Compiler Construction

Every computer scientist learns several techniques for building syntax-directed compilation during his or her studies. Few of them ever build a whole compiler. But there are many problems besides building compilers that can be solved using the techniques of formal language analysis. Some practical examples for the applicability of lexical and syntactical analyzers are user input checking, tools for code and text analysis, readers for file exchange formats or ASCII object file formats. The analysis techniques are well known, but applying them isn't easy for the occasional compiler constructor. The following patterns are a modest start for a compiler building pattern language and can only be considered as basis for a comprehensive language on compiler construction.

This paper is directed at people who have already learned about compiler construction and formal languages, but don't have much experience in this field. The reader should know about the structure of a compiler and what are attributed grammars, otherwise I recommend reading [Aho, Sethi, Ullman] or [Fischer, LeBlanc]. This is no course in compiler construction or formal languages.

Overview

A compiler transforms a source program in a source language (e. g. a programming language) into a target program in a target language (e. g. an executable file containing machine code). This transformation is performed in several parallel or sequential steps. Those steps are:

1. Lexical analysis:
   The stream of characters is read and separated into tokens (symbols). Tokens are the basic lexical units of the language.
2. Syntax analysis:
   The tokens are combined to grammatical sentences.
3. Semantic analysis:
   The source program is examined for semantic errors; type information for further examination and the subsequent code generation is collected.
4. Generation of intermediate languages:
   A computer internal representation for the source program is constructed. To this internal representation further checks and optimizations can be applied. This step can be omitted. Sometimes several intermediate languages are implemented in one compiler to achieve a gradual approximation between the source and the target language, which reduces the complexity of each transformation. The first intermediate language is close to the source language, the last is close to the target language.
5. Code optimizing:
   An intermediate language is examined and transformed to allow the generation of the best possible code.
6. Code generation:
   Generation of target (e. g. machine) code and allocation of memory to variables.

In parallel to these processing steps there are another two important tasks to accomplish: managing symbols and error handling. The symbol table is built up in parallel with all the other phases of compilation. It holds all the symbols and the collected or computed information related to them and makes them available to all stages of compilation. In every phase errors can be detected. The compiler has to report them and possibly try to continue analyzing despite the errors (error recovery).

Some small parse tasks only need step 1 and 2. Instead of semantic analysis and building an intermediate language, an object file reader might just fill out some data structures, and a code analysis tool might just collect some information.

The different phases of the compiling process get mapped to the static structure of the compiler. There is a lexical analyzer, a syntax analyzer, a semantic evaluator, implementations of the symbol table and the intermediate languages and a code generator. For the implementation of each of these components there are a lot of patterns. This language only refers to some aspects of the analysis phase of the compilation process.

Terms:

Terminal symbol (symbol, terminal):
: a sequence of characters having a certain meaning, the basic lexical units of the language, e. g. punctuators or keywords.

Terminal class:
: a set of symbols that are constructed equally, e. g. numbers or identifiers, usually described in form of regular expressions in the input language of scanner generators.

Nonterminal symbol:
: represents a sequence of symbols, described in derivation rules.

Context-free grammar:
: a set of terminal symbols, a set of nonterminal symbols, a set of production rules and one special symbol as a start nonterminal. A production gives the rule for producing a nonterminal symbol and consists of the nonterminal symbol (on the left side), an arrow and a sequence of terminal and/or nonterminal symbols.

Top-down analysis:
: reconstruction of a leftmost derivation by beginning with the start nonterminal and by guessing the next derivation step when looking at the next token in the input.

LL(k)-grammar:
: sentences of the grammar can be recognized by reading them from left to right through left canonical derivation beginning at the start symbol using k symbols in advance.

Bottom-up analysis:
: recognizing a language by reading the input and reducing parts of a read sentential form whenever a valid phrase is recognized.

LR(k)-grammar:
: sentences of the grammar can be recognized by reading from left to right through right canonical derivation using k symbols in advance.

