers (integers > 0) can be represented using:

$$\begin{aligned} \textit{non\_zero\_digit} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \textit{digit} &\rightarrow 0 \mid \textit{non\_zero\_digit} \\ \textit{natural\_number} &\rightarrow \textit{non\_zero\_digit}\ \textit{digit} \ast \end{aligned}$$

or as `[1-9][0-9]*` in most common regex syntaxes

Regular Expressions Definition

Empty string: ϵ is a RE denoting the empty string.

Symbol: If $a \in \Sigma$ then $\mathtt{a}$ is a RE denoting the symbol a.

Concatenation: If $\mathtt{A}$ and $\mathtt{B}$ are REs, then $\mathtt{AB}$ is a RE denoting the concatenation of each of the strings represented by $\mathtt{A}$ with each of the strings denoted by $\mathtt{B}$.

Regular Expressions Definition Continued

Alternation: If `A` and `B` are REs, then `A|B` is a RE denoting any of the strings represented by `A` or by `B`.

Kleene closure: If `A` is a RE, then `A*` is a RE denoting 0-or-more of the strings represented by `A`.

By default, closure has the highest precedence, followed by concatenation, followed by alternation. Parentheses can be used to override the default precedence.

Regular Expressions Syntactic Sugar

Optional: If A is a RE, then A? denotes an optional occurrence of a string denoted by A (equivalent to A | ϵ).

Positive Closure: If A is a RE, then A+ denotes one-or-more occurrences of the strings denoted by A (equivalent to AA$\star$).

Character Set: If $x_1, x_2, x_3 \ldots \in \Sigma$, then [$x_1 x_2 x_3 \ldots$] denotes any one of the symbols x_1, x_2, x_3 ....

Note that $+$ and $\star$ are also referred to as greedy quantifiers; they are *greedy* in that they match as much as possible without preventing the rest of the regex from matching.

Regular Expressions Syntactic Sugar Continued

Character Set Range: If x_i and x_j belong to the ordered vocabulary Σ, then `[`x_i`-`x_j`]` denotes any one of the symbols between x_i and x_j (inclusive bounds).

Negated Character Set: If `[...]` is a character-set, then `[^...]` denotes all those characters in Σ which are not in `[...]`.

Regular Expression Examples

- `while` is a RE denoting the token **while**.
- `[0-9]+` is a RE denoting non-negative integers.
- `[1-9][0-9]*|0` is a RE denoting non-negative integers without any non-significant leading zeros.
- `[-+]?[0-9]+` is RE denoting a optionally-signed integer.
- `[_a-zA-Z][_a-zA-Z0-9]*` is a RE denoting a C identifier.

Scanners

- A scanner is a program which transforms a *character-stream* into a *token-stream* while (usually) removing non-significant whitespace and comments.
- A token usually contains 2 essential fields: a token *type* or *kind* like `identifier`, `int_constant`, `add_op` and a lexeme containing the text of word like `var_name`, `123`, `+`.
- A scanner may be written by hand but is often automatically generated using programs called scanner generators. Examples of typical scanner generators are `lex` and `flex`. Scanner generators are constructed using the theory of finite automata (beyond the scope of this course).

Practical Regular Expressions

Most current programming languages have regular expressions available as part of the language or as part of the standard language library.

- Most modern languages use a syntax for regular expressions first popularized by Perl referred to as Perl Compatible Regular Expressions or simply PCRE.
- Most syntaxes have extensions well beyond those described in these transparencies like non-greedy (`*?` and `+?`) and possessive (`*+` and `++`)quantifiers, capturing groups (using parentheses).
- Many languages allow regex literals like `/[a-zA-Z_][0-9a-zA-Z_]/`.
- Languages like Java which represent regex using strings suffer from backslashitis. For example, the regex `\\` is represented using the string literal `"\\\\"`.

