Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

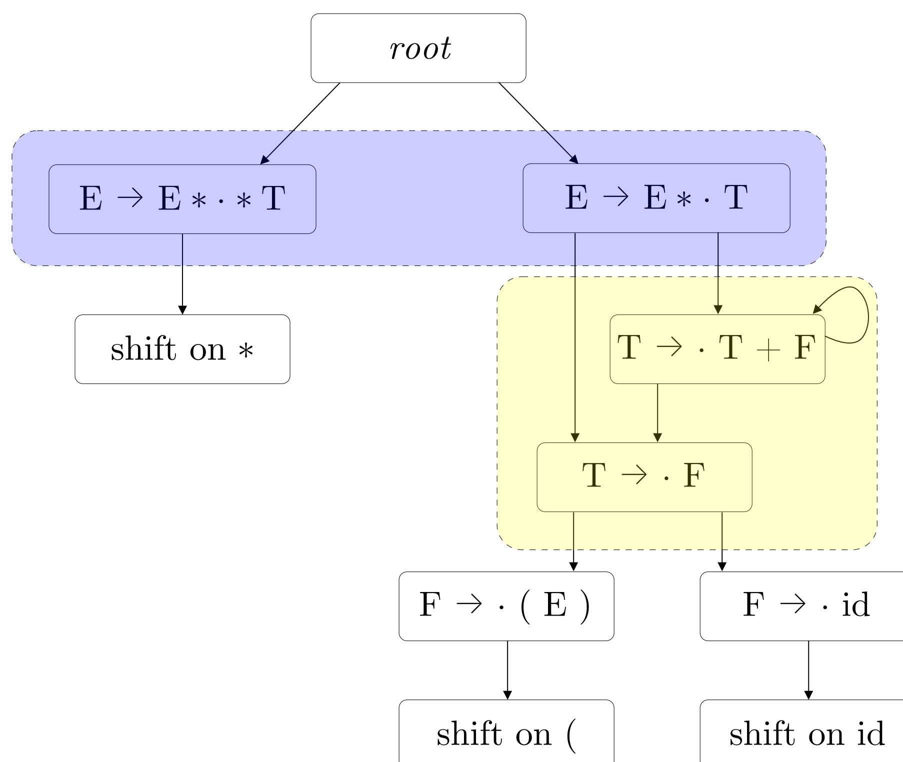
Lehrstuhl Prof. Dr.-Ing. Snelting

A Typed Recursive Ascent-Descent Backend for HAPPY

Bachelorarbeit von

David Knothe

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Sebastian Graf

Abgabedatum: 23. Oktober 2020

Abstract

Top-Down-Parser sind schnell und simpel und können oft von Hand geschrieben werden; für komplexere Grammatiken werden jedoch Bottom-Up-Parser benötigt. Diese werden typischerweise mittels Parsergeneratoren erstellt und haben eine große Anzahl an Zuständen. Rekursiver Aufstieg-Abstieg kombiniert die Top-Down- und Bottom-Up-Techniken, um die Anzahl der Zustände soweit wie möglich zu reduzieren, während die volle Mächtigkeit des Bottom-Up-Parsens beibehalten wird.

Wir erweitern den Haskell-Parsergenerator HAPPY um ein rekursives Aufstieg-Abstieg-Backend. Dabei erzeugen wir Code im continuation-passing style, wodurch die generierten Parser sowohl effizient als auch voll typsicher sind. Dadurch erzielen wir signifikante Geschwindigkeitsverbesserungen im Vergleich zu klassischen tabellenbasierten LALR-Parsern, wie sie von HAPPY generiert werden.

Wir integrieren hierbei alle wesentlichen Features von HAPPY, was es uns ermöglicht, den ebenfalls von HAPPY generierten Parser des Glasgow Haskell Compilers (GHC) durch eine rekursive Aufstieg-Abstieg-Variante zu ersetzen. Hierdurch erreichen wir Performance-Verbesserungen beim Parsen von Haskell-Code von ungefähr 10%.

While top-down parsers are fast and simple and can often be written by hand, parsing more complex grammars requires bottom-up parsers. They often have numerous states and cannot be written by hand but are typically created by parser generators. Recursive Ascent-Descent parsing combines top-down and bottom-up parsing to minimize the number of states as much as possible, while still obtaining the full power of bottom-up parsing.

We extend the popular Haskell parser generator HAPPY by a recursive ascent-descent backend. Thereby we utilize continuation-passing style, which makes our generated parsers both efficient and strongly typed. Consequently, we achieve significant speed improvements compared to regular table-based LALR parsers.

We integrate all essential features of HAPPY, which allows us to replace the parser of the Glasgow Haskell Compiler (GHC), which is also generated by HAPPY, by a recursive ascent-descent variant. As a result, we achieve performance improvements of around 10% when parsing Haskell code.

Contents

1. Introduction **7**
1.1. Contributions . . . 8

2. Preliminaries And Related Work **9**
2.1. LL And LR Parsing . . . 9
2.2. (Generalized) Left Corner Parsing . . . 10
2.3. Recursive Ascent-Descent . . . 10
2.4. Scala-bison . . . 16
2.5. Happy and GHC . . . 16
2.6. Typed Continuation-Based Parsing . . . 17

3. State Generation **21**
3.1. Recognition Points . . . 21
3.2. About RAD States . . . 24
3.2.1. RAD-Completion . . . 25
3.3. Unambiguous Nonterminals . . . 26
3.4. Algorithmic Generation . . . 27
3.4.1. State Skeleton Creation . . . 29
3.4.2. Finalization – Transforming Actions . . . 31
3.5. Default Actions And Happy's Error Token . . . 41

4. Code Generation **43**
4.1. Algorithmic Generation . . . 43
4.1.1. Parsing a Nonterminal . . . 43
4.1.2. Parsing a Terminal . . . 44
4.1.3. Rule Functions . . . 45
4.1.4. States . . . 46
4.1.5. Top-Level Entry Points . . . 52
4.1.6. Semantic Actions . . . 53
4.2. GHC-Specifics . . . 54
4.2.1. Monadic Lexers . . . 54
4.2.2. Higher-Rank Types . . . 57

5. Evaluation **61**
5.1. Experimental Results . . . 61
5.1.1. Parser Only . . . 62
5.1.2. Parser-Lexer Combination . . . 63

5.1.3. GHC: Parsing Haskell 65
5.2. LL-Ness 67

6. Conclusion **71**
6.1. Future Work 71

A. Generated Code Examples
A.1. LALR . . . .
A.2. RAD . . . .

1. Introduction

Context-free grammars are commonly used to specify the syntax of computer programming languages. A context-free grammar consists of *production rules* – rules that specify how certain nonterminals can be replaced by strings of terminals and other nonterminals. A *parser* takes a list of terminals and creates a valid derivation, beginning from the grammar's start symbol. For some classes of context-free grammars there exist efficient parsers: the most prominent ones are LL(k) and LR(k) grammars. These emit LL, LR and LALR parsers, which can be used to parse an input string, for example source code of a computer program.

Both LL(k) and LALR(k) parsers are prominent and are used in real compilers. Those can either be implemented in a table-based form or via mutually recursive functions in the host language ("directly-executable form").

All of these variants have certain drawbacks: Directly-executable LL parsers are fast, but LL parsers, especially LL(1) parsers, can only recognize a very limited set of context-free grammars. LALR parsers on the other hand are more powerful. Because they consist of numerous states, they are most often only found in table-based form, which results in a performance penalty. Directly-executable LALR parsers would be faster, but would come along with large binary code size because they would contain a function for every single state.

Horspool [1] introduced *Recursive Ascent-Descent Parsing* which combines the advantages of recursive ascent and recursive descent parsing: While being just as powerful as LALR parsers, Recursive Ascent-Descent (or just *RAD*) parsers have significantly fewer states. They switch from bottom-up parsing to top-down parsing when possible, and only use bottom-up parsing where required.

Independently of and unrelated to RAD parsing, Hinze and Paterson [2] described the concept of continuation-based parsing: directly executable code can be generated for LL and LALR parsers so that they are completely stackless – partial parsing results are not stored on a stack but are passed around via continuation functions. In addition, these parsers are fully well-typed.

In this thesis we combine the two orthogonal approaches of recursive ascent-descent parsing and of continuation-based parsing to create **powerful**, **fast** and **sufficiently small general-purpose** parsers:

1. *Powerful* and *general-purpose*, because a RAD(k) parser can parse all LALR(k) grammars.

2. *Fast*, because the directly-executable, continuation-based form brings speed improvements over classical, table-based forms.

3. *Sufficiently small*, because the compiled code of a continuation-based RAD parser is often smaller than the compiled code of an equivalent continuation-based LALR parser.

We extended the popular Haskell parser generator tool HAPPY by a typed, continuation-based recursive ascent-descent parser generator. As HAPPY is used to generate the parser for the Glasgow Haskell Compiler (GHC), it was our main aim of this work to improve GHC itself by generating a parser with better performance than the one which is currently in use.

Throughout this paper, we only consider grammars and parsers with a single token of lookahead, like LL(1) and LALR(1). While the ideas can be extended to grammars and parsers with more lookahead, we deliberately chose LALR(1) as this is both the most common form of generated parsers and also the form that HAPPY uses, which allows us to simply use and modify HAPPY's LALR(1)-states.

1.1. Contributions

- We introduce the reader to the concepts of recursive ascent-descent parsing and compare them with LL and LALR parsing in section 2. We then unite recursive ascent-descent parsing with continuation-based parsing in section 4 when we describe the process of continuation-based code generation for a recursive ascent-descent parser.

- We describe, in detail, how recursive ascent-descent states are generated algorithmically in section 3. We provide more detail than other papers covering recursive ascent-descent parsing. We also point out and fix an error in the recursive ascent-descent parser generator SCALA-BISON.

- We implement a continuation-based recursive ascent-descent backend for HAPPY and evaluate its performance and other metrics in comparison to different parser variants in section 5.1. We take a special look on the Haskell grammar and the Glasgow Haskell Compiler.

- We introduce a notion of "LL-ness" of an LR grammar which measures, intuitively, how much a parser can make use of recursive descent when parsing the grammar. The more "LL" a grammar is, the fewer states are required in general for a recursive ascent-descent parser.

- We present some possibilities for further performance fine-tuning and optimization in section 6.

2. Preliminaries And Related Work

2.1. LL And LR Parsing

For any context-free grammar, there exists a nondeterministic pushdown automaton that accepts exactly the language which is generated by the grammar. This pushdown automaton can be converted to a deterministic LL or LR parser when working with an LL or an LR grammar, respectively. This process is described in detail in various books, for example in Compiler Design [3].

LL(1) parsers derive the input in a top-down way. An LL(1) parser is predictive: it starts with the grammar's start symbol and decides at each step which production to follow, solely based on the next input token, the *lookahead token*.

An LR or LALR parser follows multiple production candidates at once and only decides as late as possible which of these productions to actually reduce. This is typically realized with an LALR automaton. It consists of LALR states, each of which has a set of *core items*. An item describes the current position of the process of parsing a production and looks like this: $[A \to \alpha \ \cdot \ \beta]$. α is the string of symbols which has already been derived, while β must still be derived before the rule $A \to \alpha\ \beta$ can be reduced.

Because an LALR parser can follow multiple production candidates at once, LALR states can have multiple core items. An exemplary LALR state could contain the following core items: $[A \to B \ \cdot \ C]$ and $[A \to B \ \cdot \ D]$. The nonterminal B has already been parsed, but it is not yet clear whether the production $A \to B\ C$ or $A \to B\ D$ is active.

In addition to its core items, an LALR state has a set of completion (or closure) items. If a state contains the item $[A \to \ldots \ \cdot \ T\ \ldots]$, the parsing position is immediately before the T. Now, because the parser might begin parsing a T, the parser is additionally in all positions of the form $[T \to \cdot\ \ldots]$ simultaneously. Therefore, all items of the type $[T \to \cdot\ \ldots]$ must be in the state's completion. The full completion set is then derived through the reflexive transitive closure of this operation.

A note on notation: We will use lower-case Latin letters (like a, b, x, id) for terminals and upper-case Latin letters for nonterminals. We will further use the word "symbol" to refer to any terminal or nonterminal.

Greek letters like α, β, ω are used to denote arbitrary (possibly empty) strings of symbols. The empty word is depicted as ε.

2.2. (Generalized) Left Corner Parsing

As mentioned above, an LL parser recognizes each rule immediately at its left end, while an LR, SLR ("simple LR") or LALR parser recognizes each rule at its right end, after all terminals and nonterminals of the rule have been gathered. An *LC parser*, as introduced by Rosenkrantz and Lewis [4], only parses the very first symbol of a rule in bottom-up-style and recognizes the rule thereafter, switching to top-down-style. LR grammars are in general not LC and cannot be parsed by an LC parser.

Demers [5] introduced GLC parsing, a generalization of LC parsing which does not switch from bottom-up to top-down after parsing the *first* symbol, but at arbitrary user-defined switch points. These points are typically chosen as the *earliest possible* positions where GLC parsing remains deterministic. This minimizes the number of required states. The class of GLC grammars contains all the LL, LR, LALR and SLR classes, as these are just special cases of GLC grammars: An LL parser is a GLC parser where the switch points are always at the left end, and an LR parser is a GLC parser where they are always at the right end.

2.3. Recursive Ascent-Descent

Demers originally introduced GLC parsing only for SLR grammars. Horspool then extended these ideas to LR and LALR grammars, calling the resulting parsers XLC and LAXLC (where the "X" denotes *extended*).

Recursive Ascent-Descent then describes the implementation of an XLC or LAXLC parser in directly-executable form, contrary to table-based form. We restrict ourselves to directly-executable LAXLC(1) parsers and refer to them as RAD(1) or just RAD.

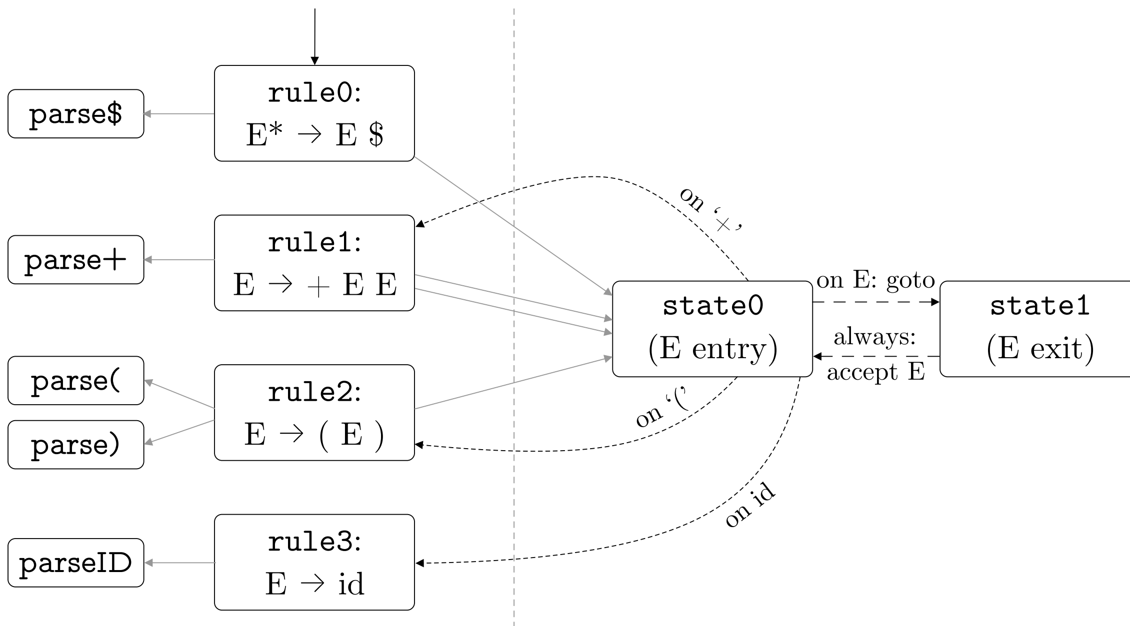
Recursive Ascent-Descent parsers use *recognition points* to switch from bottom-up (recursive ascent) to top-down (recursive descent) parsing. The recognition point of a rule is the leftmost position at which the correct production can be determined unambiguously. This is often much earlier than at the very right end.

