LISS – The Language and the Compiler

Daniela da Cruz and Pedro Rangel Henriques

Departamento de Informática
Universidade do Minho, CCTC, Braga, Portugal,
danieladacruz@di.uminho.pt

Abstract. LISS, Language of Integers, Sequences and Sets, is a toy-programming language that allows us to operate with atomic or structured integers values (literals and variables) developed in the context of a Compilers Course, to teach parsing and code generation (a complete compiler) for a traditional imperative and block-structured programming language.
LISS has the traditional integer, and array (`of integers`) data types. But beside that, the language includes dynamic sequences and sets of integers; sets are defined in comprehension, so the programmer can define and use infinite sets. More recently, the language was extended with other unusual data types in imperative languages: complex, polynomial, point, polygon, and tree.
The language (LISS) syntax and (static and dynamic) semantics is completely specified via an attribute grammar. The compiler, that generates Assembly for the VM virtual stack-machine, is automatically produced with the AG-based compiler generator LISA.
It is our purpose in this paper to introduce the extension made to LISS type syntax. LISS Compiler will also be presented, emphasizing the approach followed to cope with the unusual data types.

1 Introduction

The definition of new languages—to specialize a generic one for some application domain, or to generalize some operations/constructors that can be used in a broader domain—is a challenge for programmers. However, the design of a *nice* language requires some domain knowledge and good-practices. The development of a processor for the new language (many times called, the *language implementation* task) is complex; starting from the scratch, is not a simple task and can lead to a time consuming and bad solution; nowadays there are a supporting theory, methods and tools to accomplish it successfully.

In parallel and similarly, to teach *language processing* is another challenging and complex task. Centering the course on the definition and implementation of *domain specific languages* (DSL) makes life easier for the teacher because smaller and much more motivating projects can the proposed and handle completely. But the study of a complete compiler is richer in many senses and is an item that should not be discarded. However, once again, it is an hard work that requires the existence of an appropriate *pedagogical kit*.

This paper focus on LISS Compiler (LissC) aiming at bringing into discussion its value as a learning object for Compiler Courses, or even as a test bed for research projects on Language Processing.

The implementation of LissC involves the specification of: the source language, LISS; the target language, VM Assembly; and the translation-scheme. It, also, requires the choice of: a translation method—Syntax-directed Translation (SDT), or Semantic-directed Translation (SemDT); the data structures adequate to store the information necessary for the translation process; and the tools that will be used for automatic generation of the compiler modules.

One of the claims of this paper is that Semantic-directed Translation is much more convenient—in fact we defend the compilation model based on a *front-end* (FE) that analyzes the source text and builds an error-free *intermediate representation* (*IR*), an *abstract syntax tree* (*AST*) decorated with attributes, and a *back-end* (BE) that traverses it and generates the output code. This approach makes the conception and the specification tasks much easier; both the compiler developer and the teacher—that need to start the work describing the compiler architecture and then specify each component—take profit of that model.

To support that approach, we argue that an *attribute grammar* [Knu68], AG, (see also[Cou84,Eng84,DJL88,Kas91]), is a good (complete, coherent, declarative) specification formalism that allows to define rigorously in a systematic way the language syntax, static semantics and also the dynamic semantics (the translation); moreover, the AG definition provides mechanisms to validate the completude and soundness of each language specification. Along the article some examples are given to show that the AG description of *scope analysis*, *type checking* or *code generation* becomes easy, systematic and clear.

We also claim that a virtual machine [Dor75,Cra06] is a good choice as the target of the translation, if efficiency is not a strong requirement. Besides the fact that it is interesting to conceive and implement a virtual machine, the generation of code for a virtual machine guarantees its portability and makes the translation much easier; mainly the *instruction selection phase* becomes simpler because in a virtual machine we avoid alternative operations to do the same thing.

LISS—that stands for *Language of Integers, Sequences and Sets*—is a Pascal-like [Wir76] imperative (or procedural) programming language. The language is designed to process—*input/output, store, and operate*—atomic or structured integer values; those values can be constants or variables. LISS includes control statements and subprograms to allow the programmer to write structured code.

In order to illustrate one of the cleverest features of a compiler—*data type implementation*—LISS language was conceived with powerful and unusual types.

`Integer` and `Array of Integer` are the traditional types supported by LISS. The basic language also supports: `Sequence of Integers`—ordered collection with dynamic size (that is, that does not need a predefined dimension), and whose components can be selected with list or array operators; Set of Integers—unordered collection defined in comprehension (by a boolean expression over integers). More recently, LISS was extended with the following data types, that

will be introduced in the the paper: complex; polynomial; point and polygon; and tree.

In the paper, we introduce LISS language in section 2; after a general overview of the language statements, the type system is described. The compiler generation, using an attribute grammar specification and LISA tool [MZLA99,MLAZ00], is discussed in section 3; examples of attribute evaluation rules and contextual conditions are given for type checking and scope analysis. VM [Fil06] target machine (*a virtual stack machine with an heap mechanism for dynamic memory management*) and the translation-scheme are presented in section 5. Final remarks appear in section 6.

