University of The Gambia

Computer Sciences Department

CPS358: Compilers

J. Boillat

Keywords: Lexical Analysis, Regular Expressions, Syntax Analysis, Context-Free Grammars, Semantic Analysis, Intermediate Code Generation, Code Optimisation, Code Generation, Bytecode, Compilers, Interpreters

Abstract

An introduction to the modern theory of compilers. Topics include regular expressions, context-free grammars, compilers and interpreter.

Chapter 1

Introduction

1.1 Language Processors

A *compiler* is a program that can read a program in one language (source language) and translate it in another language (target language) (see figure 1-1). An important role of the compiler is to report any errors in the source program that is detected during the translation process.

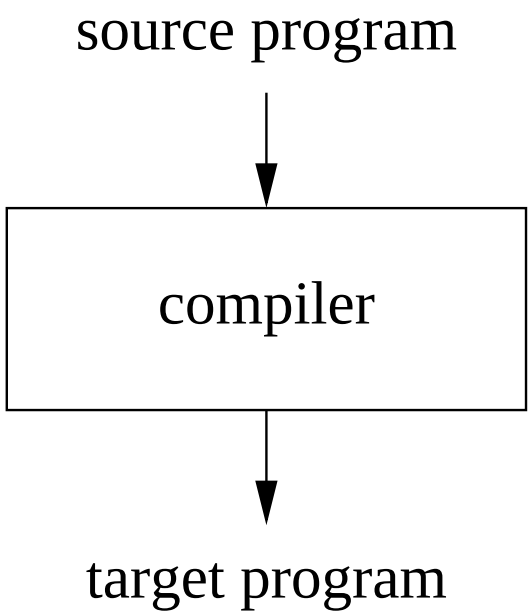


Figure 1-1: A compiler

If the target is an executable machine-language program, it can be called by the user and produce output (see figure 1-2).

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user (see figure 1-3).

The machine-language target produced by a compiler is usually much more faster than an interpreter at mapping inputs to outputs. An interpreter, however. can usually give better

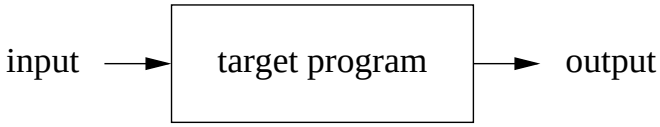


Figure 1-2: Running the target program

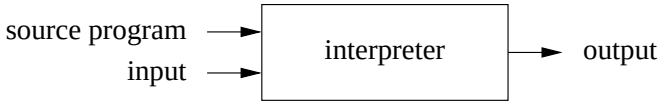


Figure 1-3: A interpreter

error diagnostics than a compiler, because it executes the source program statement by statement.

JAVA language processors combine compilation and interpretation. A JAVA program may first be compiled into an intermediate form called *bytecode*. The bytecode is interpreted by a *virtual machine* (see figure 1-4).

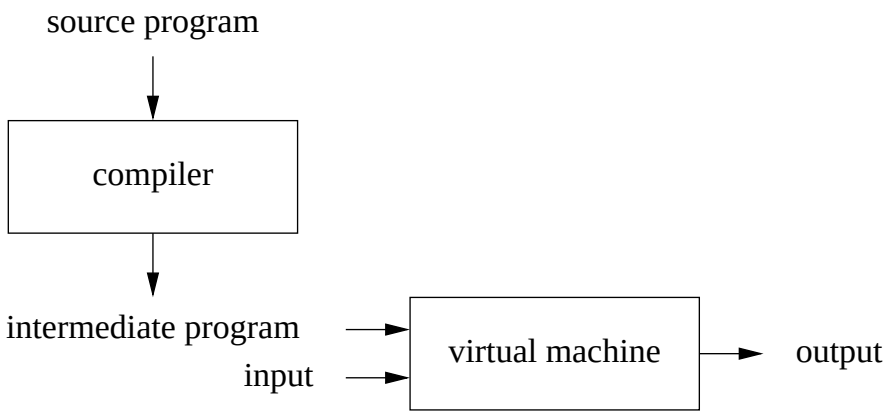


Figure 1-4: A hybrid compiler

1.2 The Structure of a Compiler

Compiling consists of two parts (see figure 1-5): *analysis* and *synthesis*.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from intermediate representation and the information in the symbol table.

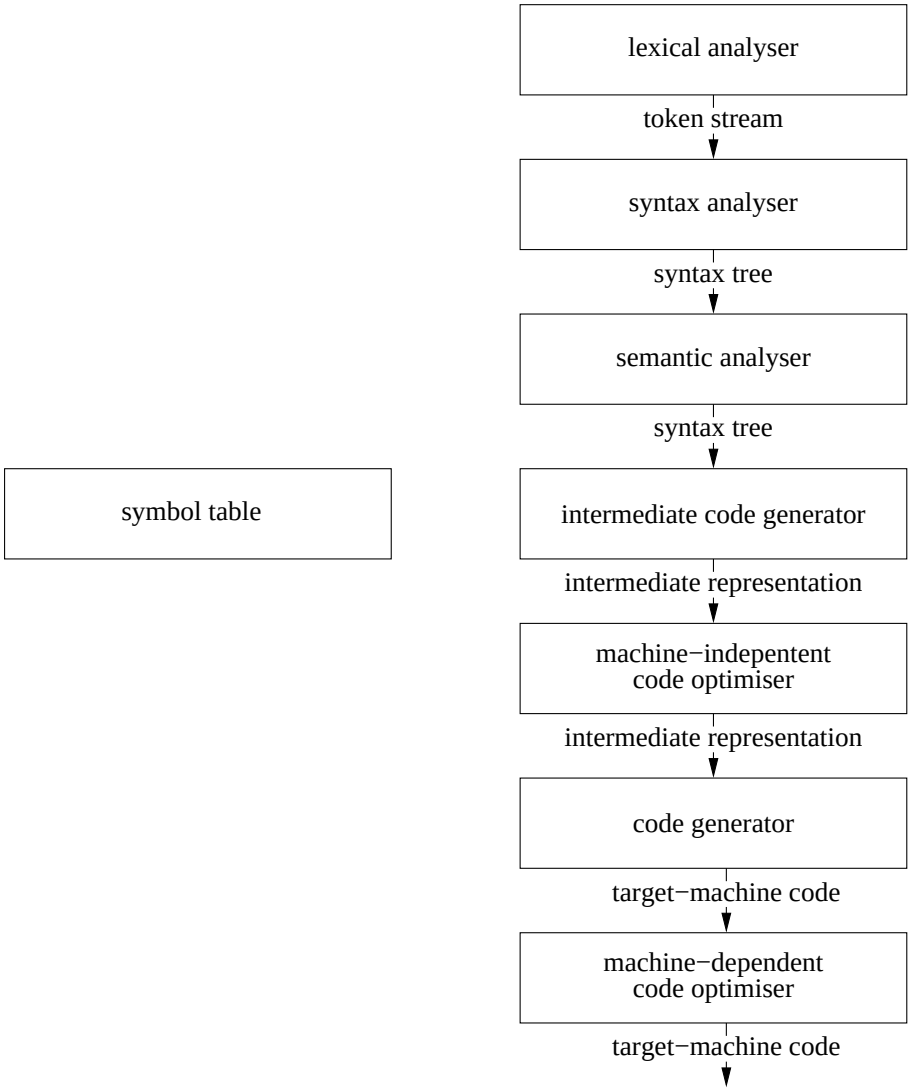


Figure 1-5: Phases of a compiler

1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyser reads a stream of characters making up the source program and groups the characters into meaningful sequences called *lexemes* for each lexeme the lexical analyser produces as output a *token* of the form

< token-name, value >

that it passes on the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during the syntax analysis, and the second component *value* points to an entry in the symbol table. The value may be empty.

For example, suppose that a source program contains the assignment statement involving floating point variables.

```
position = initial + rate * 60
```

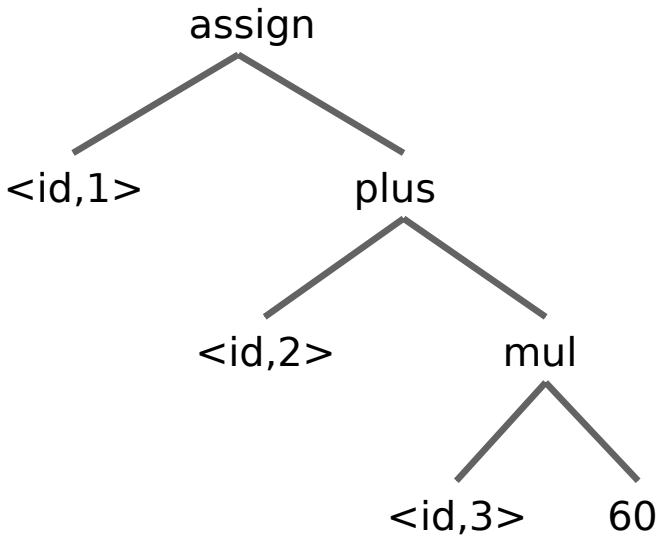
The assignment statement after lexical analysis may be following sequence of tokens

```
<id,1> <=> <id,2> <+> <id,3> <*> <60>
```

1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first component of the tokens to create a tree-like intermediate representation that depicts the grammatical structure. This intermediate structure is called *syntax tree*.

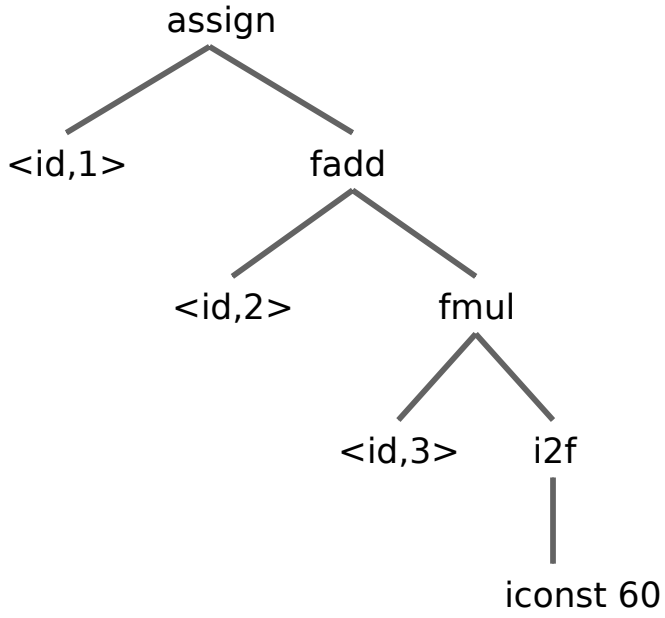
Here the syntax tree for the token stream of section 1.2.1:



1.2.3 Semantic Analysis

The *semantic analysis* uses the syntax tree and the informations in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation.

Here the modified syntax tree for the syntax tree of section 1.2.2:



1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into machine code (for any machine).

We consider here *three-address code* which consists of a sequence of assembler-like instructions with three operands per instruction.

The output of the intermediate code generator for the modified syntax tree of section 1.2.3 consists of following three-address code sequence:

```
i2f 60 null t1
fmul id3 t1 t2
fadd id2 t2 t3
icopy t3 null id1
```

1.2.5 Code Optimisation

The machine-independent code-optimisation phase attempts to improve the intermediate code so that better target code will result. There is a great amount of code optimisation different compilers perform.

In the output of section 1.2.4, the optimiser could deduce that the conversion of `60` from integer to floating point can be done once. Moreover, `t3` is used only once to transmit its value to `id3`, so the optimiser could produce following output:

```
fmul id3 60.0 t1
fadd id2 t1 id1
```

1.2.6 Code Generation

The code generator takes as input an intermediate representation and maps it into the target program. If the language is machine code, registers or memory locations are selected for each variable used by the program. Then the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Here a translation of the intermediate code of section 1.2.5 into JAVA bytecode:

```
ldc 60.0
fload 3
fmul
fload 2
fadd
fstore 1
```

Chapter 2

Regular Expressions

In computing, regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a scanner generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- the sequence of characters `car` in any context, such as `car`, `cartoon`, or `bicarbonate`
- the word `car` when it appears as an isolated word
- the word `car` when preceded by the word `blue` or `red`
- a dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits
- A FA (NFA or DFA) is a *blueprint* for constructing a machine recognising a regular language.
- A regular expression is a *user-friendly*, declarative way of describing a regular language.
- Example: $01^* + 10^*$
- Regular expressions are used in e.g.
  1. UNIX grep command
  2. UNIX Lex (Lexical analyser generator) and Flex (Fast Lex) tools.

2.1 Regular Expressions

A regular expression defines a set of strings over an alphabet A, i.e. regular expressions define languages.

Definition 2.1 [Regular Expression]

Regular Expressions *(RE) are defined by induction as follows:*

Basis: *Atomic expressions:* ϵ *is a RE and* $\emptyset$ *is a RE.*
$L(\epsilon) = \{\epsilon\}$, *and* $L(\emptyset) = \emptyset$.

If $a \in \Sigma$, *then* a *is a RE.*
$L(a) = \{a\}$.

Induction: *If* E *is a RE, then* (E) *is a RE.*
$L((E)) = L(E)$.

Choice expression: If E *and* F *are RE's then* $E + F$ *is a RE.*
$L(E + F) = L(E) \cup L(F)$.

Sequence expression: If E *and* F *are RE's then* $E.F$ *is a RE.*
$L(E.F) = L(E).L(F)$.

Iteration expression: If E *is a RE's then* E^* *is a RE.*
$L(E^*) = L(E)^*$.

Notation 2.1 [Dot] We write EF instead of $E.F$

Example 2.1 [Regular Expression] RE for

$$L = \{w \in \{0,1\} : 0 \quad \text{and} \quad 1 \quad \text{alternate in} \quad w\}$$

$$(01)^* + (10)^* + 0(10)^* + 1(01)^*$$

or, equivalently

$$(\epsilon + 1)(01)^*(\epsilon + 0)$$

Remark 2.1 [Precedence] Order of precedence for operators: Star ($*$), Dot (.), Plus ($+$)

2.1.1 Other Notations

Traditional UNIX regular expression syntax followed common conventions but often differed from tool to tool. Following tables contain the syntax of some regular expressions used in JAVA[1]. Note the extensive usage of escape sequences (\) since regular expressions are defined as JAVA strings.

[1] See `java.util.regex.Pattern` in the JAVA API documentation

Construct	Matches
Characters	
`x`	The character x
`\\`	The backslash character
`\u`*hhhh*	The character with hexadecimal value `0x`*hhhh*
`\t`	The tab character
`\n`	The newline
`\r`	The carriage-return character
Character classes	
`[abc]`	`a`, `b`, or `c` (simple class)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	`a` through `z` or `A` through `Z`, inclusive (range)
`[a-d[m-p]]`	`a` through `d`, or `m` through `p`: `[a-dm-p]` (union)
`[a-z&&[def]]`	`d`, `e`, or `f` (intersection)
`[a-z&&[^bc]]`	`a` through `z`, except for `b` and `c`: `[ad-z]` (subtraction)
`[a-z&&[^m-p]]`	`a` through `z`, and not `m` through `p`: `[a-lq-z]` (subtraction)
Predefined character classes	
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[\t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`
Greedy quantifiers	
X`?`	X, once or not at all
X`*`	X, zero or more times
X`+`	X, one or more times
X{n}	X, exactly n times
X{n`,`}	X, at least n times
X{n`,`m}	X, at least n but not more than m times
Logical operators	
XY	X followed by Y
X`|`Y	Either X or Y
`(`X`)`	X, as a capturing group

Exercise 2.1 [RE 1] *Give a regular expression that denotes the language*

$$L = \{ab\} \cdot (\{aa, bb\}^* \cup (\{b\} \cdot \{a, b\}^+))$$

Exercise 2.2 [RE 2] *Give a regular expression for the set of binary strings where the number of* 0*'s is a multiple of* 3*.*

Exercise 2.3 [RE 3] *Give a regular expression for the set of strings over the alphabet* $\Sigma = \{a, b\}$ *that start and end with* a*.*

Exercise 2.4 [RE 4] *Give a regular expression for the set of strings over the alphabet* $\Sigma = \{a, b\}$ *that do not contain the substring* aba*.*

Exercise 2.5 [RE 5]

What is the following JACACC *regular expression (see chapter A) used for?*

```
"\""(   (~["\"","\\","\n","\r"])
  | ("\\"
      ( ["n","t","b","r","f","\\","'","\""]
      | ["0"-"7"] ( ["0"-"7"] )?
      | ["0"-"3"] ["0"-"7"] ["0"-"7"]
      )
    )
)*
"\""
```

2.1.2 Equivalence of FA's and RE's

Theorem 2.1 [Equivalence of FA's and RE's] *For every DFA* $A = (Q, \Sigma, \delta, q_0, F)$ *there is a RE* R*, s.t.*

$$L(R) = L(A)$$

2.2 Algebraic Laws for Regular Expressions

Definition 2.2 [Regular Language] *A language* L *is* regular *if it is the language of some RE* R*, i.e.* $L = L(R)$

Theorem 2.2 [Algebraic Laws for RE's]

- $E + F = F + E$ *, that is,* $+$ *is commutative.*
- $(E + F) + G = E + (F + G)$*. That is,* $+$ *is associative.*
- $(EF)G = E(FG)$ *, that is, concatenation is associative.*
- $E(F + G) = EF + EG$*,* $(E + F)G = EG + FG$ *, that is, the concatenation is distributive over* $+$*.*
- $\epsilon E = E\epsilon = E$ *, that is,* ϵ *is the neutral element for concatenation.*
- $E^{**} = E^*$ *, that is,* $*$ *is idempotent.*

□

Chapter 3

Context Free Grammars

We have seen that many languages cannot be regular. Thus we need to consider larger classes of languages.

Context-Free Languages (CFL's) played a central role natural languages since the 1950's, and in compilers since the 1960's.

Context-Free Grammars (CFG's) are the basis of BNF-syntax.

Today CFL's are increasingly important for XML and their DTD's or Schema's.

3.1 Context Free Grammars

A grammar is a set of formation rules that describe which strings formed from the alphabet of a formal language are syntactically valid, within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics.

A grammar consists of a set of string rewriting rules with an assigned start symbol; the language described is the set of strings that can be generated by applying these rules arbitrarily, starting with the start symbol.

Definition 3.1 [Context Free Grammars] *A* context-free grammar *is a quadruple* $G = (V, T, P, S)$ *where*

- V *is a finite set of* variables *or* nonterminal symbols.
- T *is a finite set of* terminal symbols.
- P *is a finite set of* productions *of the form* $A \to \alpha$ *, where* A *is a variable and* $\alpha \in (V \cup T)^*$
- S *is a designated variable called the* start symbol.

Remark 3.1 [Grammar] There is a more general definition of a grammar, allowing productions to have the form $\beta \to \alpha$, where $\alpha \in (V \cup T)^*$ and $\beta \in (V \cup T)^*$.

Remark 3.2 [Chomsky Hierarchy] The *Chomsky hierarchy* is a containment hierarchy of classes of formal grammars . This hierarchy of grammars was described by Noam

Chomsky[1] in 1956 [Cho56].

