CSA1458- Compiler Design for sdd Capstone Project.

Title:

Parse Table Composition Separate Compilation and Binary Extensibility of Grammars

Guided By:

Dr.Gnanajayaraman
(Course Faculty)

SSE,SIMATS
ComputerScience&Engineering

Project by:

Mourya.P
(192211654)

SSE, SIMATS.

Aim:

The aim of this project is to implement a parsing table, a fundamental data structure in compiler design, to facilitate efficient parsing of input strings according to a given grammar.

Abstract:

Module Systems ,separate compilation ,deployment binary compo-
Nents ,anddynamiclinkinghaveenjoyedwideacceptanceinprogramminglan-

gauges and systems.In Contrast,thesyntaxoflanguagesisusuallydefinedina non-modular bay,cannot be compiled separately,cannot easily be combined with the syntax of other languages,and cannot be deployed as component for later composition.Grammarformalismsthatdosupportmodulesusewholeprogram compilation.
Current Extensible Compilers Focus On Source-level extensibility,which requires userstocompilethecompilerwithaspecificconfigurationofextensions.Acom-poundparserneedstobegeneratedforeverycombinationofextensions.Thegen-eration parse tablesis expensive,whichisaparticularproblemwhenthecom-positionconfigurationisnotfixedtoenableuserstochooselanguageextensions.
Inthispaperweintroducean Algorithm For *parse table composition* to support separate compilation grammar to *parse table components* .Parse Table Com-ponentscanbecomposed(linked)efficciently at runtime,i.e.just before parsing. Whiletheworst-casetimecomplexityofparsetablecompositionisexponential (like the complexity of parse table generation itself) ,forrealisticlanguagecombi-nationscenariosinvolvinggrammarsforreallanguages,ourparsetablecomposi-tionalgorithmisanorderofmagnitudefasterthancomputationoftheparse

Introduction:

The parsing table serves as a crucial component in the parsing process, enabling the efficient analysis of input strings based on a specified grammar. The project outlines the methodology employed in constructing the parsing table and discusses its significance in parsing algorithms. Additionally, the project presents experimental results demonstrating the effectiveness and performance of the implemented parsing table**.**

Literature View:

This section will review existing literature and research on parsing tables, parsing algorithms, and their applications in compiler construction. It will include an analysis of different parsing techniques and their relative advantages and limitations.

1. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific lan

guage prototyping. In: 35th Annual Hawaii International Conference on System Sciences

(HICSS'02), Washington, DC, USA, IEEE Computer Society Press (2002) 282.

2.Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA '07, ACM Press

(2007) 1–18.

3.van Wyk, E., Bodin, D., Huntington, P.: Adding syntax and static analysis to libraries via.

About:

Parsers are typically components of other software rather than a complete piece of software in their own right. They are used to break up structured text into meaningful trees of information that can be processed by other software. Parsers are probably most commonly used as part of a compiler. They can take textual data like a C# program or a C++ program and turn it into a tree of data based on the grammar of the language.

Parsers are also used to process Internet based textual data like JSON, XML and HTML. Your web browser uses a parser so it can turn an HTML document into its corresponding layout and display elements and the JavaScript into runnable code. Your e-commerce website uses a parser to turn

XML AJAX data into database information. Parsers are fundamental to just about everything you do on the Internet, but they exist behind other software.

In this article, we are going to explore the LL(1) parsing algorithm used to create simple generated parsers.

After scouring the Internet looking for alternatives to The Dragon Book (a daunting read), I found out there really isn't a complete tutorial on building parsers available online. All the information is there, but scattered, confusing, and sometimes contradictory. My hope is that future coders looking to develop parser generators can use this to get their feet under them and understand the key concepts and how to put them together.

The goal is not to produce a parser generator, but to show how the algorithm works. To that end, we will be making a runtime parser. We won't be making a fast parser, or a full featured parser, but we will lay the groundwork for making that happen, with simple to understand code.

Composition of LR(0) Parse Tables

We discussed the -NFA variation of the LR(0) parse table generation algorithm to in

troduce the ingredients of parse table composition. LR(0) -NFA's are much easier to

compose than LR(0) DFA's. A naive solution to composing parse tables would be to only

construct -NFA's for every grammar at parse table generation-time and at composition time merge all the station states of the -NFA’s and run the subset construction algorithm.

Unfortunately, this will not be very efficient because subset construction is the expensive

part of the LR(0) parse table generation algorithm. The -NFA’s are in fact not much

more than a different representation of a grammar, comparable to a syntax diagram.

The key to an efficient solution is to apply the DFA conversion to the individual parse

table components at generation-time, but also preserve the -transitions as metadata in the

resulting automaton, which we refer to as an -DFA. The -transitions of the -DFA can

be ignored as long as the automaton is not modified, hence the name -DFA, though there

is no such thing as a deterministic automaton with -transitions in automata theory. The

-transitions provide the information necessary for reconstructing a correct DFA using

subset construction if states (corresponding to new productions) are added to the -DFA.

In this way the DFA is computed per component, but the subset construction can be rerunpartially where necessary. The amount of subset reconstruction can be reduced by making use of information on the nonterminals that overlap between parse table components.

Also, due to the subset construction applied to each component, many states are already

part of the set of -NFA states that corresponds to a DFA state. These states do not have

to be added to a subset again.

Background

LL(1) parsing is arguably the simplest form of context-free parsing. It parses a document top-down and creates a left-derivation tree from the resulting input. It works almost identically to recursive-descent parsing.