We will now discuss the structure and functionality of a Recursive Ascent-Descent parser, accompanied by exemplary grammars. Consider the following grammar:

$$
\begin{aligned}
0 &: E^\star \rightarrow\ E\ \$ \\
1 &: E\ \ \rightarrow\ +\ E\ E \\
2 &: E\ \ \rightarrow\ (\ E\ ) \\
3 &: E\ \ \rightarrow\ id
\end{aligned}
$$

Here, $E^\star$ is the start symbol, and \$ stands for the eof-token. Note that this grammar is LL(1), meaning the active rule can always be determined at its very beginning, only based on the next input token – there are no rule ambiguities at any time. In other words, the recognition point of any rule is at the very left.

Figure 2.1.: The structure of a RAD parser of a simple LL(1) grammar. On the left is the recursive descent part, on the right the recursive ascent part.



The structure of the corresponding RAD parser can be seen in figure 2.1. On the left is the recursive descent part, consisting of one function for parsing each rule and each terminal. On the right is the recursive ascent part, consisting of RAD states, one function per state.

Recursive descent mode is started by parsing a production rule, i.e. by calling a rule function. Then, all symbols on the rule's right-hand side are parsed, one after the other, by calling their respective parsing functions:

- A terminal is parsed by simply comparing the expected terminal with the next token from the input, and failing the parse if the token does not match. For example, a single + is parsed by calling `parse+`. This is similar to how terminals are parsed in top-down parsers.

- A nonterminal is parsed by entering a special state (entry state) which is designed to parse this nonterminal. Here, the recursive ascent part begins. Once the nonterminal has been parsed, the parser arrives at a special (exit) state which will then *accept* the nonterminal and return control back to the recursive descent part, i.e. to the calling rule function, which then proceeds with its next symbol.
  For example, a single E is parsed by calling `state0`, the entry state of E. We will also use `parseE` as an alias for `state0`.

For example, parsing rule 1 ($E \to +\ E\ E$) would yield one call to `parse+`, and then two calls to `parseE` (i.e. `state0`). The outgoing arrows of `rule1` in figure 2.1

show exactly these function calls.

Because the above grammar is LL(1), its RAD parser works exactly like an LL(1) parser. Consider state 0 – it has the following actions (as seen in figure 2.1):

1. on + announce rule 1
2. on (announce rule 2
3. on *id* announce rule 3
4. on E goto state 1

State 0 decides, based on the next input token, which rule is active. The corresponding rule function is then called (the rule is *announced*) and parses all symbols of the rule successively.

E's exit state, state 1, doesn't do anything interesting – it just accepts the partial parse of E every time.

This changes when looking at a grammar that is not quite LL(1). Consider this modified grammar:

$$
\begin{aligned}
0 : E^{\star} &\rightarrow\ E\ \$ \\
1 : E &\rightarrow\ E\ +\ E\ ; \\
2 : E &\rightarrow\ (\ E\ ) \\
3 : E &\rightarrow\ id
\end{aligned}
$$

When beginning to parse an E it is unclear whether rule 1 is active or not. This can only be decided when seeing the next token *after* an E has been parsed.

It follows that the recognition point of rule 1 is not at the very left but after the first E. This is depicted as follows: $E\ \rightarrow\ E\ \bullet\ +\ E\ ;$ – we use small dots for items and large dots for recognition points inside rules.

Here is the grammar, again, with its recognition points marked:

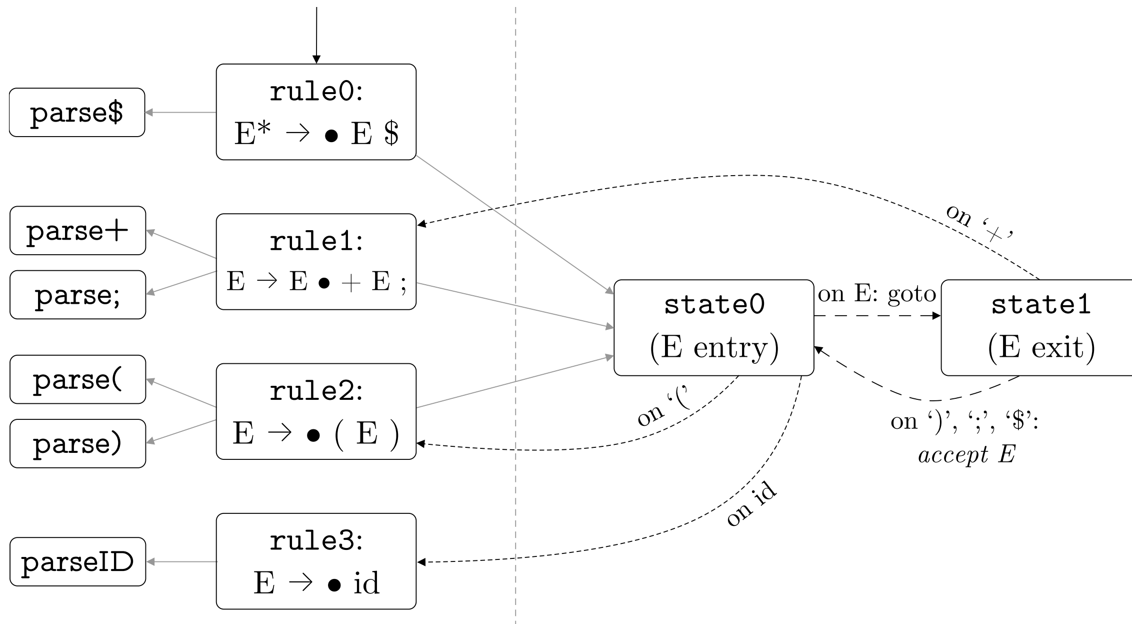
$$
\begin{aligned}
0 : E^{\star} &\rightarrow\ \bullet\ E\ \$ \\
1 : E &\rightarrow\ E\ \bullet\ +\ E\ ; \\
2 : E &\rightarrow\ \bullet\ (\ E\ ) \\
3 : E &\rightarrow\ \bullet\ id
\end{aligned}
$$

Now, when a rule is announced, it is possible that part of the rule's right-hand side has already been consumed: a rule is always announced at its recognition point. Consider rule 1: once rule 1 is recognized, the parse has already advanced *beyond* the first E. Therefore, `rule1` only parses the symbols after the first E, i.e. after its recognition point.

The other rules have their recognition point at the very left. Nothing changes here: they parse all symbols on their right-hand side, one after the other.

Figure 2.2 shows the structure of the full RAD parser. This time, state 1 (E's exit state) has a nontrivial announce action: after an E was parsed, state 1 decides whether the parse of this E has finished, or whether it should continue with rule 1.

Figure 2.2.: The structure of a RAD parser of a grammar which is not LL(1). On the left is the recursive descent part, on the right the recursive ascent part. The recognition point of each rule is marked by a dot.



State 0

Items:

$[_ \to \cdot\ E]$ (core)
$[E \to \cdot\ E\ +\ E]$
$[E \to \cdot\ (\ E\ )]$
$[E \to \cdot\ id]$

Actions:

on (	announce rule 2
on *id*	announce rule 3
on E	goto state 1

State 1

Items:

$[_ \to E\ \cdot]$ (core)
$[E \to E\ \cdot\ +\ E]$ (core)

Actions:

on +	announce rule 1
on)	accept E
on ;	accept E
on $	accept E

These RAD states have so-called *artificial items*: State 0 has the core item $[_ \to \cdot\ E]$ which signalizes that state 0 parses exactly an E and that the parser is

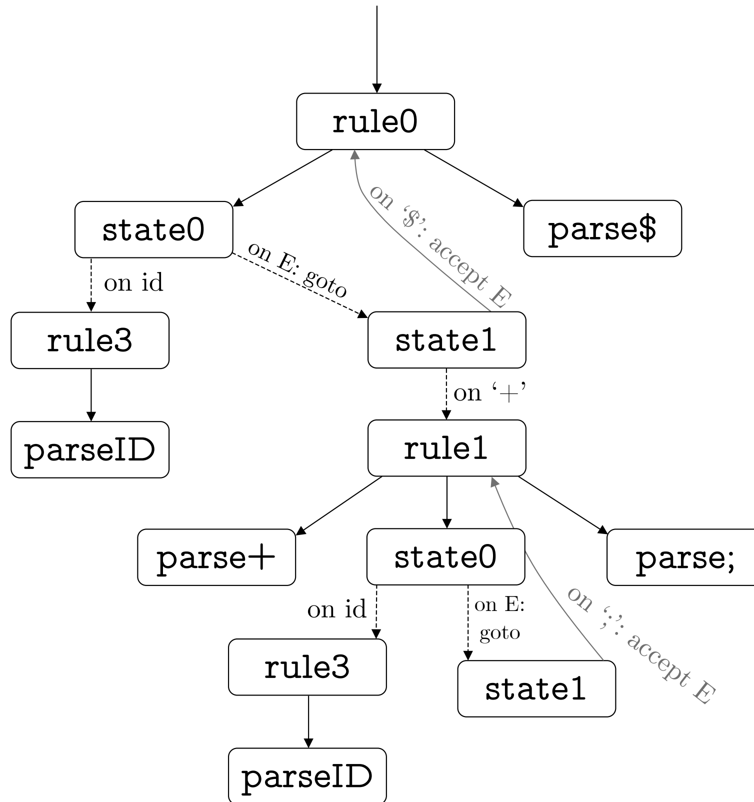
currently located just before this E. State 1 resembles the state *after* parsing this E and therefore has the corresponding artificial core item $[_ \rightarrow E\ \cdot]$. These items have no direct analogy in an LR parser as they do not belong to an actual rule.

The lookahead sets of these artificial items differ from those in an LALR parser. Section 3.4.2 addresses the task of calculating these lookahead sets.

The following table, together with figure 2.3, showcases the call tree when parsing the expression "1 + 2 ;", step by step. For brevity, we abbreviate `state0` with `s0` and `state1` with `s1`.

Function call stack	**Input**	**Action**
	1 + 2 ;	begin parse by calling top-level rule
`rule0`	1 + 2 ;	$(E^\star \rightarrow \bullet\ E\ \$)$ $\rightarrow$ parse E, then \$
`rule0 s0`	1 + 2 ;	on *id* announce rule 3
`rule0 s0 rule3`	1 + 2 ;	$(E \rightarrow \bullet\ id) \rightarrow$ parse *id*
`rule0 s0 rule3 parseID`	1 + 2 ;	consume "1"
`rule0 s0 rule3`	+ 2 ;	rule 3 has finished
`rule0 s0`	+ 2 ;	on E goto state 1
`rule0 s0 s1`	+ 2 ;	on + announce rule 1
`rule0 s0 s1 rule1`	+ 2 ;	$(E \rightarrow E\ \bullet\ +\ E\ ;)$ $\rightarrow$ parse +, E and ;
`rule0 s0 s1 rule1 parse+`	+ 2 ;	consume +
`rule0 s0 s1 rule1`	2 ;	continue with rule 1 $\rightarrow$ parse E
`rule0 s0 s1 rule1 s0`	2 ;	on *id* announce rule 3
`rule0 s0 s1 rule1 s0 rule3`	2 ;	$(E \rightarrow \bullet\ id) \rightarrow$ parse *id*
`rule0 s0 s1 rule1 s0 rule3 parseID`	2 ;	consume "2"
`rule0 s0 s1 rule1 s0 rule3`	;	rule 3 has finished
`rule0 s0 s1 rule1 s0`	;	on E goto state 1
`rule0 s0 s1 rule1 s0 s1`	;	on ; accept E
`rule0 s0 s1 rule1 s0`	;	E was accepted; return
`rule0 s0 s1 rule1`	;	continue with rule 1 $\rightarrow$ parse ;
`rule0 s0 s1 rule1 parse;`	;	consume ;
`rule0 s0 s1 rule1`	\$	rule 1 has finished
`rule0 s0 s1`	\$	on \$: accept E
`rule0 s0`	\$	E was accepted; return
`rule0`	\$	continue with rule 0 $\rightarrow$ parse \$
`rule0 parse$`	\$	consume \$
`rule0`		rule 0 has finished
		parse was successful

Figure 2.3.: A parse tree of a grammar which is not LL(1). The parse tree resembles the expression "1 + 2 ;".



1. The parse begins by calling the top-level rule function, `rule0`. As rule 0 is $E^{\star} \rightarrow \bullet\ E\ \$$, rule 0 first parses the E by calling `state0`, and after that parses the final eof-token by calling `parse$`. (Note that the recognition is at the very left and therefore all symbols of rule 0's right-hand side are parsed.)

2. `state0` reads the current lookahead token and sees an *id*. This triggers the action *on id announce rule 3* and rule 3 is announced by calling `rule3`.

3. As rule 3 is $E \rightarrow \bullet\ id$, `rule3` just calls `parseID`. `parseID` consumes the current token (an *id*) and passes control back to `rule3`. Because rule 3 has finished, it passes control back to `state0`.

4. Now, the goto action of state 0 is executed: *on E goto state 1.* This is done by calling `state1`.

5. State 1 now decides whether the current parse of E has finished, or whether we are actually in the middle of rule 1. Because the next token is a +, rule 1 is announced by calling `rule1`. Note that the dot of state 1's second core item, $[E \rightarrow E\ \cdot\ +\ E]$, is exactly at the recognition point of rule 1! This makes

sense, because a rule is always announced at the position of its recognition point – it cannot be announced earlier because it would still be ambiguous at this moment whether the rule is actually in use, and it cannot be announced later because this would require unnecessary extra states.

6. Announcing rule 1 now parses the remaining symbols of rule 1, beginning at the recognition point. This means, a "+", an "E", and a ";" are parsed successively.

7. `parse+` consumes the current token (a +). `parseE` then again calls `state0`, which parses the current E via `rule3` and `parseID`. Now the goto action is executed, and control goes to `state1`. State 1 sees the next token, a ";", and decides to *accept* the E and control goes back to rule 1, which parses the final ";".

8. Now `rule1` returns and control goes back to `state1`, and this time state 1 decides to accept the E after seeing the eof-token "$". Control is now back at `rule0`. The parse ends after `rule0` has consumed the final eof-token.

As opposed to an LALR parser, it is relatively easy for a human to understand what is happening here. This is because the RAD parser is as similar to an LL parser as possible, and only uses LALR where required, as little as possible.

An LALR(1) parser of this grammar would require eleven states – the RAD parser has only two states, and performs the rest of the work via the top-down rule functions, just like a directly-executable LL(1) parser.

2.4. Scala-bison

Apart from Horspool himself, we only found one other author who actually implemented a Recursive Ascent-Descent parser generator: Boyland and Spiewak [6] created a tool called SCALA-BISON which generates Recursive Ascent-Descent code in Scala. SCALA-BISON takes a ".y" grammar file, invokes BISON and reads the LALR states from the BISON output. Those are then converted to RAD states and eventually to executable Scala code.

The SCALA-BISON paper clarified some ideas and algorithms of Horspool. For example, it suggested a concrete, sensible algorithm for calculating the recognition points of a grammar. Especially the open-source implementation [7] of the SCALA-BISON tool contained several helpful hints and details and helped us understand some more complex questions and algorithms regarding recursive ascent-descent state generation.

2.5. Happy and GHC

HAPPY [8] is an LALR(1)-parser generator written in Haskell which generates Haskell code. It takes a grammar file which is similar to a BISON ".y"-file. It then parses

this file (using HAPPY itself) and creates LALR-states, action tables and goto tables which are then converted to table-based parsing code.

The user of HAPPY can choose between several modes of operation. In normal mode, the parser just takes a list of tokens, and returns a result (or stops by calling the user-supplied function `happyError`.) In monadic mode, the user can supply a custom monad in which all parsing functions are wrapped.

Additionally, instead of providing a full list of tokens, the user may supply a lexer function which returns the next token each time it is called. In combination with a monadic parser, this could be used to keep track of errors and line numbers or switch between different contexts during parsing. This mode (combined monadic lexer) is also used to generate the parser for the Glasgow Haskell Compiler, GHC [9]. For lexing, the tool ALEX [10] is used, which fittingly exports such a lexer function which is required by HAPPY.

Later, when generating RAD states for an LR grammar, we use HAPPY's LALR states, i.e. action and goto tables. This has the advantage that shift-reduce or reduce-reduce conflicts have already been resolved by HAPPY or by specifications of the grammar author. There is no need to consider these conflicts ourselves as we can just take HAPPY's action and goto tables and process them further.

2.6. Typed Continuation-Based Parsing

In their paper, Hinze and Paterson [2] define the function `parse` as the inverse of the `flatten` function which flattens an AST to a string. Starting from this definition, they derive sensible definitions for each state function, each shift and goto action, and develop a parsing machinery which uses continuation-passing style to pass around parsing results and semantic values. This leads to a stackless parser which is implemented via mutually recursive state functions.