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognised by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always-halting Turing machine.
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \to \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \to \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognised by a linear bounded automaton (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input).
- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \to \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognised by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.
- Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \to \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a DFA.

Every regular language is context-free, every context-free language, not containing the empty string, is context-sensitive, every context-sensitive language is recursive and every recursive language is recursively enumerable.

Notation 3.1 [BNF] Let A be a nonterminal. If there are more than one A-productions, i.e.:

$$\begin{array}{lcl} A & \to & w_1 \\ A & \to & w_2 \\ & \cdots & \\ A & \to & w_k \end{array}$$

We write

$$A \to w_1 | w_2 | \cdots | w_k$$

This notation is called *BNF*-Notation (Backus-Naur Form).

Notation 3.2 [EBNF] The *EBNF*-Notation is an extension of the BNF-Notations:

[1] Avram Noam Chomsky, born 1928, is an American linguist, philosopher, cognitive scientist, political activist, author, and lecturer. He is an Institute Professor and professor emeritus of linguistics at the Massachusetts Institute of Technology.

- We write $A \to (w)^*$ for the productions $A \to wA|\epsilon$. Zero or more repetitions of the string w
- We write $A \to (w)^+$ for the productions $A \to wA|w$. One or more repetitions of the string w
- We write $A \to (w)^?$ for the productions $A \to w|\epsilon$. Optional w.

Example 3.2 [G_{pal}] Let's define L_{pal} the language of palindromes inductively:

Basis: ϵ, 0, and 1 are palindromes.

Induction: If w is a palindrome, so are $0w0$ and $1w1$.

Circumscription: Nothing else is a palindrome.

The language of palindromes $G_{pal} = (\{P\}, \{0,1\}, A, P)$, where $A = \{P \to \epsilon|1|0|0P0|1P1)$.

Example 3.3 [G_{ari}] The grammar for arithmetic exprexssions

$$G_{ari} = (\{E,N\}, \{0,1,+,-,*,/,(,)\}, A, E)$$

where A is the following set of productions:

$$\begin{array}{rcl} E & \to & E+E \\ E & \to & E-E \\ E & \to & E*E \\ E & \to & E/E \\ E & \to & (E) \\ E & \to & N \\ N & \to & 1 \\ N & \to & 0 \\ N & \to & N0 \\ N & \to & N1 \end{array}$$

Example 3.4 [G_{re}] Regular expressions over $\{0,1\}$ can be defined by the grammar $G_{re} = (\{E\}, \{0,1,.,+,^*,(,)\}, A, E)$ where A is the following set of productions:

$$\begin{array}{rcl} E & \to & 0 \\ E & \to & 1 \\ E & \to & E.E \\ E & \to & E+E \\ E & \to & E^* \\ E & \to & (E) \end{array}$$

3.1.1 The Language of a CFG

There are two ways for testing if a string in Σ^* belong to the language of a given variable:

- *Recursive inference*, using productions from body to head
- *Derivation*, using productions from head to body.

Recursive inference

Example 3.5 [L_{pal}] We show here that the string 001010100 belongs to the language L_{pal} of G_{pal}.

String	Production
001010100	$P \to 1$
0010P0100	$P \to 0P0$
001P100	$P \to 1P1$
00P00	$P \to 0P0$
0P0	$P \to 0P0$
P	

Example 3.6 [L_{ari}] We show here that the string $1 + 1 * 10$ belongs to the language L_{ari} of G_{ari}.

String	Production
$1 + 1 * 10$	$N \to 1$
$N + 1 * 10$	$N \to 1$
$N + N * 10$	$N \to 1$
$N + N * N0$	$N \to N0$
$N + N * N$	$E \to N$
$E + N * N$	$E \to N$
$E + E * N$	$E \to N$
$E + E * E$	$E \to E * E$
$E + E$	$E \to E + E$
E	

Example 3.7 [L_{re}] We show here that the string $0^* + 0.1$ belongs to the language L_{re} of G_{re}.

String	Production
$0^* + 0.1$	$E \to 0$
$E^* + 0.1$	$E \to 0$
$E^* + E.1$	$E \to 1$
$E^* + E.E$	$E \to E^*$
$E + E.E$	$E \to E.E$
$E + E$	$E \to E + E$
E	

Derivation

Definition 3.2 [Derivation] *Let $G = (V, T, P, S)$ be a CFG, and let $\alpha A \beta$ be a string of terminals and nonterminals, with A a nonterminal. Let $A \to \gamma$ be a production of G. The replacement of A by γ in the string $\alpha A \beta$ results in the string $\alpha \gamma \beta$ and is called* derivation.

Notation 3.3 [Derivation] We write

$$\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$$

or simply

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

if G is well understood

We may extend $\Rightarrow$ to represent zero, one, or many derivation steps, much as the transition function δ of an FA was extended to $\hat{\delta}$.

Definition 3.3 [Reflexive and transitive Closure of Derivation] ***Basis:*** *For any string α of terminals and nonterminals, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. That is any string derives itself.*

Induction: *If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. That is if α becomes β by zero or more steps, and one step takes β to γ, then α can become γ.*

Remark 3.3 [Other Interpretation] $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means that there is a sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$ for some $n \geq 1$, such that

1. $\alpha = \gamma_1$
2. $\beta = \gamma_n$
3. For $i = 1, 2, \ldots, n-1$, we have $\gamma_i \Rightarrow \gamma_{i+1}$

Example 3.8 [L_{pal}] We show here that the string 001010100 belongs to the language of G_{pal}.

$$P \Rightarrow 0P0 \Rightarrow 00P00 \Rightarrow 001P100 \Rightarrow 0010P0100 \Rightarrow 001010100$$

Example 3.9 [G_{ari}] We show here that the string $1 + 1 * 10$ belongs to the language of G_{ari}.

$$E \Rightarrow E{+}E \Rightarrow N{+}E \Rightarrow 1{+}E \Rightarrow 1{+}E{*}E \Rightarrow 1{+}N{*}E \Rightarrow 1{+}1{*}E \Rightarrow 1{+}1{*}N \Rightarrow 1{+}1{*}1N \Rightarrow 1{+}1{*}10$$

Example 3.10 [G_{re}] We show here that the string $0^* + 0.1$ belongs to the language of G_{re}.

$$E \Rightarrow E + E \Rightarrow E^* + E \Rightarrow 0^* + E \Rightarrow 0^* + E.E \Rightarrow 0^* + 1.E \Rightarrow 0^* + 1.0$$

Remark 3.4 [Derivation Choice] At each step we might have several rules to choose from, e.g. $E + E \Rightarrow E^* + E$ versus $E + E \Rightarrow E + E.E$

Remark 3.5 [Derivation Fails] Not all choices lead to successful derivations of a particular string, for instance $E \Rightarrow E.E$ won't lead to a derivation of $0^* + 0.1$

Definition 3.4 [Leftmost Derivation] *At each step of the* leftmost derivation *the leftmost nonterminal will be replaced by one of its rule-bodies.*

Definition 3.5 [Rightmost Derivation] *At each step of the* rightmost derivation *the rightmost nonterminal will be replaced by one of its rule-bodies.*

Example 3.11 [G_{ari} (Leftmost)] Leftmost derivation of the string $1 + 1 * 10$.

$$E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} N+E \underset{lm}{\Rightarrow} 1+E \underset{lm}{\Rightarrow} 1+E*E \underset{lm}{\Rightarrow} 1+N*E \underset{lm}{\Rightarrow} 1+1*E \underset{lm}{\Rightarrow} 1+1*N \underset{lm}{\Rightarrow} 1+1*N0 \underset{lm}{\Rightarrow} 1+1*10$$

Example 3.12 [G_{ari} (Rightmost)] Rightmost derivation of the string $1+1*10$.

$$E \underset{rm}{\Rightarrow} E+E \underset{rm}{\Rightarrow} E+E*E \underset{rm}{\Rightarrow} E+E*N \underset{rm}{\Rightarrow} E+E*N0 \underset{rm}{\Rightarrow} E+E*10 \underset{rm}{\Rightarrow} E+N*10 \underset{rm}{\Rightarrow} E+1*10 \underset{rm}{\Rightarrow} N+1*10 \underset{rm}{\Rightarrow} 1+1*10$$

Example 3.13 [G_{re} (Leftmost)] Leftmost derivation for the string $0^*+0.1$.

$$E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} E^*+E \underset{lm}{\Rightarrow} 0^*+E \underset{lm}{\Rightarrow} 0^*+E.E \underset{lm}{\Rightarrow} 0^*+1.E \underset{lm}{\Rightarrow} 0^*+1.0$$

Example 3.14 [G_{re} (Rightmost)] Rightmost derivation for the string $0^*+0.1$.

$$E \underset{rm}{\Rightarrow} E+E \underset{rm}{\Rightarrow} E+E.E \underset{rm}{\Rightarrow} E+E.1 \underset{rm}{\Rightarrow} E+0.1 \underset{rm}{\Rightarrow} E^*+1.0 \underset{rm}{\Rightarrow} 0^*+1.0$$

Definition 3.6 [Language of a CFG] *If $G(V,T,P,S)$ is a CFG, then the language of G is*

$$L(G) = \{w \in T^* : S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

i.e. the set of strings over T^* derivable from the start symbol S.

If G is a CFG, we call $L(G)$ a context-free language.

Example 3.15 [L_{pal}] The language $L(G_{pal})$ is a context-free language.

Theorem 3.1 [L_{pal}]

$$L(G_{pal}) = \{x \in \{0,1\}^* : w = w^R\}$$

Proof: ($\supseteq$)-direction.) Suppose $w = w^R$. We show by induction on $|w|$ that $w \in L(G_{pal})$:

Basis: $|w| = 0$ or If $|w| = 1$.. Then w is ϵ, 0 or 1. Since $P \to \epsilon$, $P \to 0$, and $P \to 1$ are productions, we conclude that $P \overset{*}{\underset{G_{pal}}{\Rightarrow}} w$ in all base cases.

Induction: Suppose $|w| \geq 2$. Since $w = w^R$, we have $w = 0x0$, or $w = 1x1$, and $x = x^R$. We know from induction hypothesis that $P \overset{*}{\underset{G_{pal}}{\Rightarrow}} x$.

Then either

$$P \underset{G_{pal}}{\Rightarrow} 0P0 \overset{*}{\underset{G_{pal}}{\Rightarrow}} 0x0 = w$$

i.e. $w \in L(G_{pal})$ or

$$P \underset{G_{pal}}{\Rightarrow} 1P1 \overset{*}{\underset{G_{pal}}{\Rightarrow}} 1x1 = w$$

i.e. $w \in L(G_{pal})$

($\subseteq$-direction) We assume that $w \in L(G_{pal})$ and must show that $w = w^R$.

Since $w \in L(G_{pal})$, we have $P \overset{*}{\underset{G_{pal}}{\Rightarrow}} w$.

We do an induction on the length of $\overset{*}{\underset{G_{pal}}{\Rightarrow}}$

Basis: The derivation $P \overset{*}{\underset{G_{pal}}{\Rightarrow}} w$ is done in one step. Then w must be ϵ, 0, or 1, all palindromes.

Induction: Let $n \geq 1$, and suppose the derivation takes $n + 1$ steps. Then we must have

$$P \underset{G_{pal}}{\Rightarrow} 0P0 \overset{*}{\underset{G_{pal}}{\Rightarrow}} 0x0 = w$$

or

$$P \underset{G_{pal}}{\Rightarrow} 1P1 \overset{*}{\underset{G_{pal}}{\Rightarrow}} 1x1 = w$$

where the second derivation is done in n steps.

By induction hypothesis x is a palindrome, and the inductive proof is complete. □

Definition 3.7 [Sentential Form] *Let $G(V, T, P, S)$ be a CFG, and $\alpha \in (V \cup T)^*$. If*

$$S \overset{*}{\Rightarrow} \alpha$$

we say that α is a sentential form.

If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, we say that α is a left-sentential form *and if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, we say that α is a* right-sentential form

Definition 3.8 [Sentential Form] *$L(G)$ is the set of sentential forms that are in T^*.*

Example 3.16 [G_{re}]

$0^* + E.E$ is a left-sentential form.

$$E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} E^* + E \underset{lm}{\Rightarrow} 0^* + E \underset{lm}{\Rightarrow} 0^* + E.E$$

$E + 0.1$ is a right-sentential form.

$$E \underset{rm}{\Rightarrow} E + E \underset{rm}{\Rightarrow} E + E.E \underset{rm}{\Rightarrow} E + E.1 \underset{rm}{\Rightarrow} E + 0.1$$

Exercise 3.1 [L_{eq}] *Design a context-free grammar for L_{eq} be the language of strings with equal number of zero's and one's.*

Exercise 3.2 [Parentheses] *Design a grammar for all and only all strings of round and square parentheses that are balanced.*

Exercise 3.3 [Lisp] *In* LISP *there are two fundamental data types:* atoms *and* lists. *A list is a finite ordered sequence of elements, where each element is in itself either an atom or a list, and an atom is a number or a symbol. A list may be empty. A symbol is essentially a unique named item, written as an alphanumeric string in source code, and used either as a variable name or as a data item in symbolic processing.*

For example, the list `(FOO (BAR 1) 2 ())` *contains four elements: the symbol* `FOO`*, the list* `(BAR 1)`*, the number 2 and the empty list* `()`*.*

Design a grammar for LISP*.*

Exercise 3.4 [Regular Expression] *The following grammar generates the language of a regular expression:*

$$\begin{array}{lcl} S & \rightarrow & A1B \\ A & \rightarrow & 0A|\epsilon \\ B & \rightarrow & 0B|1B|\epsilon \end{array}$$

1. *Which regular expression is described by the grammar?*
2. *Give leftmost and rightmost derivation of* 00101

Exercise 3.5 [CFG] *Give a context-free grammar that generates the language*

$$L = \{a^n b^{n+1} : n \in \mathbb{N}\}$$

Exercise 3.6 [Derivation] *Consider the context free grammar:*

$$S \rightarrow SS+ \quad | \quad SS* \quad | \quad a$$

Give the leftmost and the rightmost derivation for the string $aa+aa+*$.

3.2 Parse Trees

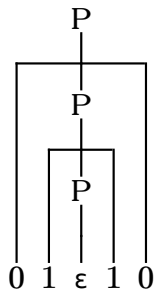
Definition 3.9 [Parse Tree] *Let $G(V,T,P,S)$ be a CFG. A tree is a* parse tree *for G if:*

1. *Each interior node is labelled by a variable in V.*
2. *Each leaf is labelled by a symbol in $V \cup T \cup \{\epsilon\}$. Any ϵ-labelled leaf is the only child of its parent.*
3. *If an interior node is labelled A, and its children (from left to right) labelled*

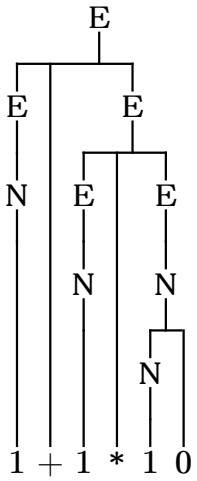
   $$X_1, X_2, \ldots, X_k$$

   then $A \rightarrow X_1 X_2 \ldots X_k \in P$.

Example 3.17 [G_{pal}] Following parse tree shows the derivation of $P \underset{G_{pal}}{\overset{*}{\Rightarrow}} 0110$



Example 3.18 [G_{ari}] Following parse tree shows the derivation of $E \underset{G_{ari}}{\overset{*}{\Rightarrow}} 1 + 1 * 10$



Definition 3.10 [Yield of a Parse Tree]

The yield *of a parse tree is the string of leaves from left to right.*

Important are those parse trees where:

1. The yield is a terminal string.
2. The root is labelled by the start symbol

We shall see the the set of yields of these important parse trees is the language of the grammar.

3.2.1 Inference, Derivation and Parse Trees

Theorem 3.2 [Inference, Derivation and Parse Trees] *Let $G(V,T,P,S)$ be a grammar, and let A be a variable in G. The following are equivalent:*

1. *The recursive inference procedure determines that terminal strings w in the language of variable A.*

2. $A \overset{*}{\Rightarrow} w$.
3. $A \underset{lm}{\overset{*}{\Rightarrow}} w$.
4. $A \underset{rm}{\overset{*}{\Rightarrow}} w$.
5. *There is a parse tree with root A and yield w.*

□

3.3 Ambiguity in Grammars and Languages

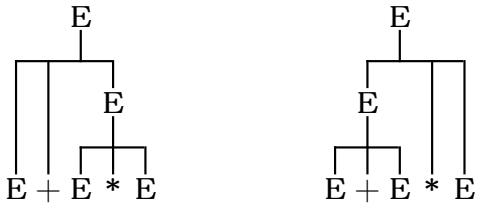
Example 3.19 [G_{ari}] In the grammar G_{ari}, the sentential form $E + E * E$ has two derivations:

$$E \Rightarrow E + E \Rightarrow E + E * E$$

and

$$E \Rightarrow E * E \Rightarrow E + E * E$$

This gives us two different parse trees:



The first parse tree states that the multiplication has a higher priority than the addition. The second parse tree states exactly the contrary.

In the same grammar, the string $0 + 1$ has several derivations

$$E \Rightarrow E + E \Rightarrow N + E \Rightarrow 0 + E \Rightarrow 0 + N \Rightarrow 0 + 1$$

and

$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow E + 1 \Rightarrow I + 1 \Rightarrow 0 + 1$$

but only one parse tree

The mere existence of several derivations is not dangerous, it is the existence of several parse trees that ruins a grammar.

Definition 3.11 [Ambiguous Grammar] *Let $G(V,T,P,S)$ be a CFG.We say that G is* ambiguous *if there is a string in T^* that has more than one parse tree.*

If every string in $L(G)$ has at most one parse tree, G is said to be unambiguous.

3.3.1 Removing Ambiguity in Grammars

Good news: Sometimes we can remove ambiguity *by hand*

Bad news: There is no algorithm to do it

More bad news: Some CFL's have only ambiguous CFG's

Example 3.20 [G_{ari}] Consider the grammar G_{ari}. There are two problems:

1. There is no precedence between $*$ and $+$.
2. There is no grouping of sequences of operators, e.g. is $E + E + E$ meant to be $E + (E + E)$ or $(E + E) + E$.

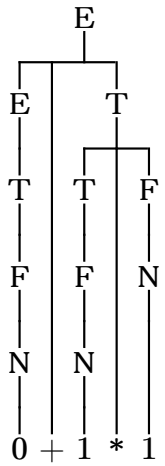
Solution: We introduce more variables, each representing expressions of same *binding strength*.

1. A factor F is an expression that cannot be broken apart by an adjacent $*$, $/$, $-$ or $+$. Our factors are
   (a) Numbers N
   (b) A parenthesised expression.
2. A term T is an expression that cannot be broken by $+$ and $-$. For instance $1 * 10$ can be broken by $11*$ i.e. $11 * 1 * 10 = (11 * 1) * 10$. It cannot be broken by $+$, since e.g. $11 + 1 * 10$ is (by precedence rules) same as $11 + (1 * 10)$.
3. The rest are expressions, i.e. they can be broken apart with $*$, $/$, $-$ or $+$.