2 LISS language and its Data Types

LISS language follows an procedural and verbose, Pascal-like, style but it is equipped with unusual data types. Aiming at making LISS a valuable aid for teaching compilers, a special attention was paid to the definition of: static (*compile time*) and dynamic (*run time*) *type checking*; block nesting and *scope analysis*; *operational semantics* of the non-standard data types; and *operational semantics* of the input/output instructions.

In this section, we present the basic statements of the language—assignment, read/write and control statements (conditional/cyclic). We also present LISS data types. Their syntax is introduced by fragments of the BNF grammar, included in this section; then, in section 3, the static semantics is formalized via an attribute grammar (written in LISA notation)—we chose the relevant subset of attributes, and just some evaluation rules or contextual conditions are included; after that we discuss, in section 5, the dynamic semantics aspects showing the the schemas followed to generate assembly code.

The following aspects of LISS language should be taken into account in the rest of the presentation:

- All variables are initialized, when they are declared, with a type-dependent default value according to table 2.

Type	Default value
boolean	false
integer, polynomial	0
complex	0+0i
array, polygon, line, rectangle	[0,...,0]
set	{}
sequence, tree	nil
circle	center—(0,0), rad—0
point	(0,0)

- Alternatively, variables can be initialized explicitly with a different value. For example:

```
a = -4, b, c= 5 -> integer;
v1, v2 = [10,-20,30,-40] -> array size 4;
```

– Variable types are just the pre-defined ones; LISS does not support any *type definition* mechanism.

All variables involved in the statements, inside the body of the program, must be declared before their use, according to the syntactic rules below:

```
VariableDeclaration → Vars "->" Type ';'
Vars                → Var | Vars ',' Var
Var                 → IDENTIFIER ValueVar
ValueVar            → ϵ | '=' InicVar
Type                → "integer" | "boolean" | "set" | "sequence"
                    | "array" "size" Dimension | "complex" | "polynomial"
                    | "point" | "square" | "polygon" | "circle"
                    | "tree"
Dimension           → NUMBER | Dimension ',' NUMBER
```

2.1 The basic language

Assignment and I/O. The assignment operation is defined for every type; however the value to assign (given by the expression on the right hand side) and the variable to be assigned (on the left hand side) should have the same type.

The read operation (that assigns to a variable a value obtained from the *standard input* file) is defined just for variables of type integer (atomic values).

However, the write operation (that sends a value to *standard de output* file) is defined for any type.

Control statements. LISS language includes the traditional control statements, conditional (two variants) and cyclic (also two variants), as follows:

```
ControlStat → IfStat | CaseStat
            | WhileStat | ForStat
```

We just detail the cyclic statement `for` because it is different from the conventional one, where a control variable (CV) takes values in a given range stepping by a default or explicit increment. In LISS there are 4 different ways to define the values range for the control variable: `in` — the CV takes values in a given integer interval defined by the lower and upper bounds (by default the step is 1, but a different (positive or negative) increment can be set); `inArray` — the CV is assigned with all the elements of an array, from the lower to the upper index; `inSequence` — similar to the previous, but taking values from a sequence; `inFunction` — the CV takes values from the co-domain of a given 1-variable function, computed in a subrange of its domain.
Additionally, a condition can be given to filter the values in the specified interval.
The grammar fragment below defines the cycle `for`:

```
ForStat             → "for" '(' Interval ')' Step Satisfy
                        '{' Statements '}'
Interval            → IDENTIFIER TypeInterval
TypeInterval        → "in"         Range
                    | "inArray"    IDENTIFIER
                    | "inSequence" IDENTIFIER
                    | "inFunction" IDENTIFIER "with" Interval
Range               → Minimum ".." Maximum
Minimum, Maximum    → NUMBER  | IDENTIFIER "with" Interval
Step                → ϵ  | "stepUp" NUMBER | "stepDown" NUMBER
Satisfy             → ϵ  | "satisfying" Expression
```

Notice that `Satisfy` is a condition, so `Expression` should be of type `boolean`. The next 2 examples help to clarify the possible ways to control the iterative statement `for`:

1. Using an integer range (`in`), the interval 2000 to 'c', stepping by a decrement of 2 (`stepdown`), with an additional filter (`satisfying`);

   ```
   for (a in 2000..c) stepdown 2 satisfying vector[a]==a {...}
   ```

2. Using elements of the co-domain of a function `f` computed in the interval 1 to 10 with increments by 1.

   ```
   for (b inFunction f(x) with x in 1..10) {...}
   ```

Block structure in LISS (Sub-programs). LISS is not a mono-block language; instead, it is possible to organize the code (for reuse, or just for the sake of clarity) splitting the statements into sub-programs. One program unit can contain several sub-programs, and sub-programs can also declare another sub-programs in a *Pascal-like nesting* strategy.

A sub-program, before ending, can return or not a value, behaving as a *function* or a *procedure*.