Semantic actions (semantic rules, output actions):
: a piece of code in the grammar, that is performed when the symbol it is related to is recognized. Semantic actions usually are used to compute attribute values.

Overview of the pattern language

Even if the techniques of compiler construction are well known, building a compiler is time-consuming and error prone. Moreover, the syntax and semantics of many languages often change frequently. To build stable and easy to maintain compiler parts quickly, the first pattern suggests *Use Generators* for generating parsers, scanners, semantic evaluators and code generators. The pattern *Choose A Good Parser Generator* defines features a good parser generator should have. To use a generator the source language has to be described in the generator's language description language which then is used as input to the generator. Most language description languages must be formulated as attributed grammars. Even if no generator is used, describing the language in a grammar can help make a good design for the scanner and the parser. *Top Down Grammar Decomposition* shows a way to build a context-free grammar. Grammars that aren't LL(1) can be made recognizable by LL(1) parsers by *Differentiating In The Scanner*. Compilers process languages in several steps which usually work interlocked. The first step is the lexical analysis, followed by the syntax analysis, then the semantic evaluation and the generation of intermediate languages or code. In every step the compiler should find as many errors as possible as early as possible, in order not to continue

unnecessary analysis when successful recognition isn't possible, and to guarantee the correct input to the next processing step. One instance of this rule is *Prefer Syntactical Checking To Semantic Checking*.

Pattern: Use Generators

Context

A given language needs a compiler, which will consist of a scanner, a parser, a semantic evaluator and maybe a code generator.

Problem

Should the compiler parts be generated or hand coded?

Forces

- Hand coded scanners and skillfully written parsers and code generators are more efficient than generated ones.
- Scanning often takes one third or even half of the compiling time.
- Generated components can be replaced by hand coded ones later if necessary.
- Using a generator, respectively using the generator's input language, must be learned.
- A language description can be useful for documentation.
- Handcoding analyzers tempts developers to optimize too early.
- The parts of the compiler following the analysis phase (like symbol table administration, intermediate languages, code generation) require more creativity than lexical and syntactical analysis.
- Testing if the language and the compiler are correct requires testing the correctness of the produced target programs. The earlier that a target program produced by the compiler can be tested, the earlier the language description can be verified.

Solution

Use generators to build parts of the compiler. The methods for lexical and syntactical analysis are well known and their construction can be automated. Parser generators often also generate semantic evaluators together with the parser. The difficult parts of the compiler construction are the ones that follow the analysis phase. Using a code generator allows early testing of compiled files and finding errors made in an earlier part of the compiler.

Symbol table, intermediate languages, optimization and good code generation require a lot more creativity. But the scanner and the parser are the prerequisites for all the other processing steps. The temptation to optimize the scanner and parser is big, hence lot of time is invested in writing the analyzers. This time could be better used for working out a good symbol handling or code optimization. Furthermore, a language description is easier to maintain than code. Using a code generator generator shortens the development process for the second part of the compiler. The language description implemented in the scanner and parser and the other processing steps of the compiler can be tested earlier. If the resulting compiler turns out to be too slow the generated components can be replaced by hand coded ones, especially hand coded scanners and code generators are usually a lot more efficient than generated ones. *Choose A Good Parser Generator* gives some advice for selecting a parser generator.

Pattern: Choose A Good Parser Generator

Context:

A parser generator should be used (maybe also a scanner generator). Quite a lot of such generators are available. Usually parser generators produce parsers that work according to a syntax-directed bottom-up (for LR or LALR grammars) or a top-down method (for LL grammars).

Problem:

Should effort be put into the selection of a parser generator, or is it sufficient to take the next best generator, that is usually YACC? What are the criteria of a good parser generator?