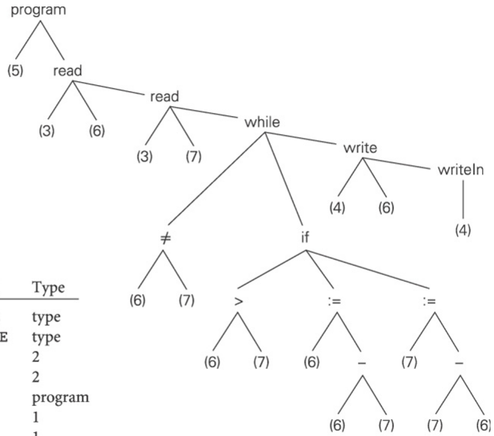
Keywords versus Reserved Words

- Most modern languages use *reserved words* which are usually alphabetical words which are **reserved** for denoting specific program constructs and cannot be used as identifiers denoting variables or functions.
- Some languages like PL/I use *keywords* which denote specific program constructs only within specific contexts. In other contexts, they may be used as general identifiers denoting variables or functions.
- Keywords allow easy language-subsetting, but complicate implementation.

Syntax Analysis

Syntax analysis takes a token-stream (from lexical analysis) and extracts a phrase structure from the stream. The extracted structure is often represented as a tree

GCD Structure Tree



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Regex Not Enough to Specify Syntax

- Regular expressions cannot be used to specify nested constructs used in most programming languages. For example, expressions can contain nested expressions nested to an arbitrary depth, statements can contain nested statements nested to an arbitrary depth.
- Syntax is specified using Context Free Grammars (CFGs). Direct or indirect recursion in CFG's allow specifying constructs which are nested to an arbitrary depth.

An Example Grammar

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \text{id} \mid \text{number} \mid - \ \textit{expr} \mid (\ \textit{expr} \ ) \\
& & \textit{expr op expr} \\
\textit{op} & \rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Note the recursive rules where *expr* is defined in terms of *expr*.

Context Free Grammar Definition

A CFG consists of a 4-tuple $\langle \mathcal{T}, \mathcal{N}, \mathcal{R}, S \rangle$ where

$\mathcal{T}$ is a set of terminal symbols.

$\mathcal{N}$ is a set of non-terminal symbols with $\mathcal{T} \cap \mathcal{N} = \emptyset$.

$\mathcal{R}$ is a set of production rules consisting of pairs $\langle\, n \in \mathcal{N}, (\mathcal{N} \cup \mathcal{T})* \,\rangle$.

S is a distinguished start-symbol belonging to $\mathcal{N}$.

Context Free Grammar Definition Continued

For the previous grammar, $T = \{+, -, *, /, \text{id}, \text{number}\}$, $N = \{\mathit{expr}, \mathit{op}\}$, $S = \mathit{expr}$ and R is the set of pairs:

$$\begin{array}{l}
\langle\ \mathit{expr}, \text{id}\ \rangle \\
\langle\ \mathit{expr}, \text{number}\ \rangle \\
\langle\ \mathit{expr}, -\mathit{expr}\ \rangle \\
\langle\ \mathit{expr}, (\ \mathit{expr}\ )\ \rangle \\
\langle\ \mathit{expr}, \mathit{expr}\ \mathit{op}\ \mathit{expr}\ \rangle \\
\langle\ \mathit{op}, +\ \rangle \\
\langle\ \mathit{op}, -\ \rangle \\
\langle\ \mathit{op}, *\ \rangle \\
\langle\ \mathit{op}, /\ \rangle
\end{array}$$

$$\begin{array}{rcl}
\mathit{expr} & \rightarrow & \text{id} \mid \text{number} \mid -\ \mathit{expr} \mid (\ \mathit{expr}\ ) \\
 & & \mathit{expr}\ \mathit{op}\ \mathit{expr} \\
\mathit{op} & \rightarrow & + \mid - \mid * \mid /
\end{array}$$

Derivations

Starting with the start symbol, repeatedly replace a non-terminal with the RHS of some rule for that non-terminal, until we have only terminal symbols.