We modify now the set P of productions of G_{ari} as follows

$$
\begin{array}{lcl}
E & \to & T \\
E & \to & E + T \\
E & \to & E - T \\
T & \to & F \\
T & \to & T * F \\
T & \to & T/F \\
F & \to & (E) \\
F & \to & N \\
N & \to & 1 \\
N & \to & 0 \\
N & \to & 0N \\
N & \to & 1N
\end{array}
$$

Now the only parse tree for $0 + 1 * 1$ will be



Exercise 3.7 [Ambiguity] *Is the following grammar ambiguous (I represents a statement of a programming language, and E is a Boolean expression)?*

$$
\begin{array}{rcl}
I & \rightarrow & \mathit{if}\ E\ \mathit{then}\ I \\
 & | & \mathit{if}\ E\ I\ \mathit{then}\ I\ \mathit{else}\ I \\
 & | & s_1 \mid s_2 \mid \ldots \mid s_n \\
E & \rightarrow & e_1 \mid e_2 \mid \ldots \mid e_m
\end{array}
$$

Exercise 3.8 [Ambiguity]

Consider the grammar

$$S \rightarrow aS \mid aSbS \mid \epsilon$$

This grammar generates all and only all strings of a's and b's such that every prefix has at least as many a's as b's. This grammar is ambiguous.

1. *Show that the string aab has two rightmost derivations.*
2. *Find an unambiguous grammar.*

Chapter 4

Parsing

A program verifying the language generated by a grammar is called *parser*. It is quite easy to write a JAVA-Program that will check if a string w belongs to the language $L(G)$ of some CFG G. We can just use backtracking, but this method is not efficient [ASU07].

To do this, we will introduce some restrictions to the grammars, such that fast parsing is possible.

To illustrate that process, we will start with the grammar of example 4.21.

Example 4.21 [Grammar for Expressions]

$$\begin{aligned} E &\to E + E \\ E &\to E - E \\ E &\to E * E \\ E &\to E/E \\ E &\to (E) \\ E &\to \text{num} \end{aligned}$$

or (BNF notation)

$$E \to \quad E + E \quad | \quad E - E \quad | \quad E * E \quad | \quad E/E \quad | \quad (E) \quad | \quad \text{num}$$

Remark 4.1 [Grammar for BNF] An important grammar is G_{BNF}, the grammar for BNF-Grammars:

$$\begin{aligned} bnf &\to production\ \ bnf \ \mid\ \epsilon \\ production &\to \text{<NT>}\ \ \text{"}\to\text{"}\ \ choice \\ choice &\to sequence \ \mid\ sequence\ \ \text{"|"}\ \ choice \\ sequence &\to factor \ \mid\ factor\ \ sequence \\ factor &\to \text{<NT>} \ \mid\ \text{<T>} \end{aligned}$$

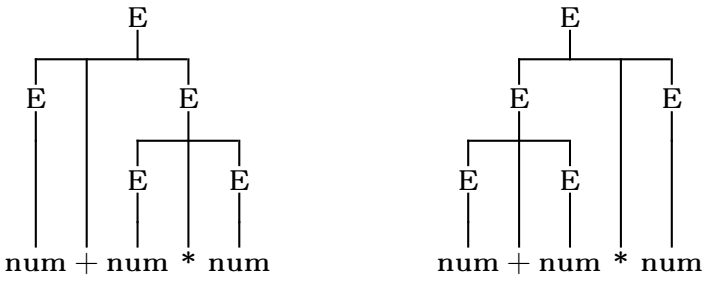
4.1 Writing a Grammar

4.1.1 Ambiguity

We know from example 3.19 that sentential forms may have different parse trees, leading to ambiguity. Grammars that produce more than one leftmost derivation or more than

one rightmost derivation for the same sentence are not suitable.

For example, the grammar of example 4.21 has two different parse trees for num + num ∗ num:



For most parsers, it is desirable that the grammar be made unambiguous, for if it is not, we cannot uniquely determine which parse tree to select for a sentence.

We know from section 3.3.1 how to remove ambiguity.

Example 4.22 [Ambiguity] Removing ambiguity in grammar 4.21.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & E - T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & T / F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & \text{num}
\end{array}
$$

or (BNF notation)

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \text{num}
\end{array}
$$

4.1.2 Lexical Versus Syntactic Analysis

Everything that can be described by a regular expression can be also described by a grammar. In compiler construction, it is usual to use both:

1. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularising the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as a grammar, e.g. num: $(0 + 1)*$.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analysers can be constructed automatically [Les75], [JCC96] from regular expressions than from arbitrary grammars.

4.1.3 Eliminating of Left Recursion

Definition 4.1 [Left Recursive Grammar] *A grammar is* left recursive *if it has a nonterminal A such that there is a derivation $A \stackrel{*}{\Rightarrow} A\alpha$ for some string α.*

Top-down parsers cannot handle left recursion.

Example 4.23 [Left Recursion] Removing left recursion in grammar 4.22.

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \\ E' & \rightarrow & -TE' \\ E' & \rightarrow & \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \\ T' & \rightarrow & /FT' \\ T' & \rightarrow & \epsilon \\ F & \rightarrow & (E) \\ F & \rightarrow & \text{num} \end{array}$$

or (BNF notation)

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid -TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid /FT' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$

The grammar has been obtained by eliminating immediate left recursion from the grammar 4.23. The left recursive productions $E \rightarrow E + E|E - E|T$ are replaced by $E \rightarrow TE'$ and $E' \rightarrow +TE'| - TE'|\epsilon$. The new productions for T and T' are obtained similarly.

Immediate left recursion can be eliminated by the following technique. First group the productions as

$$A \rightarrow A\alpha_1| \cdots |A\alpha_m|\beta_1| \cdots |\beta_n$$

where no β_i begins with A. Then replace the A-productions by

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A'| \cdots |\beta_n A' \\ A' & \rightarrow & \alpha_1 A'| \cdots |\alpha_m A'|\epsilon \end{array}$$

Notice that the left recursion has been transformed into a right recursion.

This procedure does not eliminate left recursion involving derivations of two or more steps.

Algorithm 4.1 [Eliminating Left Recursion] Input: Grammar G with no cycles or ϵ-productions

Output: An equivalent grammar with no left recursion.

Method: Apply following algorithm to G. The resulting non-left recursive grammar may have ϵ-productions:

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
for (each i from 1 to n) {
    for (each j from 1 to n) {
        replace each production of the form $A_i \rightarrow A_j\gamma$
            by the production $A_i \rightarrow \delta_1\gamma|\cdots|\delta_k\gamma$, where
            $A_j \rightarrow \delta_1|\cdots|\delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

4.1.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough input has been seen that we can make the right choice.

Example 4.24 [Left Factoring] Left factoring of grammar 4.22

$$\begin{array}{lcl} E & \rightarrow & EE'' \\ E & \rightarrow & T \\ E'' & \rightarrow & +T \\ E'' & \rightarrow & -T \\ T & \rightarrow & TT'' \\ T'' & \rightarrow & *F \\ T'' & \rightarrow & /F \\ T & \rightarrow & F \\ F & \rightarrow & (E) \\ F & \rightarrow & \text{num} \end{array}$$

or (BNF notation)

$$\begin{array}{lcl} E & \rightarrow & EE'' \mid T \\ E'' & \rightarrow & +T \mid -T \mid \\ T & \rightarrow & TT'' \mid F \\ T'' & \rightarrow & *F \mid /F \mid F \\ F & \rightarrow & (E) \mid \text{num} \end{array}$$

Algorithm 4.2 [Left Factoring] Input: Grammar G.

Output: An equivalent left-factored grammar.

Method: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ – i.e., there is a nontrivial common prefix – replace all of the A-productions $A \rightarrow \alpha\beta_1|\cdots|\alpha\beta_n|\gamma$, where γ represent all alternatives that do not begin with α, by

$$\begin{array}{lcl} A & \rightarrow & \alpha A'|\gamma \\ A' & \rightarrow & \beta_1|\cdots|\beta_n \end{array}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have common prefix.

4.1.5 Non-Context-Free language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using CFG's alone.

Example 4.25 [Declarations] The language in this example abstracts the problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form wcw, where the first w represent the declaration of an identifier w, c represents an intervening program fragment, and the second w represents the use of the identifier.

The abstract language is $L_1 = \{wcw | w \in \Sigma^*\}$. It is possible to prove that L_1 cannot be the language of some CFG G.

Exercise 4.1 [Left Factoring] *Left factor the following grammar:*

$$E \rightarrow int \quad | \quad int + E \quad | \quad int - E \quad | \quad E - (E)$$

Exercise 4.2 [Left Recursion] *Eliminate left-recursion from the following grammar:*

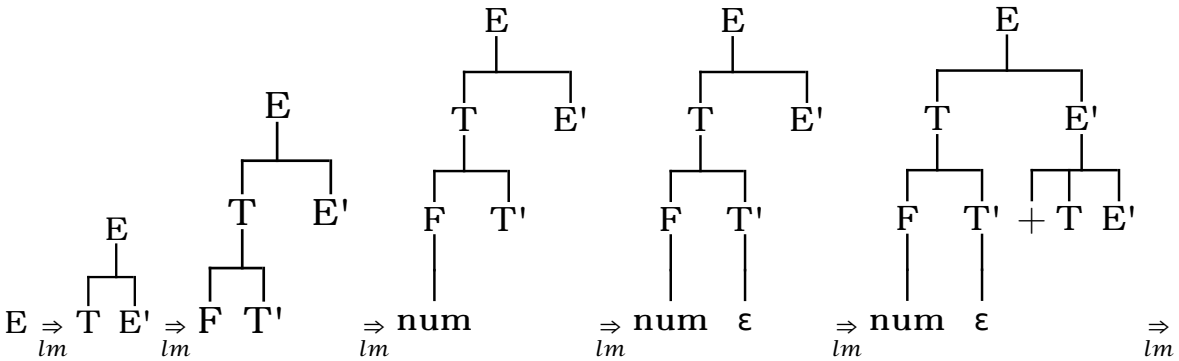
$$\begin{array}{lcl} A & \rightarrow & A + B \quad | \quad B \\ B & \rightarrow & int \quad | \quad (A) \end{array}$$

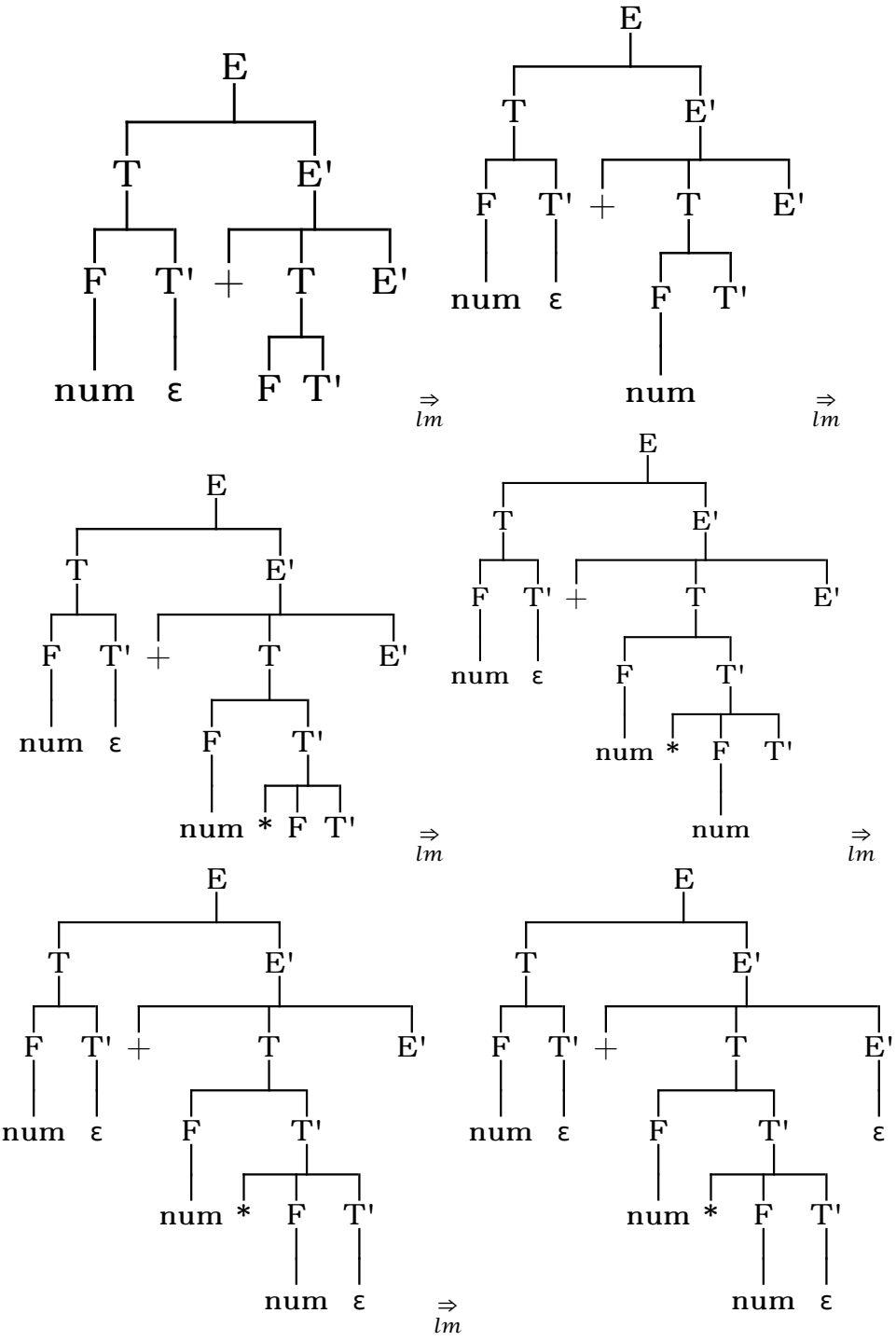
4.2 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example 4.26 [Top-Down Parsing] Sequence of parse trees for the input num+num∗num according to the grammar

$$\begin{array}{lcl} E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \quad | \quad -TE' \quad | \quad \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \quad | \quad /FT' \quad | \quad \epsilon \\ F & \rightarrow & (E) \quad | \quad \text{num} \end{array}$$





4.2.1 Recursive-Descent Parsing

Definition 4.2 [Recursive-Descent Parsing] *A* recursive-descent parser *program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Following pseudo code describes the program:*

void $A()$ *{*
Choose an A-production, $A \rightarrow X_1 \cdots X_k$

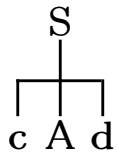
```
    for (i = 0 to k) {
        if ( X_i is nonterminal) {
            call procedure X_i();
        } else if ( X_i equals the current input symbol a) {
            advance the input to the next symbol;
        } else {
            /* an error occurred */
        }
    }
}
```

Note that this pseudo code is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

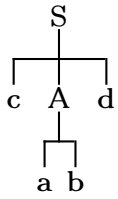
Example 4.27 [Recursive-Descent Parsing] Consider the Grammar

$$\begin{aligned} S &\rightarrow cAd \\ A &\rightarrow ab|a \end{aligned}$$

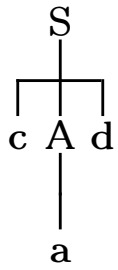
To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labelled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree



The leftmost leave labelled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf la baled A. We expand A using the first alternative $A \rightarrow ab$



We have a match for the second symbol a of w, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labelled b. Since b does not match d, there is an error and we go back to A to see whether there is another alternative for A. The second alternative produces the tree

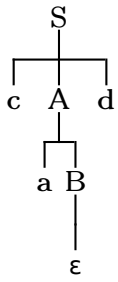


and we are done.

Remark 4.2 [Recursive-Descent Parsing] The grammar 4.27 has not been left factorised. Following factorised version will allow recursive-descent parsing without backtracking.

$$\begin{array}{lcl} S & \rightarrow & cAd \\ A & \rightarrow & aB \\ B & \rightarrow & b|\epsilon \end{array}$$

The parse tree for the input $w = cad$:

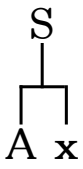


Remark 4.3 [Recursive-Descent Parsing] A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go an infinite loop. That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input.

Example 4.28 [Recursive-Descent Parsing] Consider the Grammar

$$\begin{array}{lcl} S & \rightarrow & Ax \\ A & \rightarrow & x|\epsilon \end{array}$$

To construct a parse tree top-down for the input string $w = x$, begin with a tree consisting of a single node labelled S, and the input pointer pointing to x, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree



Since A may start with x, we expand A and get

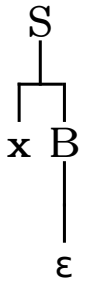
We consume x. The remaining of the input string is empty but there still a leaf x in the syntax tree that not been matched. There is an error and we have to go back, replacing A by its second alternative.

and we are done.

Remark 4.4 [Recursive-Descent Parsing] The grammar 4.28 has not been left factorised. Following factorised version will allow recursive-descent parsing without backtracking.

$$\begin{array}{lcl} S & \rightarrow & xB \\ B & \rightarrow & x|\epsilon \end{array}$$

The parse tree looks as follows:



4.2.2 FIRST and FOLLOW

The construction of top-down parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. FIRST and FOLLOW allow us to choose which production to apply, based on the next input.

Definition 4.3 [FIRST] *Let $G = (V, T, P, S)$ be a CFG ant let $w \in (V \cup T)^*$ be a string of grammar symbols.*

$$FIRST(w) = \begin{cases} \{a \in T : w \stackrel{*}{\Rightarrow} av, v \in (V \cup T)^*\} \cup \{\epsilon\} & \text{if } w \stackrel{*}{\Rightarrow} \epsilon \\ \{a \in T : w \stackrel{*}{\Rightarrow} av, v \in (V \cup T)^*\} & \text{else} \end{cases}$$

Definition 4.4 [FOLLOW] *Let $G = (V, T, P, S)$ be a CFG ant let $A \in V$ be a nonterminal.*

$$FOLLOW(A) = \{a \in T : S \stackrel{*}{\Rightarrow} \alpha A a \beta, \alpha, \beta \in (V \cup T)^*\}$$

Algorithm 4.3 [FIRST] Let $G = (V, T, P, S)$ be a CFG.

1. If X is a terminal, then FIRST$(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_{i-1}). If ϵ is in all of FIRST(Y_j), $j = 1, 2, \ldots k$, then add ϵ to FIRST(X)
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X)

Algorithm 4.4 [FOLLOW] Let $G = (V, T, P, S)$ be a CFG.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right end marker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is an element of FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example 4.29 [FIRST/FOLLOW] Consider the non-left-recursive grammar 4.23. Then:

1. FIRST(F) = FIRST(T) = FIRST(E) = \{(, num\}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, num and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).
2. FIRST(E') = $\{+, -, \epsilon\}$. Follows directly from the three E'-productions.
3. FIRST(T') = $\{*, /, \epsilon\}$. Follows directly from the three T'-productions.
4. FOLLOW(E) = FOLLOW(E') = \{), \$\}. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is an element of FOLLOW(E). For FOLLOW(E'), note this nonterminal appears only at the end of bodies of E-productions. Thus FOLLOW(E') must be the same as FOLLOW(E).
5. FOLLOW(T) = FOLLOW(T') = $\{+, -,), \$\}$. Notice that T appears in bodies only followed by E'. Thus everything, except ϵ that is in FIRST(E') must be in FOLLOW(T), that explains the symbols + and -. However, since FIRST(E') contains ϵ, and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the end of the T-productions, it must be that FOLLOW(T) = FOLLOW(T')

6. FOLLOW(F) = $\{+, -, *, /,), \$\}$. Reasoning analogous to point (5).

Remark 4.5 [Computing FIRST and FOLLOW] It is difficult to compute FIRST and FOLLOW without using a computer. Most compiler construction tools like JAVACC [JCC96] can compute FIRST and FOLLOW and also detect left-recursions automatically.