They begin by defining `parse` as a set-valued function via `parse = flatten`$^{-1}$, meaning `parse` may return zero or multiple results. This is reasonable – when the string is ill-formed, parsing has no result (`flatten` is not surjective), and if the grammar is ambiguous, parsing may have multiple results (`flatten` is not injective). They later transform `parse` and all related functions to make them deterministic (single-valued) and show that this always works as expected for LR grammars.

We will now just consider the deterministic case. The function `parse` now simply has the (Haskell) type `[Token] -> r`: it consumes a list of tokens and returns an (unspecified at first) value of type `r`. (Why not `Maybe r`? When parsing fails, we don't want to return `Nothing` but rather directly fail with an error: this makes the resulting parser much simpler.) We define a type synonym for the type of `parse` via `type Parser r = [Token] -> r`.

Now consider a state with one core item, for example $[A \to b \cdot C]$. The task of this state is to parse the following C. This means, the task of the corresponding function

$\texttt{state}_{[A \to b \cdot C]}$ is to parse the following C from the token list and return the parsed semantic value. This parsed value could be an AST, a string, or something different, but we give it the type $\overline{\texttt{C}}$. Thus $\texttt{state}_{[A \to b \cdot C]}$ takes a list of tokens and returns the semantic value of type $\overline{\texttt{C}}$ and the remaining list of tokens – those which have not been consumed while parsing the C. So this function could have the following type signature:

```
state[A→b·C] ::  [Token] -> (C̄, [Token])
```

Hinze and Paterson [2] noticed that this can be solved much more elegantly using so-called continuation-passing style:

```
state[A→b·C] ::  (C̄ -> [Token] -> r) -> [Token] -> r
```

Instead of returning the remaining tokens and the parsed value of type $\overline{\texttt{C}}$, these are both passed into a continuation. This continuation can further process them and then return some arbitrary value of any type `r`, which is then also the result of the original function.

This allows parsing to take place in a completely continuation-driven manner by simply using *different* state functions – states that are reached through a shift or goto action – as the continuation to any state function. For example, after the above state parses a C, a normal LALR parser would execute a goto action and switch into a different state (with a core item $[A \to b\ C\ \cdot]$.) This can be realized by using this *new* state function, $\text{state}_{[A \to bC \cdot]}$, as a continuation to the original one, $\text{state}_{[A \to b \cdot C]}$. This process then continues throughout the whole parser, up to a place where parsing is finally halted by *accepting* the parse after the full token list was consumed and a correct derivation was built.

Notice how the type of the above function can be expressed in terms of `Parser r`:

```
state[A→b·C] ::  (C̄ -> Parser r) -> Parser r
```

This makes its purpose even more explicit: It parses a C, *creates* a value of type $\overline{\texttt{C}}$ from it (by calling a semantic action, see below), and passes this value to the continuation function which processes the rest of the input.

When a state has multiple core items from different rules, when entering the state it is not yet known which of these rules is actually active. This is only decided after parsing the next symbol, or at an even later point. Expanding on the above idea, states with *multiple* core items can simply define *multiple* continuations and call the continuation of the core item which is decided to be the correct one. For example, a state S with the two core items $[A \to \cdot\ B]$ and $[A \to \cdot\ C]$ has two continuations and after actually parsing either a B or a C it calls the correct one, which then continues parsing in the correct context. The type of this function would be the following:

```
state_S ::  (B̄ -> Parser r) -> (C̄ -> Parser r) -> Parser r
```

One other thing we need to consider are items with multiple symbols after the dot. Considering the item $[A \to b\ \cdot\ c\ D\ e]$, a corresponding state S_1 would parse the c and then call a state S_2 with core item $[A \to b\ c\ \cdot\ D\ e]$, which would, after parsing the D, call a state S_3 with core item $[A \to b\ c\ D\ \cdot\ e]$. This leads to the following conclusion about the types of these state functions:

```
state_S1 ::  (c̄ -> D̄ -> ē -> Parser r) -> Parser r
state_S2 ::  (D̄ -> ē -> Parser r) -> Parser r
state_S3 ::  (ē -> Parser r) -> Parser r
```

Why? `state`$_{S_2}$ can be used in the completion of `state`$_{S_1}$ like this:

```
state_S1 cont = let c = ... in cont c state_S2
```

Here, c is a value of type `c̄` (which was obtained by taking the next token from the input stream). Calling the continuation with a value of type `c̄` and a function of type `D̄ -> ē -> Parser r` exactly matches its type. The same is then true for state$_{S_2}$ and state$_{S_3}$:

```
state_S2 cont = let d = ... in cont d state_S3
```

The remaining task of creating a working parser is threefold:

- determine the concrete semantic types of the symbols
- consider shift, reduce and goto actions and semantic actions and define corresponding functions
- implement the state functions.

We will briefly talk about the first two points. The subsequent example will then give a good intuition on how a complete parser will look like and work. The exact procedure can, of course, be studied in-depth in the paper of Hinze and Paterson [2].

We talked of semantic values and types of symbols and referred to them by just placing a dash over the symbol. But how do we know what these types are concretely? These types are actually specified by the creator of the grammar. They know that, for example, the type of the semantic value produced by an E should be `Expr`, because they defined `Expr` to be a suitable type depicting an AST. The type could also be a simple `Int` or whatever else – the main thing being that it is specified by the user.

Also, what is the type of a terminal? If the terminal has an associated value, like an *id* token, the type may be an `Int`. Other tokens may have no sensible semantic value, like a + token. Here, a parent type `Token` must exist (and must be specified

by the user) which can be used as the semantic type of *all* tokens. Tokens like *id* could use their own type like `Int`, but *every* token can always use the `Token` type if nothing more specific is known.

Let us briefly consider semantic actions. After a rule was fully parsed, its right-hand-side components are reduced to the rule's left-hand-side nonterminal. For example, when reducing $E \to E + T$, values of type $\overline{\mathtt{E}}$, $\overline{\mathtt{+}}$ and $\overline{\mathtt{T}}$ (or concretely, `Expr`, `Token` and `Term`) are taken and transformed to a single value of type $\overline{\mathtt{E}}$ (`Expr`). This is done by the semantic action of this rule. Working with continuation-passing style, this would look as follows:

```
actionE→E+T ::  (Ē -> Parser r) -> Ē -> +̄ -> T̄ -> Parser r
actionE→E+T k v1 v2 v3 = k (Plus v1 v3)
```

Here, `k` is the continuation while `v1`, `v2`, `v3` are the semantic values of the corresponding symbols. The `Plus` constructor then creates an `Expr` given an `Expr` and a `Term`. The semantic value of the + is not used because it is uninteresting. Finally, the continuation is called with the freshly created value of type `Expr`.

We now give an example implementation of two LALR state functions. The code should speak for itself – note that the tokens which are compared in the `case` statements are exactly the shift actions of the states, and the `g` functions are exactly the goto actions.

```
-- [T -> ( · E )]
state5 :: (Expr -> Token -> Parser r) -> Parser r
state5 k ts = case ts of
  t@(TokenInt v):tr -> state4 (action4 g5 v) tr -- on INT shift to state 4
  t@(TokenOB):tr -> state5 (action3 g5 t) tr -- on ( shift to state 5
  _ -> error
  where
    g4 x = state8 (action1 g4 x) (k x) -- on E goto state 8
    g5 x = state3 (action2 g4 x) -- on T goto state 3

-- k1: [E -> E · + T]
-- k2: [T -> ( E · )]
state8 :: (Token -> Term -> Parser r) -> (Token -> Parser r) -> Parser r
state8 k1 k2 ts = case ts of
  t@(TokenPlus):tr -> state7 (k1 t) tr -- on + shift to state 7
  t@(TokenCB):tr -> state9 (k2 t) tr -- on ) shift to state 9
  _ -> error
```

3. State Generation

The process of state generation proceeds in two main steps: After calculating the recognition points for all rules, RAD(1) states are generated based on existing LALR(1) states.

3.1. Recognition Points

Demers [5] defined the concept of recognition points when introducing Generalized Left Corner Parsing. The recognition point is, as formulated by Horspool, "the point in the rule's right-hand side which has been reached before the parser has unambiguously determined that it is this particular rule that is being matched" [1]. This is closely related to *free positions* – positions at which a semantic action can freely be inserted. Purdom and Brown [11] gave a graph-based algorithm to determine free positions.

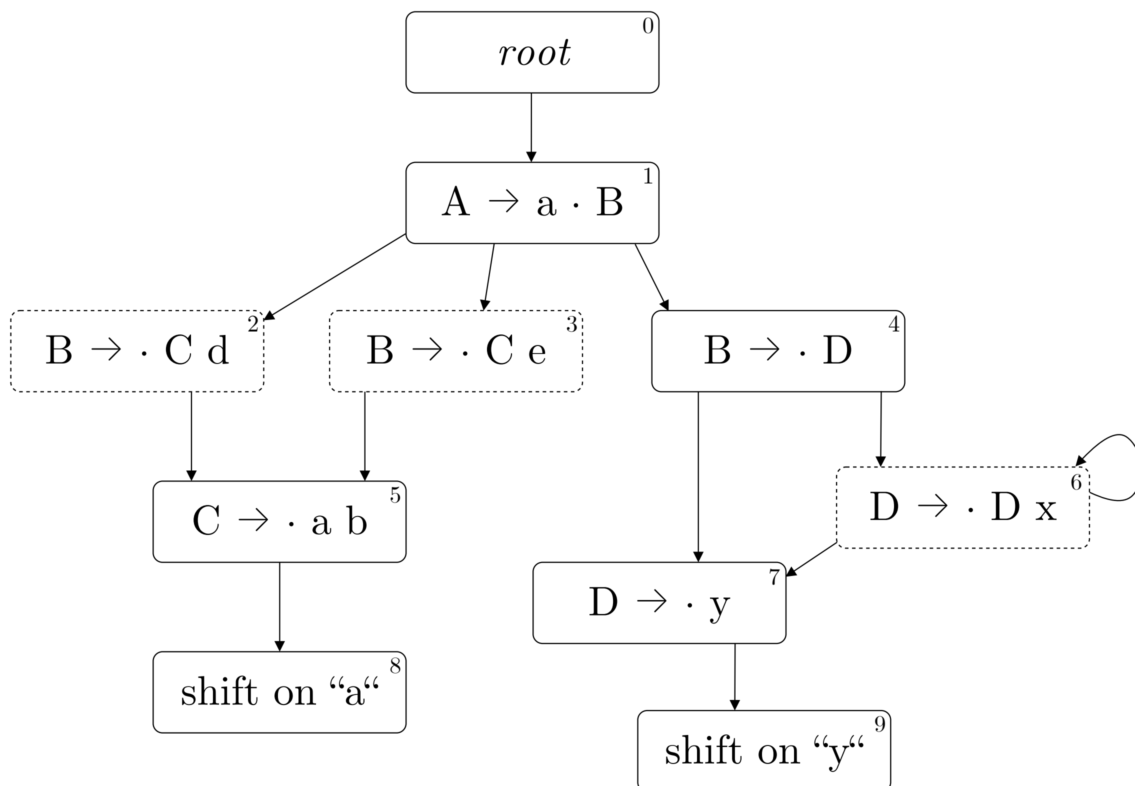
Scala-bison [6] uses a modified version of Purdoms and Browns algorithm to calculate free positions and derive the recognition points for all rules. We present it here.

In an LALR(1) parser, an LALR state depicts a single moment during the parsing process. If the state has multiple core or completion items, the parser is in all these positions simultaneously. The next token then decides what happens and either a shift or reduce action is performed. We define an item to be *free* with respect to an LALR state if the next executed action always safely determines whether this item is active or not. If an item is conversely *non-free*, this is only decided in a subsequent LALR state.

This can easily be visualized and calculated with a graph. We construct one graph per LALR state. The graph for the LALR state L is created as follows:

- Create one root vertex *root*.
- For each item c in the completion of L (the core is included in the completion), create one vertex $v(c)$.
- For each shift and reduce action of L, create one vertex.
- For each item c in the core of L, create an edge from *root* to $v(c)$.
- Create an edge $v1 \to v2$ between item vertices $v1$ and $v2$ iff $v2$ is of the form $[X \to \cdot\ \ldots]$, where the token after the dot in $v1$ is X. For example, there would be an edge from $[A \to\ b\ \cdot\ C]$ to $[C \to \cdot\ x\ y]$.

Figure 3.1.: An example graph of an LALR state with the single core item $[A \to a \cdot B]$. All non-free items have a dashed border.



- Create an edge between an item vertex v and a shift or reduce action vertex iff the action is directly caused by this item. A shift is directly caused by an item I if the shift is performed on the token which is after the dot in I. A reduce is caused by I when the reduced rule is the rule of I and when I has its dot at the right end.

Now, an item is *non-free* with respect to L iff there exists a leaf node (i.e. an action) which can be reached from the item's vertex, but is not *dominated* by it.

Figure 3.1 shows such a graph for an LALR state with the single core item $[A \to a \cdot B]$. Node 1 is free because every path from *root* to one of the leaf nodes (8 and 9) passes through node 1. Node 2 on the other hand is non-free because, even though node 8 is reachable from node 2, it is not dominated by node 2: there exists a path from *root* to node 8 which does *not* pass through node 2. Analogously, node 3 is non-free. Node 6 is also non-free: it does not dominate node 9 even though it reaches it. All other items are free.

If an item is non-free in at least one state graph, it is called non-free. Only if it is free in every state graph where it appears, it is called free.

We can now define the recognition point of a rule to be the leftmost position inside

the rule which is free and where all following positions are free. For example, when the item $[B \to \cdot\ C\ d]$ is non-free, the recognition point of the corresponding rule $B \to C\ d$ cannot be at position 0. It could either be at position 1 or at the end, depending on whether the item $[B \to C\ \cdot\ d]$ is free or not.

In an LALR(1) grammar, the rightmost item of a production is always free (as for example shown by Horspool [1]) – this is because, when an LALR state has an item with its dot at the rightmost position, it *must* decide whether to reduce this item's rule oder not. If it couldn't decide this at that moment, the grammar wouldn't be LALR(1).

Exactly if all rules have their recognition point at the very left, the grammar is LL(1).

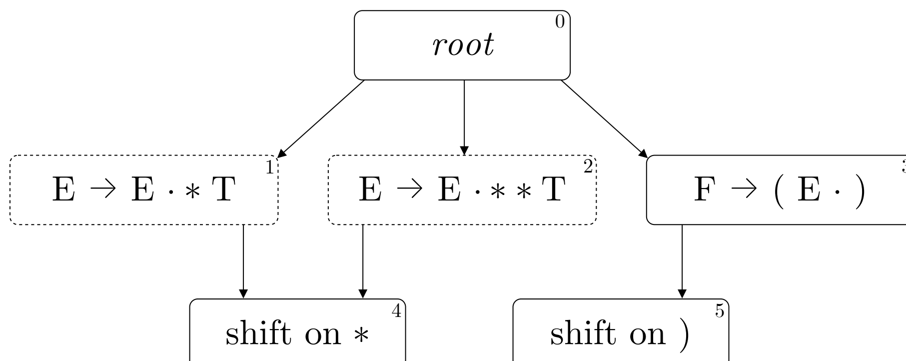
Finally we present an exemplary grammar with its recognition points. The grammar is a simple expression grammar extended by an exponentiation operation, denoted by "$**$".

$$
\begin{array}{lllll}
0: & E^\star \to & E\ \$ & \qquad & E^\star \to\ \bullet\ E\ \$ \\
1: & E \to & E\ *\ T & & E \to E\ *\ \bullet\ T \\
2: & E \to & E\ *\ *\ T & & E \to E\ *\ \bullet\ *\ T \\
3: & E \to & T & & E \to\ \bullet\ T \\
4: & T \to & T\ +\ F & & T \to T\ \bullet\ +\ F \\
5: & T \to & F & & T \to\ \bullet\ F \\
6: & F \to & (\ E\ ) & & F \to\ \bullet\ (\ E\ ) \\
7: & F \to & id & & F \to\ \bullet\ id
\end{array}
$$

Five of the eight rules have their recognition point at the very left (position 0), while no rule has its recognition point at the very right.

The graph in figure 3.2 justifies the position of the recognition points for rule 1 and 2. Consider the LALR state S with the core items $[E \to E\ \cdot\ *\ T]$, $[E \to E\ \cdot\ *\ *\ T]$ and $[F \to (\ E\ \cdot\ )]$ (and no further completion items) and its graph:

Figure 3.2.: The graph of state S. All non-free items have a dashed border.