$$
\begin{aligned}
\mathit{expr} &\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} \\
&\Rightarrow \mathit{expr}\ \underline{\mathit{op}}\ \mathrm{id} \\
&\Rightarrow \underline{\mathit{expr}} + \mathrm{id} \\
&\Rightarrow \mathit{expr}\ \mathit{op}\ \underline{\mathit{expr}} + \mathrm{id} \\
&\Rightarrow \mathit{expr}\ \underline{\mathit{op}}\ \mathrm{id} + \mathrm{id} \\
&\Rightarrow \underline{\mathit{expr}} * \mathrm{id} + \mathrm{id} \\
&\Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
$$

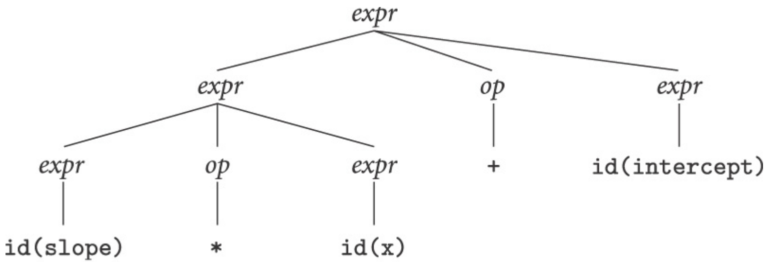
Derivations Continued

- $\Rightarrow$ represents a single step in the derivation; $\Rightarrow*$ represents 0-or-more steps in the derivation.
- If we replace the *right-most* (*left-most*) non-terminal at each step, we have a *right-most* (*left-most*) derivation.
- Each intermediate form is called a sentential form.
- The final sentential form is the yield of the derivation which is a *sentence* of the language.
- The language defined by a grammar is the set of all terminal sentential forms derived from the start symbol: $\mathcal{L} = \{x \in \mathcal{T}* \mid S \Rightarrow* x\}$.

Parse Trees

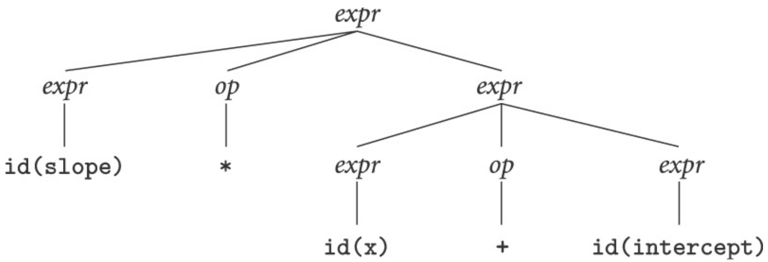
- A parse tree is a graphical representation of a derivation.
- Root of parse tree is start symbol.
- If $A \Rightarrow \alpha$ is a derivation step, then add each symbol in α as the children of the node corresponding to A.

Example Parse Tree



Parse Tree for `slope*x + intercept`.

Alternate Parse Tree



Alternate Parse Tree for `slope*x + intercept`.

Ambiguous Grammars

- A grammar is ambiguous if there is a sentence derived by the grammar which has multiple parse trees.
- Ambiguous grammars are not useful for specifying the concrete syntax of programming languages.
- Transform grammar to remove ambiguity; alternatively, some parsers allow specifying *disambiguation rules*.

Associativity and Precedence

- Binary operator $\otimes$ is left-associative if $a \otimes b \otimes c = (a \otimes b) \otimes c$.
- Binary operator $\otimes$ is right-associative $a \otimes b \otimes c = a \otimes (b \otimes c)$.
- Binary operator $\otimes$ has precedence over binary operator $\oplus$ if $a \otimes b \oplus c = (a \otimes b) \oplus c$ and $a \oplus b \otimes c = a \oplus (b \otimes c)$.

Usual arithmetic associativity and precedence:

- Lowest precedence left-associative $+$ and $-$; then left-associative $*$ and $/$; then unary minus; right associative $**$ or ^ (for exponentiation) has highest precedence. Overriden by parentheses.
- Exceptions: In APL, all operators have equal precedence and are evaluated right-to-left. In Microsoft Excel and Unix `bc`, unary minus has higher precedence than exponentiation, i.e. `-2^2` $=$ `(-2)^2` $=$ `4`.