Exercise 4.3 [First and Follow] *Consider the context free grammar:*

$$S \rightarrow SS+ \quad | \quad SS* \quad | \quad a$$

Compute FIRST and FOLLOW.

4.3 LL(1)-Grammars

We start withe two important rules:

Rule 4.1 [FIRST] A CFG G satisfies the FIRST-rule if for each production $A \rightarrow w_1|w_2|...|w_n$,

$$\text{FIRST}(w_i) \cap \text{FIRST}(w_j) = \emptyset \quad \text{for} \quad i \neq j$$

Rule 4.2 [FOLLOW] A CFG G satisfies the FOLLOW-rule if for each nonterminal A such that $\epsilon \in \text{FIRST}(A)$,

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$$

Definition 4.5 [LL(1)-Grammars] *A* LL(1)*-grammar is a grammar satisfying the rules 4.1 and 4.2.*

Remark 4.6 [LL(1)-Grammars] Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for LL(1)-grammars. The first L in LL(1) stands for scanning the input from left to right, the second L for producing a leftmost derivation, and the 1 for using one input symbol of lookahead at each step to make parsing decisions.

Remark 4.7 [LL(k)-Grammars] The first L in LL(k) stands for scanning the input from left to right, the second L for producing a leftmost derivation, and the k for using k input symbols of lookahead at each step to make parsing decisions.

Definition 4.6 [PREDICT] *Let $A \rightarrow w$ be a production.*

$$PREDICT(A \rightarrow w) = \begin{cases} (FIRST(w) - \epsilon) \cup FOLLOW(w) & if \quad \epsilon \in FIRST(w) \\ FIRST(w) & otherwise \end{cases}$$

PREDICT($A \rightarrow w$) contains all terminal expected when processing production $A \rightarrow w$.

Definition 4.7 [LL(1) Parsing] *A* recursive LL(1) parser *program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. The program uses a variable called* lookahead *pointing to the actual position of the input string. Following pseudo code describes the program:*

```
void A() {
    if (∃A → X_1 ··· X_k : lookahead ∈ PREDICT(A → X_1 ··· X_k)) {
        for (i = 0 to k) {
            if ( X_i is nonterminal) {
                call procedure X_i()
            } else if ( X_i = lookahead) {
                lookahead = nextSymbol()
            } else {
                /* syntax error (unexpected token) */
            }
        }
    } else {
        /* syntax error (unexpected token) */
    }
}
```

Note that this pseudo code is now deterministic, since the grammar is LL(1).

Example 4.30 [Recursive Parser for Expressions]

Following JAVA-classes contain a complete implementation of a recursive parser for the grammar 4.23.

The terminal symbols of the grammars are implemented as tokens. The tokens are defined using following regular expressions:

```
token: LPAR "\("
token: RPAR "\)"
token: PLUS "+"
token: MINUS "\-"
token: TIMES "\*"
token: DIV "/"
token: NUM "[0-9][0-9]*"
```

JAVA-Class `Token`

```
 1  public class Token {
 2    public String lexem;
 3    public int type;
 4    public int startLine;
 5    public int startColumn;
 6    public int endLine;
 7    public int endColumn;
 8
 9    public Token(String lexem, int type, int startLine,
10        int startColumn, int endLine, int endColumn) {
11      this.lexem = lexem;
12      this.type = type;
13      this.startLine = startLine;
14      this.endLine = endLine;
15      this.startColumn = startColumn;
```

```
16      this.endColumn = endColumn;
17    }
18
19    public String toString() {
20      return "<[" + lexem + "]," + type + "," + startLine
21        + "," + startColumn + "," + endLine + ","
22        + endColumn + ">";
23    }
24  }
```

Here follows a simple scanner.

JAVA-Class Scanner

```
 1  public class Scanner implements Constants {
 2
 3    private final int[][] packedTable = {
 4      { -40, 1, 2, 3, 4, -1, 5, -1, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7,
 5        7, -70 },
 6      { -128 }, { -128 }, { -128 }, { -128 }, { -128 }, { -128 },
 7      { -48, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, -70 }
 8    };
 9    public int[][] table = null;
10    public int start = 0;
11    public boolean[] accepting = { false, true, true, true, true,
12       true, true, true };
13    public int[] type = { -1, 1, 2, 5, 3, 4, 6, 7 };
14    private final int states;
15    public final int INPUTS = 128;
16    private int line = 1;
17    private int column = 1;
18    private final PushbackReader reader;
19
20    public Scanner(PushbackReader reader) {
21      this.reader = reader;
22      this.states = packedTable.length;
23      this.table = new int[this.states][INPUTS];
24      unpackTable();
25    }
26
27    public Token getNextToken() throws Exception {
28      return execute();
29    }
30
31    public Token execute() throws Exception {
32      Token token = null;
33      int state = start;
34      int matchedState = SE;
35      String lexem = "";
36      int startLine = line;
37      int startColumn = column;
```

```
38      int endLine = line;
39      int endColumn = column;
40      boolean found = false;
41      try {
42        int i = reader.read();
43        String s = "";
44        if (i == -1) {
45          token = new Token("", 0, startLine, startColumn,
46            endLine, endColumn);
47          found = true;
48        }
49        while (i != -1) {
50          char c = (char) i;
51          s += "" + c;
52          if (c == '\n') {
53            line++;
54            column = 1;
55          } else {
56            column++;
57          }
58          state = table[state][c];
59          if (state != SE && accepting[state]) {
60            lexem += s;
61            s = "";
62            matchedState = state;
63            endLine = line;
64            endColumn = column;
65            token = new Token(lexem, type[matchedState],
66              startLine, startColumn, endLine, endColumn);
67            found = true;
68          } else if (state == SE) {
69            break;
70          }
71          i = reader.read();
72        }
73        if (found) {
74          if (s.length() != 0) {
75            reader.unread(s.toCharArray());
76          }
77          line = endLine;
78          column = endColumn;
79        } else if (i == -1) {
80          throw new RuntimeException("unexpected end of file "
81            + "after reading [" + s + "] at line " + endLine
82            + ", column " + endColumn);
83        } else {
84          throw new RuntimeException("lexical error found "
85            + "after reading [" + s + "] at line " + endLine
86            + ", column " + endColumn);
87        }
```

```
88      } catch (IOException e) {}
89      return token;
90    }
91
92    public void unpackTable() {
93      for (int i = 0; i < packedTable.length; i++) {
94        int j = 0;
95        for (int n = 0; n < packedTable[i].length; n++) {
96          int r = packedTable[i][n];
97          if (r >= 0) {
98            table[i][j] = r;
99            j++;
100         } else {
101           r = -r;
102           while (r > 0) {
103             table[i][j] = -1;
104             r--;
105             j++;
106           }
107         }
108       }
109     }
110   }
111 }
```

Helper class.

JAVA-Class `Constants`

```
1  public interface Constants {
2
3    public final int SE = -1;
4    public final int EOF = 0;
5    public final int LPAR = 1;
6    public final int RPAR = 2;
7    public final int PLUS = 3;
8    public final int MINUS = 4;
9    public final int TIMES = 5;
10   public final int DIV = 6;
11   public final int NUM = 7;
12   public final String[] names = { "EOF", "LPAR", "RPAR",
13     "PLUS", "MINUS", "TIMES", "DIV", "NUM" };
14 }
```

For the implementation of the scanner, we use algorithm 4.7 and the results of example 4.29. The method `match()` is used to consume the next input symbol.

- Implementation of the E-productions: There is no choice. In the implementation of `E()` we expect `lookahead` to belong to FIRST(E) and we call `T()` and `Ep()`, otherwise, there is an error.

- Implementation of the E'-productions: There is a choice. If `lookahead` is +, we eat + and call `T()` and `Ep()`, if `lookahead` is −, we eat − and call `T()` and `Ep()`. Since FIRST(E') contains ϵ, we must accept) and \$ (they belong to FOLLOW(E')), but we do nothing (they will be processed by other procedures). Any other value of `lookahead` is an error.

JAVA-Class `Parser`

```
1  public class Parser implements Constants {
2
3    private static Token lookahead;
4    private final Scanner scanner;
5
6    public Parser(Scanner scanner) throws Exception {
7      this.scanner = scanner;
8      lookahead = this.scanner.getNextToken();
9    }
10
11   private void match(int type) {
12     if (lookahead.type == type) {
13       try {
14         lookahead = scanner.getNextToken();
15       } catch (Exception e) {
16         e.printStackTrace();
17       }
18     } else {
19       throw new RuntimeException();
20     }
21   }
22
23   public void E() {
24     if (lookahead.type == LPAR || lookahead.type == NUM) {
25       T();
26       Ep();
27     } else {
28       throw new RuntimeException(expected(lookahead,
29         "[LPAR] or [NUM]"));
30     }
31   }
32
33   public void T() {
34     if (lookahead.type == LPAR || lookahead.type == NUM) {
35       F();
36       Tp();
37     } else {
38       throw new RuntimeException(expected(lookahead,
39         "[LPAR] or [NUM]"));
40     }
41   }
42
```

```
43    public void F() {
44      switch (lookahead.type) {
45        case LPAR:
46          match(LPAR);
47          E();
48          match(RPAR);
49          break;
50        case NUM:
51          match(NUM);
52          break;
53        default:
54          throw new RuntimeException(expected(lookahead,
55            "[LPAR] or [NUM]"));
56      }
57    }
58
59    public void Ep() {
60      switch (lookahead.type) {
61        case PLUS:
62          match(PLUS);
63          T();
64          Ep();
65          break;
66        case MINUS:
67          match(MINUS);
68          T();
69          Ep();
70          break;
71        case RPAR:
72        case EOF:
73          break;
74        default:
75          throw new RuntimeException(expected(lookahead,
76              "[PLUS], [MINUS], [RPAR] or [EOF]"));
77      }
78    }
79
80    public void Tp() {
81      switch (lookahead.type) {
82        case TIMES:
83          match(TIMES);
84          F();
85          Tp();
86          break;
87        case DIV:
88          match(DIV);
89          F();
90          Tp();
91          break;
92        case PLUS:
```

```
93          case MINUS:
94          case RPAR:
95          case EOF:
96            break;
97          default:
98            throw new RuntimeException(expected(lookahead,
99                "[PLUS], [MINUS], [TIMES], [DIV], [RPAR] or [EOF]"));
100       }
101     }
102
103     private String expected(Token token, String s) {
104       String r = "unexpected token [" + names[token.type]
105         + "] at line " + token.startLine;
106       r += " column " + token.startColumn;
107       r += " was expexting " + s;
108       return r;
109     }
110   }
```

Here a small tester:

JAVA-Class `Main`

```
1   public class Main {
2
3     public static void main(String[] args) {
4       try {
5         PushbackReader reader =
6           new PushbackReader(new FileReader(args[0]));
7         Scanner scanner = new Scanner(reader);
8         Parser parser = new Parser(scanner);
9         parser.E();
10      } catch (Exception e) {
11        e.printStackTrace();
12      }
13    }
14  }
```

Exercise 4.4 [LL(1)] *Consider the context free grammar:*

$$S \to SS+ \quad | \quad SS* \quad | \quad a$$

1. *Transform the grammar into an LL(1) grammar.*
2. *Compute FIRST and FOLLOW for the transformed grammar.*

4.4 LL(1)-Grammars in EBNF

Let $G = (V, T, P, S)$ be a LL(1) grammar and assume that the productions are written in EBNF. We show here how to implement each EBNF constructs ($w, w_i \in (V \cup T)^*$). $P(w)$ denotes the procedure associated with w. See also [Wir96].

1. We associate the following procedure $A()$ to a production $A \rightarrow w_1|w_2|\ldots|w_n$:

```
void A() {
    if (lookahead ∈ (FIRST(w_1) − {ϵ})) {
        P(w_1)
    } else if (lookahead ∈ (FIRST(w_2) − {ϵ})) {
        P(w_2)
    } else if ...
    ...
    } else if (lookahead ∈ (FIRST(w_n) − {ϵ})) {
        P(w_n)
    } else if (lookahead ∈ FOLLOW(A)) {
        nothing()
    } else {
        error()
    }
}
```

2. We associate the following procedure $A()$ to a production $A \rightarrow w_1 w_2 \ldots w_n$:

```
void A() {
    P(w_1)
    P(w_2)
    ...
    P(w_n)
}
```

3. We associate the following procedure $A()$ to a production $A \rightarrow w^*$:

```
void A() {
    while (lookahead ∈ (FIRST(w) − {ϵ})) {
        P(w)
    }
}
```

4. We associate the following procedure $A()$ to a production $A \rightarrow w^+$:

```
void A() {
    do {
        P(w)
    } while (lookahead ∈ (FIRST(w) − {ϵ}))
}
```

5. We associate the following procedure $A()$ to a production $A \rightarrow w^?$:

```
void A() {
    if (lookahead ∈ (FIRST(w) − {ϵ})) {
```

```
        P(w)
    }
}
```

6. We associate the following procedure $A()$ to a production $A \to B$ ($B \in V$):

```
void A() {
    B()
}
```

7. We associate the following procedure $A()$ to a production $A \to \epsilon$:

```
void A() {
    nothing()
}
```

8. We associate the following procedure $A()$ to a production $A \to a$ ($a \in T$):

```
void A() {
    if (lookahead = a) {
        match(a)
    } else {
        error()
    }
}
```

4.5 Implementation with JavaCC

Consider following grammar G (see also example 4.26):

$$\begin{array}{lcl} E & \to & TE' \\ E' & \to & +TE' \mid -TE' \mid \epsilon \\ T & \to & FT' \\ T' & \to & *FT' \mid /FT' \mid \epsilon \\ F & \to & (E) \mid \text{num} \end{array}$$

The first step will be the transformation of G into EBNF notation (since JavaCC uses that notation):

$$\begin{array}{lcl} E & \to & T \ (+T \mid -T)* \\ T & \to & F \ (*F \mid /F)* \\ F & \to & (E) \mid \text{num} \end{array}$$

The translation of G into a JavaCC specification is almost one to one:

JavaCC-specification `NakedParser`

```
1  PARSER_BEGIN(NakedParser)
2
3  package utg.calculator.parser;
4
5  public class NakedParser {}
6
7  PARSER_END(NakedParser)
8
9  SKIP :
10 {
11   " " | "\t" | "\n" | "\r" | "\f"
12 }
13
14 TOKEN :
15 {
16   < NUMBER: ["0"-"9"] (["0"-"9"])* >
17 }
18
19 void e() :
20 {}
21 {
22   t() ("+" t() | "-" t() t())*
23 }
24
25 void t() :
26 {}
27 {
28   f() ("*" f() | "/" f())*
29 }
30
31 void f() :
32 {}
33 {
34   "(" e() ")" | < NUMBER >
35 }
```

The parser may be tested (after compilation withJAVACC) using following JAVA class:

JAVA-class `TestNakedParser`

```
1  package utg.calculator.parser;
2
3  import java.io.StringReader;
4
5  public class TestNakedParser {
6
7    public static void main(String[] args) throws ParseException {
8      String test = "1 + 2 * (3 + 4) / 5";
9      new NakedParser(new StringReader(test));
10     NakedParser.e();
11   }
```

```
12  }
```

The program does nothing, except testing the syntax of the input string.

In a second step we would like to compute the value of the input string, to transform it into preorder and postorder notation. This can be done within JAVACC (see A.6).

It is even possible to write an entire compiler that fits within the semantic actions of a JAVACC compiler generator. However, such a compiler is difficult to maintain. And this approach constraints the compiler to analyse the program in exactly the order it is parsed.

To improve modularity, it is better to separate issues of syntax (parsing) from the issues of semantics (type-checking and translation to machine code).

One way to do this is for the parser to produce a parse tree.

4.5.1 Abstract Syntax Tree AST

An *abstract syntax tree* (AST), or just *syntax tree*, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct such as an if-cond-then expression may be denoted by a single node with two branches.

This makes abstract syntax trees different from concrete syntax trees, traditionally called parse trees, which are often built by the parser part of the source code translation and compiling process (despite a perhaps unintuitive naming). Once built, additional information is added to the AST by subsequent processing, e.g., semantic analysis.

We first need an abstract JAVA class to define the interface of the nodes of the abstract syntax tree:

Abstract JAVA-class `ASTNode`

```
1  package utg.calculator.parser.ast;
2
3  public abstract class ASTNode {
4
5    public int priority;
6
7    public abstract int eval();
8
9    public abstract String postorder();
10
11   public ASTNode(int priority) {
12                 this.priority = priority;
13   }
14 }
```

The attribute `priority` is used for pretty printing.

It is highly recommended to provide the abstract tree with an identity transformation (here the `toString()` method) that prints out the syntax tree. In this example, the abstract syntax and the concrete syntax are equivalent. Thus the identity transformation

should produce the original input string up to some whitespaces and parentheses pairs. This is a proof, that nothing got lost during the generation of the abstract syntax tree.

Moreover, applying the identity transformation again on the output of the first identity transformation must produce exactly the same output.