Concerning scope rules, the relevant aspects in LISS sub-programs are: variable declared in the program unit are global, unless re-declared inside a sub-program; variables declared in a sub-program are local, but they can be accessed in inner blocks.

The syntax to define a sub-program is ruled out by the following grammar fragment (notice that it follows precisely the same philosophy used for the program unit).

```
SubProgram_Definition → "subProgram" IDENTIFIER
                          '(' FormalArgs ')' ReturnType FBody
FBody                 → '{'
                                  "declarations" Declarations
                                  "statements" Statements
                                  Return '}'
FormalArgs            → ϵ | FArgs
FArgs                 → FormalArg  | FArgs ';' FormalArg
```

```
FormalArg          → IDENTIFIER "->" Type
ReturnType         →  ϵ  | "->" Type
Return             →  ϵ  | "return" Expression
```

Integers and Booleans. Values of type `integer` are positive or negative integer numbers including zero; over `integer` type the usual arithmetic operators `+, -, *, /, %` are allowed.

Values of type `boolean` are the truth values `true` and `false`; over `boolean` type, the logic operators `!, and, or` are allowed.

Relational operators `==, !=, <, >, <=, >=`, that take integer and return boolean values, are also defined.

Arrays (static lists). LISS supports Arrays—indexed collections of integer values such that each value is uniquely addressed by a combination of one, two or more integer indexes (depending on the array dimension). The number of dimensions and the maximum size of elements in each dimension is fixed at declaration time; this is why array is said to be a *static* structured type.

The operations defined over `array` type are: **indexing**, denote by '[' ']' that selects an element given the array id and the index values; **cardinality**, that computes the number of elements in the array; **assignment**, that copies all the elements of an array to another one; and **write**, that outputs all the elements in the array, according to its dimensions.

Arrays can be initialized in the declaration, giving all or part of the elements in each dimension. For example, consider an array of dimension $4x2$; it could be initialized in the following way:

```
array1 := [[1,2],[4]] -> Array size 4,2
```

that is equivalent to the initialization below:

```
array1 := [[1,2],[4,0], [0,0], [0,0]] -> Array size 4,2
```

The grammar fragment below defines the syntax for `array` declaration and initialization.

```
ArrayDefinition     → '[' ArrayInitialization ']'
ArrayInitialization → '[' Elem ']'
                     |  ArrayInitialization ',' Elem
Elem                → NUMBER | ArrayDefinition
```

Sequences (dynamic lists). In LISS a value of type `sequence` is a list (ordered collection) of integers. Similar to a uni-dimensional `array` (also called a vector), a sequence does not have a fixed size; its actual size is not define at declaration time (as for `array` values) but it grows dynamically at run time.

The operations defined over `sequence` type are: **add** inserts a new element at the head or at tail of a sequence; **head** and **tail** select the head or the tail of

a sequence; **indexing**, denoted '[' ']', selects the element at the index position; **remove** deletes a given element; **empty** verifies if the sequence is null; **length** computes the number of elements in the sequence; **in** checks if a given value belongs to the sequence; **assignment** copies all the elements of a sequence to a new one; and **write** outputs all the elements in a sequence.
As in arrays, dynamic lists can be initialized at declaration time, according to the following grammar rules:

```
SequenceDefinition     → '[' SequenceInitialization ']'
SequenceInitialization → ϵ | Values
```

Sets. A value of type `set` is a collection of integers, without repeat numbers. In LISS a `set` is defined by comprehension, instead of enumeration.
The operations defined over `set` type are: **union**; **intersection**; **in**, all of them with the known mathematical meaning. **assignment** is also allowed, as well as **write**, but in both cases working with the conditional expression that describes the set instead of set elements.
The syntax for `set` declaration and initialization is:

```
SetDefinition          → '{' SetInitialization '}'
SetInitialization      → ϵ | IDENTIFIER '|' Expression
```

Notice that `Expression` should be a boolean expression like: `x < 6 && x > -7 || x > 10`.

2.2 The extension

In this section, we introduce the recent data types added to the basic LISS language.

Geometric values. In LISS some geometric shapes—`point`, `line`, `rectangle`, `polygon` and `circle`—can now be handled.
The operations defined over `set` type are: **draw**; **zoom**, using a given percentage; **rotate**; and **assign**; **write**.
The syntax for geometric shapes declaration and initialization is:

```
Point                  → '(' Constant ',' Constant ')'
CircleDefinition       → '(' CircleInitialization ')'
CircleInitialization   → ϵ
                         | Constant ',' Point
                         | Constant ',' Constant ',' Constant
LineDefinition         → '[' LineInitialization ']'
LineInitialization     → ϵ
                         | Point ',' Point
SquareDefinition       → '[' SquareInitialization ']'
SquareInitialization   →  ϵ
                         | Point ',' Point ',' Point ',' Point
```

```
PolygonDefinition        → '[' PolygonInitialization ']'
PolygonInitialization    →  ϵ  | LstPoints
```

