COS341
Project Advice

Pre-reading: RecSPL Lang sheet

Attached below [PDF] is the language specification sheet for the small programming language **RecSPL** (version 2024), for which you will have to build your own compiler during this semester.

- **SPL** stands for "**S**tudents' **P**rogramming **L**anguage", and **Rec** stands for "**Rec**ursive": You see that in this year's version (2024) the SPL does *not* offer any WHILE commands for Loops, because all repetitions must be programmed purely recursively in RecSPL.

Please carefully study the attached language specification sheet together with your project partner to whom you were allocated in today's lecture (unless you have voluntarily chosen to do the project all alone by yourself).

- As you see, I have kept the "design" of RecSPL so simple that the "Lexing" of its tokens will not be difficult; and also its "Parsing" will be quite straightforward, too **:)**

Further instructions about **how to** build your own Compiler for the given RecSPL will follow soon in due course...

- At the end of the semester, you should be able to write a small RecSPL program, compile it with your own compiler, and let the generated output code actually run and "do stuff"; that will be a HAPPY DAY when you'll see that it really works **:)**

Attachment:

RecSPL_2024.pdf

As **students had detected** and rightly pointed out - for which I thank them - there's **a small bug in the project grammar**. *In fact I had posted the bugfix already last week (Thursday or Friday), but apparently the ClickUp system seems to have totally "swallowed" that posting from last week; it's nowhere visible any more.* Thus I'm re-doing the work and re-post it again now in this message :)
The **bug** is that the **non**-void functions were **not** given their necessary return statement. (Void-functions obviously do not need any such return statement.)

The **bugfix** is as follows: Into the project grammar you simply **insert** this one additional production rule:

COMMAND ::= **return** ATOMIC

Thereby, **return** is now a new terminal token (in fact a *reserved keyword*) **for which your Lexer must also be prepared** and adjusted.

This simple bugfix allows us now to write a return-statement into the final line of the ALGO of a non-void function :)

- **Comment:** Obviously this simple bugfix now also allows us to write the most silly return statements everywhere into an ALGO - even in the middle of the main program where does not make any sense at all - but we can still deal with this semantic problem in the semantic analysis phase of the compiler after the parser has produces a syntax tree.
- *"Normally" a function without return - or a return in the middle of main - would (should) be regarded as a syntax problem to be picked up already by the parser;* but to follow that route I

would be forced to completely rewrite the project grammar on which students have already started working; that would not be good. Therefore we will deal with wrongly placed return-commands as "semantic" problems in the semantic analysis phases after the parsing.

I hope this helps :)

This message shall answer for everybody a question which several students had already asked me individually during the recent days:

The semester-project's assessment value of 15 total maximal points [as per study-guide: page 7] is "composed" of the following sub-values, which take the different technical difficulties of the various sub-tasks into account:

- **Lexer = 1 Point**
- **Parser = 2 Points**
- **Scope-Analyser = 3 Points**
- **Type-Checker = 4 Points**
- **Code-Generator = 5 Points**

As the sub-tasks are getting increasingly more difficult, also the due rewards for doing these sub-tasks are getting progressively higher.

- **With "Distinction" being approximately 11 out of 15 points (precisely 11.25 = 15 * 0.75), you see that no "Distinction" can be obtained without Code-Generator: that's fair enough, as every proper compiler also needs a code-generator after all.**

For example: If some project students are making not more than only a Lexer and a Parser - i.e.: only Synax without any Semantics - then they cannot get more than maximally 1+2=3 Points for such work.

Part 1: Lexer

The **RecSPL input file**, which your Lexer software must analyse, will be given as a plain ***.txt** file; for the purpose of experimenting and testing you can easily create such ***.txt** files (containing some RecSPL program code) by yourself.

How you implement your Lexer - in which programming language, and with which methods and techniques - is *entirely up to your own choice,* as long as only the following **requirements** are met:

- IF the input ***.txt** file contains any **lexical errors** (which corresponds, in theory, to the underlying DFA getting 'stuck' in a non-accepting state), then your **Lexer** software must "throw" a reasonably understandable **Error-Message** back to the User.
- IF the input ***.txt** file does not contain any lexical errors, then your **Lexer** software must **create**, **write**, and **store** (as its output) an **XLM file** which the Parser can later use as its input.