JAVA-class ASTAdd

```
1  package utg.calculator.parser.ast;
2
3  public class ASTAdd extends ASTNode {
4
5    public ASTNode op1;
6    public ASTNode op2;
7
8    public ASTAdd(ASTNode op1, ASTNode op2) {
9      super(2);
10     this.op1 = op1;
11     this.op2 = op2;
12   }
13
14   public String toString() {
15     return op1 + " + " + op2;
16   }
17
18   public int eval() {
19     return op1.eval() + op2.eval();
20   }
21
22   public String postorder() {
23     return op1.postorder() + " " + op2.postorder() + " +";
24   }
25 }
```

JAVA-class ASTSub

```
1  package utg.calculator.parser.ast;
2
3  public class ASTSub extends ASTNode {
4
5    public ASTNode op1;
6    public ASTNode op2;
7
8    public ASTSub(ASTNode op1, ASTNode op2) {
9      super(2);
10     this.op1 = op1;
11     this.op2 = op2;
12   }
13
14   public String toString() {
15     return op1 + " - " + op2;
16   }
```

```
17
18    public int eval() {
19      return op1.eval() - op2.eval();
20    }
21
22    public String postorder() {
23      return op1.postorder() + " " + op2.postorder() + " -";
24    }
25  }
```

JAVA-class ASTMul

```
 1  package utg.calculator.parser.ast;
 2
 3  package utg.calculator.parser.ast;
 4
 5  public class ASTMul extends ASTNode {
 6
 7    public ASTNode op1;
 8    public ASTNode op2;
 9
10    public ASTMul(ASTNode op1, ASTNode op2) {
11      super(2);
12      this.op1 = op1;
13      this.op2 = op2;
14    }
15
16    public String toString() {
17      if (op1.priority > priority && op2.priority > priority) {
18        return "(" + op1 + ") * (" + op2 + ")";
19      } else if (op1.priority > priority) {
20        return "(" + op1 + ") * " + op2;
21      } else if (op2.priority > priority) {
22        return op1 + " * (" + op2 + ")";
23      } else {
24        return op1 + " * " + op2;
25      }
26    }
27
28    public int eval() {
29      return op1.eval() * op2.eval();
30    }
31
32    public String postorder() {
33      return op1.postorder() + " " + op2.postorder() + " *";
34    }
35  }
```

JAVA-class ASTDiv

```
1  package utg.calculator.parser.ast;
```

```
 2
 3  public class ASTDiv extends ASTNode {
 4
 5    public ASTNode op1;
 6    public ASTNode op2;
 7
 8    public ASTDiv(ASTNode op1, ASTNode op2) {
 9      super(1);
10      this.op1 = op1;
11      this.op2 = op2;
12    }
13
14    public String toString() {
15      if (op1.priority > priority && op2.priority > priority) {
16        return "(" + op1 + ") / (" + op2 + ")";
17      } else if (op1.priority > priority) {
18        return "(" + op1 + ") / " + op2;
19      } else if (op2.priority > priority) {
20        return op1 + " / (" + op2 + ")";
21      } else {
22        return op1 + " / " + op2;
23      }
24    }
25
26    public int eval() {
27      return op1.eval() / op2.eval();
28    }
29
30    public String postorder() {
31      return op1.postorder() + " " + op2.postorder() + " /";
32    }
33  }
```

The implementation of the JAVACC specifications is decorated with some semantic actions thus generating the AST structure:

JAVACC-specification ASTParser

```
 1  PARSER_BEGIN(ASTParser)
 2
 3  package utg.calculator.parser;
 4
 5  import utg.calculator.parser.ast.ASTAdd;
 6  import utg.calculator.parser.ast.ASTDiv;
 7  import utg.calculator.parser.ast.ASTMul;
 8  import utg.calculator.parser.ast.ASTNode;
 9  import utg.calculator.parser.ast.ASTNum;
10  import utg.calculator.parser.ast.ASTSub;
11  public class ASTParser {}
12
13  PARSER_END(ASTParser)
```

```
14
15  SKIP :
16  {
17    " " | "\t" | "\n" | "\r" | "\f"
18  }
19
20  TOKEN :
21  {
22    < NUMBER: ["0"-"9"] (["0"-"9"])* >
23  }
24
25  ASTNode e() :
26  {
27    ASTNode result = null;
28    ASTNode node = null;
29  }
30  {
31    result = t()
32    (
33      "+" node = t()
34      {
35        result = new ASTAdd(result, node);
36      }
37    |
38      "-" node = t()
39      {
40        result = new ASTSub(result, node);
41      }
42    )*
43    {
44      return result;
45    }
46  }
47
48  ASTNode t() :
49  {
50    ASTNode result = null;
51    ASTNode node = null;
52  }
53  {
54    result = f()
55    (
56      "*" node = f()
57      {
58        result = new ASTMul(result, node);
59      }
60    |
61      "/" node = f()
62      {
63        result = new ASTDiv(result, node);
```

```
64        }
65      )*
66      {
67        return result;
68      }
69    }
70
71    ASTNode f() :
72    {
73      ASTNode result = null;
74      Token t = null;
75    }
76    {
77      (
78        "(" result = e() ")"
79      |
80        t = < NUMBER >
81        {
82          result = new ASTNum(Integer.parseInt(t.image));
83        }
84      )
85      {
86        return result;
87      }
88    }
```

The parser may be tested (after compilation withJAVACC) using following JAVA class:

JAVA-class `TestASTParser`

```
1   package utg.calculator.parser;
2
3   import java.io.StringReader;
4
5   import utg.calculator.parser.ast.ASTNode;
6
7   public class TestASTParser {
8
9     public static void main(String[] args) throws ParseException {
10      String test = "(1 + 3) + 2 * (3 + 4) / 5";
11      new ASTParser(new StringReader(test));
12      ASTNode node = ASTParser.e();
13      System.out.println(node + " = " + node.eval());
14      System.out.println(node.postorder());
15    }
16  }
```

The test program outputs the original string (identity transformation) computes its value (`eval()`) and transforms the input string into postfix notation.

4.5.2 Visitor for the AST

The maintenance of the implementation of the abstract syntax tree is not simple (especially if the AST consists of many node types). If a new operation should be added, we must write a corresponding method in each of the tree classes.

4.5.3 The Visitor Design Pattern

In object-oriented programming and software engineering, the *visitor design pattern* is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. Thus, using the visitor pattern helps conformance with the open/closed principle.

In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specialisations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

While powerful, the visitor pattern is more limited than conventional virtual functions. It is not possible to create visitors for objects without adding a small callback method inside each class.

The idea is to use a structure of element classes, each of which has an `accept()` method that takes a visitor object as an argument. visitor is an interface that has a `visit()` method for each element class. The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete visitor classes can then be written that perform some particular operations, by implementing these operations in their respective `visit()` methods.

4.5.4 The Visitor Implementation

The implementation of the AST nodes is easier, only keeping the structural aspects of the nodes:

Abstract JAVA-class `ASTVNode`

```
1  package utg.calculator.parser.astv;
2
3  public abstract class ASTVNode {
4
5    public int priority;
6
7    public abstract Object accept(ASTVisitor visitor, Object o);
8
9    public ASTVNode(int priority) {
10     this.priority = priority;
11   }
12 }
```

JAVA-class `ASTVAdd`

```
 1  package utg.calculator.parser.astv;
 2
 3  public class ASTVAdd extends ASTVNode {
 4
 5    public ASTVNode op1;
 6    public ASTVNode op2;
 7
 8    public ASTVAdd(ASTVNode op1, ASTVNode op2) {
 9      super(2);
10      this.op1 = op1;
11      this.op2 = op2;
12    }
13
14    public String toString() {
15      return op1 + " + " + op2;
16    }
17
18    public Object accept(ASTVisitor visitor, Object o) {
19      return visitor.visitAdd(this, o);
20    }
21  }
```

JAVA-class ASTVSub

```
 1  package utg.calculator.parser.astv;
 2
 3  public class ASTVSub extends ASTVNode {
 4
 5    public ASTVNode op1;
 6    public ASTVNode op2;
 7
 8    public ASTVSub(ASTVNode op1, ASTVNode op2) {
 9      super(2);
10      this.op1 = op1;
11      this.op2 = op2;
12    }
13
14    public String toString() {
15      return op1 + " - " + op2;
16    }
17
18    public Object accept(ASTVisitor visitor, Object o) {
19      return visitor.visitSub(this, o);
20    }
21  }
```

JAVA-class ASTVMul

```
 1  package utg.calculator.parser.astv;
 2
```

```
 3  public class ASTVMul extends ASTVNode {
 4
 5    public ASTVNode op1;
 6    public ASTVNode op2;
 7
 8    public ASTVMul(ASTVNode op1, ASTVNode op2) {
 9      super(1);
10      this.op1 = op1;
11      this.op2 = op2;
12    }
13
14    public String toString() {
15      if (op1.priority > priority && op2.priority > priority) {
16        return "(" + op1 + ") * (" + op2 + ")";
17      } else if (op1.priority > priority) {
18        return "(" + op1 + ") * " + op2;
19      } else if (op2.priority > priority) {
20        return op1 + " * (" + op2 + ")";
21      } else {
22        return op1 + " * " + op2;
23      }
24    }
25
26    public Object accept(ASTVisitor visitor, Object o) {
27      return visitor.visitMul(this, o);
28    }
29  }
```

JAVA-class ASTVDiv

```
 1  package utg.calculator.parser.astv;
 2
 3  public class ASTVDiv extends ASTVNode {
 4
 5    public ASTVNode op1;
 6    public ASTVNode op2;
 7
 8    public ASTVDiv(ASTVNode op1, ASTVNode op2) {
 9      super(1);
10      this.op1 = op1;
11      this.op2 = op2;
12    }
13
14    public String toString() {
15      if (op1.priority > priority && op2.priority > priority) {
16        return "(" + op1 + ") / (" + op2 + ")";
17      } else if (op1.priority > priority) {
18        return "(" + op1 + ") / " + op2;
19      } else if (op2.priority > priority) {
20        return op1 + " / (" + op2 + ")";
```

```
21      } else {
22        return op1 + " / " + op2;
23      }
24    }
25
26    public Object accept(ASTVisitor visitor, Object o) {
27      return visitor.visitDiv(this, o);
28    }
29  }
```

The interface ASTVisitor defines a virtual function for each AST class:

Abstract JAVA-class ASTVisitor

```
1   package utg.calculator.parser.astv;
2
3   public abstract class ASTVisitor {
4
5     public abstract Object visitAdd(ASTVAdd add, Object o);
6
7     public abstract Object visitDiv(ASTVDiv div, Object o);
8
9     public abstract Object visitMul(ASTVMul mul, Object o);
10
11    public abstract Object visitNum(ASTVNum num, Object o);
12
13    public abstract Object visitSub(ASTVSub sub, Object o);
14  }
```

Note that all visit() methods and the accept() method have an Object parameter and an Object return value. The Objectreturn value enables the visitor to transfer values bottom up in the AST structure. The Object parameter enables the visitor to transfer values top down in the AST structure.

The EvalVisitor is used for computing the value of the input string:

JAVA-class EvalVisitor

```
1   package utg.calculator.parser.astv;
2
3   public class EvalVisitor extends ASTVisitor {
4
5     public Object visitAdd(ASTVAdd add, Object o) {
6       return (Integer) add.op1.accept(this, o)
7           + (Integer) add.op2.accept(this, o);
8     }
9
10    public Object visitDiv(ASTVDiv div, Object o) {
11      return (Integer) div.op1.accept(this, o)
12          / (Integer) div.op2.accept(this, o);
13    }
14
```

```
15    public Object visitMul(ASTVMul mul, Object o) {
16      return (Integer) mul.op1.accept(this, o)
17          * (Integer) mul.op2.accept(this, o);
18    }
19
20    public Object visitNum(ASTVNum num, Object o) {
21      return num.value;
22    }
23
24    public Object visitSub(ASTVSub sub, Object o) {
25      return (Integer) sub.op1.accept(this, o)
26          - (Integer) sub.op2.accept(this, o);
27    }
28  }
```

The PostfixVisitor is used for transforming the input string into postfix notation:

JAVA-class PostorderVisitor

```
1   package utg.calculator.parser.astv;
2
3   public class PostorderVisitor extends ASTVisitor {
4
5     public Object visitAdd(ASTVAdd add, Object o) {
6       return add.op1.accept(this, o) + " " + add.op2.accept(this, o)
7           + " +";
8     }
9
10    public Object visitDiv(ASTVDiv div, Object o) {
11      return div.op1.accept(this, o) + " " + div.op2.accept(this, o)
12          + " /";
13    }
14
15    public Object visitMul(ASTVMul mul, Object o) {
16      return mul.op1.accept(this, o) + " " + mul.op2.accept(this, o)
17          + " *";
18    }
19
20    public Object visitNum(ASTVNum num, Object o) {
21      return num.value;
22    }
23
24    public Object visitSub(ASTVSub sub, Object o) {
25      return sub.op1.accept(this, o) + " " + sub.op2.accept(this, o)
26          + " -";
27    }
28  }
```

The parser may be tested (after compilation withJAVACC) using following JAVA class:

JAVA-class TestASTVParser

```
1  package utg.calculator.parser;
2
3  import java.io.StringReader;
4
5  import utg.calculator.parser.astv.ASTVNode;
6  import utg.calculator.parser.astv.EvalVisitor;
7  import utg.calculator.parser.astv.PostorderVisitor;
8  import utg.calculator.parser.astv.PreorderVisitor;
9
10 public class TestASTVParser {
11
12   public static void main(String[] args) throws ParseException {
13     String test = "(1 + 3) + 2 * (3 + 4) / 5";
14     new ASTVParser(new StringReader(test));
15     ASTVNode node = ASTVParser.e();
16     System.out.println(node + " = "
17       + node.accept(new EvalVisitor(), null));
18     System.out.println(node.accept(new PreorderVisitor(), null));
19     System.out.println(node.accept(new PostorderVisitor(), null));
20   }
21 }
```

The test program outputs the original string (identity transformation) computes its value and transforms the input string into postfix notation.

Chapter 5

The MC Language

In this section we will define a C-like programming language calls MC (Mini C language).

5.1 The MC Grammar

```
program                 ::= (variableDeclaration)*
                            (functionDeclaration)*
                            (statement)*
                          ;

variableDeclaration      ::= "int" <ID> ";"
                           ;

functionDeclaration      ::= "int" <ID> "(" funcParam ")" blockStatement
                           ;

parameterDeclaration     ::= ("int" <ID> ("," "int" <ID>)*)?
                           ;

blockStatement           ::= "{" (varDeclaration)* (statement)* "}"
                           ;

statement                ::=   <ID> "=" addExpression ";"
                             |
                               whileStatement
                             |
                               ifStatement
                             |
                               "return" andExpression ";"
                             |
                               "print" "(" andExpression ")" ";"
                             |
                               blockStatement
                           ;

whileStatement           ::= "while" "(" orExpression ")" statement
```

```
                          ;

ifStatement             ::= "if" "(" orExpression ")" statement
                            ("else" statement)?
                          ;

orExpression            ::= andExpression ("||" andExpression)*
                          ;

andExpression           ::= relExpression ("&&" relExpression)*
                          ;

relExpression           ::= addExpression
                            (
                             (">" | ">=" | "==" | "!=" | "<" | "<=")
                             addExpression
                            )?
                          ;

addExpression           ::= mulExpression
                            (("+" | "-") mulExpression)*
                          ;

mulExpression           ::= negExpression
                            (("*" | "/") negExpression)*
                          ;

negExpression           ::= ("-" | "!")? priExpression
              ;

priExpression           ::=   <INTEGER>
                            |
                              "(" orExpression ")"
                            |
                              <ID> ("(" (arguments)? ")")?
                            ;

arguments               ::= andExpression ("," andExpression)*
```

5.2 The Semantics of MC

5.2.1 Variable Declaration

A declaration

```
"int" <ID> ";"
```

defines an identifier (variable) of type integer. There are no boolean variables.

5.2.2 The Assignment Statement

In the assignment statement

```
<ID> "=" addExpression ";"
```

the value of `addExpression` is computed and assigned th the variable `<ID>`.

5.2.3 The While Statement

In the `while` statement

```
"while" "(" orExpression ")" statement
```

the statement is executed repeatedly so long as the value of the expression remains `true`. The test takes place before execution of the statement.

5.2.4 The If Statement

In the `if` statement

```
"if" "(" orExpression ")" statement ("else" statement)?
```

the expression is evaluated and if it is `true`, the first substatement is executed. The second substatement (if present) is evaluated if the expression is `false`. The `else` ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

5.2.5 The Return Statement

A function returns to its caller by means of the `return` statement

```
"return" andExpression ";"
```

The value of the expression is returned to the caller of the function.

5.2.6 The Print Statement

The `prints`

```
"print" "(" andExpression ")" ";"
```

outputs the value of the expression.

5.2.7 The Block Statement

So that several statements can be used where one is expected, the block statement is provided.

```
"" (varDeclaration)* (statement)* ""
```

If any of the identifiers in the declarations were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force

5.2.8 Expressions

The value of expressions is computed left to right. The negation and the not operator have priority 1, multiplication and division priority 2, addition and subtraction priority 3, relational operators priority 4, logical and priority 5 and logical or priority 6. 1 is the highest priority, 6 the lowest.

5.2.9 Comments

A MC program may contain comments. They are defined like in the JAVA language.

5.3 Example

Following MC program may be used to compute the factorial function:

```
int f(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * f(n-1);
  }
}

print(f(7));
```

Chapter 6

An interpreter and a Compiler for MC

6.1 The Scanner and the Parser

Use a scanner/parser generator tool to generate a first naked parser for MC. Note that the grammar may not be suitable for the generator . We use JAVACC here. A JAVACC specification may look like:

```
PARSER_BEGIN(NakedParser)

public class NakedParser {}

PARSER_END(NakedParser)

SKIP :
{
  " " | "\t" | "\n" | "\r" | "\f"
}

SKIP :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT> SKIP :
{
  < "\n" | "\r" | "\r\n" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT> SKIP :
{
  < "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_MULTI_LINE_COMMENT> SKIP :
```

```
{
  < ~[] >
}

TOKEN :
{
  < ELSE: "else" >
| < IF: "if" >
| < INT: "int" >
| < WHILE: "while" >
| < RETURN: "return" >
| < PRINT: "print" >
}

TOKEN :
{
  < INTEGER: ["0"-"9"] (["0"-"9"])* >
}

TOKEN :
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER: ["a"-"z","A"-"Z","_"]>
|
  < #DIGIT: [ "0"-"9"] >
}

void program() :
{}
{
  (LOOKAHEAD(3) variableDeclaration())*
  (LOOKAHEAD(3) functionDeclaration())*
  (statement())*
}

void variableDeclaration() :
{}
{
  "int" <IDENTIFIER> ";"
}
void functionDeclaration() :
{}
{
  "int" <IDENTIFIER> "(" parameterDeclarations() ")" block()
}

void parameterDeclarations() :
{}
{
```

```
  ("int" <IDENTIFIER> ("," "int" <IDENTIFIER>)*)?
}

void block() :
{}
{
  "{" (variableDeclaration())* (statement())* "}"
}

void statement() :
{}
{
  <IDENTIFIER> "=" orExpression() ";"
|
  whileStatement()
|
  ifStatement()
|
  "return" orExpression() ";"
|
  "print" "(" orExpression() ")" ";"
|
  block()
}

void whileStatement() :
{}
{
  "while" "(" orExpression() ")"
  statement()
}

void ifStatement() :
{}
{
  "if" "(" orExpression() ")"
  statement()
  ("else" statement())?
}

void orExpression() :
{}
{
  andExpression() ("||" andExpression())*
}

void andExpression() :
{}
{
  relExpression() ("&&" relExpression())*
```

```
}

void relExpression() :
{}
{
  addExpression()
  (">" | ">=" | "==" | "!=" | "<" | "<=")
  addExpression())?
}

void addExpression() :
{}
{
  mulExpression() (("+" | "-") mulExpression())*
}

void mulExpression() :
{}
{
  negExpression() (("*" | "/") negExpression())*
}

void negExpression() :
{}
{
  ("-" | "!")? priExpression()
}

void priExpression() :
{}
{
  <INTEGER>
|
  "(" orExpression() ")"
|
  LOOKAHEAD(2) <IDENTIFIER>
|
  LOOKAHEAD(2) <IDENTIFIER> "(" (arguments())? ")"
}

void arguments() :
{}
{
  orExpression() ("," orExpression())*
}
```

Note that we had to add some LOOKAHEAD instructions since variable and function declarations have the same prefix.