Notice that `Constant` is a variable or an integer.
Some examples of the declaration and initialization of geometric shapes is presented below.

```
point1 = (1,2), point2 -> point;
ray = 2 -> integer;
circle1 = (ray, point1), circle2 = (raio,0,0) -> circle;
line1 = [point1, point2] -> line;
square = [(0,0), (0,1), (1,0), (1,1)] -> rectangle;
poly1 = [(0,0), (0,2), (1,1), (2,2), (1,6)] -> polygon;
```

Polynomials. `Polynomial` values of any degree, with integer coefficients and exponents, are allowed.
The operations defined over `polynomial` type are: **add**, **subtract**, and **derivation**, the usual arithmetic operators; **assignment**; and **write**.
The syntax for `polynomial` type declaration and initialization is:

```
PolynomialDefinition  → Monomial
                        | PolynomialDefinition ('+' | '-') Monomial
Monomial              → Coefficient VarDegree
Coefficient           →  ϵ  | NUMBER
VarDegree             → ('*' | ϵ) IDENTIFIER '^' NUMBER
```

Complex. `Complex` numbers with integer in the *real* and *complex parts*, can also be handled in LISS.
The operations defined over `complex` type are: **add, multiplication, subtract, division**, the usual arithmetic operators; **real part** and **imaginary part**, to select both parts of the complex number; **assignment**; and **write**.
The syntax for `complex` type declaration and initialization is:

```
ComplexDefinition  → Real ('+' | '-') Imaginary
Real               → Coefficient
Imaginary          → Coefficient 'i'
```

Notice that `Coefficient` is an integer value.

Binary trees. Another primitive data type in LISS is the `binary search tree`.
The operations defined over `binary tree` type are: **add** inserts an element in the search tree; **delete** removes a given element of the tree; **find** search for a given element in the tree; **in-order**, **pre-order** e **post-order**, the three usual tree-traversals; **assignment**; and **write**.
The syntax for `Binary Tree` type declaration and initialization is:

```
TreeDefinition  → "NIL"
                | tree '(' Constant ',' TreeDefinition
                                    ',' TreeDefinition ')'
```

Notice that `Constant` is an integer value.

3 Compiler Development

In this section we present the compiler generator LISA [MZ03] used to produce automatically LISS Compiler, LissC; then we present the philosophy (as detailed in [MHK+02]) underlying the development of an attribute grammar [Tie80]. The instantiation of that approach for the LISS Compiler will be illustrated in the next two sections with *attribute grammar* fragments that describe type checking, scope analysis, and code generation.

3.1 The compiler generator LISA

LISA (Language Implementation System based on Attribute grammars) is a compiler-compiler, or a system that generates automatically a compiler/interpreter from formal *attribute grammar*-based language specifications. As LISA accepts as input an *attribute grammar* (in a single file), it generates all the modules of the compiler: lexical, syntactical and semantic analyzers, and code generator. By default, the compiler generated outputs (prints) the value of all synthesized attributes associated with the grammar root, or star-symbol.

LISA is platform independent; the system is written in JAVA and compilers ara also generated in JAVA.

LISA environment offers some menu options to generate and run the compiler, and provides an editor to write the *attribute grammar*, and some visual inspectors to analyze/debug the *attribute grammar*. Namely: a graphical representation of the lexer, more precisely of the transition function of the DFA (Deterministic Finite Automata) generated from the regular expressions that describe the terminal symbols; a visualizer for the syntax diagram describing the context-free grammar; and a visualizer for the Attribute Dependency Graph local to each production.

Besides its main task (generate the intended processor), LISA also produces somes tools to work with the new language: a structural editor to create or inspect source programs; a syntax tree visualizer, to see the parsed structure of a given program; and a semantic animator that presents the attributed abstract syntax tree for a given program and shows its decoration, this is, the attribute evaluation process and the tree reduction.

3.2 LISS attribute grammar, the approach

This subsection is intended to emphasize the expressive power of the *attribute grammar*-based approach to language formalization, as well as its simplicity and systematization.

To specify LISS Compiler, we chose 5 main attributes[1]: symbolTable, Scope, Code, Errors, and Comments. Those attributes, described below, are associated with the majority of grammar non-terminals, and support the specification of the language semantic constraints (contextual conditions) and code generation. Other attributes included in the *attribute grammar* are auxiliary and are used just to keep temporary information local to some productions.

1. (in|out)symbolTable — inherited/synthesized attribute to map the identifiers (*variable, function, or procedure names*) declared all over the program (including subprograms) into their description; the attribute is implemented as an hashtable, where the *key* is the identifier (`name`) and the associated *value* is a tuple composed by: the `class`, the `type`, the `memory-address`, and the `scopeLevel`[2];
2. (in|out)Scope — inherited/synthesized attribute to control the nesting level;
3. (in|out)Errors — inherited/synthesized attribute to concatenate the messages generated each time an error is detected during semantic analysis, this is each time a contextual condition is violated (for instance: *type* or *scope rules* not obeyed);
4. (in|out)Code — inherited/synthesized attribute to contain the assembly code generated as long as the program is being translated;
5. (in|out)Comments — inherited/synthesized attribute to concatenate the comments to include in the assembly code generated, in order to help understanding the compiler's output; actually, a comment line is generated, for each declaration or statements in the source program, containing the source text (of the declaration/statement) and its position (line/column), so that it becomes easy to relate the input with the corresponding output code.