Forces:

- There is never enough time for software development, it shouldn't be wasted by comparing tools.
- A generator which is difficult to use is very time-consuming.
- The best known parser generator is YACC. It generates parsers that analyze bottom up using the shift- reduce-algorithm for LR(1)-grammars. It is available on most operating systems, free and frequently used. There are books describing how to use it ([S. C. Johnson] or [A. T. Schreiner]).
- LL(1)-analysis methods are easier to understand than LR(1)-methods, LL(1)-errors are easier to recognize and to remove than LR(1) errors and the LL(1) analyzing algorithms are more efficient than the LR-analyzing algorithms.
- YACC inserts artificial nonterminal symbols for every semantic action into a grammar. These nonterminal symbols can destroy the LR(1) quality of the language, that means that semantic actions can't be put everywhere in the attributed grammar. Most other generators making LR-parsers have similar restrictions when attributing a grammar. Generators that produce LL(1)-parsers do not limit the position of semantic actions in the attributed grammar.
- LR-parsers can also recognize LL-languages, but not vice versa.
- Most programming languages are LL(1), some are LR (e. g. Ada). Testing the language's qualities by hand isn't possible for larger languages. Parser generators test the language's qualities when trying to generate a parser from the language description. For that purpose the language must be described in the parser generator's description language.
- The language description language of YACC isn't very user-convenient, because the context-free grammar has to be formulated in BNF, the names of the attributes can't be chosen freely and no local variables can be declared in nonterminal rules.

Solution:

YACC is well-known and freely available, but it is worth looking for a generator which makes it easier for the user to specify the source language. The language description language is the interface between the compiler constructor and the generator. The easier the description language, the easier a language description is built up and maintained. The effort to pick out a generator only has to be made once, then it can be used often over a long period of time.

Desired features of a parser generator are:

- The language description should be in EBNF not BNF.
- Semantic actions should be allowed everywhere in the grammar. There are also parser generators that generate bottom-up parsers, that do not limit the position of semantic actions by performing semantic actions after syntax analyzing.
- It should be possible to declare local variables per nonterminal. If there are recursive rules in the grammar, local variables put aside the need to save attributes on a stack, when a recursive rule is entered several times.

- There should be a scanner generator, of which the scanners should cooperate with the parsers generated by the parser generator.
- A common language to formulate the input for the scanner and the parser generator helps keep the language descriptions consistent.
- The generated analyzers should be easily adaptable. Some generate their code into „frame files“. The „frame files“ contain for example procedures for reading the input stream or handling the namelist for scanners, or a parse algorithm for parsers. They can be adapted by the compiler constructor according to his specific needs.
- If available use a generator family, which can generate parsers which implement different parsing algorithms out of the same language description. (Coco-2 ([Dobler, Pirklbauer], [Rechenberg, Möseneböck]) and Cocktail are two examples for such user-convenient generators I worked with. Both of them implement several parsing algorithms.)

When the generator is chosen together with its language description language, a context free-grammar can be built with *Top Down Grammar Decomposition.* Subsequently the grammar must be attributed and provided with semantic actions.

Pattern: Top Down Grammar Decomposition

Context:

This pattern starts with a brief description of attributed grammars to remind you what you learned about formal languages. If you still don't understand the examples, read [Wirth], [Aho, Sethi, Ullman] or [Fischer, LeBlanc].

A language can be specified in the form of a grammar. A grammar consists of terminals, nonterminals, a start symbol and productions. Terminals are the basic symbols of a language, nonterminals are special symbols that denote sets of strings. The start symbol is the nonterminal that denotes the whole language. Productions, also called rewriting rules, define how nonterminals are built up from one another and from the terminals.

Example for a grammar in BNF:

(The term on the left side is the name of the nonterminal described by the production. Every rule is terminated by a '.'. The '|' means „or“, the ε stands for the empty string. The productions are enumerated for later use.)

Calculator ->	Expression ';'.	(production 1)
Expression ->	Factor Term .	(production 2)
Term ->	'+' Factor Term	(production 3)
	| '-' Factor Term	(production 4)
	| ε.	(production 5)
Factor ->	constant	(production 6)
	| '(' Expression ')'.	(production 7)

In our example the terminals are ';', '+', '-' and „constant“, which denotes a class of terminal symbols, lets say all integers. The terms in capital letters are the nonterminals. 'Calculator' is the start symbol and denotes the whole language. The string on the right side of each ' ->' gives the production rule for each nonterminal. Although the above grammar is rather small it can be used to generate an infinite set of valid sentences and also to recognize them.