The specification above is still producing a JAVACC warning:

```
Choice conflict in [...] construct at line 206, column 3.
Expansion nested within construct and expansion following construct
have common prefixes, one of which is: "else"
Consider using a lookahead of 2 or more for nested expansion.
```

The warding is caused by the fact that the parser does not know to which `if` an `else` is belonging to. If we ignore the warning, the conflict is resolved by connecting an `else` with the last encountered `else`-less `if`. This is exactly what we required in the semantics of MC (see section 5.2.4).

6.2 The Abstract Syntax Tree (AST)

It is possible to write an entire compiler that fits within the semantic actions of a JAVACC parser. However, such a compiler if difficult to read and maintain.

To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking, interpreting, translation to machine code).

One way to do it is to produce an *abstract syntax tree*. An abstract syntax tree makes a clean interface between the parser and the later phases of a compiler. The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

The abstract syntax tree is different from the parse tree. It just contains all necessary informations for the later phases of compilation but not more. An abstract syntax contains ambiguities and is not suitable for parsing.

In JAVA the AST can be implemented using the abstract class `ASTNode`:

```
public abstract class ASTNode {

  public String position = "";

  public abstract Object accept(ASTVisitor visitor, Object o);
}
```

The attribute `position` remembers the line number of the corresponding source code. The method `accept` enables the separation of of algorithms from the AST upon which they operate.

The root node of the AST is the `ASTProgram` node.

All other nodes are listed here:

ASTDeclatarion	ASTStatement	ASTExpression
ASTVarDeclaration	ASTAssignStatement	ASTOr
ASTParDeclaration	ASTWhileStatement	ASTAnd
ASTFunDeclaration	ASTIfStatement	ASTLt
ASTDeclatarion	ASTReturnStatement	ASTLe
	ASTPritnStatement	ASTNe
	ASTBlock	ASTEq
		ASTGt
		ASTGe
		ASTAdd
		ASTSub
		ASTMul
		ASTDiv
		ASTNeg
		ASTNot
		ASTVariable
		ASTConstant
		ASTCall

The classes `ASTDeclatarion`, `ASTAssignStatement` and `ASTExpression` are abstract classes. `ASTExpression` defines an attribute `priority` corresponding to the priority of the operator.

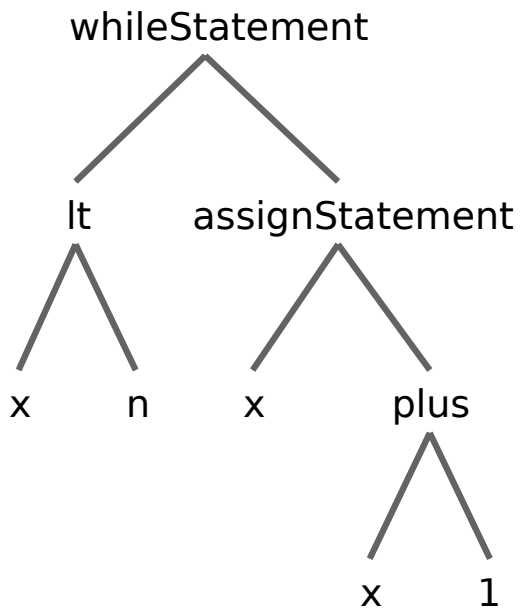
All AST node are similar, they just contain attributes for each of their subtrees, e.g.

```
public class ASTDiv extends ASTExpression {

  public final ASTExpression op1;
  public final ASTExpression op2;

  public ASTDiv(ASTExpression op1, ASTExpression op2) {
    super(2);
    this.op1 = op1;
    this.op2 = op2;
  }
}
```

Example 6.31 [AST] The instruction `while (x < n) { x = x + 1; }` will be translated into following AST:



6.2.1 The Visitor

To implement the visitor pattern, we need to add the `accept()` method to the nodes of the AST:

```
public class ASTDiv extends ASTExpression {

  public final ASTExpression op1;
  public final ASTExpression op2;

  public ASTDiv(ASTExpression op1, ASTExpression op2) {
    super(2);
    this.op1 = op1;
    this.op2 = op2;
  }

  @Override
  public Object accept(ASTVisitor visitor, Object o) {
    return visitor.visitDiv(this, o);
  }
}
```

The `accept` method gives access to the algorithm operating on the node (`visit`).

The abstract class `ASTVisitor` contains the virtual methods giving access to all nodes of the AST:

```
public abstract class ASTVisitor {

  public abstract Object visitAdd(ASTAdd add, Object o);
```

```
public abstract Object visitAnd(ASTAnd and, Object o);

public abstract Object visitAssignStatement(
    ASTAssignStatement assignStatement, Object o);

public abstract Object visitBlock(ASTBlockStatement block, Object o);

public abstract Object visitCall(ASTCall call, Object o);

public abstract Object visitDiv(ASTDiv div, Object o);

public abstract Object visitEq(ASTEq eq, Object o);

public abstract Object visitFunDeclaration(
    ASTFunDeclaration funDeclaration, Object o);

public abstract Object visitGe(ASTGe ge, Object o);

public abstract Object visitGt(ASTGt gt, Object o);

public abstract Object visitIfStatement(ASTIfStatement ifStatement,
    Object o);

public abstract Object visitIntegerConstant(
    ASTIntegerConstant integerConstant, Object o);

public abstract Object visitLe(ASTLe le, Object o);

public abstract Object visitLt(ASTLt lt, Object o);

public abstract Object visitMul(ASTMul mul, Object o);

public abstract Object visitNe(ASTNe ne, Object o);

public abstract Object visitNeg(ASTNeg neg, Object o);

public abstract Object visitNot(ASTNot not, Object o);

public abstract Object visitOr(ASTOr or, Object o);

public abstract Object visitParDeclaration(
    ASTParDeclaration parDeclaration, Object o);

public abstract Object visitPrintStatement(
    ASTPrintStatement printStatement, Object o);

public abstract Object visitProgram(ASTProgram program, Object o);

public abstract Object visitReturnStatement(
    ASTReturnStatement returnStatement, Object o);
```

```
    public abstract Object visitSub(ASTSub sub, Object o);

    public abstract Object visitVarDeclaration(
        ASTVarDeclaration varDeclaration, Object o);

    public abstract Object visitVariable(ASTVariable variable, Object o);

    public abstract Object visitWhileStatement(
        ASTWhileStatement whileStatement, Object o);

}
```

All `visit` methods return a pointer to the corresponding node of the AST.

6.2.2 The Parser

We have to modify the parser, such that it can generate an AST. We show here the necessary modification to generate an `ASTAdd` node:

```
ASTExpression addExpression() :
{
  ASTExpression result = null;
  ASTExpression e = null;
  Token t;
}
{
  result = mulExpression()
  (
    t = <ADD> e = mulExpression()
    {
      result = new ASTAdd(result, e);
      result.setPosition(t);
    }
  |
    t = <SUB> e = mulExpression()
    {
      result = new ASTSub(result, e);
      result.setPosition(t);
    }
  )*
  {
    return result;
  }
}
```

6.2.3 The Identity Transformation

It is highly recommended to write an algorithm that reconstructs (up to some indentations) the source code from the AST thus insuring that nothing got forgotten.

This is done With the *identity transformation*

Here some examples:

```
@Override
public Object visitAdd(ASTAdd add, Object indent) {
  return binaryOperator("+", add.priority, add.op1, add.op2,
      (String) indent);
}
```

The method `visitAdd` receives an `ASTAdd` node `add` to process and an indentation string `indent`. The method `binaryOperator` will output the expression according to the priorities of the subtrees.

```
private String binaryOperator(String lexem, int priority,
   ASTExpression op1, ASTExpression op2, String indent) {
  if (op1.priority > priority && op2.priority > priority) {
    return "(" + op1.accept(this, indent) + ")    " + lexem + " ("
        + op2.accept(this, indent) + ")";
  } else if (op1.priority > priority) {
    return "(" + op1.accept(this, indent) + ")" + lexem
        + op2.accept(this, indent);
  } else if (op2.priority > priority) {
    return op1.accept(this, indent) + " " + lexem + " ("
        + op2.accept(this, indent) + ")";
  } else {
    return op1.accept(this, indent) + " " + lexem + " "
        + op2.accept(this, indent);
  }
}
```

6.3 Semantics

The semantic analysis is separated into three parts

1. Variable declarations and variable scope.
2. Type checking
3. Detection of unreachable code

6.3.1 Variable Scope

The scope analysis will detect all variables in a MC program and assign them virtual addresses. Furthermore, the algorithm will detect all variables that have not been declared before usage.

The Symbol Table

The main problem of scope analysis is that the same variable name may have different meanings. Let consider following MC program:

```
int x;
x = 1;
if (x == 1) {
  int x;
  x = 2;
  print(x);
}
print(x);
```

The variable `x` is declared twice, once outside and once inside of the `if` statement. Thus the first `print` statement will output `2` and the second `print` statement will output `1`. Semantically, we are dealing here with two different variables. Thus the first `x` and the second `x` will be assigned two different virtual addresses[1].

We have to build a structure called *symbol table*. A symbol table is a mapping of identifiers to their types, addresses, values, etc. Symbol tables are usually implemented using hash tables which are very efficient. Note that symbol tables must support entering a new identifier in the begin of the scope and deletion at the end of the scope, thus recovering the preceding identifier with the same name.

We use following JAVA classes:

```
class Binder{

  int value;
  String prevtop;
  Binder tail;

  Binder(int value, String prevtop, Binder tail) {
    this.value = value;
    this.prevtop = prevtop;
    this.tail = tail;
  }

public class Table<T> {

  private final Dictionary<String, Binder> table =
    new Hashtable<String, Binder>();

  /** the last symbol defined until the last beginScope */
  private String top = null;

  /** remembers all last top symbols */
```

[1] If MC would also support the type `float`, it would be allowed to declare the second `x` to be a `float` variable

```
  private Binder marks = null;

  /**
   * Gets the object associated with the specified symbol in the Table.
   */
  public T resolve(String name) {
    Binder e = table.get(name);
    if (e == null)
      return null;
    else
      return e.value;
  }

  /**
   * Puts the specified value into the Table, bound to the specified Symbol.
   */
  public void define(String name, int value) {
    table.put(key, new Binder(value, top, table.get(key)));
    top = key;
  }

  /**
   * Remembers the current state of the Table.
   */
  public void beginScope() {
    marks = new Binder(null, top, marks);
    top = null;
  }

  /**
   * Restores the table to what it was at the most recent beginScope that has
   * not already been ended.
   */
  public void endScope() {
    while (top != null) {
      Binder e = table.get(top);
      if (e.tail != null)
        table.put(top, e.tail);
      else
        table.remove(top);
      top = e.prevtop;
    }
    top = marks.prevtop;
    marks = marks.tail;
  }
}
```

Associating a symbol x with a value is called binding. When the binding $x \rightarrow b$ is entered into the table (`table.define(x,b)`), `x` is hashed and a `Binder` object $x \rightarrow b$ is placed at the head of a linked list for that hash value. If the table already contained a binding

$y \rightarrow b'$, that would still be in the list, hidden by $y \rightarrow b$. This is important because it will support the implementation of *undo*.

There must also be an auxiliary stack, showing in what order the identifiers were pushed into the symbol table. When $y \rightarrow b$ is entered, then x is pushed onto this stack. A `beginScope()` operation pushed a special marker `null` onto the stack. Then to implement `endScope()`, symbols are popped off the stack down and including the topmost marker. As each symbol is popped, the head binding in the linked list is removed.

The auxiliary stack is integrated into the `Binder` by having a global variable `top` showing the most recent symbol bound to the table. Then pushing is accomplished by copying `top` into the `prevtop` attribute of the `Binder`. Thus this stack is threaded through the binders.

Implementation

To avoid subsequent use of symbol tables, all relevant informations for variables and functions are stored into the corresponding AST node as an `Entry` object.

When a variable declaration is encountered, an `Entry` object is creating containing the address of the variable and this `Entry` object is also entered into the table.

```
public Object visitVarDeclaration(ASTVarDeclaration varDeclaration,
    Object o) {
  VarEntry entry = new VarEntry(varDeclaration.name);
  entry.address = address++;
  varDeclaration.entry = entry;
  varTable.define(varDeclaration.name, entry);
  return null;
}
```

When a variable is encountered, we look at the table for the corresponding `Entry` object and store it into the AST node. If no `Entry` object can be found, the variable has not been declared and we throw an exception.

```
public Object visitVariable(ASTVariable variable, Object o) {
  VarEntry entry = varTable.resolve(variable.name);
  if (entry == null) {
    throw new RuntimeException("variable " + variable.name + " at "
        + variable.position + " has not been declared");
  }
  variable.entry = entry;
  return null;
}
```

We do also the same with functions. But we have to take care of following problem: since recursion can occur, a function may be used before it has been declared. Thus we are forced to first visit the headers of the function declarations before we can visit their bodies.

6.3.2 Type Checking

Type checking only involves declarations and expressions.

Type checking is easy. When a variable declaration is encountered, the type information is written into the `Entry` of the variable.

When an expression is encountered, the type of its operands computed and the results are compared. If the language supports many types, a lot of code may be necessary is the types of the operands are compatible.

```
public Object visitAdd(ASTAdd add, Object o) {
  Type t1 = (Type) add.op1.accept(this, null);
  Type t2 = (Type) add.op2.accept(this, null);
  if (!Type.checkType(IntType.intType(), t1, t2)) {
    throw new RuntimeException("incompatible types near "
        + add.position);
  }
  return t1;
}
```

6.3.3 Computing Reachable Code

Any statement following a `return` statement cannot be executed and is thus useless. To detect unreachable code, we need to test if a statement contains a `return` statement and mark the following statements as unreachable.

```
public Object visitReturnStatement(
    ASTReturnStatement returnStatement, Object o) {
  boolean reachable = (Boolean) o;
  if (!reachable) {
    m.error("unreachable statement at " + returnStatement.position
        + "\n" + returnStatement);
  }
  // a return statement always contains a return statement
  return true;
}
```

The main job is the analysis of block statements:

```
public Object visitBlock(ASTBlockStatement block, Object o) {
  boolean reachable = (Boolean) o;
  if (!reachable) {
    m.error("unreachable statement at " + block.position + "\n"
        + block);
  }
  reachable = true;
  boolean hasReturn = false;
  for (int i = 0; i < block.statements.size(); i++) {
    boolean hr = (Boolean) block.statements.elementAt(i).accept(this,
```

```
          reachable);
      hasReturn = hr || hasReturn;
      if (hasReturn) {
        reachable = false;
      }
    }
    return hasReturn;
  }
```

6.4 The Interpreter

The basic idea behind executing instructions is called the *fetch-decode-execute* cycle. First, we load an instruction from code memory. Then, we decode the instruction to find the operation and operands. Finally, we execute the operation. Rinse and repeat. Ultimately the interpreter runs out of instructions in the main program, or it executes a halt instruction.

Our interpreter will executes programs by constructing an AST from the source code and walking the tree.

6.4.1 Interpreter Memory

Interpreters have one global memory space but multiple function spaces (assuming the language we are interpreting has functions). Each function call creates a new space to hold parameters and local variables.

For simplicity, we can normalise all these spaces by treating them as dictionaries. Even fields are really just variables stored within an instance's memory space. To store a value into a space, we map a name to that value. A memory space is the run-time analog of a scope from static analysis. Thus we will use the unique virtual address as variable name.

The interpreter keeps track of the function spaces by pushing them onto the stack. Upon returning from a function, the interpreter pops the top space off the stack. This way, parameters and local variables pop in and out of existence as we need them.

Interpreters also need global memory for the instructions to be executed. We can use the AST itself as code memory.

The global memory can be implemented as follows:

```
public class MemorySpace<Value> {

  protected MemorySpace<Value> parent = null;

  protected final Map<Integer, Value> members =
    new HashMap<Integer, Value>();

  public Value get(Integer i) {
    return members.get(i);
  }
```

```
  public void put(Integer i, Value value) {
    members.put(i, value);
  }
}
```

The function spaces stack can be implemented as a linked list. As soon as a function is called, a corresponding function space is created and linked to the former top memory space.

```
public class FunctionSpace<Value> extends MemorySpace<Value> {

  public FunctionSpace(MemorySpace<Value> parent) {
    this.parent = parent;
  }

  @Override
  public Value get(Integer i) {
    if (members.get(i) != null) {
      return members.get(i);
    } else if (parent != null) {
      return parent.get(i);
    } else {
      return null;
    }
  }
}
```

6.4.2 The Interpreter

Operators

Computing the value of a binary operation is quite simple: we compute the value of the operands and return the result according to the operator.

```
public Object visitAdd(ASTAdd add, Object o) {
  int r = (Integer) add.op1.accept(this, o)
      + (Integer) add.op2.accept(this, o);
  return r;
}
```

Variables

We just keep track of the actual top memory space in a variable `memory`. The value of `memory` will change each time we call a function.

```
public Object visitVariable(ASTVariable variable, Object o) {
  int r = memory.get(variable.entry.address);
  return r;
}
```

Statements

```
public Object visitAssignStatement(
    ASTAssignStatement assignStatement, Object o) {
  int r = (Integer) assignStatement.expression.accept(this, o);
  memory.put(assignStatement.variable.entry.address, r);
  return r;
}

public Object visitPrintStatement(ASTPrintStatement printStatement,
    Object o) {
  int r = (Integer) printStatement.expression.accept(this, o);
  return r;
}

public Object visitReturnStatement(
    ASTReturnStatement returnStatement, Object o) {
  int r = (Integer) returnStatement.expression.accept(this, o);
  return r;
}

public Object visitIfStatement(ASTIfStatement ifStatement, Object o) {
  int r = 0;
  if ((Integer) ifStatement.expression.accept(this, o) != 0) {
    r = (Integer) ifStatement.thenStatement.accept(this, o);
  } else if (ifStatement.elseStatement != null) {
      r = (Integer) ifStatement.elseStatement.accept(this, o);
  }
}

public Object visitWhileStatement(ASTWhileStatement whileStatement,
    Object o) {
  int r = 0;
  while ((Integer) whileStatement.expression.accept(this, o) != 0) {
    r = (Integer) whileStatement.statement.accept(this, o);
  }
  return r;
}
```

6.5 The Compiler

Our compiler is generating directly JASMIN JAVA assembler code (see section B).

6.5.1 The JAVA Virtual Machine

Stack

A JAVA virtual machine stack stores frames. A JAVA virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return.

The following exceptional condition is associated with JAVA virtual machine stacks: If the computation requires a larger JAVA virtual machine stack than is permitted, the JAVA virtual machine throws a `StackOverflowError`.

Thus, when generation code, the expected size of the stack must be computed.

We will use the global integer variable `actStack` for the actual stack size and `maxStack` for the maximal stack size so far. We also need the method `as` that adjusts the stack size when a bytecode instruction is executed.

```
public void as(int i) {
  actStack += i;
  if (actStack > maxStack)
    maxStack = actStack;
}
```

Instructions that push elements onto the stack will call `as` with a positive argument, instructions that pop elements from the stack will call `as` with a negative argument.

