DIE SCHRIFTART SMALL CAPS (KAPITÄLCHEN). Die Schriftart sans serif (serifenlos).

Inhaltsverzeichnis

TODO:

Welchen typen unterstuetzen wir Alle Ausdruecke werden bei uns in der Klasse Symboltable ueberprueft auf Korrektheit.

1 Parser

In our recursive-descent Parser we implemented all EBNF production rules as methods. The production rules of our EBNF allows alternatives, repetitions and optional parts. Figure (1) shows a dependence diagram of the parser methods.

To ensure that our EBNF is LL(1) conform, we created the a list with the first sets. The parser compares the actual token with the first-sets symbol and takes decisions between alternatives. To guarantee that the decision between two alternatives is correct, the first set of the alternatives must be disjunctive. We checked our EBNF with the Parser tool: coco, to ensure we are are LL(1) conform.

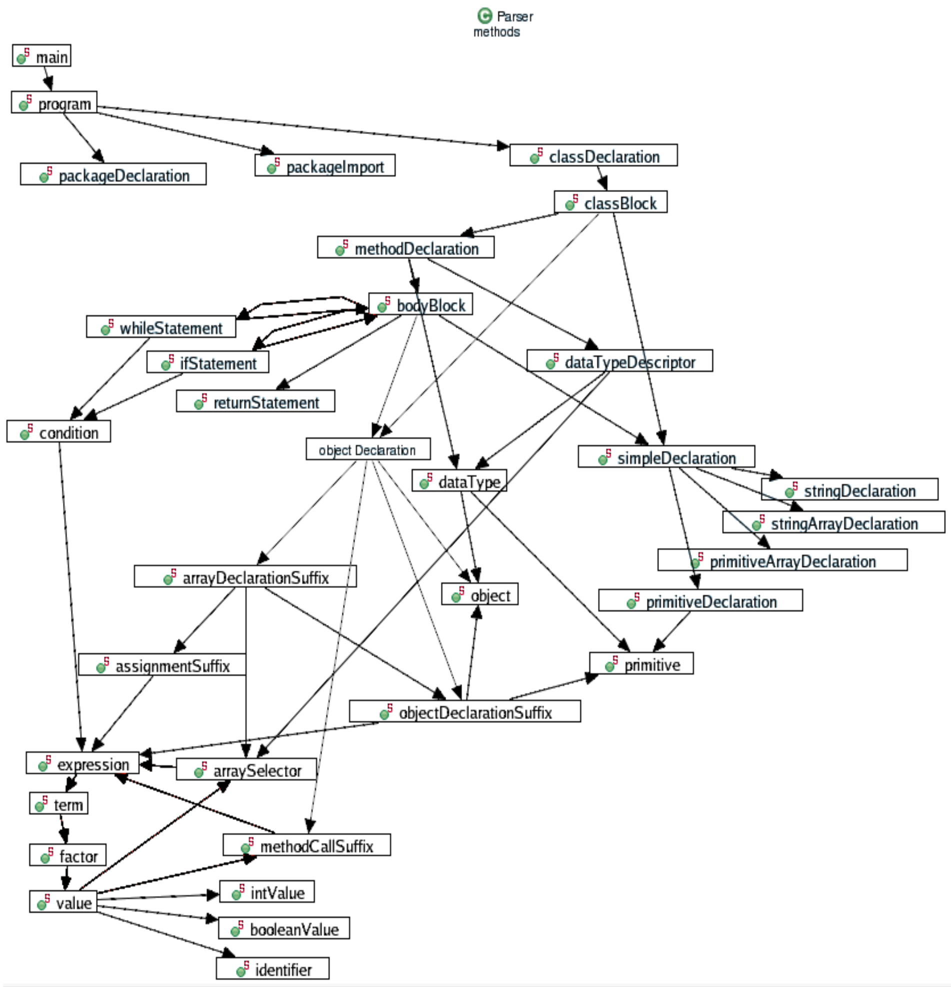

Abbildung 1: parser methods dependencies

1.1 First Symbols(first sets)

program	“package” “import” “public”
packageDeclaration	“package”
packageImport	“import”
classDeclaration	“public”
identifier	simpleIdentifier
classBlock	simpleIdentifier “public” “String” “String[]” “int” “boolean” “char” “int[]” “boolean[]” “char[]”
objectDeclaration	simpleIdentifier
simpleDeclaration	“String” “String[]” “int” “boolean” “char” “int[]” “boolean[]” “char[]”
methodDeclaration	“public”
object	simpleIdentifier
objectDeclarationSuffix	simpleIdentifier
primitiveDeclaration	“int” “boolean” “char”
primitiveArrayDeclaration	“int[]” “boolean[]” “char[]”
stringDeclaration	“String”
stringArrayDeclaration	SString[]"
datatype	simpleIdentifier SStringintbooleanchar"
datatypeDescriptor	simpleIdentifier SStringintbooleanchar"
bodyBlock	simpleIdentifier SStringString[]whileif returnintbooleancharint[]boolean[]char[]"
primitive	ïntbooleanchar"
assignmentSuffix	equal