One immediately sees that the items $[E \to E \ \cdot \ * \ T]$ and $[E \to E \ \cdot \ * \ * \ T]$ are both non-free, which means that the recognition point of the respective rules it *at least* at position two.

This grammar is used as an example in the rest of this work, and the appendix contains complete code examples for this grammar.

3.2. About RAD States

As discussed in section 2.3, when in top-down-mode, a RAD parser parses a nonterminal by entering a special entry state which is designed to parse just this nonterminal, switching to bottom-up mode. The parse of this nonterminal ends by reaching the nonterminal's exit state and accepting the nonterminal.

This means that there are three different types of states in a RAD parser: entry states, exit states and auxiliary states (which are neither entry nor exit states). We will describe them and how they are constructed in the following sections.

Any RAD state consists of the following parts:

- A set of core items
- A set of completion items
- A set of shift actions, announce actions and accept actions
- A set of goto actions.

In addition, entry and exit states have a designated nonterminal – this will be discussed later.

- Shift and goto action are known from LR and LALR parsers and work identically: on seeing a specific token, the parser may *shift* and enter a different (or the same) state. When coming back from a state (which has been entered either through a shift or an announce action), a goto action may be performed and a new state is entered.
- An announce action *announces* a specific rule after seeing, but without consuming, a specific token. When a rule is announced, the remaining symbols of the rule – all symbols after the recognition point – are parsed in a top-down fashion via switching to the recursive descent part.
  RAD states do not have reduce actions, as a reduce action is just a special case of an announce action – in particular, when a rule has its recognition point at the very right, an announce action behaves similar to a reduce action, as no further symbols have to be parsed and the rule is reduced to its left-hand side. If the recognition point is not at the very right, further symbols are parsed before reducing the rule's full right-hand side to its left-hand side.

- Accept actions are only present in exit states and denote that the partial parse of the exit state's nonterminal has ended and control is passed back to the active rule function which entered the nonterminal's entry state. Like announce actions, accept actions do not consume the current token.

The main difference between entry, exit and auxiliary states is, beneath their purpose, the presence of an *artificial item* in the state's core. An entry state with the nonterminal NT always contains the artificial core item $[_ \rightarrow \cdot\ NT]$, while a respective exit state contains the artificial core item $[_ \rightarrow NT\ \cdot]$ – auxiliary states do not contain an artificial core item. The left-hand side "_" signifies that this parse does **not** depend on the rule or context in which NT was encountered – *every* top-down parse of NT proceeds identically, irrespective of the rule or state that was previously active! This is, of course, the reason that a RAD parser often has significantly less states than a respective LALR parser: a RAD parser "reuses" states where possible.

3.2.1. RAD-Completion

Let's consider core and completion items. An item in an LALR state's core or completion can either produce more completion items (if its dot is before a nonterminal), produce a shift action (if its dot is before a terminal), or produce a reduce action (if its dot is at the right end). The recognition graphs from above visualize exactly this idea. In particular, every item with a dot in front of a nonterminal NT recursively results in new items in the completion, all of the form $[NT \rightarrow \cdot\ \ldots]$.

In a RAD state, when an item's dot is at the recognition point, the respective rule will be announced when seeing a matching token. Instead of entering a specific state that depends on the exact token, *every* matching token will announce the rule and begin a top-down parse of its remaining symbols. In other words: An item of the form $[A \rightarrow \cdot\ B]$ (where the dot is at the recognition point) will **not** yield any shift or reduce actions for any item of the form $[B \rightarrow \cdot\ \ldots]$ – instead of shifting or reducing to a specific state, rule $A \rightarrow \bullet\ B$ is simply announced. This means that items of the form $[B \rightarrow \cdot\ \ldots]$ **do not** belong in the RAD state's completion.

We therefore characterize the *RAD-completion* of a set of items as follows: An item with its dot before a nonterminal NT is only further completed (by all rules of the form $[NT \rightarrow \cdot\ \ldots]$) if the dot is *before* its recognition point.

This will later lead to the fact that all RAD states only have core and completion items where the dot is at or before the recognition point – never afterwards.

We remark that the artificial item of an entry state, $[_ \rightarrow \cdot\ NT]$, has no notion of recognition point and is therefore always completed at least one step with all items of the form $[NT \rightarrow \cdot\ \ldots]$.

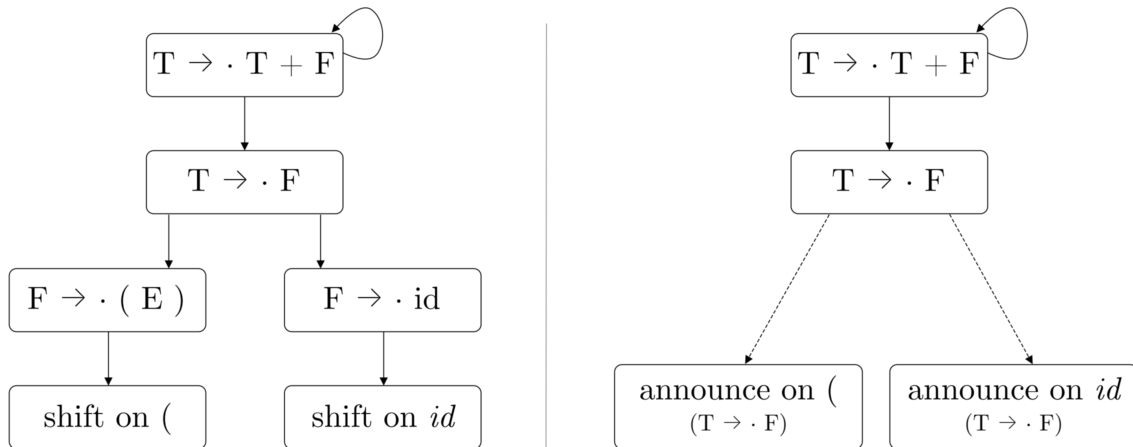
As an example, recall the extended expression grammar from above:

$0: E^\star \to E\ \$$	$E^\star \to \bullet\ E\ \$$
$1: E \to E * T$	$E \to E * \bullet\ T$
$2: E \to E * * T$	$E \to E * \bullet * T$
$3: E \to T$	$E \to \bullet\ T$
$4: T \to T + F$	$T \to T \bullet + F$
$5: T \to F$	$T \to \bullet\ F$
$6: F \to (E)$	$F \to \bullet\ (E)$
$7: F \to id$	$F \to \bullet\ id$

Consider a state with the single core item $[T \to \cdot\ T\ +\ F]$. This item will yield one more completion item: $[T \to \cdot\ F]$. The item $[T \to \cdot\ F]$ is not further completed as its dot is already at the recognition point.

Now, intuitively, shift and reduce actions of an LALR state which emerged from completion items that are *no longer* in the RAD-completion of a respective RAD state will get transformed into announce actions. While this idea will be formalized later, one can receive illustrative insight using the state graphs from above: In figure 3.3, the actions *shift on (* and *shift on id* are transformed into announce actions. In particular, the item from which these actions emerged in the state graph – $[T \to \cdot\ F]$ – will be announced. The dot is exactly at the recognition point, which is of course no coincidence.

Figure 3.3.: Left: state graph of an LALR state with the core item $[T \to \cdot\ T\ +\ F]$. Right: state graph of a respective RAD state with the same core item.



3.3. Unambiguous Nonterminals

We now shortly introduce the notion of *unambiguous nonterminals*, as they have been called by SCALA-BISON. A nonterminal NT is called *unambiguous* in a grammar if there exists some rule R in which NT appears somewhere after R's recognition point.

In the extended expression grammar from above, every nonterminal (expect for

$E^\star$) is unambiguous. The following grammar shows that this is not always the case:

$$
\begin{array}{llll}
0: & A^\star \to A \; \$ & \quad & A^\star \to \bullet \, A \; \$ \\
1: & A \to B \, * \, C & & A \to B \, * \bullet \, C \\
2: & A \to B \, * \, * \, C & & A \to B \, * \bullet \, * \, C \\
3: & B \to b & & B \to \bullet \, b \\
4: & C \to c & & C \to \bullet \, c
\end{array}
$$

A and C are unambiguous because they appear after the recognition point of some rule, while B does not and is therefore not an unambiguous nonterminal.

Note that the start symbol S of any grammar is always unambiguous as we always consider the extended grammar containing the extra production $S^\star \to S \; \$$, having the new start symbol $S^\star$.

A direct consequence is the following: If a nonterminal NT is unambiguous then there exists at least one LALR state $S =: entry(NT)$ which has a core or completion item I where both:

- I's dot is at or after the recognition point and
- I's dot is immediately before NT.

This is clear because, per definition, there exists such an item I. As long as the rule is both reachable from the start symbol and can actually produce a string of terminals (which we both assume because we can just remove all rules that don't), it follows that this item must exist in some LALR state.

Analogously holds: If a nonterminal NT is unambiguous then there exists at least one LALR state $S =: exit(NT)$ which has a core or completion item I where both:

- I's dot is at or after the recognition point and
- I's dot is immediately *after* NT.

As an example, using the small grammar from above: A is unambiguous. The only rule containing A is $A^\star \to \bullet \, A \; \$$. Therefore, there exists an LALR state $entry(A)$ which has the item $[A^\star \to \cdot \, A \; \$]$ and an LALR state $exit(A)$ which has the item $[A^\star \to A \, \cdot \, \$]$.

3.4. Algorithmic Generation

We can now describe the algorithmic generation of all RAD states. The following algorithm is mostly borrowed from the SCALA-BISON code and paper.

1. Begin with a grammar and its set of LALR states.
2. Calculate all recognition points and unambiguous nonterminals.

3. For each unambiguous NT, create two *RAD state skeletons*: an entry and an exit state (as described below).
4. These entry and exit state skeletons which have just been created are now extended to full RAD states. In this *finalization* process, new state skeletons may be created.
5. If there are new state skeletons, extend all of them to full RAD states by *finalizing* them. This may again yield **new** state skeletons.
6. Repeat step 5 until no new state skeletons have been created.
7. Once there are no more state skeletons, the algorithm has finished. It has produced a set of (finalized) RAD states.

Why do we need the distinction between state skeletons and finalized states?

A state skeleton just describes three fundamental properties of a RAD state: its type, its associated LALR state and its set of core items. The state skeleton does not yet contain any shift, announce, accept and goto actions.

These actions are then created during the finalization process, by using and modifying the actions of the associated LALR state. During this action modification process, it may become necessary to create a new RAD state, e.g. as a target for a shift action. If this is the case, we can just create a new state skeleton for this target. It will then be finalized later.

As an example, a finalized RAD state could look as follows:

Entry state for A

Items:

$[_ \to \cdot A]$ (core)
$[A \to \cdot x + B]$
$[A \to \cdot x + + B]$
$[A \to \cdot C]$
$[A \to \cdot D]$

Actions:

on x	shift to state 15
on c	announce rule 3: $A \to \bullet\ C$
on d	announce rule 4: $A \to \bullet\ D$

The skeleton would only contain the information about the type (entry), the associated LALR state and the core.

The recursive finalization process of a set of RAD states and skeletons can be described as follows:

```
recFinalize :: [Skeleton] -> [RADState] -> ([Skeleton], [RADState])
recFinalize [] states = ([], states)
recFinalize skeletons states =
  let finalized = map finalize skeletons
      newStates = map fst finalized
      newSkeletons = concatMap snd finalized
  in recFinalize newSkeletons (states ++ newStates)

-- "finalize" finalizes a skeleton to a full state, and possibly
↪  creates more skeletons
finalize :: Skeleton -> (RADState, [Skeleton])
```

We describe the two main processes in detail: creation of state skeletons, and finalization of state skeletons.

3.4.1. State Skeleton Creation

There are two situations in which a state skeleton is created:

1. At the beginning, an entry and exit skeleton is created for each unambiguous nonterminal.
2. During the finalization process, further state skeletons are created.

State skeletons created in the first way are of type *entry* or *exit*, while skeletons created in the second way are always *auxiliary* states.

Every RAD state is associated to an LALR state. In the finalization process, this state will be used as a foundation for the action creation. We call this LALR state the *associated* state and denote it with $assoc(S)$, where S is a RAD state or skeleton.

So, a state skeleton is created with three parameters: its type (entry, exit or auxiliary), its associated LALR state and a set of core items. These core items can be different than the core items of the associated state – for example, entry states have the *single* core item $[_ \rightarrow \cdot\ NT]$.

The completion of a RAD state is always calculated via the previously-defined *RAD-completion* and only depends on a state's core.

Entry And Exit States

We now look at the creation of entry- and exit-skeletons for an unambiguous nonterminal.

An unambiguous nonterminal NT yields two skeletons: S_{entry} and S_{exit}.

We first consider S_{entry} which is specified as follows:

- **type**: entry

- **assoc. state**: $entry(NT)$
- **core**: $\{[_ \to \cdot\ NT]\}$

Here we make use of the properties of $entry(NT)$: For an unambiguous nonterminal NT, there exists an LALR state $entry(NT)$ which has an item of the form $[X \to \ldots \cdot NT\ \ldots]$ with the dot at or after the recognition point. In an LALR parser, this state has the job to begin parsing NT when it sees a matching token in $first(NT)$. Then, control is passed to a different state, depending on the token that was read. The state may also have other different, unrelated items in its core. Later, when finalizing the entry-skeleton, we just add and transform the actions and gotos which are actually relevant for parsing NT – the other actions and gotos are ignored in the finalization process.

Therefore, it suffices to include the single core item $[_ \to \cdot\ NT]$ and later add all relevant actions.

Before considering S_{exit}, we think about the following general question: When having an LALR state L and a goto action *on* S *goto* G, where S is any symbol, what core items does G have?

When L has a core or completion item $I = [X \to \ldots \cdot S \ldots]$, where the dot is before the symbol S, then G needs to have a *core* item with the dot after this S, i.e. $[X \to \ldots S \cdot \ldots]$.

Hinze and Paterson [2] therefore introduced the following operation:

$$Q + X := \{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in Q\},$$

where Q is a set of items, X a terminal or nonterminal, and α and β are strings of symbols of arbitrary length.

Then, the core of G can simply be defined as $core(G) := completion(L) + S$.

Now we can consider S_{exit}. Of course, we will later need a goto action for S_{entry}: *on* NT *goto* S_{exit}. We specify S_{exit} as follows:

- **type**: exit
- **assoc. state**: $exit(NT)$
- **core**: $completion(S_{entry}) + NT$

Here, $completion(S_{entry})$ is the completion (which is calculated via the above-defined RAD-completion) of S_{entry}'s core.

Consider the extended expression grammar:

$$
\begin{array}{llll}
0: & E^{\star} \rightarrow E\ \$ & \quad & E^{\star} \rightarrow \bullet\ E\ \$ \\
1: & E \rightarrow E\ *\ T & & E \rightarrow E\ *\ \bullet\ T \\
2: & E \rightarrow E\ *\ *\ T & & E \rightarrow E\ *\ \bullet\ *\ T \\
3: & E \rightarrow T & & E \rightarrow \bullet\ T \\
4: & T \rightarrow T\ +\ F & & T \rightarrow T\ \bullet\ +\ F \\
5: & T \rightarrow F & & T \rightarrow \bullet\ F \\
6: & F \rightarrow (\ E\ ) & & F \rightarrow \bullet\ (\ E\ ) \\
7: & F \rightarrow id & & F \rightarrow \bullet\ id
\end{array}
$$

We now state, as an example, the core and completion items of the entry and exit skeletons for all unambiguous nonterminals, E, T and F. The completion is no additional property of a skeleton or RAD state as it is always directly calculated via the RAD-completion of the skeleton's core.

	core	additional completion
$S_{entry}(E)$	$\{[_ \rightarrow \cdot\ E]\}$	$\{[E \rightarrow \cdot\ E\ *\ T],\ [E \rightarrow \cdot\ E\ *\ *\ T],\ [E \rightarrow \cdot\ T]\}$
$S_{entry}(T)$	$\{[_ \rightarrow \cdot\ T]\}$	$\{[T \rightarrow \cdot\ T\ +\ F],\ [T \rightarrow \cdot\ F]\}$
$S_{entry}(F)$	$\{[_ \rightarrow \cdot\ F]\}$	$\{[F \rightarrow \cdot\ (\ E\ )],\ [F \rightarrow \cdot\ id]\}$

	core	add. compl.
$S_{exit}(E)$	$\{[_ \rightarrow E\ \cdot],\ [E \rightarrow E\ \cdot\ *\ T],\ [E \rightarrow E\ \cdot\ *\ *\ T]\}$	$\emptyset$
$S_{exit}(T)$	$\{[_ \rightarrow T\ \cdot],\ [T \rightarrow T\ \cdot\ +\ F]\}$	$\emptyset$
$S_{exit}(F)$	$\{[_ \rightarrow F\ \cdot]\}$	$\emptyset$

The additional core items of an exit state, beneath $[_ \rightarrow NT\ \cdot]$, are always introduced due to left-recursive rules of the form $NT \rightarrow NT \ldots$, or mutually recursive rules of the form $NT \rightarrow A \ldots$ and $A \rightarrow NT \ldots$.