An attributed grammar can be seen as a recognition process for a formal language. In a simplified way the process for top-down parsing can be described as follows:

Every production is processed from left to right. In the course of this process every symbol read is checked if it corresponds to the next terminal symbol in the grammar. A nonterminal symbol in the grammar is a branch to the recognition of another rule. Alternatives in the grammar are selected by looking at the next n (usually one) symbols. (This process is different for bottom-up recognition, but this isn't important to understand the recognition abilities of attributed grammars.)

Example:

We want to know if the sentence '2 + 3;' is a valid sentence of our language described by the above grammar.
Starting with the start symbol we follow the productions to find out if the string is a valid sentence

Calculator ->	apply production 1 for Calculator
Expression ';' ->	apply production 2 for Expression
Factor Term ';' ->	apply production 6 for Factor
constant Term ';' ->	read the first symbol, check if it is a constant
2 Term ';' ->	apply production 3 for Term, after checking the next input symbol
2 '+' Factor Term ';' ->	read the next symbol, check if it is a '+'
2 + Factor Term ',' ->	apply production 6 for Factor
2 + constant Term ';' ->	read the next symbol, check if it is a constant
2 + 3 Term ';' ->	apply production 5 for Term after checking the next input symbol
2 + 3 ε ';' ->	read next symbol, check if it is a ';'
2 + 3 ;	the string is recognized as sentence of the language

Usually simultaneously to the recognition semantic actions are performed, that calculate the attributes of a symbol. In case of a compiler these are parts of the symbol table and an intermediate language. These calculations can be described in the form of attributes to the grammar's symbols and in semantic actions inserted into the grammar. The result is an attributed grammar.

An attributed grammar is built up as follows:

1. The syntactical structure of the input language is formulated in a context-free grammar.
2. The attributes (semantic values) of the symbols are established.
3. The actions to calculate the symbol's attributes are specified.
4. The grammar is „attributed", whereby the compiler constructor specifies the input and output attributes for each symbol and where semantic actions have to be performed.

Back to the example:

We assume the only attribute of a constant is its value. The operation signs and the ';' need no attributes. The start nonterminal's attribute denotes the values calculated from its right side. The Term needs two attributes, one input attribute to get the value of the expression part already evaluated and an output attribute to return the newly calculated value. It is easy to insert the actions to calculate the right sides in our small grammar:

$Calculator_{\uparrow value}$ ->	$Expression_{\uparrow value}$ ';'.	
$Expression_{\uparrow newVal}$ ->	$Factor_{\uparrow value}$ $Term_{\downarrow value \uparrow newVal}$.	
$Term_{\downarrow calcVal \uparrow newVal}$ ->	'+' $Factor_{\uparrow value}$ $Term_{\downarrow calcVal \uparrow newVal}$	sem calcVal = calcVal + value; endsem
	| '-' $Factor_{\uparrow value}$ $Term_{\downarrow calcVal \uparrow newVal}$	sem calcVal = calcVal - value; endsem

| ε sem newVal = calcVal; endsem .

Factor$_{\uparrow value}$ -> constant$_{\uparrow value}$
| '(' Expression$_{\uparrow value}$ ')'.

Average languages are of course much more complicated. They are described either in the form of a grammar, in prose text and/or in examples.

Problem:

A context-free grammar for a formal language should be built up, either for documentation or for use as an input for a compiler generator.

Forces:

- The language description language specifies restrictions about how to formulate the description, e. g. EBNF versus BNF or the analyzing algorithm works on LR(1) or on LL(1).
- Maybe there is a formal description of the language that could be used as a grammar, after it has been adapted according to the language description language's formalism.
- Formal descriptions by authors who don't know much about formal languages are usually bad.
- The grammar should be clear, compact and easy to understand to be maintainable and useful for documentation.