2 Syntax Error Handling

The Compiler is able to detect a certain amount of Syntax Errors. We distinguish between two error levels : week and strong errors.

- **Week Errors** will be fixed by the parser.
- **Strong Errors** can not be fixed.

2.1 Week Errors (Warnings)

Week errors are missing tokens which will be inserted by the parser. The code generator never get notice about those errors. When a week error occurs a warning will be displaced, the token will be inserted, parsing and code generation continues. Compiler is able to detect and correct following week errors:

- any missing ";"
- any missing "){}]"
- near all missing "(ëxcept in an object declaration, because we have to distinguish between "[änd "("
- in class declarations a missing “class” will be fixed.
- in method declarations when return type is missing (TODO !!!)
- in method declarations when “static” is missing
- in parameterlists we will detect and correct missing ","

2.2 Strong Errors

Strong Errors can be syntax errors as well as invalid symbols or invalid words. When a strong error occurs the parser displaces a error message and the code generation will be stopped. The parser goes to the next strong symbol and continue parsing. In our parser implementation we have sync operation implemented in three methods. See Figure (1), marked methods (TODO) serves as synchronisation points. Synchronization works this way:

- one of the synchronization methods recognizes a strong error in one of its called method
- the error will be printed and code generation will stop
- the synchronization methods x fetches new tokens until the next “first set” token of method x is fetched

Strong Errors can be:

- misplaced tokens. When the current token does not correspond to EBNF rules, except of week errors.
- any missing identifiers.

- any illegal terminal symbols, like identifier starting with a digit. This kind of error detects the scanner, and a error token will be delivered to the parser.

  For example, if we had a production

  ```
  A = a b c.
  ```

  for which the input was

  ```
  a x c
  ```

  the parser reports

  ```
  TODO Message
  ```

3 Symboltable

The Parser produces for every field (global variables, methods) and local variables an entry in the Symboltable. The Symboltable is implemented as a Java Linked List. We distinguish between different scopes. The global scope and local scopes. Every method declaration has an entry in the gobal scope. The method's variables (local vars) are not in the same list (global scope), they are in a separate linked list which is an attribute of the methode. TODO FIGURE

The Symboltable is representet by the symboltable list (SymbolTableList.java) which consists of cells (SymbolTableCell.java). A cell consists of:

`String name`	Name of the entry (Variable name, Method name,...)
`ClassType classType`	kind of the entry: variable, array, procedure
`TypeDesc type`	data type: int, bool, char, String
`String value`	the value of the variable, when available
`SymbolTableList methodSymbols`	only methods and arrays have a sub list with symbol table cells
`int offset`	offset is negative
`int size`	in 4 bytes (word)

4 Type Savety

4.1 Naming convention

We ensure that variable names in a separate scope are unique. Local variables can have the same name as global variables.

4.2 Operator on incompatible Types

- will be ensured by EBNF
-

4.3 Typechecking

5 Memorymanagement in comPiler

ComPiler compiles code for RISC-based architectures. It has 32 registers each with size 32 Bit (4 Byte). Register 0 (R0) has always the value 0. Register 31 (R31) is reserved for return addresses. In a branch instruction the PC is stored in R31.

register 0
register 1 - 29
instruction register
return address

The instruction register (IR) holds the current instruction being executed.
The program counter (PC) contains the address of the instruction to be fetched next.
The stack pointer (SP) indicates the top element of a register-based stack. That means that on some occasions registers are accessed like a stack and the SP points out the next free register.
The memory for the activation frames is organized like a stack. Each frame is an entry. The SP indicates the next free memory and the FP the base address of the current frame.

5.0.1 Organization of an activation frame

An activation frame is a special memory context used for a procedure and its local variables. It's created by a branch instruction (e.g. a procedure call). The base address of the activation frame is also the base address for all local variables declared here. Because this base address is highly important and one needs to access it effiecently it's saved in the frame pointer (FP). As the former PC was saved at the branch instruction, we use it as our return address and save it in R31.