Enforcing Associativity/Precedence via Grammar

- Introduce extra non-terminals for each precedence level.
- Have lower-precedence operators higher in grammar (closer to the start symbol).
- For left-associative (right-associative) operators use left-recursive (right-recursive) rules.

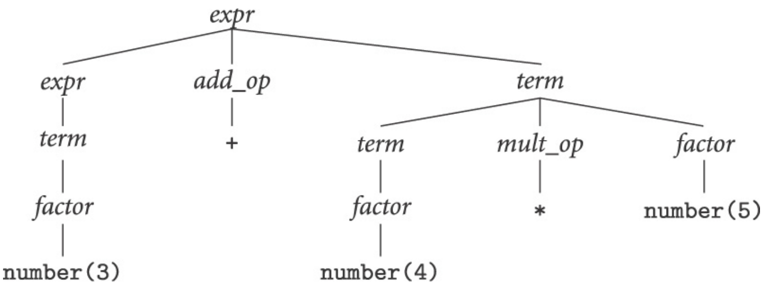
Transformed Grammar for Arithmetic Expressions

Use non-terminal *expr* for $+$, $-$ level; non-terminal *term* for $\star$, $/$ level; non-terminal *factor* for primitives (number, id), unary-minus or parenthesized expressions.

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \textit{term} \mid \textit{expr add\_op term} \\
\textit{term} & \rightarrow & \textit{factor} \mid \textit{term mult\_op factor} \\
\textit{factor} & \rightarrow & \text{id} \mid \text{number} \mid - \textit{factor} \mid (\textit{expr}) \\
\textit{add\_op} & \rightarrow & + \mid - \\
\textit{mult\_op} & \rightarrow & \star \mid /
\end{array}
$$

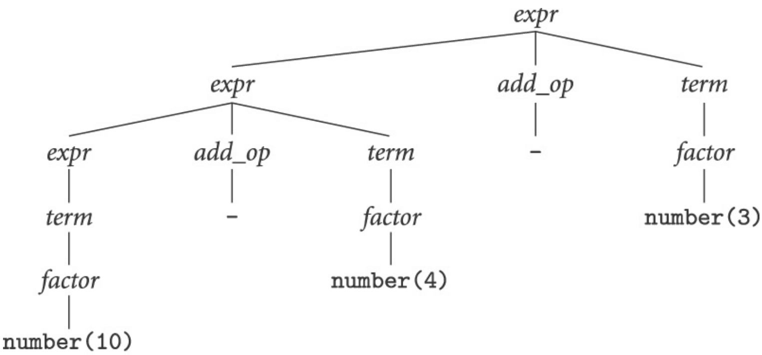
Ignored exponentiation operator.

Precedence Parse Tree



Parse Tree for `3+4*5`.

Associative Parse Tree



Parse Tree for Left-Associative `10-4-3`.

Abstract Syntax Tree

Abstract Syntax Tree: Extract interesting structure of tree by removing intermediate grammar symbols:

```
   +
  / \
 /   \
3     *
     / \
    /   \
   4     5
```

AST for `3 + 4 * 5`.

Can be represented linearly using Lisp S-expression `(+ 3 (* 4 5))` or Prolog term `+(3, *(4, 5))`.

Parsers

- A parser is a program which given a *token stream* and a CFG produces (explicitly or implicitly) a *parse tree* or *AST*.
- Many parsers are top-down (write start-symbol to terminals) or bottom-up (write terminals to start-symbol).
- Parsers are usually constructed by parser generators like `yacc`, `bison` (bottom-up generators which generates C), `javacc`, `antlr` (top-down generators which generates java).

Recursive-Descent Parsing

Not strictly relevant to this course, but a very useful technique to know.

- Recursive-descent is a simple way of writing parsers manually.
- A recursive-descent parser is a *top-down* parser which *descends* into derivation using a set of mutually-*recursive* functions.
- Structure of recursive-descent parsing program mirrors CFG.
- Rather severe restrictions on class of CFG's which can be handled using this technique.