The chief advantage it has over recursive-descent parsing is that it's slightly easier for a machine to generate an LL(1) table than it is to produce recursive functions. In practice, it is generally faster to use tables than a bunch of recursive methods. Recursive-descent lends itself well to hand written parsers. The performance of each vary due to different advantages and disadvantages (they're more efficient in some areas than others) but they are very close, in practice, so the differences in relative performance are little more than academic.

Regardless of underlying algorithm, in order for any parser to work, it must have some kind of grammar. Almost everything we produce for parsing will be generated from this grammar. The grammar describes the "language" we are looking to parse. We're only looking to parse context-free languages so far, so we will need a context-free grammar, or CFG.

(Introduction expanded as per a discussion with David O'Neil in the comments.)

Let's start with a run down of the parsing process. We'll explore this small C fragments.

Above CFG grammars were alluded to but grammars themselves typically don't understand individual characters in the input stream. They work with tokens, which are small text fragments - lexemes essentially, with an associated symbol/label attached to them.

The above might be broken down into these tokens:

```
keyword: int

(whitespace)

identifier: main

lparen: (

rparen: )

(whitespace)

lbrace: {

(whitespace)

identifier: printf
lparen: (

string: "Hello World"

rparen: )

semi: ;

(whitespace)

keyword: return

(whitespace)

int: 0

semi: ;

(whitespace)

rbrace: }
```

Note on the left hand side of the colon is the symbol "attached" to the fragment of text on the right hand side of the colon on each line.

The grammar won't care what the value is. It only cares about the left hand side parts - the symbols. This is what the grammar sees:

```
keyword,identifier,lparen,rparen,lbrace,identifier,

lparen,string,rparen,semi,keyword,int,semi,rbrace
```

These are called terminal symbols, or terminals. They are leaves of our parse tree. We will be creating these from input text in lesson 2. Non-terminals are the nodes in the tree. We use them to impose a hierarchy on these lists of terminals we get from the input.

For now, we are going to focus on the CFG grammar itself. We'll leave the terminals as "abstract" until we get to the next lesson.

At its simplest, a CFG is a series of rules. The rules themselves are composed of elements called symbols which can either be "terminal" or "non-terminal". We'll be using lowercase to represent terminals and uppercase to represent non-terminals but in practice it doesn't matter. This is simply for clarity.

A rule takes the form of:

```
A -> B c
```

Here, A is the left hand side, and B c are two symbols (non-terminal and terminal respectively) appearing on the right hand side. A symbol appearing on the left-hand side of any rule is automatically a non-terminal. Since all of ours are uppercase, only uppercase words will appear on the left hand side of any rule.

We can have multiple rules with the same left hand side.

```
A -> b
```

```
A -> c
```

This is sometimes written as the shorthand:

```
A -> b | c
```

While the left hand side is called the Non-terminal, the right hand side is called a derivation of a rule.

When we have multiple derivations with the same left hand non-terminal, it means the rule can be resolved in multiple ways. It's kind of like a choice. The above might be spoken as "A is derived as the terminal b or c."

When we have multiple symbols appearing on the right hand side/derivation as in the first example, that represents a sequence of symbols, such as "A is derived as the non-terminal B followed by the terminal c".

A rule can be empty, or nil, which means it has no right hand side such as:

```
A ->
```

This is usually useful for combining it with other A rules in order to make them optional such as:

```
A -> c
```

```
A ->
```

Which indicates that A may be derived as the terminal c or nothing.

Rules can be recursive, such that the left hand side is referenced within its derivation on the right hand side. This creates a kind of "loop" in the grammar, which is useful for describing repeating constructs, such as argument lists in function calls.

```
A -> c A
```

The only issue with LL - including LL(1) - is that a rule cannot be left recursive or it will create an infinite loop, such that the following is not allowed:

```
A -> A c
```

There are workarounds for this which you must do in order for the grammar to be valid for an LL(1) parser. Massaging the grammar to work with an LL(1) parser is known as *factoring*. There are even ways to do it programatically which are beyond the scope of this tutorial. However, even with left-factoring and left recursion elimination, not all grammars can be parsed with LL(1). It's best for simple languages.

The series of rules taken together make up the grammar, and define an ovararching outline/heirarchy for our language.

The structure and form of the non-terminals are defined by the rules. The terminals are not yet defined - only referenced in the rules. Defining the terminals will be covered in the next lesson, since the approach for defining terminals is different.

The grammar need only refer to symbols by their handle. It needs no other direct information about them other than what is provided by the rules. In our code, we will simply be using strings to represent the symbols, but they can be integers, or guids or whatever your heart desires, as long as the values act as unique handles.

That leads to a pretty simple data structure for a CFG overall.

A rule has a string field for the left hand side, and a list of zero or more strings for the right hand side/derivation.

A grammar meanwhile, is just a list of these rules.

The clever bit comes in what we do with them. Roughly, there are five major steps we need to perform.

The first thing we need to do is know how to determine which symbols are non-terminals and which are terminals even though they're all just strings. That's pretty easy. Anything that appears on the left hand side of any rule is a non-terminal. All the rest of the symbols referenced by the rules are terminal. Remember that while non-terminals are defined by these rules, the terminals are defined elsewhere. Right now, they're "abstract".

The second thing we need to do is conceptually "augment" the grammar by adding two reserved terminals, "#EOS" (often represented as "$" in other tutorials) which is the end of input stream marker, and "#ERROR" which indicates an unrecognized bit of input in the stream. These have no user supplied definition. They are produced while reading the input as necessary and consumed by the parser.