5.1 local variables

ComPiler offers local hiding of variables. That means, that procedures can contain variables that cannot be seen outside the procedure. Even if there exists a variable with the same name outside the procedure, these two don't interfere.

So what happens when a branch instruction occurs?

When the instruction is executed, the PC is stored in R31. Then the code generator jumps to the next instruction-address indicated by the branch instruction. Now we have entered a special memory context for procedures (an activation frame). In this frame all memory is managed that is needed in the procedure

Local variables have the following properties:

- They have negative offsets to their baseaddresses

6 The symbolfile

The symbolfile (file-extension is .sym)is a sequential representation of the symboltable. All entries are representations of the symbols found in the sourcecode.
Because the symbolfile is just read one time during compilation, it cannot be seen as a performance factor. So we decided to write the symbolfile in xml-format as it is a good representation of objects.

6.1 Structure

The root-tag of the symbolfile is *<symbolfile>*. After the xml-header and the root-tag the symbolfile starts with a list of the so-called *module anchors*. Every module anchor represents a module that's imported. In xml-syntax this lookes like that:

```
<modules>
        <module>
                <name> modulename 1 </name>
        </module>
        <module>
                <name> modulename 2 </name>
        </module>
        <module>
                <name> modulename 3 </name>
        </module>
</modules>
```

Listing 1: module anchors

After the module anchors the actual symboltable representation starts. It is enclosed in a *<symbols>*-tag There are 3 different elementtypes that can be described here:

- variables
- arrays
- methods

6.1.1 variables

A variable has 2 properties:

- one of the primitive datatypes like *boolean*, *int* or *char*
- the name of the variable in the sourcecode

```
<variable>
        <type> type </type>
        <name> name </name>
</variable>
```

Listing 2: variables

6.1.2 arrays

An array has 3 properties

- one of the primitive datatypes like *boolean*, *int* or *char*
- the name of the variable in the sourcecode
- the number of elements in the array

```
<array>
        <type> type </type>
        <name> name </name>
        <size> size </size>
</array>
```

Listing 3: arrays

6.1.3 methods

An array has 3 properties and contains another symboltable with the parameters of the method.

- the return value of the method as one of the primitive datatypes like *boolean*, *int* or *char*
- the name of the method in the sourcecode
- the number of elements in the array

So the description of a method is a recursive search through the symboltable.

```
<method>
        <type> type </type>
        <name> name </name>
        <size> size </size>
        <symbols>
                <variable>
                        <type> type </type>
                        <name> name </name>
                <variable>
        </symbols>
</method>
```

Listing 4: methods

Teil I
The linker

The linker is a intermediate step between processing the sourcecode and executing a binary. It combines objectfiles from the compiler to one binary for the

virtual machine.
A linker is needed if a compiler wants to provide *separate compilation*. That means that a program can import methods from a precompiled module. The program just needs a symbolfile to find out which methods the module offers.
But a problem arises when the compiler wants to create a binary from the program-sourcecode. The compiler knows that the called method exists in the module (from the symbolfile) but it doesn't know the content of that method.
So before the program can be executed the linker has to copy the relevant data from the affected objectfiles together into one binary. The objectfiles contain all the information the linker needs to fullfill that work.

7 The objectfile

The objectfile is a intermediate file created by the compiler, that already contains generated code and a lot of meta-information. The linker reads this objectfile and creates a binaryfile.
The reason for that intermediate step is, that if modules are imported the compiler doesn't have all the data he needs to create a valid and executable binary.
So the objectfile offers the needed information like the program entry point, which methods are loaded from modules and were to find that methods in these modules.
Based in that information the linker can link the compiled code in the objectfiles together and create one executable binary.

7.1 Structure

The structure of an objectfile is quite simple as one can see below. Its basic datasize is 32 bit. That means all data except character are represented and stored as 32 bit integers. A character is a 1-byte-value (8-bit).

magic word
branch instruction to main method
lenght of offset table
offset table
lenght of fixup table
fixup table
opCode

Abbildung 2: structure of an objectfile

7.1.1 magic word

The magic word identifies the objectfile. Its value has to be 0 represented by a 32 bit integer value.

7.1.2 branch instruction to main method

This is an integer-representation of the branch-instruction to the address of the main-method in this objectfile.