Frame

A frame is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception). Frames are allocated from the JAVA virtual machine stack of the thread creating the frame. Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame. Thus the size of the frame data structure depends only on the implementation of the JAVA virtual machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the current frame, and its method is known as the current method. The class in which the current method is defined is the current class. Operations on local variables and the operand stack are typically with reference to the current frame.

A frame ceases to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. On method return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Thus we will need to count the number of local frame variables when generating the bytecode instructions.

6.5.2 Expressions

Arithmetic Expressions

In a first step, the code for the operands will be generated (`accept`). Finally, we generate code for the operator. At evaluation, the operands will push their results onto the stack, the operator will pop them and push the result onto the stack.

```
public Object visitAdd(ASTAdd add, Object o) {
  add.op1.accept(this, null);
  add.op2.accept(this, null);
  emt("iadd");
  as(-1); // pop 2 operands and push result
  return null;
}
```

Boolean Expressions

The virtual machine does not have boolean variables. thus the value `1` will be used for `true` and the value `0` will be used for `false`.

The method `label()` generates a new (unique) label at each call.

```
public Object visitAnd(ASTAnd and, Object o) {
  String lEnd = label();
  String lFalse = label();
  and.op1.accept(this, null);
  emt("ifeq " + lfalse);
  as(-1);
  and.op2.accept(this, null);
  emt("ifeq " + lfalse);
  as(-1);
  emt("iconst_1"); // true
  as(1);
  emt("goto " + lEnd);
  eml(lFalse);
  emt("iconst_0"); // false
  as(1);
  eml(lEnd);
  return null;
}

  public Object visitEq(ASTEq eq, Object o) {
  String lTrue = label();
  String lEnd = label();
  eq.op1.accept(this, o);
  eq.op2.accept(this, o);
  emt("isub");
  as(-1);
  emt("ifeq " + lTrue);
```

```
    as(-1);
    emt("iconst_0"); // false
    as(1);
    emt("goto " + lEnd);
    eml(lTrue);
    emt("iconst_1"); // true
    as(1);
    eml(lEnd);
    return null;
  }
```

6.5.3 Statements

Assignment Statement

The value of the expression is popped from the stack and stored into memory. Here we have to take care whether the variable is global or local, since there are two different kinds of addressing.

```
  public Object visitAssignStatement(
      ASTAssignStatement assignStatement, Object o) {
    assignStatement.expression.accept(this, null);
    ASTVariable variable = assignStatement.variable;
    if (variable.entry.global) {
      emt("putstatic utg/mcc/bytecode/A/" + variable.entry.name + " "
          + variable.entry.type.signature());
    } else {
      emt("istore " + variable.entry.address);
    }
    as(-1);
    return false;
  }
```

Print Statement

The print statement simply calls the standard `Java println()` method. Since this method is not static, a pointer to the `PrintStream` object `System.out` must be pushed onto the stack before the `println()` can be executed.

```
  public Object visitPrintStatement(ASTPrintStatement printStatement,
      Object o) {
    emt("getstatic java/lang/System/out Ljava/io/PrintStream;");
    as(1);
    printStatement.expression.accept(this, o);
    emt("invokevirtual java/io/PrintStream/println(I)V");
    as(-1);
    return false;
  }
```

Return Statement

The JAVA virtual machine does not accept unreachable code. Thus we will have a single `ireturn` statement at the end of each method code. So we just jump to the `ireturn` statement at the end of the function code.

```
public Object visitReturnStatement(
    ASTReturnStatement returnStatement, Object o) {
  returnStatement.expression.accept(this, o);
  // now the return value is on top of the stack
  as(-1);
  emt("goto " + returnLabel);
  return true;
}
```

If Statement

We generate first the code for expression (test). At evaluation, the top of the stack will be 1 (expression is true) or 0 (expression is false). We just jump at the right place.

```
public Object visitIfStatement(ASTIfStatement ifStatement, Object o) {
  String lElse = label();
  String lEnd = label();
  ifStatement.expression.accept(this, null);
  emt("ifeq " + lElse);
  as(-1);
  ifStatement.thenStatement.accept(this, null);
  emt("goto " + lEnd);
  eml(lelse);
  if (ifStatement.elseStatement != null) {
    ifStatement.elseStatement.accept(this, null);
  }
  eml(lEnd);
  return null;
}
```

While Statement

We generate first the code for expression (test). At evaluation, the top of the stack will be 1 (expression is true) or 0 (expression is false). We just jump at the right place. At the end of the statement part, we jump to the beginning of the `while` loop and test again.

```
public Object visitWhileStatement(ASTWhileStatement whileStatement,
    Object o) {
  String lStart = label();
  String lEnd = label();
  eml(lStart);
  whileStatement.expression.accept(this, null);
```

```
    emt("ifeq " + lEnd);
    as(-1);
    whileStatement.statement.accept(this, null);
    emt("goto " + lStart);
    eml(lend);
    return null;
  }
```

There is a more efficient implementation of the `while` statement (see section B.3.9).

6.5.4 Variables

Just notice that global and local variable are loaded in a different way.

```
  public Object visitVariable(ASTVariable variable, Object o) {
    if (variable.entry.global) {
      emt("getstatic utg/mcc/bytecode/A/" + variable.entry.name + " "
          + variable.entry.type.signature());
    } else {
      emt("iload " + variable.entry.address);
    }
    as(1);
    return null;
  }
```

6.5.5 Functions

Function Call

At function call, we first need to push all actual parameters values (expression) onto the stack. In the next step, we just call the function. Do not forget the class name in the call. The function will be executed and push its result onto the stack.

```
  public Object visitCall(ASTCall call, Object o) {
    for (int i = 0; i < call.parameters.size(); i++) {
      call.parameters.elementAt(i).accept(this, null);
    }
    emt("invokestatic utg/mcc/bytecode/A/" + call.entry.name
        + call.entry.type.signature());
    as(1 - call.entry.formals);
    return null;
  }
```

Function Declaration

A function (method) declaration needs a special header and a special footer containing informations about the maximal stack size and the number of local variables (parameters included) of the function.

Take care, if the stack size is too small, the JAVA verifier may tell you that the stack is too large! If the the stack is too large, the verifier will not generate any error.

To avoid unreachable code, the function will have a single labelled `ireturn` statement as last instruction.

```
public Object visitFunDeclaration(ASTFunDeclaration funDeclaration,
    Object o) {
  actualFunction = funDeclaration;
  returnLabel = label();
  actStack = 0;
  maxStack = 0;
  em(".method public static " + funDeclaration.name
      + funDeclaration.entry.type.signature());
  funDeclaration.block.accept(this, null);
  eml(returnLabel);
  emt("ireturn");
  as(-1);
  em(".limit locals " + funDeclaration.entry.addresses);
  em(".limit stack " + maxStack);
  em(".end method");
  em("");
  actualFunction = null;
  return null;
}
```

6.5.6 The MC Program

The code of the JAVA class generated for the MC program needs also a special header containing informations concerning the class name and the super class name.

We need to generate code for each global variable, and for the methods. The code of the main MC program (statements) is embedded into the method `exec`. Please don't create MC programs containing an `exec` method.

```
public Object visitProgram(ASTProgram program, Object o) {
  // class header
  em(".bytecode 50.0");
  em(".source " + source);
  em(".class public utg/mcc/bytecode/A");
  em(".super java/lang/Object");
  em("");
  // global variables
  for (int i = 0; i < program.varDeclarations.size(); i++) {
    program.varDeclarations.elementAt(i).accept(this, null);
  }
  em("");
  // class constructor
  em(".method public <init>()V");
  emt(".limit locals 1");
```

```
    emt(".limit stack 1");
    emt("aload_0");
    emt("invokespecial java/lang/Object/<init>()V");
    emt("return");
    em(".end method");
    em("");
    // functions
    for (int i = 0; i < program.funDeclarations.size(); i++) {
      program.funDeclarations.elementAt(i).accept(this, null);
    }
    // executer header
    em(".method public static exec()V");
    actStack = 0;
    maxStack = 0;
    // statements of the mc program
    for (int i = 0; i < program.statements.size(); i++) {
      program.statements.elementAt(i).accept(this, null);
    }
    // executer footer
    emt("return");
    emt(".limit locals " + program.entry.locals);
    emt(".limit stack " + maxStack);
    em(".end method");
    return null;
  }
```

6.5.7 Execution of the MC Program

Using the JASMIN assembler and the JAVA reflection API, it is possible to execute the generated code immediately as follows:

```
public class Exec {

  public static void exec() {
    // create the java class using jasmin
    String jasmin_args[] = new String[3];
    jasmin_args[0] = new String("-d");
    jasmin_args[1] = new String("../bin");
    jasmin_args[2] = new String("A.j");
    jasmin.Main.main(jasmin_args);
    // run the generated class
    try {
      Class<?> a = Exec.class.getClassLoader().loadClass(
          "utg.mcc.bytecode.A");
      Method m = a.getMethod("exec", (Class<?>[]) null);
      m.invoke(null, (Object[]) null);
    } catch (Exception e) {
      e.printStackTrace();
    }
```

```
  }
}
```

Another way to execute the MC program is first to generate the class file with JASMIN as shown above. And to use following class to execute the class:

```
public class DoIt {

  public static void main(String [] args) {
    utg.mcc.bytecode.Exec.exec();
  }
}
```

6.5.8 Boolean Expressions Variant 2

There is a more efficient way to generate code for `if` statements, `while` statements and boolean expressions. It is not necessary to handle boolean expressions like integer expressions. We just need to know the labels `lTrue` and `lFalse` where where need to jump if the the expression is `true` or `false`.

The visitors of boolean expressions receive a true/false pair `(lTrue, lFalse)`; as a parameter of the `visit` method.

```
public Object visitOr(ASTOr or, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  String lNext = label();
  or.op1.accept(this, new TrueFalse(tf.t, lNext));
  eml(lNext);
  or.op2.accept(this, tf);
  return null;
}

public Object visitAnd(ASTAnd and, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  String lNext = label();
  and.op1.accept(this, new TrueFalse(lNext, tf.f));
  eml(lNext);
  and.op2.accept(this, tf);
  return null;
}

public Object visitNot(ASTNot not, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  not.op.accept(this, new TrueFalse(tf.t, tf.t));
  return null;
}
```

The comparison operators generate the jumps to the corresponding labels.

```
public Object visitEq(ASTEq eq, Object o) {
  TrueFalse tf = (o != null ? (TrueFalse) o : null);
  eq.op1.accept(this, null);
  eq.op2.accept(this, null);
  emt("isub");
  as(-1);
  emt("ifeq " + tf.t);
  as(-1);
  emt("goto " + tf.f);
  return null;
}
```

The most labels are generated inside the `if` and the `while` statement.

```
  public Object visitIfStatement(ASTIfStatement ifStatement, Object o) {
  String lTrue = label();
  String lFalse = label();
  String lEnd = label();
  TrueFalse tf = new TrueFalse(lTrue, lFalse);
  ifStatement.expression.accept(this, tf);
  eml(lTrue);
  ifStatement.thenStatement.accept(this, null);
  emt("goto " + lEnd);
  eml(lFalse);
  if (ifStatement.elseStatement != null) {
    ifStatement.elseStatement.accept(this, null);
  }
  eml(lEnd);
  return null;
}

public Object visitWhileStatement(ASTWhileStatement whileStatement,
    Object o) {
  String lStart = label();
  String lEnd = label();
  String lTrue = label();
  eml(lStart);
  TrueFalse tf = new TrueFalse(lTrue, lEnd);
  whileStatement.expression.accept(this, tf);
  eml(lTrue);
  whileStatement.statement.accept(this, null);
  emt("goto " + lStart);
  eml(lEnd);
  return null;
}
```

6.5.9 Download

The complete source code of the MC compiler/interpreter can be downloaded from `prog/compilers.j`

Appendix A

JAVACC

In practise, the scanning and parsing phases of a compiler are handled by code that is generated by a parser generator. One parser generator for Java is called JAVACC. Here's a tutorial[1].

A.1 Introduction

JAVACC is a lexer and parser generator for LL(k) grammars. You specify a language's lexical and syntactic description in a `.jj` file, then run javacc on the `.jj` file. You will get seven java files as output, including a lexer and a parser.

This page helps you get started using JAVACC. You still need to read the on line documentation to do anything really useful with the tool though.

We'll look at three things you can do with JAVACC

1. Do a simple syntax check only
2. Make an actual interpreter
3. Generate code

A.2 Getting Started

Make sure you have (or get) *at least* version 4.0.

The home page for JAVACC is `https://javacc.dev.java.net/`.

Download it. Unzip it. You may want to make the `javacc` script accessible from your path.

A.3 A First Example: Syntax Checking

Let's start with the language of integer expressions with addition and multiplication, with addition having lower precedence. The typical LL(1) grammar for this is:

[1] From `http://www.cs.lmu.edu/\char126\relaxray/notes/javacc/`

```
Microsyntax
  SKIP -> [\x0d\x0a\x20\x09]
  NUM -> [0-9]+
  TOKEN -> [+*()] | NUM
Macrosyntax
  E -> T ("+" T)*
  T -> F ("*" F)*
  F -> NUM | "(" E ")"
```

Make a file called `Exp1.jj` like so:

JAVACC-Specification `SyntaxChecker`

```
1   PARSER_BEGIN(SyntaxChecker)
2
3   public class SyntaxChecker {
4     public static void main(String[] args) {
5       try {
6         new SyntaxChecker(new java.io.StringReader(args[0])).S();
7         System.out.println("Syntax is okay");
8       } catch (Throwable e) {
9         System.out.println("Syntax check failed: " +
10          e.getMessage());
11      }
12    }
13  }
14
15  PARSER_END(SyntaxChecker)
16
17  SKIP:  { " " | "\t" | "\n" | "\r"          }
18  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
19
20  void S(): {} { E() <EOF>       }
21  void E(): {} { T() ("+" T())*    }
22  void T(): {} { F() ("*" F())*    }
23  void F(): {} { <NUM> | "(" E() ")" }
```

This is pretty much a minimal example. You must define a parser class between the markers `PARSER_BEGIN` and `PARSER_END`, and you must specify tokens in a `TOKEN`: clause, and parsing methods must be defined, one for each non-terminal in the grammar. The parsing functions look rather like the EBNF for a grammar: you'll just notice non-terminals look like function calls, tokens look like themselves, and you have the usual EBNF metasymbols like "|" (choice), "*" (Kleene star), and "+" (Kleene plus).

To process the JJ file, invoke

```
javacc Exp1.jj
```

and notice seven files are output. The one called `SyntaxChecker.java` should be of some interest; take a look at it! (Yeah, it's pretty ugly, but you didn't write it yourself, right?)

Some example runs:

```
$ javacc Exp1.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Exp1.jj . . .
Parser generated successfully.
$ javac *.java
Note: SyntaxChecker.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java SyntaxChecker "100"
Syntax is okay
$ java SyntaxChecker "8 * 3 * (2 + 2345234)"
Syntax is okay
$ java SyntaxChecker "(((3)) * 12)"
Syntax is okay
$ java SyntaxChecker "this doesn't work"
Syntax check failed: Lexical error at line 1, column 1.
                     Encountered: "t" (116),
 after : ""
$ java SyntaxChecker "(x = 3) + 111"
Syntax check failed: Lexical error at line 1, column 2.
                     Encountered: "x" (120),
 after : ""
$ java SyntaxChecker "1 2"
Syntax check failed: Encountered "2" at line 1, column 3.
Was expecting one of:
  <EOF>
  "+" ...
  "*" ...
```

If you want to get fancy with lexing, you can add comments to the language. For example:

```
SKIP: {
  " "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n"|"\r")>
}
```

For lexical analysis, JAVACC allows other sections besides `TOKEN` and `SKIP`. There are also ways to make "private" tokens, and write complex regular expressions. See the JAVACC documentation for details. Also see the mini-tutorial on the JAVACC site for tips on writing lexer specifications from which JAVACC can generate efficient code.

A.4 Left Recursion

JAVACC cannot parse grammars that have left-recursion. It just can't. Suppose you tried to use the non-LL(k) grammar for the language above:

```
Macrosyntax
  E -> T | E "+" T
  T -> F | T "*" F
  F -> NUM | "(" E ")"
```

If you changed your `.jj` file accordingly and ran JAVACC, you would see this:

```
$ javacc BadLeftRecursion.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file BadLeftRecursion.jj . . .
Error: Line 22, Column 1: Left recursion detected: "T... --> T..."
Error: Line 21, Column 1: Left recursion detected: "E... --> E..."
Detected 2 errors and 0 warnings.
```

You must remove left recursion before writing your grammar rules in JAVACC. The general approach is to replace rules of the form

```
A -> a | Ax
```

with

```
A -> a (x)*
```

A.5 Lookahead

Let's add identifiers and assignment expressions to our language:

```
E -> id ":=" E | T ("+" T)*
T -> F ("*" F)*
F -> NUM | ID | "(" E ")"
```

If you naively write JAVACC for this grammar, JAVACC will say something like

```
$ javacc NotEnoughLookahead.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file NotEnoughLookahead.jj . . .
Warning: Choice conflict involving two expansions at
     line 24, column 16 and line 24, column 32 respectively.
     A common prefix is:
     Consider using a lookahead of 2 for earlier expansion.
Parser generated with 0 errors and 1 warnings.
```

What this means is that when expanding an E and looking at an id, we wouldn't know if that id is starting an assignment or is just a variable, unless we examine not just the id, but also the following token. So the parser needs to look at the next *two* symbols.

Understand that *choice points* can appear at various places within a grammar. They obviously appear at "|" separators, but also in `()*`, `()+` and `()?` constructs too.

JAVACC-Specification `SyntaxChecker1`

```
1  PARSER_BEGIN(SyntaxChecker1)
2
3  public class SyntaxChecker1 {
4    public static void main(String[] args) {
5      try {
6        new SyntaxChecker1(new java.io.StringReader(args[0])).S();
7        System.out.println("Syntax is okay");
8      } catch (Throwable e) {
9        System.out.println("Syntax check failed: " +
10         e.getMessage());
11     }
12   }
13 }
14
15 PARSER_END(SyntaxChecker1)
16
17 SKIP: { " " | "\t" | "\n" | "\r" }
18 TOKEN: {
19   "(" | ")" | "+" | "*" | ":="
20   | <NUM: (["0"-"9"])+> | <ID: (["a"-"z"])+>
21 }
22
23 void S(): {} { E() <EOF>                    }
24 void E(): {} { LOOKAHEAD(2) <ID> ":=" E() | T() ("+" T())* }
25 void T(): {} { F() ("*" F())*                }
26 void F(): {} { <NUM> | <ID> | "(" E() ")"          }
```

Sometimes *no* number of lookahead tokens is sufficient for the parser. Then you'd need to use *syntactic* lookahead. For sophisticated, or bizarre, parsing, sometimes *semantic* lookahead is needed. See the JAVACC Lookahead Mini-Tutorial for details.

A.6 Writing An Interpreter

If the syntax checker above looked like it had a lot of extra markup in its parsing functions, that's because parsers are supposed to *do* stuff while parsing. Parsing functions can take in parameters, return results, and invoke blocks of arbitrary Java code. Here's how we can actually evaluate the expressions, or, as they say, *write an interpreter*.