Note that no entry and exit states are required for terminals which are *not* unambiguous – because they don't appear after any rule's recognition point, they are never parsed in top-down mode.

3.4.2. Finalization – Transforming Actions

Now we consider the finalization process: Given a **state skeleton S**, we create a finalized RAD state, and possibly new state skeletons.

A state skeleton S is defined by three properties: its type, its core and its associated LALR state $assoc(S)$.

The process of finalizing then takes the shift, reduce and goto actions of $assoc(S)$ and transforms them into shift, announce, accept and goto actions of S. In addition, entry and exit states may receive further actions. We will consider these one by one.

Before beginning, we need to modify Hinze-Paterson's definition of "+" to exclude

any items where the dot has advanced after the recognition point:

$$Q +_{rad} X := \{I \in Q + X \mid I\text{'s dot is before or at the recognition point}\}$$

Shift Actions

Consider a shift action $shift$:= *on t shift to state Q* of $assoc(S)$ where t is a terminal and Q another LALR-state. This shift action can either translate to a shift action, an announce action, or no action at all, as follows:

1. if $newCore := completion(S) +_{rad} t$ is non-empty:
   - **Create new or reuse auxiliary state** A with core = $newCore$ and associated state Q
   - **Add shift action** to S: *on t shift to state A*.
2. else, if R := `getAnnouncedRule(shift)` is non-null:
   - **Add announce action** to S: *on t announce rule R*.
3. else: discard the shift action.

We will discuss these three points now.

1. If there is an item of the form $[X \rightarrow \ldots \cdot\ t\ \ldots]$ in the completion of S, where the dot is **before** the recognition point, then a normal shift must occur, just like in an LALR parser. But we do not want to shift beyond the recognition point; if the dot is at or after the recognition point, this could lead to an announce action, but never to a shift action. This justifies the use of the "$+_{rad}$" operation (instead of "+").
   What does *create new or reuse auxiliary state* mean? We need an auxiliary state with the given core items and the given associated state (which will be relevant when finalizing the new state). If such a state was already created – with the same core and associated state – during the finalization process of another RAD state, it can (and must) be reused. This avoids creating the same state multiple times.
   If no state can be reused, a new state skeleton is created.

2. We will define `getAnnouncedRule` below. Intuitively, if the shift action belongs to a completion item of $assoc(S)$ which is *no longer* in the RAD-completion of S itself, then the corresponding rule will be announced. For example, in figure 3.4, consider a RAD state which has the two yellow items as its completion. The actions *shift on (* and *shift on id* are translated into announce actions: *announce* $T \rightarrow \bullet\ F$.

3. If a shift action neither translates to a shift nor an announce action, then it is completely unrelated to S. It came from an item in $assoc(S)$ which is neither in S's completion nor does it have a parent in the state graph of $assoc(S)$. Therefore, it is simply discarded. This is the case for the action *shift on* $*$ in figure 3.4 – it doesn't relate to any of the yellow items.

GetAnnouncedRule

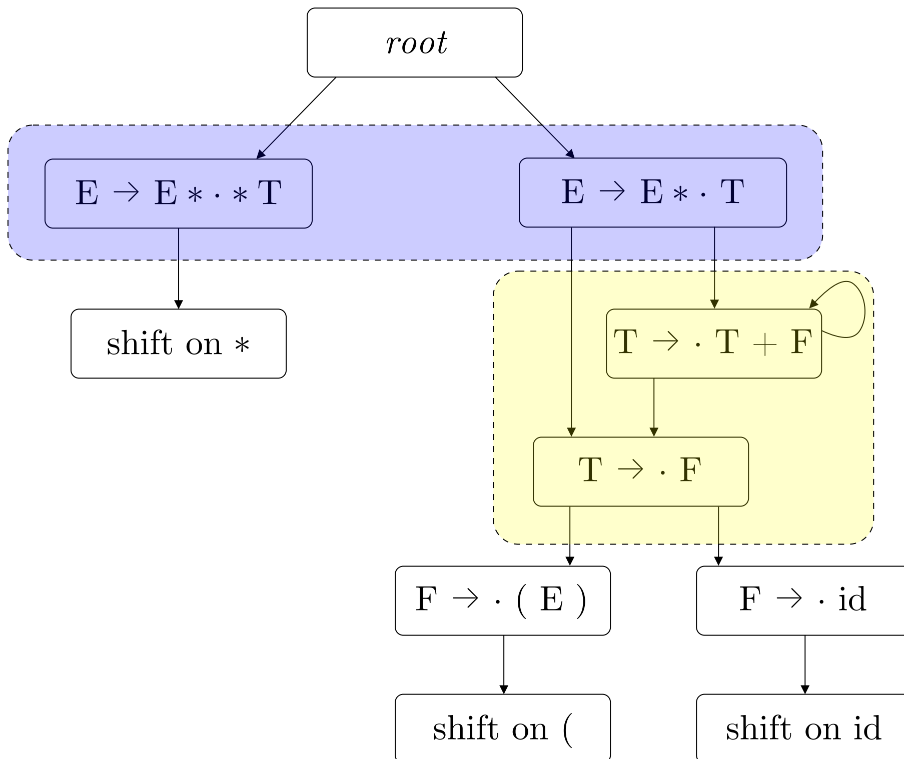
As an example of how shift actions are transformed, consider the following LALR state and its state graph.

Items:
$[E \to * \ \cdot \ T]$ (core)
$[E \to * \ \cdot \ * T]$ (core)
$[T \to \cdot \, T \ + \ F]$
$[T \to \cdot \, F]$
$[F \to \cdot \, (\ E \ )]$
$[F \to \cdot \, id]$

Actions:

on (	shift to state 2
on $*$	shift to state 11
on id	shift to state 1
on T	goto state 12
on F	goto state 5

Figure 3.4.: State graph of the LALR state from above. Yellow: completion of an entry state. Blue: core and completion of an auxiliary state.



Now let us consider an entry state with the nonterminal T. It has the single core item $[_ \rightarrow \cdot\ T]$. Its completion is marked in yellow in figure 3.4.

Now consider the shift action *shift on (*. How do we translate it?

Well, it cannot be translated to a shift action – the relevant item $[F \rightarrow \cdot\ (\ E\ )]$ is not in the (yellow) completion of our RAD state. This is because the item $[T \rightarrow\ \cdot F]$ has its core at the recognition point. In other words: this item is **ready** to announce its rule. Therefore, when encountering a (, we announce the rule $T \rightarrow \bullet\ F$.

The same happens for the action *shift on id*: On *id* we also announce the rule $T \rightarrow \bullet\ F$. What happens for the third shift action of the LALR state, *shift on* $*$? It is completely irrelevant to our entry state and should be discarded.

This gives us an intuition for what `getAnnouncedRule` should do. We will now describe it algorithmically. Thereby, `getAnnouncedRule` takes a shift or reduce action of an LALR state L and returns either a rule to be announced, or nothing ($null$).

Let S be a RAD state skeleton. Considering the state graph of $assoc(S)$, when a shift action is a (direct or indirect) child node of an item which has its dot at the recognition point, the shift action cannot be translated to a shift action in S. Instead, the rule of this item is announced. This works because the dot of the item is definitely at the recognition point (because this vertex is a cut-off point), which means the rule can be announced and the symbols after the recognition points can be parsed top-down.

We can find this item and its rule as follows:

1. begin with the shift or reduce action vertex in the recognition graph of $assoc(S)$
2. go to any parent vertex
3. if this vertex is in the RAD-completion of S *and* its dot is at the recognition point, return this item's rule
4. else:
    - if the item has no parent (i.e. it is in $assoc(S)$'s core): return null
    - if the item has a parent: go to any parent and go back to step 3.

If a vertex has multiple parents, we must follow all possible paths to the top until finding an item with the desired properties – the item is in the RAD-completion of S and has its dot at the recognition point. If there are multiple different items with this property, we can choose the one we want to announce freely, or we can output an *announce conflict* because now there are multiple, possibly different valid derivations.

Note that the artificial item of an entry or exit state cannot be announced – it is not part of the state graph.

As an example, we look at two RAD-states, **both** emerging from the same associated LALR state, shown in figure 3.4.

1. The first RAD state we consider (which already served as an example above) is an **entry state** with the **nonterminal** $\boldsymbol{T}$, having the single core item $[_ \to \cdot\, T]$ – its associated state is the LALR state from above. Its completion is marked in yellow in figure 3.4.
   First we note that none of the shift actions are translated into shift actions. The question is whether and which rule should be announced for each shift action.
   - The action *shift on* $*$ is discarded: there is no yellow parent node.
   - Both actions *shift on* (and *shift on id* are translated to *announce rule* $T \to \bullet\, F$, as this is the first yellow parent item which has its dot at the recognition point.
2. The second RAD state we consider is an **auxiliary state** with the two core items $[E \to E\ *\ \cdot\ *\ T]$ and $[E \to E\ *\ \cdot\ T]$, marked in blue in figure 3.4. Here, also, its associated state is the same LALR state from above, and none of the shift actions are translated into shift actions. But we can easily see the following:
   - The action *shift on* $*$ is translated to *announce rule* $E \to E\ *\ \bullet\ *\ T$.
   - The actions *shift on* (and *shift on id* are translated to *announce rule* $E \to E\ *\ \bullet\ T$.

As you see, this LALR state is actually used as a basis for two different RAD states! Thereby, one of these RAD states entails "reusable" parts of the LALR state, i.e. the parse of a T.

Reduce Actions

Again, let S be a state skeleton. Consider a rule $R = A \to \omega$, where ω is an arbitrary string of symbols, and a reduce action $reduce$:= *on t reduce rule R* of $assoc(S)$ where t is a terminal. The reduce action can either translate to an announce action or to no action at all, as follows:

1. if $[A \to \omega\ \cdot] \in completion(S)$:
   - **Add announce action** to S: *on t announce rule* $A \to \omega\ \bullet$.
2. else, if $R' := getAnnouncedRule(reduce)$ is non-null:
   - **Add announce action** to S: *on t announce rule* R'.
3. else: discard the reduce action.

Why?

1. As stated earlier, a reduce action is a special case of an announce action. In particular, when a rule has its recognition point at the very right, an announce action behaves similar to a reduce action, as no further symbols have to be parsed and the rule is reduced to its left-hand side.
   Also, every completion item of a RAD state has its dot before or at the recognition point. Therefore, if $[A \to \omega\ \cdot] \in \mathit{completion}(S)$, the recognition point **must** be at the very right, which means we can announce the rule (and immediately reduce it).

2. Just like in the case of shift actions, when the item which is related to the reduce action (in this case $[A \to \omega\ \cdot]$) does not belong to the RAD state's completion, the rule of the RAD-completion-item from which this reduce action emerged must be announced. Therefore we use `getAnnouncedRule` and do not give an example because this works absolutely the same way as in the shift case.

3. If the reduce action belonged to an item of $\mathit{assoc}(S)$ which is completely unrelated to S itself, it is simply ignored.

Accept Actions

Accept actions are only present in exit states. If a full parse of the nonterminal NT of the exit state S was performed, NT is accepted.

Consider, as an example, the expression grammar:

$$
\begin{array}{llll}
0: & E^\star \to\ E\ \$ & \quad & E^\star \to\ \bullet\ E\ \$ \\
1: & E\ \to\ E\ *\ T & & E\ \to\ E\ *\ \bullet\ T \\
2: & E\ \to\ E\ *\ *\ T & & E\ \to\ E\ *\ \bullet\ *\ T \\
3: & E\ \to\ T & & E\ \to\ \bullet\ T \\
4: & T\ \to\ T\ +\ F & & T\ \to\ T\ \bullet\ +\ F \\
5: & T\ \to\ F & & T\ \to\ \bullet\ F \\
6: & F\ \to\ (\ E\ ) & & F\ \to\ \bullet\ (\ E\ ) \\
7: & F\ \to\ id & & F\ \to\ \bullet\ id
\end{array}
$$

Let us consider the exit state of E. We need to accept E on every token that can come after E in some top-down parse of E. Imagine some rule has been announced, and it just parsed an E. Now, the next valid token which can follow the E must be the token which comes after the E in the currently active rule.

Which tokens could this be? E appears in the right-hand side of the following rules: $E^\star \to \bullet\ E\ \$$, $F \to \bullet\ (\ E\ )$, $E \to E\ *\ \bullet\ T$ and $E \to E\ *\ \bullet\ *\ T$. In the last two of these rules, E cannot be parsed in a top-down parse because it comes before the recognition point. Therefore the last two rules do not need to be considered.

So what tokens can follow a top-down parse of E? In the rule $E^\star \rightarrow \bullet\ E\ \$$, a "\$" can follow the parse, and in the rule $F \rightarrow \bullet\ (\ E\ )$, a ")" can follow the parse. This is already the whole top-down-follow-set of E: `rad-follow(E)` = {")", "\$"}.

We define `rad-follow(NT)` for a nonterminal NT as follows:

1. For every occurrence of NT in a rule $A \rightarrow \alpha\ NT\ \beta$, where α and β are arbitrary strings of symbols and NT appears after the recognition point:
   - If β can produce ε (i.e. $\varepsilon \in first(\beta)$): add $follow(A) \cup first(\beta) \backslash \{\varepsilon\}$ to `rad-follow(NT)`.
   - Else: just add $first(\beta)$ to `rad-follow(NT)`.

If the symbols in a rule after NT can produce ε – which is also the case if NT appears at the very right – we must of course add the full follow-set of the left-hand-side A in the list of tokens. Else, the first-set of the symbols after NT suffices.

Here, $first$ and $follow$ are functions which are already well-known from LALR and other parsers. Notice that the first step didn't just specify every rule, but every *occurrence* of NT in a rule, as NT may occur multiple times after the recognition point in the same rule.

After having defined `rad-follow`, we generate accept actions:

1. for each token $T \in$ `rad-follow(NT)`:
   - if there is not already another action for T:
     - **Add accept action** to S: *on T accept NT*

This means that we accept NT on all tokens that can possibly follow NT *in a top-down parse* **and** that have no other associated shift or announce actions. These other shift or announce actions could come from parsing the nonterminal itself and must have higher priority than the accept action – as long as we can continue parsing the nonterminal, we must do this; only when the nonterminal cannot be extended, it may be accepted.

Goto Actions

As described in section 3.4.1, an entry state S always receives a goto action to its corresponding exit state E: *on NT goto state E.*

Now let S be a state skeleton and consider a different[1] goto action *on X goto state G* of $assoc(S)$, where X is a nonterminal and G is an LALR state.

[1] Different means: in an entry state, the goto-action of NT is *not* transformed – a respective action already exists.

Similar to a shift action, after coming back from parsing an X, we want to shift the dot one position to the right in every item of the form $[A \to \ldots \cdot X \ldots]$. Because we do not want to shift past the recognition point, we use "$+_{rad}$".

Then, the goto action is transformed as follows:

1. if $newCore := completion(S) +_{rad} X$ is non-empty:
   - **Create new or reuse auxiliary state** A with core = $newCore$ and associated state G
   - **Add goto action** to S: *on* X *goto state* A.
2. else: discard the goto action.

Again, the goto action is discarded if its corresponding item is not relevant to the RAD state S. In this case, "$+_{rad}$" returns $\emptyset$ because either no item of the form $[A \to \ldots \cdot X \ldots]$ was in $completion(S)$ to begin with, or these items had their dots at their recognition points which means that one will never come back to this state after parsing an X: rather, X is parsed top-down after announcing the corresponding rule. In both cases, no goto action is required.

Epsilon-Announce Actions for Entry States

Finally, we add additional announce actions to entry states. Consider a nonterminal NT which can produce ε, i.e. $NT \xrightarrow{*} \varepsilon$, or equivalently, $\varepsilon \in first(NT)$.