In the next section (sec. 4 we present the strategy followed to specify the semantic analysis, namely *type checking* and *scope checking* tasks; they use the first three attributes above mentioned: `symbolTable`, `Scope`, and `Errors`. The other two, `Code` and `Comments` concerned with Assembly code generation, will be explained in section 5.

4 Semantic Analysis in LISS

4.1 Type checking

To discuss type checking in LISS Compiler, we will consider 2 different cases: *Variable declaration* processing; and *Assignment* processing.

[1] Notice that actually we have 10 attributes, because for each one we need an inherited and a synthesized attribute.

[2] Additional information—like array dimensions, procedure/function arguments, etc.—can also be included in the symbolTable, depending on the identifier class.

Variable declaration. To process a variable declaration in LISS, like the one exemplified below:

```
point1 = (2,3), point2, point3 -> point.
```

two actions must be done: a) each identifier recognized in the list on the left-most part of the declaration (`point1`, `point2`, `point3`) should be added to the identifier table (`outSymbolTable`) associated with its `type` (defined by the identifier at the right-most part of declaration) and its `memory-address` (automatically generated by the compiler); b) code should be generated to initialize the variable with the default or given value. If that initial value is explicitly described by an expression, then it is also necessary to check if the *expression type* and the *declared type* are *compatible* (in LISS, *compatible* means *the same*).

The strategy followed is to associate a temporary attribute, `outType`, with symbol `Type` and assign it to the attribute (`inType`) associate with symbol `Vars`. In this way, the type can be inherited by the symbol `Var` through its attribute `inType`. On one hand, the value of `inType` can be used in conjunction with the name of the variable (given by `Identifier.value`) to update the inherited table `Var.inSymbolTable`, and produce the new table, `Var.outSymbolTabel`; on the other hand, `Var.inType` can be compared with the type of the expression (given by `INICVAR.outType`) in order to check if the type constraint holds.

The *attribute grammar* fragment below (concerned with *variable declaration/initialization*) illustrates the systematic way to verify a contextual condition and generate a semantic error message[3].

```
rule Var  {
  VAR ::= #Identifier INICVAR compute {
          ...
          VAR.outSymbolTable = addIdentifier( VAR.inSymbolTable,
                                              #Identifier.value(), VAR.inType);
          VAR.outErrors = VAR.inErrors +
                          typeErrors(VAR.inType,INICVAR.outType) +
                          isRedefinedVar( VAR.inSymbolTable,
                                          #Identifier.value,VAR.inScope);
               ....                     };
          }
```

By the way, notice also another semantic check, `isRedefinedVar()`, included in `VAR.outErrors` evaluation rule; that test verifies if the declared variable name occurs in the inherited identifier table at the same level (defined by the attribute `VAR.inScope`).

Figure 1 depicts the *dependency graph* generated by LISA for the derivation rules involved; that dependency graph shows the inheritance/synthesis of attributes above described. The attributes on the left side of each symbol (*blue* color) are *inherited*; and those on the right side (*green* color) are *synthesized*.

[3] Notice that, instead of output the message at each tree node, error messages are concatenated all over the tree using the `Error` attribute; as a synthesized attribute

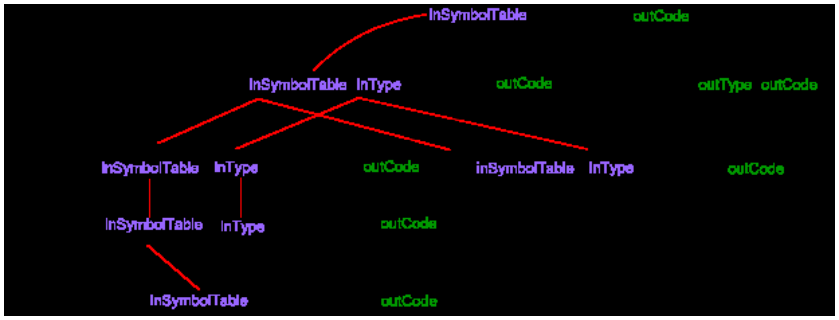


Fig. 1. Dependency Graph for Variable Declaration (as generated by LISA).

Assignment. Type checking in assignment is very similar to the one discussed above for variable initialization; so the strategy followed to specify it with attributes is also similar to the one discussed in the previous item.
The type of the variable on the left-hand side of the assign operator (`Var.outType`) is obtained looking-up for the identifier (`Var.outName`) in the attribute `inSymbolTable`, inherited by the non-terminal `Assign`. To test the contextual condition concerned with *type compatibility in assignment*, `Var.outType` is compared with the type of the expression on the right-hand side of the assign operator (the synthesized attribute `Expression.outType`); for instance, in the assignment `f = tree(6,NIL,NIL)`, we lookup in `inSymbolTable` the type of variable `f` and verify if it is compatible with the type of the expression `tree(6,NIL,NIL)` (carried out as the value of the synthesized attribute `outType`.