The third thing we need to do is be able to compute the PREDICT/FIRSTS sets. FIRSTS are the terminals that can appear at the beginning of each non-terminal. If you look at any grammar, you can see the first right hand side symbol (or empty/nil) - symbol. That's what we need. However, if it's a non-terminal, we need to get *its* FIRSTS and so on. In the end, each non-terminal should have an exhaustive list of the terminals that appear as the first right hand side, and nil if there is an empty/nil rule as shown previously.

The PREDICT sets are the same as the FIRSTS sets except they also have the rule associated with each terminal so you know where it originated. In practice, since the PREDICT sets contain all the information the FIRSTS sets contain, it's best to forgo the computation of the FIRSTS sets directly, and simply compute the PREDICT sets.

The fourth thing we need to do is compute the FOLLOWS sets. The FOLLOWS sets are the terminals that can appear directly *after* a given non-terminal. This is trickier than computing FIRSTS and PREDICT sets. You have to scan the grammar looking for other rules that reference your non-terminals in order to find what can follow them. We'll be getting into that below with our example.

Finally, the fifth thing we need to do is create the parse table. Fortunately, once we have the PREDICT and FOLLOWS sets, this step is trivial.

In this lesson, we will be declaring the classes for a CFG and performing these five steps.

Runthrough

Taking what we've outlined above, let's work through it conceptually with a specific example. We will be using the same example as Andrew Begel's excellent work so that can be referred to as well. If it were only complete, I would not be writing this tutorial, but what is there is wonderful for getting started.

Consider the following grammar:

`E -> E + T | T`

`T -> T * F | F`

`F -> ( E ) | int`

or its longhand equivalent:

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> int

This represents a grammar for a simple arithmetic language that supports *, + and ( ) operations.

The non-terminals are E, T, and F.

The terminals are +, *, (,) and int.

The grammar is also left-recursive, which as mentioned earlier, will not do. We're going to manually refactor it and eliminate the left recursion.

Take a look at the first two rules (shorthand):

E -> E + T | T

Here, for each rule where the non-terminal on the left (E) of the arrow is the same as the first part of the right-hand side of the arrow (E + T), we take the part of the rule without the E (+T) and remove it and add it into its own new rule which we'll call E':

E' -> + T

Now, after each of the new rules, add E' to the end:

```
E' -> + T E'
```

Now add an extra rule that is empty/nil:

```
E' -> + T E'

E' ->
```

Now we must fix the original E rules. To do so, we take all of the right-hand sides that *didn't* start with E, and add E' to the end of them:

```
E  -> T E'
```

If we do the same with T, we get the following grammar:

```
E  -> T E'

E' -> + T E' |

T  -> F T'

T' -> * F T' |

F  -> ( E ) | int
```

or longhand:

```
E  -> T E'

E' -> + T E'

E' ->

T  -> F T'

T' -> * F T'

T' ->
```

F -> (E)

F -> int

Note that we didn't do anything to F because it wasn't left-recursive.

Above is the modified grammar we'll be using and referring to so keep that in mind.

Now we can move on to computing the FIRSTS sets (or more generally, the PREDICT sets, but we'll be looking at FIRSTS in particular)

As before, FIRSTS are the first terminals that can appear for any non-terminal, while a PREDICT is the same but with the rule that each of the FIRSTS came from.

Let's start with an easy one, **FIRSTS(**F**)** and **PREDICT(**F**)** - the firsts and predict tables for F.

We see F-> (E) so we can say that (is one of the FIRSTS for F, and one of the PREDICT entries is **(** F-> (E)**,**(**)**

We also see F-> int so we can say that int is one of the FIRSTS for F and one of the PREDICT entries is **(** F-> int**,**int **)**

We're done with F, which has two entries. From now on, we'll only list the FIRSTS sets for readability. The PREDICT sets can be implied from the originating rule like above.

FIRSTS(E) = { (, int }
FIRSTS(E') = { +, ***nil*** }
FIRSTS(T) = { (, int }

FIRSTS(T') = { *, ***nil*** }
and finally, the one we already did:

FIRSTS(F) = { (, int }

Good. Now moving on to the FOLLOWS sets:

FOLLOWS() lets us know the terminals that can come *after* a non-terminal. This is *not* the final terminal in a non-terminal. It's the ones that can follow it. It's dicey to compute because you have to look at all the rules where a non-terminal can appear - that is, examine all of the derivations that contain the non-terminal in order to figure out what can follow it, and even then, it's not quite that simple. There is a non-obvious case having to do with nil rules.

So now we find every place our non-terminal is located on the right side of *any* of the arrows, as stated above, looking for anything that follows it. When we find a non-terminal following it, we take the FIRSTS and use those. When we find a terminal following it, we use that. We also have to handle nil/empty rules specially, as well as non-terminals that appear on the righthandmost part of a derivation.

Before we begin, we must virtually augment the grammar with a special starting rule that includes the #EOS (end of stream) terminal after our start symbol. Here, this rule would be:

S -> E #EOS

This is important so that we can tell when we've reached the end of our parse.

Now that we've augmented the grammar for the purposes of computing the FOLLOWS, we can move on.

Let's work out **FOLLOWS**(E):

Obviously, #EOS is one based on the augmented rule above. Look through all the rules in the grammar and see if E appears anywhere else. It does. It appears under one of the F rules. What follows it is the terminal), so we add that. We're done with E.