In this manner, you can keep your implementations of Lexer and Parser completely separated from each other in different "phases" of your semester-project, because **the persistent XML file will serve as an *"offline bridge"*** *between Lexer and Parser.* This design approach is very "convenient" and will make your "project life" much easier (in comparison against having to implement one huge all-in-all Compiler software). *Thus you'll even be able to switch your computer off and on again between Lexing and Parsing* **:)**

Thereby, the "tokenized" **contents of the XML** file shall be structured as follows:

<TOKENSTREAM>

<TOK>

<ID>**1**</ID> *// comment: Each token has its own unique ID number*

<CLASS>*the token's class*</CLASS> *// comment: See an example for illustration below*

<WORD>**terminal characters for the Parser**</WORD> *// comment: See example below*

</TOK>

<TOK>

<ID>**2**</ID>

<CLASS>*the token's class*</CLASS>

<WORD>**terminal characters for the Parser**</WORD>

</TOK>

... etc ...

<TOK>

... etc ...

</TOK>

</TOKENSTREAM>

EXAMPLES:

<TOK>

<ID>**126**</ID>

<CLASS>**reserved_keyword**<CLASS> *// comment: The class corresponds to some **Accept-State** of the **DFA***

<WORD>**else**</WORD>

</TOK>

...

<TOK>

<ID>**496**</ID>

<CLASS>**N**<CLASS> *// comment: That is the token-class for **N**umbers, as per given Specification Sheet*

<WORD>56.7</WORD>

</TOK>

Part 2: Parser

On the internet you can find several useful websites for analysing and processing context-free grammars. Obiously you'll have to find those websites by yourself - I won't "feed" you with their exact URLs 😅

If you have found those websites by "clever browsing", then you can - for example - **enter the grammar into the website, press a button, and the website returns to you the complete SLR parse table** for that grammar 😀

For your semester project, you are **ALLOWED to use** the helpful service provided by such grammar websites. In this way, you can be reasonably sure that your Parse Table will be correct, and you won't loose valuable time by writing the Parse Table with pen and paper, (which can be very time-consuming with all the FIRST- and FOLLOW-computations, etc.)

Obviously you will still have to program your Parser Software on the basis of the Parse Table, but at least the Parse Table itself you can get "for free" 🙂

From a Software Engineering point of view, Parser generation is indeed one of the few areas in which the "dream" of fully automatic software creation from specification has become a reality. In our case, the "software specification" is a grammar, which a parser generator "eats" as input; then you press a button, and out comes as output the entire parser software for that grammar. You'd wish your COS301 Software Engineering Project would be that easy 😁

For your **COS341 Semester Project**, however, **you may NOT use such kind of Parser Generators** that make the entire parser for you. **For your Semester Project you must "hand-code" your Parser Software by yourself**; **ONLY the above-mentioned help-websites for grammar-analysis are allowed**.

You're near the point at which you'll have to come up with a **parser** that 'eats' the XML file that contains the tokens (from the lexer) and produces either a **syntax tree** or a "syntax error!" message.

For the same good reason as before, it is highly recommendable to **store the syntax tree permanently in a new XML file**, which can then later be "eaten" by the compiler's software modules for semantic analysis (such as type checking) and target code generation.

For this purpose I'm suggesting to you the following **XML format for the syntax tree**:

<ROOT>

<UNID>number</UNID> // comment: UNID means Unique Node ID number: a number which no other node has

<SYMB>startsymbol</SYMB> // comment: Root always contains the grammar's start symbol

<CHILDREN>

<ID>number</ID> // this number is the UNID of a child

... // etc... More children ...

<ID>number</ID>

</CHILDREN>

</ROOT>

<INNERNODES>

<IN> // comment: IN means Inner Node, between the Root and the Leafs

<PARENT>number</PARENT> // this number is the UNID of the parent

<UNID>number</UNID> // this number is the node's OWN UNID

<SYMB>nonterminal</SYMB> // comment: Inner nodes always contain some Nonterminal symbol of the grammar

<CHILDREN>

<ID>number</ID> // this number is the UNID of a child

... // etc... More children ...

<ID>number</ID>

</CHILDREN>

</IN>

... // etc... More inner nodes ...

<IN>

<PARENT>number</PARENT>

<UNID>number</UNID>

<SYMB>nonterminal</SYMB>

<CHILDREN>

<ID>number</ID>

... // etc... More children ...