Solution:

Constructing a context-free grammar is like designing a procedural program. The grammar should be considered a program that recognizes sentences of a formal language. It uses procedures that correspond to nonterminal symbols in the grammar and recognize subparts of the language. Related or repeated parts of the language are combined to such procedures (nonterminals). Starting with the start nonterminal (the nonterminal representing the whole language) the grammar will be refined top down. Every new nonterminal is equivalent to a new procedure. The names of the nonterminal symbols should be as predicative as function names and comprise only units belonging together. Names often can be chosen from terms used for language parts, such as „declaration" or „parameter list".

Example:

The starting symbol of a grammar for a programming language denotes a single program. It may consist of keywords to mark the beginning and the end of the program, its name and the program's body. The first rule might look like this:

Program = 'PROGRAM' identifier ProgramBody 'END'.

Then the ProgramBody is refined. If a program in this language must have a declaration part followed by a list of statements then the second rule might be:

ProgramBody = Declarations
StatementList.

and so on, until the whole language is described.

Sometimes duplicate grammar parts can be removed by making a nonterminal for them. Be sure that the semantics of the grammar part is the same in all occurrences. Otherwise there will be problems when the grammar is attributed and provided with semantic actions. If the semantic s are not the same,

the semantic actions of the nonterminal's rule would have to distinguish the context in which the nonterminal was used.

Example:

In a programming language there is an ordinary assignment statement:

AssignmentStatement -> **identifier '=' expression**.

In a function call, the actual parameters also can be „assigned" to the formal parameters:

FunctionCall -> functionName '(' { **identifier '=' expression**} ')'.

The assignment of parameters looks like an „AssignmentStatement", but the code produced from both cases may be different. An assignment statement requires code to write a value to the memory the variable addresses, but a function call probably requires putting the value on the stack. Replacing „**identifier '=' expression"** by „AssignmentStatement" in the rule for „FunctionCall" forces the compiler constructor to distinguis between an ordinary „AssignmentStatement" and one that occurred in a „FunctionCall" in the semantic actions of the „AssignmentStatement".

Occasionally *Top Down Grammar Decomposition* isn't useful. For example, a language that specifies an object file format for an object oriented application will use objects to represent the data of the application and the object file format to describe a set of classes. In this case the language's grammar is already decomposed. The class names can be used as nonterminal symbol names. For the grammar's structure the has-a-relation of the class structure is used. The classes with only terminal symbols as member variables can be used as nonterminals of the lowest layer. The following more abstract nonterminals are made of the classes which feature these nonterminals of the lowest layer and terminals. This building process is repeated until the whole data structure of the application is described.

Building a context-free grammar is the first step towards an attributed grammar. Before you can start attributing it, the grammar must be recognizable by an analysis algorithm to be LL(1) or LR(1). All left-recursions must be removed from LL(1) grammars and LL(1) conflicts must be solved by left-factoring. The algorithms are explained in detail in [Aho, Hopcroft, Ullman] and [Fischer, LeBlanc]. Unfortunately, these algorithms make the grammar more complicated and less readable. A method to make a grammar top-down recognizable and keep it clear is introduced in *Differentiate In The Scanner*.

Pattern: Differentiate In The Scanner

Context:

A language description in the form of a context-free grammar was built up *using Top Down Grammar Decomposition*. The language description is used as input for a parser generator which generates parsers which work according to a top-down LL(1) analyzing algorithm.

Problem:

The grammar has LL(1) conflicts. If the parser was hand-coded and not generated, it would be easy to solve this problem. If required, some more symbols of the input stream could be read for solving an LL(1) conflict. But how can this problem be solved in a generated parser?