Recursive-Descent Parsing Details

- Initialize a global `lookahead` to contain the current lookahead token from the scanner.
- Have a `match(t)` function which ensures that the current lookahead matches token `t` and advances the lookahead to the next token. If the lookahead does not match `t`, then signal a syntax error.
- For each non-terminal in grammar have a corresponding function whose specification requires it to recognize a prefix of the input which corresponds to one of the rules for that non-terminal.

Recursive-Descent Parsing Details Continued

- For each non-terminal function use the current lookahead to select the appropriate rule.
- For each rule with RHS containing sequence of symbols α, process each symbol in order:
  - If the symbol is a terminal symbol `t` then call `match(t)`.
  - If the symbol is a non-terminal symbol `A` then call the function `A()` corresponding to `A`.

Easy to prove correctness (use an inductive argument with inductive hypothesis that each parsing function meets its specification for a smaller input).

Recursive-Descent Example

Consider following grammar for a list of comma-separated `ID`'s terminated by a `;`.

```
idList
  : ID idListTail
  ;
idListTail
  : ',' ID idListTail
  | ';'
  ;
```

Recursive-Descent Example Continued

Following program:

```
Token lookahead;
void idList() {
  match(ID); idListTail();
}
void idListTail() {
  if (lookahead.kind == ',') {
    match(','); match(ID); idListTail();
  }
  else {
    match(';');
  }
}
```

Consider fragment of arithmetic expression grammar:

```
expr
  : expr '+' term
  | term

void expr() {
  if (...) {
    expr(); match('+'); term();
  }
  else {  term();  }
```

Recursive Descent Problems Continued

- What is the test `(...)` in `if (...)`.
- `expr()` calls `expr()` directly without changing lookahead ... infinite loop!!
- Recursive-descent parsers cannot handle CFG's with left-recursive (direct or indirect) rules.
- Show-stopper?

Coping with Left-Recursion

Since recursive-descent parsers cannot cope with left-recursive grammars try transforming grammar to one without left-recursive rules but which describes the same language. Consider

```
expr
  : expr '+' term
  | term
```

`expr` must start with a `term`. This may be followed by 0-or-more occurrences of `'+' term`.

Coping with Left-Recursion Continued

```
expr
  : term exprRest
  ;
exprRest
  : '+' term exprRest
  | /* empty */
  ;
```

Grammar for Arithmetic Expressions

Motivation: extended example of parsing arithmetic expressions using recursive-descent parsing.
Start with a grammar for arithmetic expressions which enforces associativity and precedence (very similar to previous grammar).

```
program
   :   EOF
   |   expr '\n' program
   ;
expr
   :   expr '+' term
   |   expr '-' term
   |   term
   ;
```

Grammar for Arithmetic Expressions Continued

```
term
   :   term '*' factor
   |   term '/' factor
   |   factor
   ;
factor
   :   simple '**' factor
   |   simple
   ;
```

Grammar for Arithmetic Expressions Continued

```
simple
  : '-' simple
  |  INTEGER
  |  '(' expr ')'
  ;
```

This grammar has unary minus with higher precedence than exponentiation.

Transformed Grammar for Extended Recursive-Descent Example

```
program
   :   EOF
   |   expr '\n' program
   ;
```

Transformed Grammar for Extended Recursive-Descent Example Continued

```
expr
   :   term exprRest
   ;
exprRest
   :   '+' term exprRest
   |   '-' term exprRest
   |   //EMPTY
   ;
```

`//EMPTY` is a comment denoting that the rule matches the empty string ϵ.