<ID>number</ID>

</CHILDREN>

</IN>

</INNERNODES>

<LEAFNODES> // Here we have reached the level of the Token nodes that come from the Lexer

<LEAF>

<PARENT>number</PARENT> // this number is the UNID of the parent in the Tree

<UNID>number</UNID> // this number is the Leaf-node's OWN UNID in the Tree

<TERMINAL>

... // in here you copy&paste the XML Token from the lexer as Terminal Symbol !

</TERMINAL>

</LEAF>

... // etc... More Leaf Nodes ...

<LEAF>

<PARENT>number</PARENT>

<UNID>number</UNID>

<TERMINAL>

... // in here you copy&paste the XML Token from the lexer as Terminal Symbol !

</TERMINAL>

</LEAF>

</LEAFNODES>undefined</SYNTREE>

The unique node ID numbers can later be used as 'Foreign Keys' in the semantic Symbol Table (see Chapter #3)

Formatted without other rubbish:

<ROOT>

<UNID>number</UNID> // comment: UNID means Unique Node ID number: a number which no other node has

<SYMB>startsymbol</SYMB> // comment: Root always contains the grammar's start symbol

<CHILDREN>

<ID>number</ID> // this number is the UNID of a child

<ID>number</ID>

</CHILDREN>

</ROOT>

<INNERNODES>

<IN> // comment: IN means Inner Node, between the Root and the Leafs

<PARENT>number</PARENT> // this number is the UNID of the parent

<UNID>number</UNID> // this number is the node's OWN UNID

<SYMB>nonterminal</SYMB> // comment: Inner nodes always contain some Nonterminal symbol of the grammar

<CHILDREN>

<ID>number</ID> // this number is the UNID of a child

<ID>number</ID>

</CHILDREN>

</IN>

<IN>

<PARENT>number</PARENT>

<UNID>number</UNID>

<SYMB>nonterminal</SYMB>

<CHILDREN>

<ID>number</ID>

<ID>number</ID>

</CHILDREN>

```
        </IN>
</INNERNODES>
<LEAFNODES> // Here we have reached the level of the Token nodes that come from the Lexer
        <LEAF>
                <PARENT>number</PARENT> // this number is the UNID of the parent in the Tree
                <UNID>number</UNID> // this number is the Leaf-node's OWN UNID in the Tree
                <TERMINAL></TERMINAL>
        </LEAF>
        <LEAF>
                <PARENT>number</PARENT>
                <UNID>number</UNID>
                <TERMINAL></TERMINAL>
        </LEAF>
</LEAFNODES>undefined</SYNTREE>
```

Part 3: Scope-Analyser / Semantic Analysis

In this year's Semantics of the RecSPL, **everything is static** - i.e.: known at compilation time - i.e.: **static scoping**, as well as also **static typing**.

For this reason, a **simple Hash Table** - or a **simple Relational Database** - will suffice as Symbol Table; *it can be "thrown away" after target code generation* (and will thus not itself become part of the target code).

Since **your already-existing parser** has equiped each Tree Node with a unique Node ID, you will use these unique **IDs as "Foreign Keys"** in your Database or Hash Table **to "link" the Syntax Tree with the Symbol Table**.

A **Tree-Crawling-Algorithm** must be implemented that **"populates" the Symbol Table** with Semantic Information while "visiting" all the nodes of the Syntax Tree.

The **SEMANTIC RULES** for **Function Names** and for **Variable Names**, which your **Compiler's Semantic Analyser Component** must take into account, are given in the **PDF document attached below**.

Semantic-analysis.pdf

In our **semester project**, we want to give **system-internal unique new names** for user-defined variable-names and user-defined function-names **already in the scope-analysis-phase**, (NOT "mingled into" the later translation phase): That was the approach which I had recommended in the previous lecture, (in contrast to the approach presented by our textbook).

For this purpose, already **your scope-analyser module needs a sub-function that can generate ever-new identifiers** that were never used before.

Examples:

- Let there be two variables **X** in one scope (being the same entities to each other), and another two variables **X** in another scope (also being the same entities to each other but NOT to the afore-mentioned other two **X**), then your scope analyser would perhaps give the new names '**v136**' to the former two **X**, and perhaps '**v419**' to the latter two **X**.
- Let there be a **call** to some function **g**, which has a **g**-definition within the appropriate scope, then both the call to **g** as well as the definition of **g** itself could get an internal new name '**f561**' in its internal representation.