4.2 Scope checking

`Scope checking`—concerned with the identifiers' visibility in nested program blocks[4]—is, once again, supported on the attribute `inSymbolTable` as it keeps all the information about variables and sub-programs (in this case, we will make use of the `scopeLevel` element of the value associated with each table key).

As referred in subsection 2.1, all identifiers (variables or sub-programs) introduced in the program Declaration part are global. So, their `scopeLevel` has the initial value of 0.

On entering a block (each time the begin of a sub-program is recognized) the `scopeLevel` is incremented; and it will be decremented on leaving the block (when the end of the sub-program is recognized).

This process, controlled via the `(in|out)Scope` attribute, is enough to allow the re-declaration of identifiers in nested block: although with the same name, they

---

of the tree root, it will be automatically displayed at the end of attribute evaluation process.

[4] Remember that LISS follows a Pascal-like nesting strategy.

have different `scopeLevel` values, and so they are not confused; the one with bigger value is the active one. Identifiers with a `scopeLevel` bigger then the `scopeLevel` of the present block are not accessible.

5 Code generation

As told in the previous sections, LissC translates source programs written in LISS into the Assembly language of its target machine—in this case, the virtual machine VM (see details below).

Defining schema to generate code for assignment or control statements, or even for atomic or array data types, is a trivial task; however, the other LISS data types demand an appropriate, non-usual, translation strategy. That strategy requires the definition of a non-standard memory allocation map (to hold conveniently those structured values) and an adequate code generation. After a brief introduction to the architecture and instruction set of the VM target machine, we illustrate the approach. For that purpose, we discuss, in the rest of the section, the translation schema to handle complex and polynomial data types. Those schema are specified again via an *attribute grammar* that uses the attributes `Code`, `Comments`, and `symbolTable` defined in section 3.2.

5.1 The Virtual Machine VM

VM [Fil06], a virtual stack-machine that supports integer and float numbers, is slightly different from the usual. It contains a special stack to save the internal registers to allow the return from functions, and two *heaps* for dynamic memory allocation—one generic and the other for character strings; it also has graphic primitives. Its architecture and instruction set make VM an appropriate choice as the target for LISS Compiler.

VM Architecture. VM has the following components:

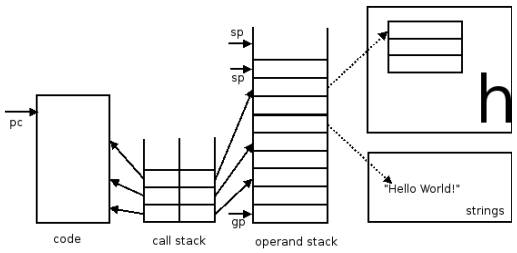
- 3 *stacks*: a **code stack** for the program (the sequence of operations to be executed); an **execution stack** for global variables, *function activation records* (with their parameters and local variables) and *working stack* (to store temporary values during computations); a **call stack** for the value of ¡*pc*,*fp*¿ registers (to control the return from the called function).
- 2 *heaps*: one for *strings*; and the other for generic structures.

Each *memory location* stores one value that can be a numbers or an address. An address can point to a location on the data/execution stack, code stack, or any of the heaps.

Three registers control the access to the different memory blocks allocated in the *execution stack*:

– The register `sp` (*stack pointer*) points to the first free cell in *working stack*.
– The register `fp` (*frame pointer*) points to the base address of local memory.
– The register `gp` (*global pointer*) contains the base address of global memory.

VM also has a register, `pc` (*program counter*), that every time points to the next instruction (in the code stack) to be fetched .
That architecture is depicted in the block diagram of figure 5.1.



VM Assembly Language. For the purpose of this paper, it is not worth-while to introduce the plain VM instruction set; we selected, an enumerate below, the most relevant operations:

– Integer Operators: `Add`, `Sub`, `Mul`, `Div`, `Mod`, `Not`, `Equal`, `Inf`, `Infeq`, `Sup`, `Supeq`;
– Data Manipulation: `PushI n`, `PushS n`, `PushG n`, `PushL n`, `LOAD n`, `STORE n`;
– Heap Operators: `Alloc n`, Free;
– Graphic Operators: `OpenDrawingArea`, `Refresh`, `Drawline`, `DrawCircle` e `DrawPolygon`;
– I/O Operators: `WriteI`, `WriteS`, `Read`;
– Control Flow Operators: `Jump`, `JZ`, `PushA`, `Call/Return`.

5.2 Translation Schema

To show how do we specify the translation rules for LISS Language with an *attribute grammar*, we discuss below code generation for 2 data types—`complex`, and `polynomial`—that are implemented following different strategies; for the second one, no Assembly code is generated. Actually, polynomials are treated at compiler time, as it happens with.

Complex. A *complex number* has a *real* and an *imaginary* parts; in our case, they are both integer numbers.
To store a complex number in the VM we chose the following *representation scheme*: an *heap* block with 2 cells, one for the real part and the other for the imaginary part. According to this memory map, all the operations over complex values will access the heap block and its 2 components.