Now consider S, the RAD entry state of NT. When seeing a token in `rad-follow(NT)`, NT must apparently be reduced to ε by announcing a matching rule. If there is a direct rule $NT \to \varepsilon$, i.e. if NT produces ε directly, we can just announce this rule. Then everything will work out: as there are no symbols to be parsed, announcing the rule does nothing. Then S executes its goto action on NT and goes to the exit state, which just accepts on a token in `rad-follow(NT)`, as seen above.

But what happens if there is no direct rule $NT \to \varepsilon$ – for example, if ε is produced indirectly, via $NT \to A$ and $A \to \varepsilon$? Here we must choose the correct rule to announce. This is in general **neither** a rule of the form $NT \to ...$ **nor** $A \to \varepsilon$.

Let's look at the following example grammar (the start symbol is S):

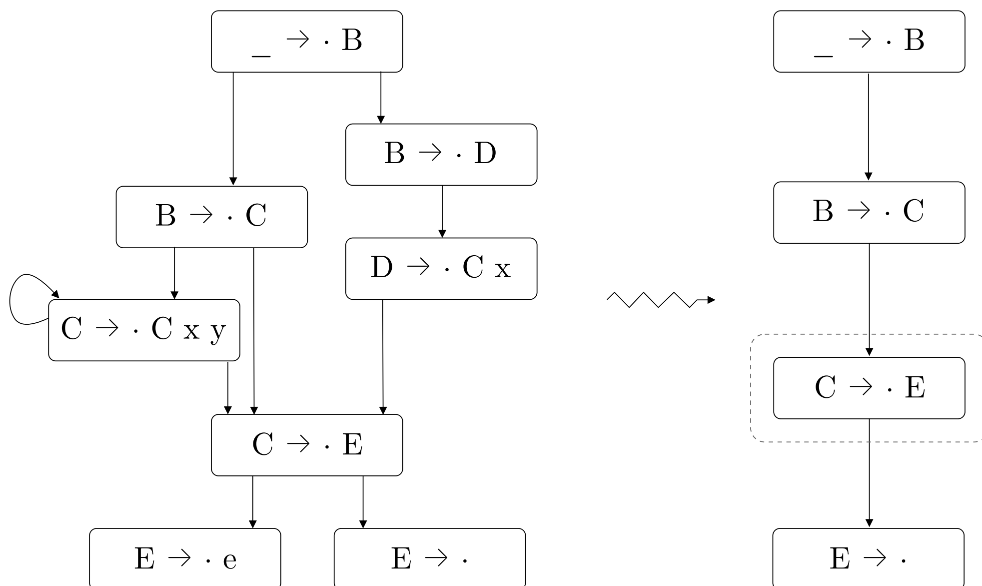
$0: S \to a\ B\ c$	$S \to \bullet\ a\ B\ c$
$1: B \to C$	$B \to C\ \bullet$
$2: B \to D$	$B \to D\ \bullet$
$3: C \to C\ x\ y$	$C \to C\ x\ \bullet\ y$
$4: C \to E$	$C \to \bullet\ E$
$5: D \to C\ x$	$D \to C\ x\ \bullet$
$6: E \to e$	$E \to \bullet\ e$
$7: E \to \varepsilon$	$E \to \bullet$

There is one epsilon-production, $E \to \varepsilon$. So, the entry-state of E would just announce the rule $E \to \varepsilon$ on all tokens in `rad-follow(E)` $= \{c, x\}$.

But now consider the entry-state of B, which has the following items:

[RAD state]: B's entry state:
$[_ \to \cdot\ B]$ (core)
$[B \to \cdot\ C]$
$[B \to \cdot\ D]$
$[C \to \cdot\ C\ x\ y]$
$[C \to \cdot\ E]$
$[D \to \cdot\ C\ x]$

Figure 3.5.: Left: the full state graph starting with $[_ \to \cdot\ B]$. Right: the reduced graph, only containing items that can produce ε.
The algorithm finally chooses the outlined item, $[C \to \cdot\ E]$, as the rule to announce.



We want to know which rule should be announced on `rad-follow(B)` $= \{c\}$. Therefore we apply the following algorithm for the nonterminal B, which is visualized in figure 3.5:

1. Build a state graph, starting with the core item $[_ \to \cdot\ B]$, and containing **all** LALR-completion items. This graph can contain items which are **not** in the RAD-completion of the state. This graph definitely contains a valid derivation of $B \xrightarrow{*} \varepsilon$ (because B can produce ε per assumption).

2. Remove all vertices whose right-hand-side does **not** produce ε, and remove

all edges connected with any of these vertices. For example, all items with a terminal in their right-hand-side are removed.

3. Now consider the remaining graph. It still contains a valid derivation of $B \xrightarrow{*} \varepsilon$, per assumption. It is seen on the right in figure 3.5.

4. If there are multiple leaf items of the form $X \to \varepsilon$, or if there are multiple ways from the top node to any leaf, output an *announce conflict*: there are multiple valid derivations of ε.

5. Else, there is a single, linear way from the top node to the leaf node (as seen on the right side of figure 3.5). We consider the remaining items one by one and find at least one rule that can be announced. It must fulfill the following conditions:

   - The dot must be at the recognition point. Because all items in the state graph have their dot at position 0, this is where the recognition point must be.
   - The item must be in the RAD-completion of S – there could be items which are in the LALR-completion of $[_ \to \cdot\ B]$ (which is what the graph consists of), but not in the RAD-completion of S.
   - The left-hand-side nonterminal of the item must have a goto action in S. This is no new condition as it follows from the first two.

6. There is always at least one valid item. Choose one – this is the rule which will be announced.

In figure 3.5, the right side shows the graph after removing items that do not produce ε. Which of these items is the rule that should be announced (which item is chosen as *valid* by steps 5 and 6 of the algorithm)?

- $[_ \to \cdot\ B]$: The artificial item cannot be announced.
- $[B \to \cdot\ C]$: This item has its dot before the recognition point and can therefore not be announced.
- $[C \to \cdot\ E]$: This item is valid.
- $[E \to \cdot]$: This item is not in the RAD-completion of S and can therefore not be announced.

One valid item remains, it is $[C \to \cdot\ E]$, so we choose to announce the rule $C \to \bullet\ E$ on each token in `rad-follow(B)`. If there were multiple valid options, we can just choose any one of them – this only changes the call sequence of state and rule functions, but not of the built derivation nor of semantic action functions.

We now point out an error in SCALA-BISON: it is this case of an indirect epsilon-production (i.e. $NT \not\to \varepsilon$, but $NT \xrightarrow{*} \varepsilon$) which is not handled by SCALA-BISON. Instead of considering whether the rule can be announced due to the position of its recognition point, SCALA-BISON **always** chooses to announce some rule of the form $NT \to \dots$. This is correct when NT produces ε directly, but not if it does so indirectly.

In the entry state of B from above, SCALA-BISON would erroneously choose to announce $B \to C$ (instead of $C \to E$), which is simply not possible because the recognition point of the rule is after the C ($B \to C\ \bullet$), but in the corresponding item of S ($[B \to \cdot\ C]$), the dot is before the C. So a SCALA-BISON-generated parser would skip a full C which eventually leads to some internal error or a wrong derivation.

In strongly-typed, continuation-based parsing code, such an error would already be detected at compile-time: skipping a symbol by accident would lead to a type error – some continuation expects a C, but announcing the rule $B \to C\ \bullet$ doesn't produce a C, et voilà, a type error.

3.5. Default Actions And Happy's Error Token

After a state was generated, we can choose an action as default action. A default action is an action which is executed if no other action matched the token. Default actions do not change the behavior of the parser, but make the states smaller, which will later lead to possibly faster execution.

Especially on RAD parsers, using default actions can often save *much* more space than on LALR parsers – this is because often, many different shift or reduce actions of an LALR state will be translated to the *same* announce action on the corresponding RAD state, which can then be taken as a single default action.

Actually, the behavior of the parser *is* changed: an erroneous token may not immediately be detected by a state which has a default action, but only later, when trying to shift over this token, or parse it directly via its `parseToken` function. Therefore, the exact state where the error occurs may change, but the token where it occurs remains the same.

With this in mind, we can define the following default actions:

- Announce actions can be defaulted: If an announce action is executed on a token where the parser should fail, the parser will fail on the next shift or parseToken action. If it wouldn't, then the announce wasn't erroneous in the first place.
- The same is true for accept actions.
- Shift actions cannot be defaulted as they consume a token, in contrast to announce and accept actions.

This means that we can take the largest accept or announce action and make it into a default action, removing as much individual actions from the state as possible, which reduces the size of the generated parser as much as possible.

Happy's Error Token

HAPPY has a special terminal, the so-called *error token*. This token works as a fallback: if no token matched the valid tokens of a state, the error token is matched and shifted automatically without consuming a token from the input. Consider a state with the items $[A \rightarrow a \ \cdot \ b]$ and $[A \rightarrow a \ \cdot \ error]$: on b, a normal shift is performed. On any other token, the error token is matched, and we come in a state of this form: $[A \rightarrow a \ error \ \cdot]$. But matching the error token does not consume the current token from the input – it will be processed by a subsequent state.

During state generation, the error token is handled just like any other token. It appears in items and in actions, i.e. *shift*, *announce* or *accept on error*.

We must consider two things when dealing with the error token: recognition points of affected rules and default actions.

- Because of their special nature, the recognition point of any rule must come after all error tokens in this rule. In other words, items with an error token after the dot are always non-free.
- Handling of error tokens can conveniently be implemented using default actions: the action of a state which is performed on the error token **must** be used as a default action. For example, *on error announce rule R* would yield the default action *announce rule R*.
  We must be careful with shift actions: The action *on error shift to state S* does translate to a default action *shift to state S*, but this default shift action does not consume a token from the input stream and is therefore a special shift action.

4. Code Generation

In the following we present the process of code generation for a RAD parser in continuation-based, strongly-typed style. The style is very similar to the style used by Hinze and Paterson [2], as presented in chapter 2.3.

As HAPPY generates Haskell code, we do the same and work with Haskell code in the following sections.

4.1. Algorithmic Generation

The continuation-based code of a RAD parser consists of the following parts:

1. User-supplied header
2. Entry points: Parse function for each parsing entry point
3. One state function per state
4. One rule function per rule
5. One parse function per terminal
6. One parse function per nonterminal
7. One semantic action function per rule
8. User-supplied footer

With the exception of the user-supplied parts, the generation of all parts will be discussed in detail in the following.

4.1.1. Parsing a Nonterminal

Consider an unambiguous nonterminal NT. As discussed in sections 3.3 and 3.4, a RAD parser generator creates an entry state and an exit state for this nonterminal. When parsing NT, it suffices to call the corresponding state function of the entry state E.

Following section 2.6, the generated parse function has the following type:

$$\texttt{parse}_{\texttt{NT}} :: (\overline{\texttt{NT}} \texttt{ -> Parser r) -> Parser r}$$

This is because $\texttt{parse}_{\texttt{NT}}$ takes a list of tokens and returns a fully parsed NT (having type $\overline{\texttt{NT}}$) and a list of remaining tokens, which can be rewritten in terms of `Parser r` using continuation-passing style (remember: `Parser r = [Token] -> r`).

Then, $\texttt{parse}_{\texttt{NT}}$ is just defined via $\texttt{parse}_{\texttt{NT}}$ `=` $\texttt{state}_{\texttt{E}}$, i.e. via NT's entry state.

4.1.2. Parsing a Terminal

Like in a top-down parser, parsing a specific terminal is equivalent to comparing the next input token with the token that belongs to the expected terminal. If the token matches, it is returned via the continuation; if not, a parse error is produced.

Just like in the nonterminal case, this function has the following type, considering some terminal T:

```
parseT ::  (T̅ -> Parser r) -> Parser r
```

Remember that $\overline{\texttt{T}}$ can either be the container type `Token` or a more specific type. This can be specified by the user of HAPPY – if a token constructor has a nonzero arity, one of its arguments can be used as the semantic value of the token.

As an example, first consider the terminal + and its associated token `TokenPlus`. The corresponding parse function looks as follows:

```
parse+ :: (Token -> Parser r) -> Parser r
parse+ k (t@(TokenPlus):tr) = k t tr
parse+ k ts = happyError ts
```

The function `parse+` takes two arguments, the continuation `k` and the token list. If the head of the token list matches the expected token, in this case `TokenPlus`, the continuation is called with the value of this token (which is `TokenPlus` itself) and with the remaining tokens, i.e. the tail of the token list. If not, an error is produced. HAPPY provides therefore the function `happyError :: [Token] -> r`.

Now consider a terminal with an associated value, such as *id*, and its associated token with the constructor `TokenID Int`. Instead of passing the whole `TokenID` constructor to the continuation, we can now just use the wrapped value of type `Int`. Hence, the corresponding parse function looks as follows:

```
parseID :: (Int -> Parser r) -> Parser r
parseID k ((TokenInt v):tr) = k v tr
parseID k ts = happyError ts
```

Here, the semantic value `v` is passed to the continuation which therefore has the type `Int -> Parser r` instead of `Token -> Parser r`.

4.1.3. Rule Functions

After having considered nonterminal and terminal functions, we can move on to rule functions.

A rule function parses the symbols of a rule after its recognition point, one after the other. After all symbols have been parsed, the continuation is called with the semantic values of every symbol.

This causes the type of the continuation function to be
$\overline{\alpha_1}$ `->` $\overline{\alpha_2}$ `-> ... ->` $\overline{\alpha_k}$ `-> Parser r`,
where $\alpha_1, \ldots, \alpha_k$ are the symbols after the recognition point.

There are two ways we can achieve the intended task of a rule function: tuple-based and continuation-based.

1. In tuple-based form, after the rule function has parsed a symbol, the remaining tokens are explicitly passed to the next symbol function while all semantic values are stored locally. Only after all symbols have been parsed, the continuation `k` is called with all semantic values.

2. In continuation-based form, the rule function parses a symbol, but instead of explicitly storing the result, it is passed to the continuation function which parses the next symbol. Thereby, a continuation chain is created, at the end of which is the call to the top-level continuation `k`.

Consider the rule $F \to \bullet$ (E). The corresponding rule function `rule0` must have the following signature:

```
rule0 :: (Token -> Expr -> Token -> Parser r) -> Parser r
```

In tuple-based form, `rule0` would look as follows:

```
rule0 :: (Token -> Expr -> Token -> Parser r) -> Parser r
rule0 k ts0 = k v1 v2 v3 ts3 where
  (v1, ts1) = parse( (,) ts0
  (v2, ts2) = parseE (,) ts1
  (v3, ts3) = parse) (,) ts2
```

`rule0` begins by parsing a (via `parse(`. Thereby, the provided continuation is `(,)`, which simply creates a tuple from the semantic value `v1` and the remaining tokens `ts1`. The remaining tokens are then used as explicit input to `parseE`, and so on.

At the end, after all three symbols were parsed, `k` is called with the semantic values `v1`, `v2` and `v3` and the token rest `ts3`.

Continuation-based form looks different:

```
rule0 :: (Token -> Expr -> Token -> Parser r) -> Parser r
rule0 k = parse( (cont1) where
  cont1 v1 = parseE (cont2 v1)
  cont2 v1 v2 = parse) (cont3 v1 v2)
  cont3 v1 v2 v3 = k v1 v2 v3
```

Here, `rule0` also begins by parsing a (. Instead of storing the result, the continuation `cont1` receives the semantic value `v1`. `cont1` then simply calls `parseE` and provides `cont2` as continuation. `cont2` calls `parse)` and with the continuation `cont3`, which now has all three semantic values as arguments. From here, `k` is called.

The token list is handled implicitly – all continuation functions receive the remaining token list as an additional argument after the semantic values. All declared functions are initially only partially applied, and are fully evaluated only when the token list is explicitly passed to `rule0`.

There are two special cases we can consider additionally to generate more concise code, depending on the number of symbols after the recognition point of the rule:

- When a rule has no symbol after its recognition point, nothing has to be done. In this case, the rule function is the identity.
  The rule $A \to B \bullet$ would emit the following function: `rule0 = id`.

- When a rule has a single symbol after its recognition point, after parsing this symbol, the continuation of the rule function is called. After noticing that the type of the rule function matches the type of the symbol's parse function, one can simply define the rule function directly via the symbol's parse function:
  The rule $A \to \bullet\ B$ would emit the function `rule0 = parseB`. The same works for a terminal after the dot: the rule $A \to \bullet\ b$ would emit the function `rule1 = parse_b`.