Forces:

- Every LL(k) grammar can be transformed into an equivalent LL(1) grammar by removing left-recursion and left-factoring of up to k symbols of two alternatives causing an LL(1) conflict.
- A grammar transformed by left-factoring gets more complex. The clear structure achieved by *TopDownDecomposition* is lost. This affects the grammar's quality consisting in changeability and usefulness for documentation.
- Methods to make non LL(1) grammars recognizable by an LL(1) parsers require creativity.
- It often isn't easy to adapt generated analyzers according to the compiler constructor's needs. Sometimes more creative methods than left-factoring aren't possible.

Solution:

Cheat the parser. Change the lexical analyzer so it cheats the parser by reporting that it has read a special symbol that is actually not in the input stream. Introduce an artificial symbol that can't be mistaken for any other terminal symbol in the grammar. Instead of left-factoring the alternatives causing the conflict, insert the artificial symbol into one of the alternatives, either instead of one of the symbols which start the conflict or in front of this symbol. This makes the alternatives begin with different terminal symbols thus solving the LL(1) conflict.

Example:
Integer constants and enumeration constants can be elements of the selection list of a case-statement in a certain language. This is the rule for the case-statement in EBNF (keywords and plain terminals are enclosed in quotation marks, nonterminals symbols start with capital letters, terminals with small letters):

```
CaseStatement =
        'CASE' Expression
        'OF' CaseList StatementList
        {CaseList StatementList}
        'END_CASE'.

CaseList =
        CaseListElement
        {CaseListElement} ':' .

CaseListElement =
        SignedInteger ['..' SignedInteger]
        | identifier.          // this identifier can only be the name of an enumeration constant

StatementList =
        identifier ':=' .... .   // start of an assignment statement, with other arbitrary statements as
                                    alternatives
```

An identifier in a CaseListElement must be an enumeration constant. An LL(1) conflict occurs whenever an identifier is read (bold in the grammar). It can't be decided whether 'identifier' is a CaseList or the beginning of an assignment statement.

To solve the conflict, invent an artificial terminal symbol, the „enumeration constant symbol", that replaces the identifier. The rule for the CaseListElement changes as follows:

```
CaseListElement =
        SignedInteger ['..' SignedInteger]
        | enumerationConstant.          // enumerationConstant is specified as a token which differs
```

// from all other tokens in the grammar.

How can the lexical analyzer be influenced so that it returns the correct symbol when needed?

1.1. The lexical analyzer can read several symbols in advance. It makes them available for the semantic evaluator when they are needed. In a semantic action before CaseList the lexical analyzer can be told to return either the token code of an ordinary identifier - if the next token is an identifier, followed by a ‘:’ and ‘=’ - or the artificial symbol otherwise.

1.2. Symbols sometimes can’t be distinguished lexically, but semantically. Enumeration constants look like variable, procedure or type names, but what distinguishes them from other identifiers is that they are contained in enumeration constant declarations. Such identifiers can be marked in the lexical analyzer, controlled by semantic actions. Thus it is possible for the lexical analyzer to recognize such identifiers as enumeration constants in the further course of analyzing. Conflicts can be avoided, when alternative rules can either start with variable or program names or with enumeration constants. In the above example the lexical analyzer can mark the name of the enumeration constant in the name list, so it can associate this name with an enumeration constant’s name. The lexical analyzer can access this information (the scanner usually manages a name list). Each time such a name occurs, the scanner returns the token code for an enumeration constant symbol.

2. Properties of a language can also be checked semantically instead of syntactically. The parser is cheated since it only knows the original language partly.

Example:
A language has some qualifiers consisting of one or two characters. The rule containing these qualifiers is:

```
ActionAssociation =
        identifier ‘(‘ (‘N’ | ‘ R’ | ‘S’ |  ‘P’ |  (‘L’ | ‘D’ | ‘ SD’ | ‘DS’ | ‘SL’) [‘,’ (duration | identifier)
                                                                                {identifier}]
                        )
                ‘)’.
```