After the consistent re-naming of all user-defined names by unique internal new names, we need not worry about scope-borders any longer, as we can now simply treat each uniquely re-named variable AS IF IT WOULD BE a global variable 😀

And by giving all function-names unique new internal names, we will later be able to simply use a unique function-name as a unique GOTO-Label, to which we can "jump" without any ambiguity when we are "calling" a function 😀👍

Part 4: Type checker

In the **attachment** I've provided a somewhat abstract 'high-level' specification of a Type System which you shall use as the basis of the implementation of the type-checker module within your compiler for RecSPL. The document clearly shows the idea of '**attributed grammars**' whereby **semantic rules are affiliated with the syntax rules** (such as in Chapter #5 of our Textbook).

- **I hope** that my type system specification is consistent and does not contain too many 'bugs' 😅
- **IF** you spot any 'bugs' in my specification then you've got to fix them before you can actually program your type-checker software along those lines 😅

In the attached PDF you can see that the specification of the Type System contains **two** recursive procedures, which are *both working on the Syntax-Tree* which the Parser has already generated:

- ***typecheck*** is the main procedure and returns true/false depending on whether (or not) an analysed RecSPL program was correctly typed.
- ***typeof*** - also recursive - is an auxiliary procedure which 'helps' the *typecheck* procedure: This auxiliary procedure has access to the already existing *Symbol Table,* in which relevant type information gets stored, and *reports* type information *to* the *typecheck* procedure by way of various characters (such as '**n**' for the numeric type, '**v**' for the void type, '**t**' for the text type, etc...)

Type-Check_2024.pdf

Part 5: Code-Generator / Translation to Target Code

As you know from a foregoing communication, the Semester Project's **Translation Phase** has a total value of **5** Points.

- This Translation Phase as a whole is further divided into two Sub-Phases (**A**,**B**) with their own assessment points as follows.

Sub-Phase A [**2** Points]:

- Translation **from Syntax-Tree to the** non-executable **intermediate code of Chapter #6**.
- **You can already now BEGIN implement this sub-phase** with the knowledge that you already have: *a little bit of additional advice will be provided very soon...*

Sub-Phase B [**3** Points]:

- **Final** translation **from the intermediate code of Sub-Phase A**, **to executable code**, WITH a simulated Call-Stack **along the lines of Chapter #9**:
- *The syntax of the executable final target language, plus some further technical advice on the final translation, will be announced by Prof.G after Chapter #9 has been completely "treated" in our forthcoming Lectures.*

This arrangement will enable Students to possibly achieve a "**Distinction**" mark [with 12 out of 15 Points] for their Semester Project even without Sub-Phase B, **if only Sub-Phase A is correctly implemented** without any errors.

IMPORTANT REMINDER!

- Because our Tutors will need TWO WEEKS time to assess and mark all the many project submissions, projects cannot be submitted any later than **two weeks BEFORE** the official closing date of the semester**!**
- Official **Closing Date of the Semester** is the 7th of November: On that day I must provide the Head of Department with all Semester-Marks fully completed.
- Students will have to submit their Compilers **as "executables"**, such that our Tutors will not have to waste their precious scarce time with trying to compile students' Compilers' source code.

Further details concerning the submission of the final project results will be communicated in a separate announcement in due course.

*For those students who wish to participate in **Phase 5** (**code generation**) of the Semester Project:*

- *The **requirements specification** is attached below [as PDF].*

*Inside the requirements specification you will find a **split line** which separates **Sub-Phase** 5**a** from **Sub-Phase** 5**b**, such that this same PDF document can be used by those students who only want to participate in Sub-Phase **a**, as well as also by those students who want to participate in both Sub-Phase **a** and Sub-Phase **b**.*

Transl-advice_2024.pdf

*Attached below is the Task Specification Sheet for those students who wish to fully complete their Semester Project with the final Phase **5b**) and who thus aim for a "Distinction" mark for their projects.*

- ***Please note: A little bit of own "research and tinkering" is needed for Phase 5b)**, too. Thus the task sheet does not "spoon-feed" you with all the micro-details about how to do everything exactly step-by-step. From a "Distinction"-Student at 3rd-Year Level we can surely expect a little bit of creativity* 😀