Now let's do **FOLLOWS**(E'):

This one is trickier. Nothing follows it directly in the grammar. It appears at the end of a couple of rules though, and remember we need to deal with that. For this, we need to examine the rules where it was referenced:

```
E -> T E'
```

We can see that it came from E, so if you think about it, what follows E also follows E'.

```
E' -> + T E'
```

For this one, it's the same thing, except as Andrew Begel points out, it's a tautology: What follows E' follows E' so we ignore it.

On to **FOLLOWS**(T):

Easy enough. T is followed by E'. Ergo, whichever terminals begin E' must be the terminals that follow T.

In other words, **FOLLOWS**(T) = **FIRSTS**(E')

But wait, there's a nil in there. Remember when I said nil rules need special handling? Well, here we are.

We can't have that nil that came from the FIRSTS set. Since it came from **FIRSTS**(E'), then whatever **FOLLOWS**(E') also follows this, because of our optional rule. Trust me, if you do the math, it works out this way. Remember that whatever FIRSTS it came from is whatever FOLLOWS it will have.

Applying those principles, let's do them all:

FOLLOWS(E) = { #EOS,) }
FOLLOWS(E') = { #EOS,) }
FOLLOWS(T) = { +, #EOS,) }
FOLLOWS(T') = { +, #EOS,) }
FOLLOWS(F) = { *, +, #EOS,) }

We're done with that step.

Now finally, something easy. Let's construct our parse table:

We have one column for each terminal, and one row for each non-terminal. In each cell we have a rule, or empty.

	+	*	(	)	int	#EOS
E						
E'						
T						
T'						
F						

Our parse table will look roughly like this.

Now, we need to fill in the rules. This is what our PREDICT sets we generated earlier are for.

For each row, we get the PREDICT for that row, which is the set of terminals and the rule associated with the row's non-terminal. Let's do E:

remember **FIRSTS**(E) = { (, int }

And the predicts for that are both rule E-> T E'

Now our table looks like:

	+	*	(	)	int	#EOS
E			E-> T E'		E-> T E'	
E'						
T						
T'						
F						

Let's do E' now:

FIRSTS(E') = { +, ***nil*** }

Uh oh, there's no nil in the table. What do we do now?

This is where our FOLLOWS sets come in. Any time we see a nil, we take the FOLLOWS of that non-terminal row and we use its terminals to tell us the columns to use. The rule we use comes from the PREDICT with the nil in it, so it will be an empty/nil rule. Below, we take the follows so we can see where to place our rules.

FOLLOWS(E') = { #EOS,) }

Filling in the next row of the table leaves us with:

	+	*	(	)	int	#EOS
E			E-> T E'		E-> T E'	
E'	E' -> + T E'			E' ->		E' ->
T						
T'						
F						

Now let's fill in the rest of the rows:

	+	*	(	)	int	#EOS
E			E-> T E'		E-> T E'	
E'	E' -> + T E'			E' ->		E' ->
T			T-> F T'		T-> F T'	
T'	T'->	T'-> * F T'		T'->		T'->
F			F-> (E)		F-> int	

Whew. We're done!

Coding this Mess

Now that we've explored the process conceptually, let's code it. The included project contains all 3 lessons but we're going to be focusing on the contents of the Lesson 1 folder. Please keep the source handy since we will be referring to it here sometimes only briefly. We'll be going over the 5 steps again, and this time linking them to the various bits of code.

CfgRule is a class that defines a single grammar rule, like A -> B c

In practice, it's a good idea to make your rule class implement value semantics for equality comparisons, so that rules can be used as keys in dictionaries and duplicate rules can be found, etc. The class contained therein implements value semantics.

Cfg is a class that defines a collection of rules. The Rules collection is the primary driver here.

Let's declare our grammar from above. You can see below we have to do it longhand.

C#
```
var cfg = new Cfg();

// E -> T E'

cfg.Rules.Add(new CfgRule("E", "T", "E'"));

// E' -> + T E'

cfg.Rules.Add(new CfgRule("E'", "+", "T", "E'"));
```

```
// E' ->
cfg.Rules.Add(new CfgRule("E'"));
// T -> F T'
cfg.Rules.Add(new CfgRule("T", "F", "T'"));
// T' -> * F T'
cfg.Rules.Add(new CfgRule("T'", "*", "F", "T'"));
// T' ->
cfg.Rules.Add(new CfgRule("T'"));
// F -> ( E )
cfg.Rules.Add(new CfgRule("F", "(", "E", ")"));
// F -> int
cfg.Rules.Add(new CfgRule("F", "int"));
```

Next, let's explore how our steps map out, but before we do, I want to make small note about collections which are used a lot in this code:

I typically use a FillXXXX pattern to return collections where most might use a GetXXXX method or a property.

Doing this is slightly more flexible since you can act on an existing collection. All of the FillXXXX methods take an optional result parameter. If it's not passed, or if it's null, a new collection will be created to be filled. In either case, the result is the return value from the method. So to call it like a GetXXXX method, you just omit the result parameter like foo = FillXXXX();

Briefly, let's revisit our five steps:

1. Distinguish terminal from non-terminal symbols
2. Augment the grammar with #EOS and #ERROR terminals
3. Compute the FIRSTS/PREDICT sets
4. Compute the FOLLOWS sets
5. Generate the Parse Table

Let's explore the Cfg class, which represents our context-free grammar.

Step 1 is handled by FillNonTerminals(), FillTerminals(), and FillSymbols(), each with a corresponding _EnumXXXX() private method which allows lazy enumeration of the same.

Step 2 is handled by FillTerminals() and _EnumTerminals() automatically.