4.1.4. States

Consider a RAD state S. As discussed in section 3.2, a RAD state is defined by its set of core items, shift actions, accept actions, announce actions and goto actions. Also, a RAD state may have a default action (section 3.5).

A state function must execute the correct action, depending on the next input token. When returning from another state, the appropriate goto action must be executed. Finally, when one of the state's core items was parsed, the appropriate continuation must be called.

For this to work correctly, we generate the state functions the same way as described by Hinze and Paterson [2], with minor modifications for the RAD-specific parts.

Consider the following (fictional) RAD state as an example for this section.

State 4

Items:

$[A \to x\ \cdot\ +\ B]$ (core)
$[A \to x\ \cdot\ +\ +\ B]$ (core)
$[A \to \cdot\ C]$ (core)
$[A \to \cdot\ D]$ (core)
$[C \to \cdot\ x\ y]$
$[C \to \cdot\ x\ z]$
$[C \to \cdot\ E]$
$[C \to \cdot\ F]$

Actions:

on $+$	shift to state 10
on x	shift to state 15
on d	announce rule 4: $A \to \bullet\ D$
on e	announce rule 5: $C \to \bullet\ E$
on f	announce rule 6: $C \to \bullet\ F$
on C	goto state 2

We begin with the signature of the state function. As discussed in section 2.6, this function has one continuation per core item.

Our example state has four core items and its signature looks like this:

```
state4 :: (+̅ -> B̅ -> Parser r) -> (+̅ -> +̅ -> B̅ -> Parser r) ->
(C̅ -> Parser r) -> (D̅ -> Parser r) -> Parser r
```

Different actions are executed depending on the next input token. We use a switch:

```
state4 k1 k2 k3 k4 ts = case ts of
  (TokenPlus:tr) -> ...
  (Token_x:tr) -> ...
  (Token_d:tr) -> ...
  (Token_e:tr) -> ...
  (Token_f:tr) -> ...
```

We now translate each of the actions into code.

Shift Shift actions are translated to code exactly as described by Hinze and Paterson [2] – readers of [2] may skip this paragraph.

Consider a RAD state S. A shift action *on* T *shift to state* Q takes one or more completion items where the dot is before T and shifts the dot over T; state Q has these shifted items in its core. For each of the completion items, there are two cases:

- The item is in the core of S. Then, there is a continuation in `state_S`'s function declaration – say `k1` – which is associated to this core item. It expects as first argument a value of type $\overline{\mathtt{T}}$. This means that we can **partially apply** `k1` with the semantic value of T, and pass it as a continuation to `state_Q`.
  In the state from above, there is an action *on + shift to state 10* which includes **two** core items: $[A \to x\ \cdot\ +\ B]$ and $[A \to x\ \cdot\ +\ +\ B]$ – these are associated with `k1` and `k2`. State 10's core consists of exactly these two items in shifted form: $[A \to x\ +\ \cdot\ B]$ and $[A \to x\ +\ \cdot\ +\ B]$. This translates as follows:

  ```
  state8 k1 k2 k3 k4 ts = case ts of
    (t@(TokenPlus):tr) -> state10 (k1 t) (k2 t) tr
    ...
  ```

  We must not forget to pass the token rest `tr` to `state10` after passing the partially applied continuations.

- The item is **not** in the core of S, only in its completion. Then, there is no direct continuation to partially apply. We rather construct a continuation consisting of the semantic action of the item's rule and of the goto action of its left-hand-side. This process is motivated and derived in [2].
  Consider the non-core item I and its rule $R : A \to \omega$. The continuation that we construct looks as follows: `action_R g_A`. Thereby, `action_R` is the semantic action of the rule, while `g_A` is the local (i.e. belonging to state S) goto action of A which is introduced later.
  In our example state, there is the action *on x shift to state 15* including the non-core items $[C \to \cdot\ x\ y]$ and $[C \to \cdot\ x\ z]$. This is translated as follows:

  ```
  state8 k1 k2 k3 k4 ts = case ts of
    (t@(Token_x):tr) -> state15 (action7 g_C) (action8 g_C) tr
    ...
  ```

  Here, `g_C` is the local goto action of C, while `action7` and `action8` are the respective semantic actions of the rules $C \to x\ y$ and $C \to x\ z$.

These two cases can appear together if a shift action is associated to multiple items, some of which are in the state's core and some are not.

Announce When a rule R is announced, there is exactly one item I in the core or completion of our considered RAD state S that corresponds to this rule and has its dot at the rule's recognition point.

When announcing the rule R, we call its rule function `rule_R` and pass something sensible into its continuation. We do the same as in the shift case and distinguish on whether I is in S's core or not.

- If I is in the core of S, we announce the rule and then just call the continuation associated to I.
  In the example above, we can consider the core item $[A \to \cdot\ D]$ and its action

on d announce rule 4: $A \to \bullet\ D$. The associated continuation of the item is `k4`. This translates as follows:

```
state8 k1 k2 k3 k4 ts = case ts of
  (Token_d:_) -> rule4 k4 ts
  ...
```

The type of `k4` is `D̄ -> Parser r` and therefore exactly matches the type of the continuation that is expected by `rule4`.
An important thing to notice is that an announce action never consumes the token it has read. Instead, the token is ignored and the full token list (`ts`) is passed to `rule4`.

- If I is not in the core of S, we use the same construct as above. Let $R : A \to \omega$ be the rule of I. Then we construct the continuation as follows: `action_R g_A`, where `action_R` is the semantic action of the rule, while `g_A` is the local goto action of A.
  In our example state, we have two such announce actions: *on e announce rule 5:* $C \to \bullet\ E$ and *on f announce rule 6:* $C \to \bullet\ F$. Both associated items are not in the core of S. This translates as follows:

```
state8 k1 k2 k3 k4 ts = case ts of
  (Token_e:tr) -> rule5 (action5 g_C) ts
  (Token_f:tr) -> rule6 (action6 g_C) ts
  ...
```

  Note that a rule and its semantic action appear together here.

Accept Our example state has no accept actions as it is not an exit state.
If a state S is an exit state, it has the artificial core item $[_ \to NT\ \cdot]$. It comes always at the first place and is therefore associated with the continuation `k1`.

Accepting NT is realized by simply calling the continuation `k1` – this moves control back to the entry state (which called the exit state via a goto action), which moves control back to its caller.
As an example, consider the extended expression grammar and look at the nonterminal E and its entry and exit states.
The generated code is as follows:

```
1 -- _ -> · E
2 state0 :: ((Expr) -> Parser r) -> (Parser r)
3 state0 k1 ts = case ts of
4   (TokenOB:_) -> rule3 (action3 g4) ts -- on ( announce rule 3
5   (TokenID:_) -> rule3 (action3 g4) ts -- on id announce rule 3
6   where
7     g4 x = state1 (k1 x) (action1 g4 x) (action2 g4 x) -- on E goto
      ↪  state 1
```

State 0 (E entry)

Items:
$[_ \to \cdot\ E]$ (core)
$[E \to \cdot\ E\ *\ T]$
$[E \to \cdot\ E\ *\ *\ T]$
$[E \to \cdot\ T]$

Actions:

on (	announce rule 3: $E \to \bullet\ T$
on *id*	announce rule 3: $E \to \bullet\ T$
on E	goto state 1

State 1 (E exit)

Items:
$[_ \to E\ \cdot]$ (core)
$[E \to E\ \cdot\ *\ T]$ (core)
$[E \to E\ \cdot\ *\ *\ T]$ (core)

Actions:

on $*$	shift to state 2
on)	accept E
on $	accept E

```
8
9   -- _ -> E ·
10  -- E -> E · '*' T
11  -- E -> E · '*' '*' T
12  state1 :: (Parser r) -> ((Token) -> (Term) -> Parser r) -> ((Token)
    ↪ -> (Token) -> (Term) -> Parser r) -> (Parser r)
13  state1 k1 k2 k3 ts = case ts of
14    t@(TokenTimes):tr -> state2 (k2 t) (k3 t) tr -- on * shift to
      ↪ state 2
15    (TokenCB:_) -> k1 ts -- on ) accept E
16    (TokenEof:_) -> k1 ts -- on $ accept E
```

`state1` is called (and **only** called) via `state0`'s goto action `g4`. Thereby, the first argument of `state1` is `k1`, the first (and only) continuation of `state0`. This `k1` was provided by the caller of `state0`. And the caller of `state0` is always a rule function (because state 0 is an entry state).

This means that, when `state1` decides to accept E, `k1` of `state1` is called, which translates to `k1` of `state0`, which moves control back to the caller of `state0` and the partial parse of this E is finished – control is back at the recursive descent part.

Goto This is also borrowed directly from Hinze and Paterson [2]; readers of [2] may skip this paragraph. Consider the RAD state S and a goto action *on NT goto state Q*, where NT is some nonterminal.

Each of Q's core items emerged of a completion item of S by shifting the dot over NT. For example, the completion item $[A \to \cdot\ C]$ of S induces a core item $[A \to C\ \cdot]$ in the goto-state of C.

The goto action simply calls the goto-state $\texttt{state}_\texttt{Q}$ and supplies sensible continuations. Each continuation of $\texttt{state}_\texttt{Q}$ belongs to a core item of Q and therefore

belongs to a completion item of S when shifting the dot one place to the left. We then distinguish, just as above, on whether the item is a core or a completion item of S.

Following the example of E's entry state (state 0), state 0 has one goto action: *on E goto state 1.* Thereby, state 1 has the three core items: $[_ \to E\ \cdot]$, $[E \to E\ \cdot\ *\ T]$ and $[E \to E\ \cdot\ *\ *\ T]$. Each core item of state 1 belongs to a completion item of state 0 by shifting the dot one position to the left:

1. $[_ \to E\ \cdot]$ belongs to $[_ \to \cdot\ E]$, which is a core item of state 0. It is associated to state 0's continuation `k1`.

2. $[E \to E\ \cdot\ *\ T]$ belongs to $[E \to \cdot\ E\ *\ T]$, which is no core item of state 0. Therefore, the continuation `action1 g4` is generated – `action1` is the semantic action of the rule $R : E \to E\ *\ T$, while `g4` is the goto function of E, as E is the left-hand side of rule R.

3. $[E \to E\ \cdot\ *\ *\ T]$ belongs to $[E \to \cdot\ E\ *\ *\ T]$, which is no core item of state 0. The continuation looks as follows: `action2 g4`.

This leads to the local goto-action `g4` of state 0 being declared as follows:

```
g4 x = state1 (k1 x) (action1 g4 x) (action2 g4 x) -- on E goto state 1
```

Default The last thing we consider are default actions. A default action is performed when no other action matched the current input token. A default action can either be an announce or an accept action. Additionally, HAPPY's error token can induce a shift action as a default action (shift on *error*).

Announce or Accept The default action is included in the state's switch expression as a default case. Because announce or accept actions do not consume the input token, there is no need to evaluate it – we can just execute the announce or accept action as described above.

This could look as follows:

```
state0 k1 k2 k3 ts = case ts of
  (t@(TokenPlus):tr) -> state4 (k1 t) tr
  (TokenTimes:_) -> rule4 k2 ts
  _ -> rule5 k3 ts
```

This state has a shift action on $+$ and an announce action on $*$. In addition, it has a default announce action, which is simply executed when neither $+$ nor $*$ is matched.

As described in section 3.5, this can be done for every RAD state which has at least one announce or accept action – the largest one (the one which is executed on the most number of different tokens) can be used as a default action.

Shift An item like $[A \rightarrow B \cdot \mathit{error}\ C]$ induces a default-shift action in its state. This default-shift action must be executed when no other action was matched. In contrast to a normal shift action, the next input token is not consumed.

```
state0 k1 k2 ts = case ts of
  (TokenTimes:_) -> rule4 k1 ts
  _ -> state4 k2 ts
```

Notice that the continuation of the error-item, `k2`, does not receive a semantic value as it is nonsensible to talk about the semantic value of an error terminal.

Instead of giving no value to `k2`, we could also define an explicit type for the error terminal and then just provide a dummy value as the semantic value:

```
data ErrorToken = ErrorToken

state0 k1 k2 ts = case ts of
  (TokenTimes:_) -> rule4 k1 ts
  _ -> state4 (k2 ErrorToken) ts
```

This method has the advantage that it does not alter the number of arguments in the continuation function and in related semantic actions.

4.1.5. Top-Level Entry Points

In HAPPY, the user can define multiple top-level parsing *entry points*. An entry point is a nonterminal N from where the full derivation should begin. For each entry point N, a top-level parse function is generated. There are two types of entry points: *normal* and *partial* entry points.

1. For each normal entry point N, HAPPY generates a new production $N^\star \rightarrow N\ \$$, where \$ denotes the eof-token. Then the grammar with the start symbol $N^\star$ is considered.

2. A partial parse does not need to end on the eof-token, but ends whenever the top-level nonterminal has been parsed and cannot be parsed further.
   Therefore, for each partial entry point P, HAPPY generates a new production $P^\star \rightarrow P\ \mathit{error}$, where *error* denotes HAPPY's error-token, accepting *any* following token, including the eof-token. Then the grammar with the start symbol $P^\star$ is considered.

When a grammar has multiple entry points, the user of the parser must specify which one to use. This is easily possible since there is one top-level parsing function for each entry point.

Consider a normal entry point N. We want to call the associated rule function $(N^\star \rightarrow N\ \$)$ and then extract and return the value of type $\overline{\mathtt{N}}$. Because this is a top-level function, we actually return the parsed value instead of calling a continuation.

Therefore, the entry function has the following type:

$\texttt{entry}_{\texttt{N}}$ :: Parser $\overline{\texttt{N}}$

Here we instantiate a concrete instance of `Parser` the first time! $\texttt{entry}_{\texttt{N}}$ takes a list of tokens and returns an $\overline{\texttt{N}}$.

We implement $\texttt{entry}_{\texttt{N}}$ in a straightforward way: we call the associated rule function and then just extract the value using `const` as continuation. This could look as follows:

```
entry_E :: Parser Expr
entry_E = rule0 const
```

This works because `const` has the type `a -> b -> a`, which is just what the continuation of `rule0` expects: we discard the remaining token list (which is empty anyway) and just return the value of type `Expr`.

Partial entry points work exactly the same, with the exception that the remaining token list does not need to be empty. Still, the remaining tokens are discarded, and the generated entry function looks identical to a normal entry function.

Actually, there is a small implementation detail which is different in our parser generator: the top-level rules do not contain the eof- or the error-token; this is an artifact of HAPPY. Instead of $N^\star \rightarrow N\ \$$, the top-level rule would look like this: $N^\star \rightarrow N$. Then, the burden of checking whether the token list is empty would fall on the entry function $\texttt{entry}_{\texttt{N}}$. It would then look like this:

```
entry_N :: Parser N
entry_N = rule0 (parseEof . const)
```

where `parseEof` is a function of type `Parser r -> Parser r` which just checks whether the token list is empty – it does not pass a semantic value to its continuation. If the token list is not empty, `parseEof` raises an error.

The call to `parseEof` would not be required for partial entry points.

4.1.6. Semantic Actions

Semantic action functions look just like described by Hinze and Paterson [2] and have already been addressed shortly in section 2.6.

A semantic action function takes semantic values of each symbol in a rule's right-hand side and reduces them to a single value of the semantic type of the rule's left-hand side symbol. The rule's recognition point is hereby irrelevant. For the rule $E \rightarrow E\ +\ T$, this could look as follows:

```
action1 :: (Expr -> Parser r) -> Expr -> Token -> Term -> Parser r
action1 k v1 v2 v3 = k (Plus v1 v3)
```

The expression inside the parentheses – `Plus v1 v3` – is what comes directly from the user of HAPPY. The user can supply arbitrary expressions here, as long as they match the type requirements.

An example of a generated parser in just described continuation-based style can be found in the appendix. It contains both an LALR parser, as described by Hinze and Paterson [2], and a RAD parser, using our above-described style.

4.2. GHC-Specifics

There are additional features of HAPPY and of Haskell itself that are used by GHC's Haskell-grammar that we had to take care of. The main features are monadic lexers and higher-rank types.