We present below the *attribute grammar* fragment that describes the code generation for the memory allocation and initialization related with the declaration of a variable of type `complex`.

```
rule Complex {
 COMPLEX ::= REAL SIGN IMAGINARY compute {
 COMPLEX.outComments    = "//Line " + REAL.outLine + ": " +
                          REAL.outComments + IMAGINARY.outComments;
 COMPLEX.outCode        = Alloc(2) +
                          PushGP(getAddress(COMPLEX.inSymbolTable, COMPLEX.inNameVar)) +
                          REAL.outCode +
                          Store(0) +
                          PushGP(getAddress(COMPLEX.inSymbolTable, COMPLEX.inNameVar)) +
                          PUSHI(SIGN.outCode + IMAGINARY.outCode) +
                          Store(1);
 COMPLEX.outErrors       = REAL.outErrors + IMAGINARY.outErrors;
 IMAGINARY.inSymbolTable = COMPLEX.inSymbolTable;       };
}
rule Real {
 REAL ::= epsilon compute {
         REAL.outCode     = "";
         REAL.outComments = REAL.outErrors   = "";
         REAL.outLine     = epsilon.row();    }
     | EXPRESSION compute {
         REAL.outComments = CONSTANT.outComments;
         REAL.outErrors   = getTypeErrors(CONSTANT.type,"integer");
         REAL.outCode     = EXPRESSION.outCode;
         REAL.outLine     = EXPRESSION.row(); };
}
rule Imaginary {
 IMAGINARY ::= epsilon compute  { ... }
     | EXPRESSION 'i' compute   { ...  };
}
```

The code produced by the translation scheme above allocates 2 cells (`Alloc 2`)in the heap, leaving the new address on the top of the *stack*; then pushes onto the stack the real part and stores that value on the first position of the heap block (`Store(0)`); after that, executes the same process for the imaginary part.
The code generated to allocate a new complex variable, and initialize it with the value (5-7i) is shown below.

```
ALLOC 2
STOREG 0
PUSHG 0
PUSHI 5
STORE 0
PUSHG 0
PUSHI -7
STORE 1
```

Polynomial. Consider the following polynomials defined in LISS:

```
p = 6x^5 - 2 + 4x^3;
q = 2x^7 - 3x^5 + 8x^4 - x^7 + 1;
```

In this case, the polynomials could be represented in the VM memory using a strategy based on the heap, similar to the one discussed for `complex` type. Polynomials could be implemented as a sequence of structured blocks (one per monomial) in the heap with 2 cells, one for the coefficient and another for the degree.

That approach requires that we compute, at compilation time, the polynomial size (i.e., the number of monomials) because it is necessary to allocate all the blocks at the same time (using just one `Alloc` operation) to guarantee that they are continuous positions—it is a static allocation approach, similar to the one used for *arrays*; otherwise, it would be necessary to adopt a dynamic allocation approach (similar to *linked lists*), that requires an extra cell in each block to point the next monomial, and a much more difficult memory management algorithm (hard to code in Assembly). Choosing the static approach—the alternative that sounds easier, and the more feasible from a pedagogical point of view—we get in troubles to implement the arithmetic operations over polynomials, because the size of the result is different from the operands and variable (depends on their actual values), requiring extra code to compute the size at run time and allocate it dynamically.

Taking into account that *coefficient and degree are just constants*[5] (integer numbers) and that LISS Language *does not include an operation to compute the value of the polynomial at a given point*, we decided to handle variables of type `Polynomial` at compile time, precisely as we do with variables of type `Set`. So, no memory will be allocated in the VM for polynomials, and no Assembly code will be generated, except for the `write` operation.

For each variable of type `Polynomial`, we add to the respective entrance in the identifier table (represented by the attribute `symbolTable`) an extra field that is a linked list of pairs, where the first element is the degree of the monomial and the second one is its coefficient. Monomials are stored in the list sorted by degree in decreasing order; of course, the polynomial will be reduced to the canonical form during the parsing of the its value. Therefore, the representation for the 2 polynomials, p e q above, would be:

```
p = [(5,6),(3,4),(0,-2)]
q = [(7,1),(5,-3),(4,8),(0,1)]
```

The *attribute grammar* fragment below formalizes code generation for `Polynomial` data type in LISA, according to the translation scheme described:

```
rule PolynomialDefinition {
 POLYNOMIAL ::= MONOMIAL compute   {
      POLYNOMIAL.outCode        = MONOMIAL.outCode;
```