- FillSymbols() returns all the symbols in the grammar, both non-terminal and terminal, returning non-terminals first, followed by terminals.
- FillNonTerminals() returns all the non-terminals in the grammar.
- FillTerminals() returns all the terminals in the grammar, with the reserved #EOS and #ERROR terminals being the final terminals in the collection.

As long as the Rules collection does not change, these symbols will always be returned in the same order - non-terminals first, followed by terminals, then reserved terminals, but otherwise in the order they appear in the grammar.

Step 3 is handled by FillPredicts() which fills a dictionary keyed by non-terminal that contains collections of tuples where each tuple is a rule and either a terminal symbol or nil.

Step 4 is handled by FillFollows() which fills a dictionary keyed by non-terminal that contains collections of terminal symbols.

Step 5 is handled by ToParseTable() since we can't fill existing parse tables anyway, it breaks the pattern. It returns a nested dictionary, the outer keyed by non-terminal symbol, the inner keyed by terminal symbol, with the inner's value being the rule. This represents our parse table from above.

And that's really all there is to it so far. Feel free to run and poke at the code, but in this lesson, there's not a lot of output to demonstrate - it's all just the carriage to get the rest of the parser working.

When it starts coming together in lesson 3 and we revisit the CFG, the rest will fall into place, but first we get to explore lexing/tokenizing in lesson 2, where we build a tiny, remedial regular expression engine.

System Requirements:

- Programming Language: [Specify programming language]
- Development Environment: [Specify IDE or tools]
- Memory: Sufficient memory for handling parsing table data structures and input strings.
- Processor: Standard processor capable of running the chosen programming language and development environment.

Hardware Requirements:

- ☐ Processor: A modern processor (e.g., Intel Core i5/i7, AMD Ryzen) for better performance.
- ☐ RAM: At least 4GB RAM, but preferably 8GB or more for smooth operation.

- ☐ Storage: Sufficient storage space (at least a few gigabytes) for the operating system, development tools, and project files.
- ☐ Compiler: Depending on the programming language you're using, you need an appropriate compiler. For example:
- ☐ C/C++ Compiler: GCC (GNU Compiler Collection), Clang.
- ☐ Text Editor or Integrated Development Environment (IDE): Choose a text editor or IDE for writing and editing your code. Popular options include Visual Studio Code, Sublime Text, Atom, or IDEs like JetBrains CLion for C/C++ development.

Existing System:

This section will discuss the current state of parsing tables, highlighting any limitations or challenges faced in existing implementations. It may also touch upon commonly used parsing algorithms and their reliance on parsing tables.

1. Parser Generator Tools: Such as Yacc, Bison, ANTLR, JavaCC, and others, which automate the generation of parsing tables and parser code from a formal grammar specification.
2. Programming Languages and Libraries: Developers may use general-purpose programming languages such as C, C++, Java, Python, etc., Online Resources and Communities: Websites, forums, and communities focused on compiler design, parsing techniques, and language processing provide valuable resources, tutorials, and discussions related to parsing tables and parsers.

Extensible Parsing Algorithms: For almost every single parsing algorithm extensible
variants have already been proposed. What distinguishes our work from all the existing
work is the idea of separately compiled parse table components and a solid foundation

on finite automata for combining these parse table components. The close relation of our
principles to the LR parser generation algorithm makes our method easy to comprehend
and optimize. All other methods focus on adding productions to an existing parse table,
motivated by applications such as interactive grammar development. However, for the
application in extensible compilers we do not need incremental but *compositional* parser
generation. In this way, a language extension can be compiled, checked, and deployed
independently of the base language in which it will be used. Next, we discuss a few of
the related approaches.

Horspool's [20] method for incremental generation of LR parsers is most related to
our parse table composition method. Horspool presents methods for adding and deleting
productions from LR(0), SLR, as well as LALR(1) parse tables. The work is motivated by
the need for efficient grammar debugging and interactive grammar development, where
it is natural to focus on addition and deletion of productions instead of parse table com
ponents. Interactive grammar development requires the (possibly incomplete) grammar
to be able to parse inputs all the time, which somewhat complicates the method. For SLR

follow sets Horspool uses an incremental transitive closure algorithm based on a ma
trix representation of the first and follow relations. In our experience, the matrix is very
sparse, therefore we use Digraph. This could be done incrementally as well, but due to
the very limited amount of time spend on the follow sets, it is hard to make a substantial
difference.
IPG [21] is a lazy and incremental parser generator targeting a GLR parser using
LR(0) parse tables. This work was motivated by interactive metaprogramming environ
ments. The parse table is generated by need during the parsing process. IPG can deal with
modifications of the grammar as a result of the addition or deletion of rules by resetting
states so that they will be reconstructed using the lazy parser generator. Rekers [18] also
proposed a method for generating a single parse table for a set of languages and restrict
ing the parse table for parsing specific languages. This method is not applicable to our
applications, since the syntax extensions are not a fixed set and typically provided by
other parties.
Dypgen [15] is a GLR self-extensible parser generator focusing on scoped modi-
fication of the grammar from its semantic actions. On modification of the grammar it

generates a new LR(0) automaton. Dypgen is currently being extended by its developers
to incorporate our algorithm for runtime extensibility. *Earley* [42] parsers work directly
on the productions of a context-free grammar at parse-time. Because of this the Earley
algorithm is relatively easy to extend to an extensible parser [43, 44]. Due to the lack of
a generation phase, Earley parsers are less efficient than GLR parsers for programming
languages that are close to LR. *Maya* [45] uses LALR for providing extensible syntax
but regenerates the automaton from scratch for every extension. *Cardelli*'s [22] exten
sible syntax uses an extensible LL(1) parser. Camlp4 [46] is a preprocessor for OCaml
using an extensible top down recursive descent parser.*Automata Theory and Applications.* The egrep pattern matching tool uses a DFA for ef-
ficient matching in combination with lazy state construction to avoid the initial overhead
of constructing a DFA. egrep determines the transitions of the DFA only when they are
actually needed at runtime. Conceptually, this is related to lazy parse table construction
in IPG. It might be an interesting experiment to apply our subset reconstruction in such
a lazy way. Essentially, parse table composition is a DFA maintenance problem. Surpris

ingly, while there has been a lot of work in the maintenance of transitive closures, we
have have not been able to find existing work on DFA maintenance.