Attachment:

- *Target-Code_BASIC.pdf*

And now: HAPPY CODING 😀👍

Part: Software Testing

From COS301 Software Engineering *you should (hopefully) know the importance of* ***Software Testing****. For the thorough* ***testing of Compilers*** *(which are also Software),* *the following* ***testing method*** *is highly recommendable:*

First you program a little ***auxiliary tool*** *which* ***internally "hard-codes" the given RecSPL Grammar*** *and which can automatically* ***"emit" all kinds of syntactically correct RecSPL programs*** *by way of ==> Forward Production from the Grammar's Start Symbol [as per Textbook Section 2.2: "Derivation"].*

Any such ***output from the auxiliary tool*** *you save into some* ****.txt*** *file, which will subsequently serve as INPUT for your Compiler.*

IF your auxiliary tool itself is correct, such that it emits ONLY syntactically correct RecSPL programs into the ****.txt*** *files, then your Compiler should not throw any "Syntax Error" alert when "eating" such a *.txt file as input. IF your compiler throws a "Syntax Error" alert on any correctly generated *.txt file, then there is something wrong with your Lexer and/or with your Parser.*

Next, you deliberately distort ("by hand") a file that was correctly generated by your auxiliary tool, such that the ***distorted file *.txt is no longer syntactically correct****. In Software Engineering Terminology, that is the principle of* ***Mutation Testing****. Now your Lexer and/or Parser* ***must*** *pick up the Syntax Error and* ***must*** *raise a "Syntax Error" alert: IF your Lexer or Parser would not pick up the "Mutation", then something is wrong with your Lexer and/or Parser.*

Also note: Because the above-mentioned ***auxiliary tool is a "pure" Syntax Tool,*** *it is completely* ***"blind" about whether it automatically generates Programs with or without Semantic Errors*** *(such as Type Errors, Scope Errors, and the like). Thus: for Testing the Semantic Components (e.g.: Type Checker, etc.) of your Compiler Software, you will first have to "inspect" the* ****.txt*** *files generated by your tool and - perhaps - modify them "by hand" to ensure that they are also free from Semantic Errors.*

Then, a syntactically correct input file without Semantic Errors should be translate-able to target code, whereas a syntactically correct input file with Semantic Errors should cause your Compiler's Semantic Analysis Component to raise a "Semantic Error" alert.

In general, ***a Compiler could possibly be wrong in two different ways****:*

- *Compiler is "****false positive****" wrong := It raises Error alerts* ***although*** *the* ***input*** *file was* ***good****.*
- *Compiler is "****false negative****" wrong := It does not raise any Error alerts* ***although*** *the* ***input*** *file was* ***bad****.*

When you are testing your compiler software along the above-mentioned lines, you must make sure that you will test for both "false positive" as well as also for "false negative" Software Defects:

- *That's also how the Tutors will test your Compiler Software, after Project Submission Deadline.*

In addition to my previous message, I'm here giving you a small **example of a producer program** (in C++) which is "hard-coding" **Grammar 2.4** of Textbook **Section 2.2** and which "emits" the strings which belong to the language of that grammar. That's the way in which you program an "Auxiliary Tool" which you can subsequently **use for automatically generating good Test Cases as test inputs** for your Compiler.

```
#include <iostream> // C++ String-Producer for Grammar 2.4 in Textbook Section 2.2

void produceR()
{
   int inputFromUser;
   std::cin>>inputFromUser;
   if(inputFromUser==1){ return; } // apply Rule 3 of Grammar 2.4
   else{ // apply Rule 4 of Grammar 2.4
      produceR();
      std::cout<<"b"; // this should actually be written out into a *.txt file
      produceR();
      return;
   }
};

void produceT()
{
   int inputFromUser;
   std::cin>>inputFromUser;
   if(inputFromUser==1){ // apply Rule 1 of Grammar 2.4
      produceR();
      return; }
   else{ // apply Rule 2 of Grammar 2.4
      std::cout<<"a"; // this should actually be written into a *.txt file
      produceT();
      std::cout<<"c"; // this should actually be written into a *.txt file
      return;
   }
};

int main()
{
   produceT();
   return 0; // here we should actually display the whole *.txt file
}
```