[5] Recall above the syntax, in subsection 2.2 at page 8.

```
      POLYNOMIAL.outComments    = MONOMIAL.outComments;
      POLYNOMIAL.outErrors      = MONOMIAL.outErrors;
      POLYNOMIAL.outSymbolTable = setExtraInfo(POLYNOMIAL.inSymbolTable,
                                               MONOMIAL.outCode);   }
  | POLYNOMIAL SIGN MONOMIAL compute   {
      MONOMIAL.inSign           = SIGN.outSign;
      POLYNOMIAL[0].outComments = POLYNOMIAL[1].outComments +
                                  SIGN.outComments +
                                  MONOMIAL.outComments;
      POLYNOMIAL[0].outErrors   = POLYNOMIAL[1].outErrors +
                                  SIGN.outErrors +
                                  MONOMIAL.outErrors;
      POLYNOMIAL[0].outCode     = mergePair(POLYNOMIAL[1].outCode,
                                            MONOMIAL.outCode);  };
}
rule Monomial {
     MONOMIAL ::= COEFFICIENT VARDEGREE compute {
     MONOMIAL.outComments = COEFFICIENTE.outComments +
                            VARDEGREE.outComments;
     MONOMIAL.outErrors   = COEFFICIENTE.outErrors +
                            VARDEGREE.outErrors;
     MONOMIAL.outCode     = makePair(MONOMIAL.inSign,
                                     COEFFICIENTE.outCode,
                                     VARDEGREE.outCode);     };
}
```

Concerning the 2 semantic rules above, we emphasize the auxiliary methods, `makePair` `mergePair` and `setExtraInfo`, responsible for the creation of a new pair degree,coefficient, for the merge of a pair into the existing list[6], and for the addition of that list to the `extraInfo` field of the identifier table. It is interesting to observe the use of an inherited attribute `Monomial.inSign` to carry down the coefficient sign.

Notice that the attribute `outCode`—of the JAVA type Object—is still used besides the fact that we are not generating Assembly code. The same attribute is used, however with a different semantics.

6 Conclusion

In this paper we discussed LISS Compiler: the imperative block-structured programming language, and its compilation into Assembly of a virtual stack machine. Such a language, equipped with an unusual and powerful type system (over integers), could be useful for teaching basic skills on programming, because it allows to deal with multiple concepts on data-structures and algorithmic principles. But that was not our point of view; instead we argued that this language imposes various challenges to those who want to implement it. So we think that it is a

[6] Necessary to reduce to the canonical form.

nice case-study for Compiler Courses or Compiler Development Projects. Additionally the size of its underlying grammar is reasonable to be taught during a one semester course. Both arguments make LISS a good learning instrument.

Actually a language that allows the programmer to use, as values of primitive types, linked lists (sequences) with a variable size, or sets with an infinite number of elements (once they are defined in comprehension), or complex numbers, polynomials, polygons or trees, is very convenient to show that this panoply do not requires a special processor; its implementation just needs some compilation tricks, based on memory allocation decisions and the appropriate code generation schemes.

As a second goal, we intended to show that, using a strategy to choose the attributes to associate with the context-free grammar symbols, an *attribute grammar* allows us described in a concise, coherent, systematic and clear way the syntax and semantics of the source language and its translation-scheme. We also emphasized the advantages of using a compiler generator like LISA, based on the AG specification.

The last but not the least goal, was to advocate the use of a virtual machine as a target-machine. We claimed that this approach has also a pedagogical value. In our case, we adopted the VM stack machine with an heap, which allowed us to implement the semantics of all our data types without difficulties.

References

[Cou84] B. Courcelle. Attribute grammars: Definitions, analysis of dependencies, proof methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.

[Cra06] Iain D. Craig. *Virtual Machines*. Springer Verlah, 1.st edition, Outubro 2006.

[DJL88] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars: Main results, existing systems and bibliography. In *LNCS 341*. Springer-Verlag, 1988.

[Dor75] Robert W. Doran. Architecture of Stack Machines. In Yaohan Chu, editor, *High-Level Language Computer Architectures*. Academic Press, 1975.

[Eng84] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.

[Fil06] Jean-Christophe Filliâtre. Machine virtuelle pour le project de compilation. Authors' Web Page, `http://www.lri.fr/~filliatr/index.fr.html`, 2006.

[Kas91] Uwe Kastens. Attribute grammar as a specification method. In H. Alblas and B. Melichar, editors, *Int. Summer School on Attribute Grammars, Applications and Systems*, pages 16–47. Springer-Verlag, Junho 1991. LNCS 545.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[MHK+02] Marjan Mernik, Pedro Henriques, Tomaz Kosar, Maria João Varanda, and Viljem Zumer. Object-oriented attribute grammar based grammatical approach to problem specification. Technical report, University of Minho, 2002.

[MLAZ00] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Compiler/interpreter generator system LISA. In *IEEE Proceedings of 33rd Hawaii International Conference on System Sciences*, 2000.

[MZ03] Marjan Mernik and Viljem Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, 2003.

[MZLA99] Marjan Mernik, Viljem Zumer, Mitja Lenic, and Enis Avdicausevic. Implementation of multiple attribute grammar inheritance in the tool lisa. *ACM SIGPLAN not.*, 34(6):68–75, Jun. 1999.

[Tie80] Martti Tienari. On the definition of an attribute grammar. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 408–414. Springer-Verlag, Janeiro 1980. LNCS 94.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1976.