Proposed System:

The proposed system will introduce an improved parsing table implementation, addressing any shortcomings identified in existing systems. It will detail the design and construction of the parsing table, along with its integration with parsing algorithms for efficient syntactic analysis.

Advanced Parser Generators: Future parser generator tools may incorporate advanced features such as:

1. Support for parsing Expression Grammars (PEGs) which offer more expressive power compared to traditional CFGs.
2. Integration of domain-specific optimizations to improve parsing speed and memory usage.
3. Automatic parallelization of parsing tasks to take advantage of multi-core architectures.

2. **Incremental Parsing Techniques**: Incremental parsing techniques enable parsers to efficiently update their parse trees in response to small changes in the input, rather than re-parsing the entire input from scratch. Future advancements in this area could lead to:

1. Improved performance for IDEs and real-time parsing applications.
2. Enhanced support for interactive programming environments where code is continuously modified.

Implementation:

This section will provide a detailed explanation of the implementation process, including data structures, algorithms, and any design considerations. It may include code snippets or pseudocode to illustrate key aspects of the implementation.

Result:

As a result, extensions implemented using current extensible compilers cannot be

deployed as a plugin to the extensible compiler, thus not allowing the *user* of the compiler

to select of a series of extensions. For example, it is not possible for user to select a

series of Java extensions for ableJ (e.g. SQL and algebraic datatypes) or Polyglot (e.g.

JMatch and Jedd) without compiling the compiler. Third parties should be able to deploy

language extension that do not require the compiler (or programming environment) to be

rebuilt. Therefore, methods for deploying languages as *binary components* are necessary

to leverage the promise of extensible compilers. We call this *binary extensibility*. One

of the challenges in realizing binary extensible compilers is binary extensibility of the syntax of the base language. Most extensible compilers use an LR parser, therefore this

requires the introduction of LR parse table components.

Coding & Screenshots:

LL(1) Parsing Table Program in C

The program reads grammar from a file “text.txt”.
“^” is used to represent epsilon.
The program assumes that the input grammar is LL(1).

text.txt

```
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)
```

```
#include<stdio.h>

#include<string.h>

#define TSIZE 128

// table[i][j] stores

// the index of production that must be applied on

// ith varible if the input is

// jth nonterminal
```