Transformed Grammar for Extended Recursive-Descent Example Continued

```
term
   :   factor termRest
   ;
termRest
   :   '*'  factor termRest
   |   '/'  factor termRest
   |   //EMPTY
   ;
```

Transformed Grammar for Extended Recursive-Descent Example Continued

```
factor
   :  simple '**' factor
   |  simple
   ;
simple
   : '-' simple
   |  INTEGER
   |  '(' expr ')'
   ;
```

Expression Evaluation Log

```
$ java -cp target/arith.jar \
      edu.binghamton.cs571.ArithParser
1 + 2 * 3
7
(1 + 2) * 3
9
2 * 3**3 * -2
-108
64 / 2**3 / 2
4
64 / (2**3 / 2)
16
1 + - 2
-1
```

Expression Evaluation Log Errors

```
1 + + 2
<stdin>:7:4: syntax error at '+'
1 ( 2 )
1
<stdin>:8:2: syntax error at '('
1 @ 2
1
<stdin>:9:2: syntax error at '@'
$
```

Evaluating Arithmetic Expressions using Recursive-Descent

- Have each function corresponding to an expression non-terminal return the value of that expression.
- The functions for non-terminals like `exprRest` and `termRest` introduced for left-recursion removal will have an input parameter which will accumulate the value of the expression/term seen previously.
- When the function for non-terminals like `exprRest` and `termRest` recognized the empty sequence, they should use the input parameter as the return value.

In file ArithParser.java:

```
/** token kinds for arith tokens*/ //
private static enum TokenKind {
  EOF,
  NL,
  EXP_OP,
  ADD_OP,
  SUB_OP,
  MUL_OP,
  DIV_OP,
  INTEGER,  //@tokenKind1@
  LPAREN,
  RPAREN,
  ERROR,
}
```

Scanner in Scanner.java:

```
  /** Map from regex to token-kind */ //
  private static final LinkedHashMap<String, Enum> 
    new LinkedHashMap<String, Enum>() {{
      put("", TokenKind.EOF);
      put("[ \\t]+", null);  //ignore linear whites
      put("\n", TokenKind.NL);
      put("\\*\\*", TokenKind.EXP_OP);
      put("\\+", TokenKind.ADD_OP);
      put("\\-", TokenKind.SUB_OP);
      put("\\*", TokenKind.MUL_OP);
      put("\\/", TokenKind.DIV_OP);
      put("\\d+", TokenKind.INTEGER); //@tokenMap1@
      put("\\(", TokenKind.LPAREN);
      put("\\)", TokenKind.RPAREN);
      put(".", TokenKind.ERROR);  //catch lexical e
    }};
```

```
public class ArithParser { //

  private Scanner _scanner;
  private Token _lookahead;

  ArithParser() {
    _scanner = new Scanner(PATTERNS_MAP);
    nextToken(); //prime lookahead
  }
```

Recursive-Descent Program: `program()`

```java
/** Parser for program: //
 *  program
 *      :   EOF
 *      |   expr '\n' program
 *      ;
 */
public void program() {
  if (_lookahead.kind == TokenKind.EOF) {
    match(TokenKind.EOF);
  }
  else {
    try {
      int value = expr();
      System.out.println(value);
      match(TokenKind.NL);
```

```
        catch (ArithParseException e) {
          System.err.println(e.getMessage());
          match(TokenKind.NL);
        }
        program();
      }
    }
```

Recursive-Descent Program: `expr()`

```
/** Parse expr: //
 *  expr
 *    :   term exprRest
 *    ;
 */
private int expr() {
  int t = term();
  return exprRest(t);
}
```

Recursive-Descent Program: `exprRest()`

```
/** Parse exprRest: //
 *  exprRest
 *    :  ADD_OP term exprRest
 *    |  SUB_OP term exprRest
 *    |  //EMPTY
 *    ;
 */
private int exprRest(int valueSoFar) {
  if (_lookahead.kind == TokenKind.ADD_OP) {
    match(TokenKind.ADD_OP);
    int t = term();
    return exprRest(valueSoFar + t);
  }
```

Recursive-Descent Program: `exprRest()` Continued

```
    else if (_lookahead.kind == TokenKind.SUB_OP) {
      match(TokenKind.SUB_OP);
      int t = term();
      return exprRest(valueSoFar - t);
    }
    else { //EMPTY
      return valueSoFar;
    }
  }
```