7.1.3 length of the offset table

The length of the offset table as a 32 bit integer value. The unit of this value is one 32 bit word.

7.1.4 offset table

In the offset table variable- or methodnames and their offsets in the current module are saved. That means that every exported module element stands in the offset table with its name and opCode offset in the modulefile.
The example in 3 shows the method *print*:

P	R	I	N
T	=	0	0
value			
...			

Abbildung 3: offset table

The bytes until the equal-sign form the name of the symbol. The next bytes are skipped so that only complete 32 bit words are read (in our example 2 bytes). The next 32 bit word is the offset of the instruction (opCode) for access to that element in the module. The length of the offset table (7.1.3) defines how often this operations are performed.

7.1.5 length of the fixup table

The length of the fixup table as a 32 bit integer value. The unit of this value is one 32 bit word.

7.1.6 fixup table

In the fixup table the imported symbols are listed. That means their modulename, their name and their offset in the module are stored.
The example in 4 shows a fixup table with the *print*-method from the module *Util*.

The bytes from the beginning to the first dot form the modulename. The following bytes to the equals-sign form the symbolname. The next bytes (in our example one byte) are skipped so that only 32 bit words are read.

U	T	I	L
.	P	R	I
N	T	=	0
value			
...			

Abbildung 4: fixup table

This reading-operation is performed until the number of words in the length of the fixup table (7.1.5) is read.

8 Processing of the objectfiles

As mentioned above the linker combines one or more objectfiles to one binary for execution in the virtual machine.

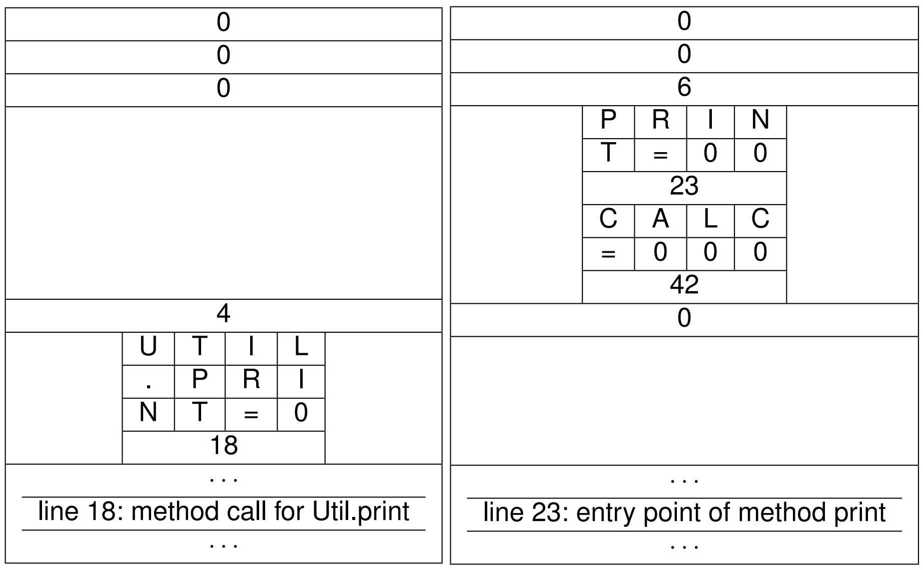

Abbildung 5: The objectfile of the main-module and the util-module

The example in 5 shows what data the linker reads from the objectfiles and how this data is processed.
The fixup table from the main-objectfile (left example) shows which information must be read from the offset table of the imported objectfile (util-objectfile) (right example). The offset-information from the fixup table determines the position in the opCode that needs to be updated (18 in our example). That means:
Line 18 of the opCode in the main-module is a call of a method in another module (module util, method print).
In the objectfile of that module (util) there is description where to find that method. This description lies in the offset table. So the linker knows now where

the command that has to be fixed in the main-module lies and what information it needs.
The information in the offset table is of course relative. How the linker creates an absolute position for that method will be described later.

9 The binaryfile

The binaryfile is an file that can be executed by the virtual machine. It hardly provides any meta-information but just executable code (opCode). This file is created when a linker links one or more objectfiles together.
The opCode is already created by the compiler and written into the objectfiles. The linker copies that opCode together into a binaryfile and fixes branchinformation in that opCode.
As we have heared before the opCode in the binary is addressed relatively. That means that the first line of code in a module can be the 100th line of code in the binary. So we need to fix that addressed, because otherwise branch-instructions would point to incorrect addresses.

9.1 Structure