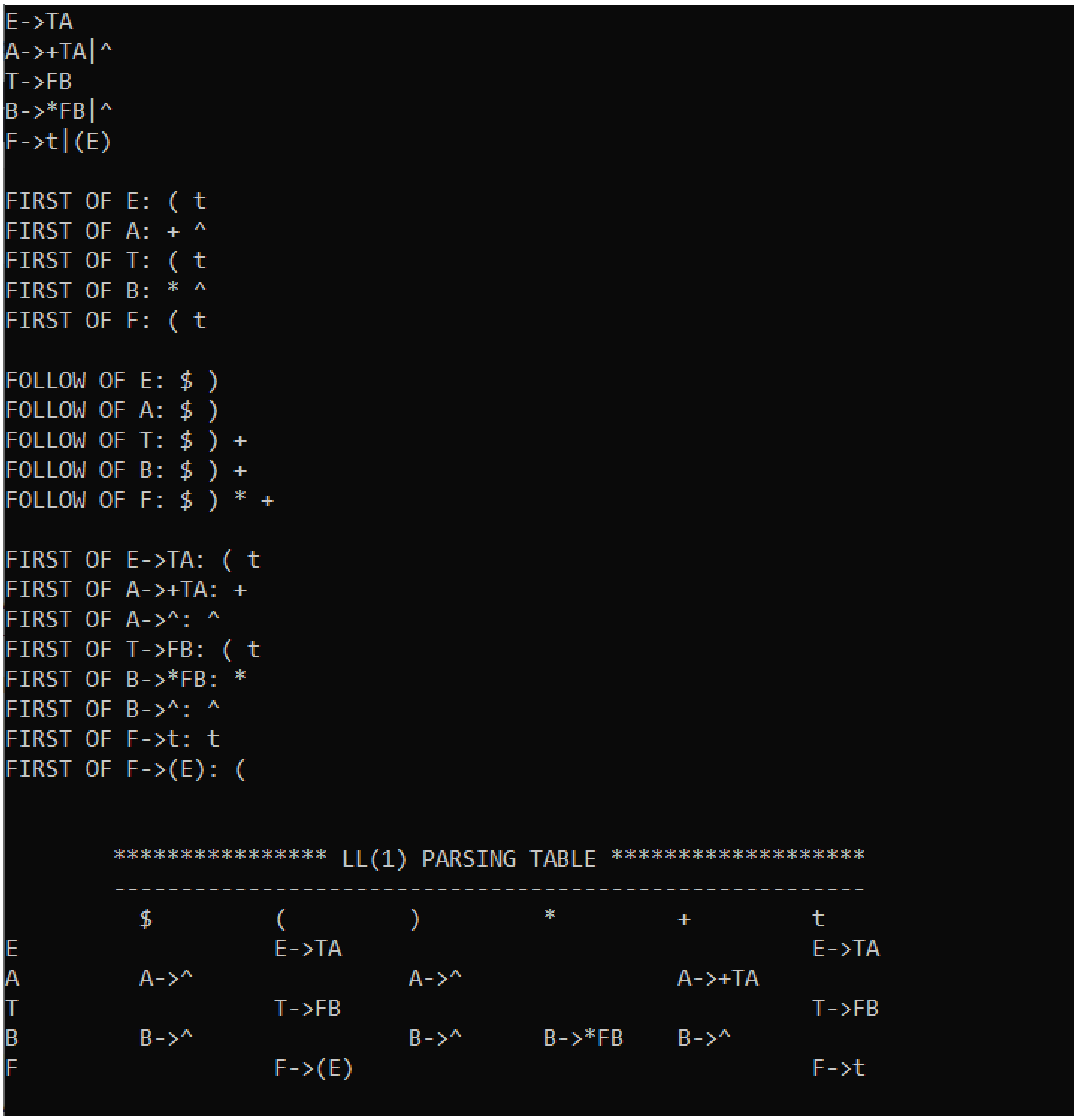
```
int table[100][TSIZE];

char terminal[TSIZE];

char nonterminal[26];

struct product {

   char str[100];

   int len;

}pro[20];

// no of productions in form A->ß

int no_pro;

char first[26][TSIZE];

char follow[26][TSIZE];

// stores first of each production in form A->ß

char first_rhs[100][TSIZE];

// check if the symbol is nonterminal

int isNT(char c) {

   return c >= 'A' && c <= 'Z';

}

// reading data from the file

void readFromFile() {
```

```
FILE* fptr;

fptr = fopen("text.txt", "r");

char buffer[255];

int i;

int j;

while (fgets(buffer, sizeof(buffer), fptr)) {

    printf("%s", buffer);

    j = 0;

    nonterminal[buffer[0] - 'A'] = 1;

    for (i = 0; i < strlen(buffer) - 1; ++i) {

        if (buffer[i] == '|') {

            ++no_pro;

            pro[no_pro - 1].str[j] = '\0';

            pro[no_pro - 1].len = j;

            pro[no_pro].str[0] = pro[no_pro - 1].str[0];

            pro[no_pro].str[1] = pro[no_pro - 1].str[1];

            pro[no_pro].str[2] = pro[no_pro - 1].str[2];

            j = 3;

        }
```

```
            else {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}
void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}
```

```
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {

  int i;

  for (i = 0; i < TSIZE; ++i) {

     if (i != '^')

         follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];

  }

}

void FOLLOW() {

  int t = 0;

  int i, j, k, x;

  while (t++ < no_pro) {

     for (k = 0; k < 26; ++k) {

         if (!nonterminal[k])    continue;

         char nt = k + 'A';

         for (i = 0; i < no_pro; ++i) {

             for (j = 3; j < pro[i].len; ++j) {

                 if (nt == pro[i].str[j]) {

                     for (x = j + 1; x < pro[i].len; ++x) {

                         char sc = pro[i].str[x];
```

```
                            if (isNT(sc)) {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['^'])
                                    continue;
                            }
                            else {
                                follow[nt - 'A'][sc] = 1;
                            }
                            break;
                        }
                        if (x == pro[i].len)
                            add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
}
                }
            }
        }
    }
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
  int i;

  for (i = 0; i < TSIZE; ++i) {

    if (i != '^') {

        first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];

    }

  }

}
```



```
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)

FIRST OF E: ( t
FIRST OF A: + ^
FIRST OF T: ( t
FIRST OF B: * ^
FIRST OF F: ( t

FOLLOW OF E: $ )
FOLLOW OF A: $ )
FOLLOW OF T: $ ) +
FOLLOW OF B: $ ) +
FOLLOW OF F: $ ) * +

FIRST OF E->TA: ( t
FIRST OF A->+TA: +
FIRST OF A->^: ^
FIRST OF T->FB: ( t
FIRST OF B->*FB: *
FIRST OF B->^: ^
FIRST OF F->t: t
FIRST OF F->(E): (


        **************** LL(1) PARSING TABLE *******************
        --------------------------------------------------------
          $         (         )         *         +         t
E                   E->TA                                   E->TA
A         A->^                A->^                A->+TA
T                   T->FB                                   T->FB
B         B->^                B->^      B->*FB    B->^
F                   F->(E)                                  F->t
```

Conclusion:

In conclusion, this project demonstrates the successful implementation of a parsing table for use in compiler design. It highlights the importance of parsing tables in parsing algorithms and discusses potential avenues for future research and improvement,we introduce an algorithm for parse table composition to support separate compilation of grammars to parse table components. Parse table components can be composed (linked) efficiently at runtime, i.e. just before parsing.While the worst-case time complexity of parse table composition is exponential(like the complexity of parse table generation itself).