The compiler constructor could make keywords out of all these characters or pairs. This would mean that either these letters could no longer be ordinary identifiers in the language because the lexical analyzer would always recognize them as keywords, or to avoid this, wherever an identifier is allowed in a grammar these keywords must be allowed as well. In the latter case the beginning of an assignment statement would look like this:

```
AssignmentStatement =
        (identifier | ‘N’ | ‘ R’ | ‘S’ |  ‘P’ |  ‘L’ | ‘D’ | ‘ SD’ | ‘DS’ | ‘SL’) ‘:=’ ..... .
```

The solution is to check the qualifiers not syntactically, but semantically. The rule changes to:

```
ActionAssociation =
        ident ‘(‘ identifier              SemanticAction:  check if identifier corresponds to one of the
                                                           above characters or pairs.
                [‘,’ (duration | identifier) {identifier}]
              ‘)’.
```

This solution contradicts the next pattern, *Prefer Syntactical Checking To Semantic Checking.*

Pattern: Prefer Syntactical Checking To Semantic Checking

Context:

A grammar is constructed by *Top Down Grammar Decomposition*. There are always some language qualities, that can't be checked syntactically, e. g. the type compatibility in expressions. But in some situations, the language's qualities can be checked either syntactically or semantically.

Example:
A language might specify expressions denoting a variable as follows:

Variable = identifier {'.' identifier | '[' Expression ']' }.
Expression = (identifier | ['+' | '-'] constant) { ('+' | '-' | '*' | '/') (identifier | constant) }.

Objects (variables) can be initialized in the language when declaring them. In a certain context a variable can be initialized with another variable, but in this case the address of the initializing variable must be computable at compile time. This means that expressions for selecting one component of an array may use only constants.

The compiler constructor could write some semantic actions that check, if the expression enclosed in '[' and ']' only contains constants. But she also could introduce a new rule for variables of which the address is computalble during the compile time, e. g:

ConstVariable = identifier {'.' identifier | '[' ConstExpression ']' }.
ConstExpression = ['+' | '-'] constant){ ('+' | '-' | '*' | '/') constant }.

Problem:

The qualities of a language can be checked syntactically or semantically. What variant should be preferred?

Forces:

- The grammar becomes more compact when checks are performed by the semantic evaluation.
- Checking syntactical restrictions semantically is more complex.
- Complex syntactical constructions and long rules affect the grammar's readability and maintainability.
- The grammar is not really a description of the language if the syntax isn't complete or correct.
- Syntactical errors are found earlier than semantic errors.
- Semantic checks allow more detailed error messages.

Solution:

Every processing step in the compiler should find as many errors as possible as early as possible to reduce the complexity of the next processing step. This is also the underlying principle of the previous pattern *Differentiate In The Scanner*. When using a parser generator, give preference to syntactic checking because this often only requires changing some rules in the grammar or adding new rules. Semantic checks requires writing code to test the language's features. Hand written code usually is longer, less clear and more error prone than generated code or tables. Furthermore the grammar is more useful as a documentation when it describes the real syntax of the language.

Literature:

[Aho, Sethi, Ullman] A. V. Aho, R. Sethi, J. D. Ullman: „Compilers" Addison Wesley 1988

[Dobler, Pirklbauer] H. Dobler, K. Pirklbauer: „Coco-2, A New Compiler Compiler“, Technical Report
TR 90/1, Institut für Informatik, Universität Linz, January 1990
[Fischer, LeBlanc] C. N. Fischer, R. J. LeBlanc: „Crafting A Compiler“, The Benjamin/Cummings
Publishing Company, 1988
[Johnson] S. C. Johnson: „Yacc, Yet Another Compiler-Compiler“, Technical Report Nr. 32
Bell Laboratories 1975
[Rechenberg, Möseneböck] P. Rechenberg, H. Moessenboeck: „Ein Compiler-Generator für Mikrocomputer“ Hanser, 1985
[Schreiner] A. T. Schreiner, H. G. Friedmann: „Compiler bauen mit UNIX“, Hanser Verlag
München-Wien, 1985
[Waite, Goos] Q. M. Waite, G. Goos: „Compiler Construction“ Springer, 1983
[Wirth] N. Wirth: „Compilerbau“, Teubner, 1977