Recursive-Descent Program: `term()`

```
/** Parse term: //
  *  term
  *    :   factor termRest
  *    ;
  */
private int term() {
  int f = factor();
  return termRest(f);
}
```

```
/** Parse termRest: //
 *  termRest
 *    :  MUL_OP factor termRest
 *    |  DIV_OP factor termRest
 *    |  EMPTY
 *    ;
 */
private int termRest(int valueSoFar) {
  if (_lookahead.kind == TokenKind.MUL_OP) {
    match(TokenKind.MUL_OP);
    int f = factor();
    return termRest(valueSoFar * f);
  }
```

```
    else if (_lookahead.kind == TokenKind.DIV_OP) {
      match(TokenKind.DIV_OP);
      int f = factor();
      return termRest(valueSoFar / f);
    }
    else { //EMPTY
      return valueSoFar;
    }
  }
```

Recursive-Descent Program: `factor()`

```java
/** Parse factor: //
 *  factor
 *      :   simple '**' factor
 *      |   simple
 *      ;
 */
private int factor() {
  int s = simple();
  if (_lookahead.kind == TokenKind.EXP_OP) {
    match(TokenKind.EXP_OP);
    s = (int)Math.pow(s, factor());
  } //@factor1@
  return s;
}
```

Recursive-Descent Program: `simple()`

```java
/** Parse simple: //
 *  simple
 *      |   SUB_OP simple
 *      |   INTEGER
 *      |   LPAREN expr RPAREN
 *      ;
 */
private int simple() {
  int value = 0;
  if (_lookahead.kind == TokenKind.SUB_OP) {
    match(TokenKind.SUB_OP);
    value = - simple();
  }
```

Recursive-Descent Program: `simple()` Continued

```
    else if (_lookahead.kind == TokenKind.INTEGER)
      value = Integer.parseInt(_lookahead.lexeme);
      match(TokenKind.INTEGER);
    }
    else if (_lookahead.kind == TokenKind.LPAREN) {
      match(TokenKind.LPAREN);
      value = expr();
      match(TokenKind.RPAREN);
    }
    else {  //@simple2@
      syntaxError();
    }
    return value;
  }
```

Recursive-Descent Program: `match()`

```
private void match(TokenKind kind) { //
  if (kind != _lookahead.kind) {
    syntaxError();
  }
  if (kind != TokenKind.EOF) {
    nextToken();
  }
}
```

```
/** Skip to end of current line and then throw ex
private void syntaxError() {
  String message =
    String.format("%s: syntax error at '%s'",
                  _lookahead.coords,
                  _lookahead.lexeme);
  while (_lookahead.kind != TokenKind.NL &&
         _lookahead.kind != TokenKind.EOF) {
    nextToken();
  }
  throw new ArithParseException(message);
}
```

Semantic Language Restrictions

- CFG's cannot describe many aspects of programming languages: for example, a variable can only be used after it is declared; the number of parameters of a function call must agree with the function declaration.
- What ever cannot be described by the syntax is lumped into semantic restrictions.
- There are some formal frameworks like attribute grammars for checking semantics. In practice, usually ad-hoc techniques are used.
- Once a program meets all lexical, syntax and semantic restrictions, it is known to be correct.
- In a compiler, the task of analyzing a program to ensure that it meets lexical, syntactic and semantic restrictions is done by the front-end. The back-end is responsible for generating code.

Semantic Specification

There are various methods for describing the semantics of a programming language (different from the more implementation-oriented semantic checking):

Natural Language Description A language definition manual attempts to describe the programming language using a natural language like English. The description attempts to be as precise as possible, but there are often ambiguities and inconsistencies. In practice, this is the most common technique. An example is the Java Language Specification.

Semantic Specification Continued

Operational Semantics There is a canonical implementation of the programming language and the language is defined by this implementation. This means that bugs in the implementation are part of the language specification. More importantly, it is not clear which aspects of the specification are essential and which result from accidental implementation details. In practice, quite a few languages have been defined this way, with Perl being a exemplar.

Semantic Specification Continued

Denotational Semantics The language is described using mathematical functions. Not terribly popular because the description is complex and inaccessible to most users of the programming language. Suited to the languages which have more mathematical backgrounds like the functional and logic programming languages.

References

Text, Chapters 1; Chapter 2 through 2.3.1 omitting 2.2