References:

1. van Wyk, E., Krishnan, L., Schwerdfeger, A., Bodin, D.: Attribute grammar-based language

extensions for Java. In: ECOOP'07. LNCS, Springer (July 2007)

2. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. In: OOPSLA '07, ACM Press

(2007) 1–18

3. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for

Java. In: CC'03. Volume 2622 of LNCS., Springer (April 2003) 138–152

4. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embedding –

a host and guest language independent approach. [47]

5. Lhot

´

ak, O., Hendren, L.: Jedd: A BDD-based relational extension of Java. In: PLDI 2004,

ACM Press (2004)

6. Millstein, T.: Practical predicate dispatch. In: OOPSLA ’04, ACM (2004) 345–364

7. Arnoldus, B.J., Bijpost, J.W., van den Brand, M.G.J.: Repleo: A syntax-safe template engine.

[47]

8. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific lan

guages. In: ICSR’98, IEEE Computer Society Press (June 1998) 143–153

9. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.16. Components for

transformation systems. In: PEPM’06, ACM (January 2006)

10. : ASF+SDF MetaEnvironment website. http://www.meta-environment.org

11. Visser, E.: Meta-programming with concrete object syntax. In Batory, D., Consel, C., Taha,

W., eds.: GPCE’02. Volume 2487 of LNCS., Springer (October 2002) 299–315

12. van Wyk, E., Bodin, D., Huntington, P.: Adding syntax and static analysis to libraries via

extensible compilers and language extensions. In: LCSD’06. (2006)

13. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition.

In: OOPSLA ’06, ACM (2006) 21–36

14. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA ’05, ACM (2005)

41–57

15. Onzon, E.: Dypgen: Self-extensible parsers for ocaml. http://dypgen.free.fr (2007)

16. Salomon, D.J., Cormack, G.V.: Scannerless NSLR(1) parsing of programming languages. In:

PLDI ’89, ACM Press (1989) 170–178

17. Tomita, M.: Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems.

Kluwer Academic Publishers (1985)

18. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, Univ. of Amsterdam

(1992)

19. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, Univ. of Amsterdam

(September 1997)

20. Horspool, R.N.: Incremental generation of LR parsers. Computer Languages **15**(4) (1990)

205–223

21. Heering, J., Klint, P., Rekers, J.: Incremental generation of parsers. IEEE Transactions on

Software Engineering **16**(12) (1990) 1344–135122. Cardelli, L., Matthes, F., Abadi, M.: Extensible syntax with lexical scoping. SRC Research

Report 121, Digital Systems Research Center, Palo Alto, California (February 1994)

23. Knuth, D.E.: On the translation of languages from left to right. Information and Control **8**(6)

(December 1965) 607–639

24. Aho, A.V., Johnson, S.C.: LR parsing. ACM Computing Surveys **6**(2) (1974) 99–124

25. Aho, A.V., Sethi, R., Ullman, J.: Compilers: Principles, techniques, and tools. Addison Wesley,

Reading, Massachusetts (1986)

26. Grune, D., Jacobs, C.J.H.: Parsing Techniques - A Practical Guide. Ellis Horwood, Upper

Saddle River, NJ, USA (1990)

27. Johnstone, A., Scott, E.: Generalised reduction modified LR parsing for domain specific lan

guage prototyping. In: 35th Annual Hawaii International Conference on System Sciences

(HICSS'02), Washington, DC, USA, IEEE Computer Society Press (2002) 282

28. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and

Computation (3rd Edition). Addison-Wesley, Boston, MA, USA (2006)

29. Leslie, T.: Efficient approaches to subset construction. Master's thesis, University of Waterloo,

Waterloo, Ontario, Canada (1995)

30. van Noord, G.: Treatment of epsilon moves in subset construction. Computational Linguistics

26(1) (2000) 61–76

31. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica **16**(4/5) (1996) 464–497

32. DeRemer, F.: Simple LR(k) grammars. Communications of the ACM **14**(7) (1971) 453–460

33. Johnstone, A., Scott, E., Economopoulos, G.: Evaluating GLR parsing algorithms. Science of

Computer Programming **61**(3) (2006) 228–244

34. DeRemer, F., Pennello, T.J.: Efficient computation of LALR(1) look-ahead sets. In: CC ’79,

New York, NY, USA, ACM Press (1979) 176–187

35. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal on Computing

1(2) (1972) 146–160

36. Eve, J., Kurki-Suonio, R.: On computing the transitive closure of a relation. Acta Informatica

8(4) (1966) 303–314

37. Nuutila, E.: Efficient Transitive Closure Computation in Large Digraphs. PhD thesis, Helsinki

University of Technology (1995)

38. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of

Language Conglomerates. PhD thesis, Utrecht University, Utrecht, The Netherlands (Jan.

2008)

39. Bravenboer, M., Tanter, E., Visser, E.: Declarative, formal, and extensible syntax definition

for AspectJ – A case for scannerless generalized-LR parsing. In: OOPSLA'06, ACM Press

(2006)

40. Grimm, R.: Better extensibility through modular syntax. In Cook, W.R., ed.: PLDI'06, ACM

Press (June 2006)

41. van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages.

[47]

42. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13**(2)

(1970) 94–102

43. Tratt, L.: The Converge programming language. Technical Report TR-05-01, Department of

Computer Science, King's College London (February 2005)

44. Kolbly, D.M.: Extensible Language Implementation. PhD thesis, University of Texas at Austin

(December 2002)

45. Baker, J., Hsieh, W.: Maya: multiple-dispatch syntax extension in java. In: PLDI ’02, ACM

Press (2002) 270–281

46. de Rauglaudre, D.: Camlp4 Reference Manual. (September 2003)

47. Lawall, J., ed.: Generative Programming and Component Engineering, Sixth International

Conference, GPCE 2007, New York, NY, USA, ACM Press (October 2007)