JAVACC-Specification `Evaluator`

```
1  PARSER_BEGIN(Evaluator)
2
3  public class Evaluator {
4    public static void main(String[] args) throws Exception {
5      int result =
6        new Evaluator(new java.io.StringReader(args[0])).S();
7      System.out.println(result);
8    }
9  }
10
```

```
11  PARSER_END(Evaluator)
12
13  SKIP:  { " " | "\t" | "\n" | "\r"          }
14  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
15
16  int S(): {int sum;}
17  {
18    sum=E() <EOF> {return sum;}
19  }
20
21  int E(): {int sum, x;}
22  {
23    sum=T() ("+" x=T() {sum += x;} )* {return sum;}
24  }
25
26  int T(): {int sum, x;}
27  {
28    sum=F() ("*" x=F() {sum *= x;} )* {return sum;}
29  }
30
31  int F(): {int x; Token n;}
32  {
33    n=<NUM> {return Integer.parseInt(n.image);}
34  |
35    "(" x=E() ")" {return x;}
36  }
```

Some example runs

```
$ javacc Exp2.jj
Java Compiler Compiler Version 4.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Exp2.jj . . .
Parser generated successfully.
$ javac *.java
Note: Evaluator.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$ java Evaluator "34"
34
$ java Evaluator "34 * 2 + ((3 + 11))"
82
```

A.7 Generating an Abstract Syntax Tree

A quick and dirty way to do what most real parsers do, namely generate an abstract syntax tree, is to embed the tree classes in the `.jj` file, like so:

JAVACC-Specification `Parser`

```
1  PARSER_BEGIN(Parser)
2
3  public class Parser {
4    public static void main(String[] args) throws Exception {
5      Exp result =
6        new Parser(new java.io.StringReader(args[0])).S();
7      System.out.println(result);
8    }
9  }
10
11 // Classes defining the Abstract Syntax Tree
12 abstract class Exp {}
13 class Num extends Exp {
14   int value;
15   Num(int v) {
16     value = v;
17   }
18   public String toString() {
19     return value + "";
20   }
21 }
22 class BinaryExp extends Exp {
23   String op;
24   Exp left, right;
25   BinaryExp(String o, Exp l, Exp r) {
26     op = o;
27     left = l;
28     right = r;
29   }
30   public String toString() {
31     return "(" + op + " " + left + " " + right + ")";
32   }
33 }
34
35 PARSER_END(Parser)
36
37 SKIP:  { " " | "\t" | "\n" | "\r"          }
38 TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
39
40 Exp S(): {Exp e;}
41 {
42   e=E() <EOF> {return e;}
43 }
44
45 Exp E(): {Exp e1; Exp e2;}
46 {
47   e1=T() ("+" e2=T() {e1 = new BinaryExp("+", e1, e2);} )*
48   {return e1;}
49 }
50
```

```
51  Exp T(): {Exp e1; Exp e2;}
52  {
53    e1=F() ("*" e2=F() {e1 = new BinaryExp("*", e1, e2);} )*
54    {return e1;}
55  }
56
57  Exp F(): {Exp e; Token n;}
58  {
59    n=<NUM> {return new Num(Integer.parseInt(n.image));}
60  |
61    "(" e=E() ")" {return e;}
62  }
```

The `toString()` method is convenient, though it might trick you into believing we made a code generator that targets LISP. But the tree is really there.

```
$ java Parser 54
54
$ java Parser 54+3
(+ 54 3)
$ java Parser "54 + 3 * 2 + 7"
(+ (+ 54 (* 3 2)) 7)
$ java Parser "54 + 3 * (2 + 7) * 33"
(+ 54 (* (* 3 (+ 2 7)) 33))
```

A.8 Integrating the Parser into a Compiler

In a real compiler, you don't dump a main method into the parser. You'd make a nice function called `parse()` perhaps, and call that from your own code. And you'd define the tree classes in their own files.

Exercise A.1 [JAVACC] *Write a* JAVACC *specification for following grammars. Run* JAVACC *and look carefully at the warning or error message.*

1. *The grammar of example 4.21.*
2. *The grammar of example 4.27.*
3. *The grammar of example 4.28.*

Exercise A.2 [Beautifier]

Develop using JAVACC *a* JAVA *beautifier, i.e. a program that reads a* JAVA *source file and produces a* XHTML *file that contains the Java source with some parts displayed in colour, for example:*

1. *Keywords in magenta.*
2. *Class/type names in purple. You may assume that class/type names start with a capital letter.*

3. *Method names in red. You may assume that method names start with a lower case letter.*
4. *Other identifiers in blue. You may assume that identifier names start with a lower case letter.*
5. *String constants in green.*
6. *Character constants in light green.*

Appendix B

JAVA Byte Code

B.1 The JAVA Virtual Machine

A JAVA *Virtual Machine* (JVM) enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as JAVA *bytecode*. This language conceptually represents the instruction set of a stack-oriented, capability architecture

A JAVA virtual machine instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation. Many instructions have no operands and consist only of an opcode.

Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively

```
do {
 fetch an opcode;
 if (operands) fetch operands;
 execute the action for the opcode;
} while (there is more to do);
```

The operands of the opcode are on the evaluation stack. The result of the opcode may be pushed on the top of the evaluation stack.

Most of the instructions in the JAVA virtual machine instruction set encode type information about the operations they perform. For instance, the `iload` instruction loads the contents of a local variable, which must be an `int`, onto the operand stack. The `fload` instruction does the same with a `float` value. The two instructions may have identical implementations, but have distinct opcodes.

For the majority of typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter: `i` for an int operation, `l` for long, `s` for short, `b` for byte, `c` for char, `f` for float, `d` for double, and a for reference. Some instructions for which the type is unambiguous do not have a type letter in their mnemonic. For instance, arraylength always operates on an object that is an array. Some instructions, such as `goto`, an unconditional control transfer, do not operate on typed operands.

B.2 JASMIN

JASMIN is an assembler for the JVM. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the JVM instruction set. It converts them into binary JAVA class files, suitable for loading by a JAVA runtime system.

JASMIN was originally created as a companion to the book *Java Virtual Machine*, written by Jon Meyer and Troy Downing and published by O'Reilly Associates [MD97]. Since then, it has become the de-facto standard assembly format for JAVA. It is used in dozens of compiler classes throughout the world, and has been ported and cloned multiple times. For better or worse, JASMIN remains the oldest and the original JAVA assembler.

The O'Reilly JVM book is now out of print. JASMIN continues to survive as a SOURCEFORGE Open Source project[1].

The JAVA bytecode examples are all written using the JASMIN syntax.

B.3 Simple JAVA Bytecode Instructions

The JVM is working with an evaluation stack. Local variables are stored in stack registers. The stack registers are numbered 0 to number of variables minus 1. The `xload i` instructions pushes the variable stored in register `i` onto the stack. The instruction `xstore i` pops the stack and writes the resulter into register `i`.

We will use following notation: `i`, `j` and `k` are integers or addresses `x`, `y` and `z` are floats `a`, `b`, `c` array references. `L` is a label.

B.3.1 Arithmetical `int` Operations

Operation	Stack	Meaning
`iadd`	`...,i,j ⇒ ...,k`	`k = i + j`
`isub`	`...,i,j ⇒ ...,k`	`k = i - j`
`imul`	`...,i,j ⇒ ...,k`	`k = i * j`
`idiv`	`...,i,j ⇒ ...,k`	`k = i / j`
`irem`	`...,i,j ⇒ ...,k`	`k = i - (i/j) * j`
`ineg`	`...,i ⇒ ...,k`	`k = -i`

B.3.2 Arithmetical `float` Operations

Operation	Stack	Meaning
`fadd`	`...,y,z ⇒ ...,x`	`x = y + z`
`fsub`	`...,y,z ⇒ ...,x`	`x = y - z`
`fmul`	`...,y,z ⇒ ...,x`	`x = y * z`
`fdiv`	`...,y,z ⇒ ...,x`	`x = y / z`
`fneg`	`...,y ⇒ ...,x`	`x = -y`

[1] JASMIN can be downloaded from `http://jasmin.sourceforge.net/`

B.3.3 Type Conversion

Operation	Stack	Meaning
i2f	...,i ⇒ ...,x	x = (float) i
f2i	...,y ⇒ ...,k	k = (int) y

B.3.4 Comparisons

Operation	Stack	Meaning
fcmpg	...,y,z ⇒ ...,k	k = 1 if x > y, k = 0 if x == y, k = -1 if x < y
fcmpl	...,y,z ⇒ ...,k	k = 1 if x < y, k = 0 if x == y, k = -1 if x > y

B.3.5 Bit Operations

Operation	Stack	Meaning
iand	...,i,j ⇒ ...,k	k = i & j
ior	...,i,j ⇒ ...,k	k = i | j
ixor	...,i,j ⇒ ...,k	k = i ^ j
ishr	...,i,j ⇒ ...,k	k = i » j
ishl	...,i,j ⇒ ...,k	k = i « j

B.3.6 Jumps

Operation	Stack	Meaning
goto L	... ⇒ ...	jump to L
ifeq L	...,i ⇒ ...	if i == 0 jump to L
iflt L	...,i ⇒ ...	if i < 0 jump to L
ifle L	...,i ⇒ ...	if i <= 0 jump to L
ifgt L	...,i ⇒ ...	if i > 0 jump to L
ifge L	...,i ⇒ ...	if i >= 0 jump to L
ifne L	...,i ⇒ ...	if i != 0 jump to L
L:	... ⇒ ...	L is a label for the next instruction

B.3.7 Assignments

Operation	Stack	Meaning
`iload n`	`... ⇒ ...,j`	load `j` from stack register `n`
`istore n`	`..., i ⇒ ...` store	`i` in stack register `n`
`fload n`	`... ⇒ ...,x`	load `y` from stack register `n`
`fstore n`	`...,x ⇒ ...`	store `x` in stack register `n`
`aload n`	`... ⇒ ...,a`	load reference `a` from stack register `n`
`astore n`	`...,a ⇒ ...`	store reference `a` in stack register `n`
`iaload n`	`...,a,i ⇒ ...,j`	load `j` from `a[i]`
`iastore n`	`...,a,i,j ⇒ ...`	store `j` at `a[j]`
`faload n`	`...,a,i ⇒ ...,x`	load `x` from `a[i]`
`fastore n`	`...,a,i,x ⇒ ...`	store `x` at `a[j]`
`ldc i`	`... ⇒ ...,i`	load constant `i`

B.3.8 Methods

Operation	Stack	Meaning
`ireturn`	`...,i ⇒ ...`	`return(i)`
`freturn`	`...,x ⇒ ...`	`return(x)`
`return`	`... ⇒ ...`	`return()`

`ireturn` and `freturn` pop the top element of the evaluation and pushes the resultat onto the preceeding evaluation stack (the stack of the caller).

B.3.9 While Statement

```
i = 100;
j = 0;
while (i > 0) {
  j = j + i;
  i = i - 1;
}
```

May be translated into following JASMIN assembler code:

```
        ldc 100
        istore 1
        ldc 0
        istore 2
        goto 16
L1:
        iload 2
        iload 1
        iadd
```

```
        istore 2
        iload 1
        ldc 1
        isub
        istore 1
L2:
        iload 1
        ifgt L1
```

B.3.10 If Statement

```
i = 100;
j = 0;
if (i > 0 && j <= 50) {
  j = j + i;
}
else {
  j = j - i;
}
```

May be translated into following JASMIN assembler code:

```
        ldc 100
        istore 1
        iconst 0
        istore 2
        iload 1
        ifle L1
        iload 2
        ldc 50
        if icmpgt L1
        iload 2
        iload 1
        iadd
        istore 2
        goto L2
L1:
        iload 2
        iload 1
        isub
        istore 2
L2:
```

B.3.11 Signatures

When working with methods or global variables, it is necessary to use signatures.

```
methodDescriptor    ::= "(" parameterDescriptor* ")"
```

```
                        returnDescriptor
                      ;
parameterDescriptor ::= fieldType
                      ;
returnDescriptor    ::= fieldType | "V"
                      ;
fieldType           ::= baseType | objectType | arrayType
                      ;
baseType            ::= "B" | "C" | "D" | "F" |
                        "I" | "J" | "S" | "Z"
                      ;
objectType          ::= "L" className ";"
                      ;
arrayType           ::= "[" fieldType
                      ;
```

where

Operation	JAVA	Bedeutung
B	byte	signed byte [8]
C	char	character [8]
D	double	double-precision IEEE 754 float [64]
F	float	single-precision IEEE 754 float [32]
I	int	integer [32]
J	long	long integer [64]
LClassName;	...	an instance of the class
S	short	signed short [16]
Z	boolean	true or false

B.3.12 Methods

Declaration of a static method f with one integer parameter, one integer array parameter and one String parameter;

```
.method static f(I[ILjava/lang/String;)I
```

Static method call

```
invokestatic A/f(I[ILjava/lang/String;)I
```

A is the name of the class where f has been declared.

B.3.13 Global Variables

Example B.32 [Integer Variables]

```
.field static i I = 22
```

Read:

```
getstatic ClassName/i I
```

Write;

```
putstatic ClassName/i I
```

Example B.33 [Integer Array]

```
.field static a [I
```

Initialisation:

```
sipush 5 ; array of length 5
newarray int
putstatic ClassName/a [I
```

Reading `a[3]`:

```
getstatic ClassName/a [I
sipush 3
iaload
```

Writing value `77` at `a[3]`:

```
sipush 3
sipush 77
iastore
```

B.4 Simple Examples

Example B.34 [Simple Program] Consider following program

```
x = 1;
x = x * 2;
y = 3;
z = x * y + 2;
t = 3;
t = z * z;
```

May be translated into following JASMIN assembler code:

```
.class public aOut
.super java/lang/Object

.method public <init>()V
aload_0
invokenonvirtual java/lang/Object/<init>()V
return
.end method

.method public static main([Ljava/lang/String;)V
ldc 1
istore 1
iload 1
ldc 2
imul
istore 1
ldc 3
istore 2
iload 1
iload 2
imul
ldc 2
iadd
istore 3
ldc 3
istore 4
iload 3
iload 3
imul
istore 4
return
.limit locals 5
.limit stack 3
.end method
```

Example B.35 [Method Call]

```
main () {
  f(3));
}

int f (int n) {
  if (n <= 1)
    return (1);
  else
    return(n*f(n-1));
}
```

May be translated into following JASMIN assembler code:

```
.source A.mcc
```

```
.class public A
.super java/lang/Object

.method public <init>()V
    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    ldc 3
    invokestatic A/f(I)I
.limit locals 1
.limit stack 1
.end method

.method static f(I)I
    iload 0 ; n
    ldc 1
    isub
    ifle L2
    ldc 0
    goto L3
  L2:
    ldc 1
  L3:
    ifeq L0
    ldc 1
    ireturn
    goto L1
  L0:
    iload 0 ; n
    iload 0 ; n
    ldc 1
    isub
    invokestatic T7/f(I)I
    imul
    ireturn
  L1:
    ldc 0
    goto L3
  L2:
    ldc 1
  L3:
    ifeq L0
    sipush 1
    ireturn
    goto L1
  L0:
    iload 0 ; n
```

```
    iload 0 ; n
    ldc 1
    isub
    invokestatic T7/f(I)I
    imul
    ireturn
  L1:
    ldc 0
    ireturn
.limit locals 1
.limit stack 3
.end method
```

B.5 JAVAP

The bytecode of any JAVA class file can be recovered using the command line command `javap -c`.

Example B.36 [Disassembling]

JAVA class:

```
public class A {

  public static int f (int n) {
    if (n <= 0) {
      return 1;
    } else {
      return n * f(n - 1);
    }
  }

  public static void main(String [] args) {
    System.out.println(f(7));
  }
}
```

JAVA bytecode:

```
Compiled from "A.java"
public class A extends java.lang.Object{
public A();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

public static int f(int);
  Code:
   0: iload_0
```

```
   1: ifgt 6
   4: iconst_1
   5: ireturn
   6: iload_0
   7: iload_0
   8: iconst_1
   9: isub
   10: invokestatic #2; //Method f:(I)I
   13: imul
   14: ireturn

public static void main(java.lang.String[]);
  Code:
   0: getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   3: bipush 7
   5: invokestatic #2; //Method f:(I)I
   8: invokevirtual #4; //Method java/io/PrintStream.println:(I)V
   11: return

}
```

Appendix C

Abbreviations

AST Abstract Syntact Tree.

BNF Backus-Naur Form.

CFG Context-Free Grammar.

CFL Context-Free Language.

DFA Deterministic Finite Automaton.

EBNF Extended Backus-Naur Form.

JVM JAVA Virtual Machine.

LL(k) A parser parsing the input from left to right, and constructing a leftmost derivation using a lookahead of k tokens.

LM Leftmost.

MC A subset of the C programing language.

NFA Nondeterministic Finite Automaton.

PDA Pushdown Automaton.

RE Regular Expression.

RM Rightmost.

Bibliography

[ALSU08] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compiler*. Pearson Studium, München, 2008.

[App98] A.W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[ASU07] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Boston, 2nd edition, 2007.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[BBB+57] J. W. Backus, R.J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, R. A. Hughes, and R. Nutt. The fortran automatic coding system. In *Proc. AFIPS 1957 Western Joint Computer Conference*, 1957.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. `http://www.chomsky.info/articles/195609--.pdf`.

[Dij60] E. W. Dijkstra. Algol 60 translation. Supplement ALGOL Bulletin 10, 1960.

[DS92a] C. Donnelly and R. Stallman. *Bison, the YACC-compatible Parser Generator*. Free Software Foundation, December 1992. Bison Version 1.20.

[DS92b] C. Donnelly and R. Stallman. *Bison, the YACC-compatible Parser Generator*. Free Software Foundation, December 1992. Bison Version 1.20.

[Fle06] P. Flener. Formal languages and automata theory, 2006. `http://user.it.uu.se/~pierref/courses/FLAT/`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, Reading, Massachusetts, 1995.

[Gos96] J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[Her92] H. Herold. *lex und yacc, Lexikalische und syntaktische Analyse*. Addison-Wesley, Bonn, 1992.

[HMU07] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Automata Theory, Languages and Computation*. Pearson Education, Boston, 3rd edition, 2007.

[JCC96] JAVACC, java compiler compiler [tm], 1996. `https://javacc.dev.java.net`.

[Les75] M.E. Lesk. Lex – a lexical analyser generator. Technical Report Computer Science Technical Report 39, AT & T Bell Laboratories, Murray Hill, N.J., 1975.

[LMB92] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates. Inc., Sebastopol, CA 95472, 2nd edition, 1992.

[LY96] T. Lindholm and F Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[MD97] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[Rm] S. Rodger and many. Jflap. `http://jflap.org`.

[Wik] Wikipedia. `http://en.wikipedia.org`.

[Wir71] N. Wirth. The programing language pascal. *Acta Informatica*, 1(1):35–63, 1971.

[Wir89] N. Wirth. *Programming in Modula-2*. Springer, 4th edition, 1989. ISBN 0387501509.

[Wir96] N. Wirth. *Compiler Construction*. Addison-Wesley, Reading, 1996. See also `http://www-old.oberon.ethz.ch/WirthPubl/CBEAll.pdf`.

Index

Contents