4.2.1. Monadic Lexers

HAPPY provides the option to wrap a monad around the parsing result, allowing for context-sensitive actions inside semantic actions. Following an example from the HAPPY user guide [12], the user could create a parser monad as follows:

```
-- User-defined types and functions
data P = Ok a | Failed String

thenP :: P a -> (a -> P b) -> P b
thenP = ...

returnP :: a -> P a
returnP a = Ok a

failP :: String -> P a
failP err = Failed err
```

This enables the use of monadic semantic actions in the grammar declaration:

```
prec : int {% if $1 < 0 || $1 > 9
                      then failP "Precedence out of range"
                      else returnP $1}
```

In this semantic action, the generated parser raises an error inside the monad when the token value is out of range.

Combined with a monadic parser, HAPPY allows to use a monadic lexer. Then the input to the parser is no longer a list of tokens – instead, every time a new input token is required, the parser calls the `lexer` function which has the following type:

```
lexer :: (Token -> P a) -> P a
```

This function reads the next token from the input and returns it via a continuation. This originally comes from ALEX [10] – a lexer generator for Haskell which is often used in conjunction with HAPPY.

When HAPPY is used with a monadic parser and lexer, our generated functions must be adapted accordingly to match new type requirements and to implement the continuation-based lexing style.

We first note that we need a special handling for the lookahead token: because announce and accept actions do not consume the lookahead token, it needs to be repeated somehow in order for the next function that calls `lexer` to receive the lookahead token instead of the next token from the input stream.

We therefore redefine the type `Parser r` as follows:

```
type Parser r = Maybe Token -> P r
```

The first argument is the *repeat token* – if we want to repeat the lookahead token which was just read, we assign it to the repeat token; else, the repeat token contains `Nothing`. We need to define two auxiliary functions:

```
lexerWrapper :: (Token -> Parser a) -> Parser a
lexerWrapper cont (Just la) = cont la Nothing
lexerWrapper cont Nothing = lexer (\t -> cont t Nothing)

repeatTok :: Token -> Parser r -> Parser r
repeatTok tok parser _ = parser (Just tok)
```

When the repeat token is non-empty, `lexerWrapper` repeats it and resets the repeat token to `Nothing`. `repeatTok` is used to update the repeat token.

It was our aim that the user of the parser does not notice anything of the wrapper type `Parser r` – the user should only ever have to work with their own type `P r` and should not notice our implementation details around the repeat token.

Therefore, user-defined functions like `happyError`, `lexer` and `thenP` need a wrapper to match the internal type requirements of our parser functions, which work with `Parser r`.

We briefly discuss how our generated functions need to change when using a monadic parser and lexer.

Parse Terminals We parse and consume the next token from the input. Because we need to consider the repeat token if it is non-empty, we need to call `lexerWrapper`:

```
parsePlus :: (Token -> Parser r) -> Parser r
parsePlus k = lexerWrapper $ \t -> case t of
  TokenPlus -> k t
  _ -> happyErrorWrapper t
```

`happyErrorWrapper` is a wrapper around the user-defined `happyError` to accommodate for our type requirements:

```
-- happyError can only receive a single token due to the monadic lexer
happyError :: Token -> P r
happyError _ = failP "Error"

happyErrorWrapper :: Token -> Parser r
happyErrorWrapper = const . happyError
```

States Similar to above, we need to call `lexerWrapper` to receive the correct token. In addition, when performing an accept, announce or a default-error-shift action, we need to repeat the lookahead token with `repeatTok`:

```
state5 :: (Token -> Parser r) -> (Token -> Term -> Parser r) ->
↪  (Token -> Term -> Parser r) -> Parser r
state5 k1 k2 k3 = lexerWrapper $ \t -> case t of
  TokenPlus -> state8 (k2 t)
  TokenMinus -> repeatTok t $ rule4 k3
  _ -> repeatTok t $ k1
```

The type of the continuations or of the state function itself does not change compared to a non-monadic lexer – `Parser r` is still used everywhere.

Semantic Actions In a monadic parser, a semantic action can either be normal or monadic. While nothing changes for normal actions, monadic actions must be considered separately.

Consider the monadic action from before:

```
prec : int {% if $1 < 0 || $1 > 9
                then failP "Precedence out of range"
                else returnP $1}
```

The result of the action is a `P r` while the continuation function of the semantic action expects a `Parser r`. We solve this as follows:

```
action123 :: (Prec -> Parser r) -> Int -> Parser r
action123 k v1 = (if v1 < 0 || v1 > 9
```

```
                                     then failP "Precedence out of range"
                                     else returnP $1)
                                    `thenWrapP` k

-- Wrapper around thenP
thenWrapP :: P a -> (a -> Parser b) -> Parser b
thenWrapP a f la = a `thenP` (flip f la)

-- Remember: thenP was user-defined and had the type:
P a -> (a -> P b) -> P b
```

The result of the user-provided code is a `P r` which we need to convert to a `Parser r`, wherefore we create a wrapper around the user-defined function `thenP`. This wrapper combines the result of the user-provided code with the continuation `k`.

Entry Points Because the entry points are ultimately called by the user of the parser, we want them to have the type `P T` instead of `Parser T` (where `T` is the type of the respective top-level nonterminal).

This looks as follows:

```
entry_E :: P Expr
entry_E = rule0 (parseEof . const . returnP) Nothing
```

`rule0` has a continuation of type `Expr -> Parser r`.

- `returnP` converts the `Expr` into a `P Expr`
- `const :: P Expr -> b -> P Expr` converts the `P Expr` into a `Parser Expr` (i.e. `Maybe Token -> P Expr`) by discarding the repeat token
- `parseEof` checks for the final eof-token
- Finally calling the function of type `Parser Expr` (i.e. `Maybe Token -> P Expr`) with the initial repeat token `Nothing` begins the parse and returns a `P Expr`, as desired.

More The other generated things – rules and parsing nonterminals – do not change. They look identical to their non-monadic counterparts.

4.2.2. Higher-Rank Types

When defining a function such as `const :: a -> b -> a`, the type variables are implicitly universally quantified: `const :: forall a b. a -> b -> a`.

The explicit use of the `forall`-keyword is not legal in standard Haskell, but can be enabled via GHC-specific language extensions (like *RankNTypes*). When using the `forall`-keyword explicitly, one can create *higher-rank* types:

```
f :: (forall a. a -> a) -> (forall b. b -> b) -> Bool
```

The occurrence of `forall`-keywords can be nested arbitrarily deep inside a type signature – the highest level of nesting which can not be resolved determines the rank of the type.

While type inference of a program only using rank-1 and rank-2 types is decidable, this problem gets undecidable once rank-3 or higher-rank types are used [13]. This means that Haskell programmers have to assist the compiler in type inference by explicitly stating the types of functions and statements which are critical for type inference.

This can become a problem in our continuation-based parser:

Consider a nonterminal N which has the following semantic type (which was defined in the ".y"-grammar file):

```
forall b. DisambECP b => [Located b]
```

This is a modified example from GHC's Haskell grammar. Here, `DisambECP` is a class. This is no problem by itself, but now consider the type of the rule function parsing the rule $A \to \bullet\ N$:

```
rule0 :: ((forall b. DisambECP b => [Located b]) -> Parser r) ->
↪  Parser r
```

Because the type of N is now two layers deep inside the left-hand side of a `->`, this type is already rank-3!

Considering a rule like $A \to \bullet\ N\ x\ y\ z$, the type of N is already 5 layers deep in the continuation function, giving the associated rule function a rank of 6.

When using types like these, GHC will struggle to compile the generated parser and fail during type inference. To fix this, we need to help GHC by stating the type of some further functions explicitly.

Look at this generated code for the Haskell grammar:

```
-- aexp1 -> aexp1 • { fbinds }
rule535 :: forall r. (Token -> (forall b. DisECP b => [b]) -> Token
↪  -> Parser r) -> Parser r
rule535 k la = parse426 cont1 la where
  cont1 :: Token -> Parser r
  cont1 v1 la = parse259 (cont2 v1) la
  cont2 :: Token -> (forall b. DisECP b => [b]) -> Parser r
  cont2 v1 v2 la = parse427 (cont3 v1 v2) la
  cont3 :: Token -> (forall b. DisECP b => [b]) -> Token -> Parser r
  cont3 v1 v2 v3 la = k v1 v2 v3 la
```

This rule parses an *fbinds*, which has a universally-quantified type.

The local continuations `cont1` through `cont3` receive prefixes of the full parse (*{ fbinds }*): `cont1` receives *{*, `cont2` receives *{ fbinds* and `cont3` receives the full *{ fbinds }*.

Therefore, it is not enough to just state the type of `rule535`, but we also need to state the types of all local continuation functions where *fbinds* appears – in this case `cont2` and `cont3` – because these cannot be inferred by GHC automatically.

Because we now explicitly use universal quantification in the type of `rule535` and its local continuations, we can no longer use universal quantification *implicitly* – which explains the leading `forall r` in the type of `rule535`. This is required so that the `r` of `Parser r` which is used in `cont1`, `cont2` and `cont3` is considered the same.

The same thing applies not only to rule functions but also to state functions and their local goto actions:

```
-- core: [_ -> · fbinds]
state202 :: forall r. ((forall b. DisambECP b => [b]) -> Parser r)
↪ -> (Parser r)
state202 k = lexerWrapper $ \t -> case t of
  (L _ ITas) -> repeatTok t $ rule652 (action652 g259)
  ...
  _ -> repeatTok t $ rule653 (action653 g259)
  where
    g259 :: (forall b. DisambECP b => [b]) -> (Parser r)
    g259 x = state643 (k x)
```

Here, both `state202` and `g259` have a higher-rank type which must be annotated explicitly in order for the generated parser to compile under GHC.

Conclusively, it is required to explicitly annotate the types of some top-level-functions and related local functions when the user specifies universally quantified or higher-rank types for some nonterminals. This is, for example, the case in GHC's Haskell-grammar.

Our generator backend does this of course automatically, and precise details can be taken from the implementation.

5. Evaluation

5.1. Experimental Results

In this section, we compare the performance of our parsers with classical HAPPY-generated ones.

We conduct performance evaluations on multiple different grammars: our example expression grammar, a JSON grammar, a C and a Rust grammar. Also, we evaluate the parsing performance of GHC itself, comparing the standard HAPPY-generated GHC parser to a RAD parser.

In the following, we will compare six different types of generated parsers:

1. `happy`: the classical LALR parser generated by HAPPY.
2. `happy -acg`: a table-based version of `happy`, using GHC-specific optimizations and string-encoded arrays for better size and performance.
3. `happy -acg --strict`: in addition to `-acg`, semantic actions are evaluated strictly, at the moment they are invoked. Removing Haskell's laziness brings another performance gain.
4. `lalr`: a continuation-based LALR parser, generated by our backend. Uses HAPPY's LALR states and generates a parser in the continuation-based style of Hinze and Paterson [2].
5. `rad`: a continuation-based RAD parser, as described in this thesis, generated by our backend.
6. `rad --optims`: this version brings two performance optimizations to the generated RAD parser: all rule functions are marked `INLINE`, and all applications of `goto` and `k` functions are eta-expanded. For example, a goto action could look like this:
`g228 x la = state751 (\la -> k1 x la) (action581 (\z la -> g229 z la) x) la`.

Disk space nowadays is cheap and not nearly as big a factor as in the past. Still, while good performance is the number one quality we strive for, we want our generated parsers to be reasonably small in comparison to HAPPY-generated parsers.

Here, "reasonably small" means "not excessively large" – as our parsers are not table-based, but have a function for each rule, state, and symbol, we cannot expect

them to be as small as HAPPY-generated ones. Instead, we want their size to be around the same order of magnitude.

We compare the size of a parser by looking at the ".o"-file generated by GHC; it contains the compiled object code of the parser module.

5.1.1. Parser Only

We begin by looking at two rather small grammars: the extended expression grammar which we used throughout this thesis, and a (slightly modified) JSON grammar.[1]

While our expression grammar has 8 rules and 9 symbols, the JSON grammar already has 42 rules and 39 symbols.

Both of these grammars were parsed using HAPPY's normal parsing style: The parsers receive a full list of tokens as input, which has previously been created by a lexer. We compare two different performance metrics: First, the time of the parser alone, receiving a fully-evaluated token list as input. Second, the combined time of lexer and parser. The difference here is that the token list is not fully evaluated when being passed to the parser, but is lazily evaluated by the lexer. This introduces additional caching effects which may have a negative impact on performance; it is also the more realistic metric to measure (as a lexer will always be involved in real-world use).

Tables 5.1 and 5.2 show the results. We used large randomly generated files (several hundred kBs) for the performance evaluation.

Table 5.1.: Expression grammar. Filesize: 686kB, 524k tokens.

Type	T(parse)	T(lex & parse)	.o-size
`happy`	105.8 ± 1.4 ms	306.1 ± 3.8 ms	120 kB
`-acg`	120.5 ± 4.0 ms	329.9 ± 4.4 ms	74 kB
`--strict`	**90.0 ± 0.1 ms**	**249.8 ± 0.3 ms**	74 kB
`lalr`	29.4 ± 1.8 ms	221.0 ± 4.0 ms	41 kB
`rad`	35.7 ± 2.1 ms	226.9 ± 4.4 ms	33 kB
`--optims`	**28.0 ± 1.9 ms**	**219.6 ± 3.7 ms**	33 kB

The execution time values are stated in the form $\bar{t} \pm \sigma$, where $\bar{t}$ is the mean of the measurements and σ is the standard deviation. We used the tool `gauge`[2] to perform the measurements. The parsers were compiled using GHC 8.8.1 and with the `-O2` flag.

In the expression grammar, the lexer performs significant work by creating *id* tokens from longer number strings such as "34" and "−12". The JSON lexer, on the

[1]The official JSON grammar, from `www.json.org`
[2]`https://hackage.haskell.org/package/gauge`

Table 5.2.: JSON grammar. Filesize: 341kB, 341k tokens.

Type	T(parse)	T(lex & parse)	.o-size
`happy`	**61.1 ± 1.8 ms**	**64.3 ± 1.4 ms**	447 kB
`-acg`	86.9 ± 3.4 ms	97.8 ± 2.3 ms	306 kB
`--strict`	66.2 ± 1.7 ms	69.4 ± 1.3 ms	297 kB
`lalr`	16.7 ± 0.9 ms	19.0 ± 0.5 ms	248 kB
`rad`	16.1 ± 0.9 ms	18.4 ± 0.5 ms	251 kB
`--optims`	**15.3 ± 0.8 ms**	**18.1 ± 0.5 ms**	261 kB

other hand, does not perform any nontrivial work and converts each character to a respective token.

Both the best respective parser of classical `happy` and our best parser are marked in bold. It can immediately be seen that the optimized RAD parser beats `happy`'s best parser significantly in both grammars. Just comparing the plain parsing times, the optimized RAD-parser is 3 to 4 times faster. In addition, the continuation-based parsers are actually smaller than their `happy`-counterparts!

5.1.2. Parser-Lexer Combination

We shift our focus onto programming languages and their grammars. We look at a C grammar[3] and a Rust grammar[4].

Both of these grammars use a monadic parser-lexer combination. This means that tokens are lexed on-demand (using `alex`). We can now no longer talk about the plain parsing time as we cannot separate the interconnected parser from the lexer.

The grammars are, in addition, much larger than those from before: the C grammar has 502 rules and 257 symbols, and the Rust grammar features a decent number of 1209 rules and 358 symbols.

We used a single large file per parser for evaluation. Tables 5.3 and 5.4 show the results. Here we see different results than before: While the RAD parser still beats the `happy`-parser in the case of Rust, this doesn't hold anymore for the C grammar. Here, the best `happy`-parser performs as well as the optimized RAD parser.

The C grammar brings a different, very interesting insight: Comparing the compiled size of our continuation-based parsers, the RAD parsers are **much** smaller than the LALR parser – up to a factor of 4 (27 MB vs 6.6 MB).

This seems a little counterintuitive at first – an LALR parser for the C grammar just has 963 states, while a RAD parser has 542 states, 502 rule functions and 257

[3]from `https://hackage.haskell.org/package/language-c`
[4]from `https://hackage.haskell.org/package/language-rust`

Table 5.3.: C grammar. Filesize: 375kB.

Type	T(lex & parse)	.o-size
`happy`	136.1 ± 1.7 ms	5.3 MB
`-acg`	139.3 ± 1.7 ms	1.3 MB
`--strict`	**108.0 ± 2.2 ms**	1.3 MB
`lalr`	118.3 ± 1.7 ms	27 MB
`rad`	115.4 ± 1.9 ms	7.8 MB
`--optims`	**108.3 ± 1.9 ms**	6.6 MB

Table 5.4.: Rust grammar. Filesize: 156kB.

Type	T(lex & parse)	.o-size
`happy`	78.9 ± 0.4 ms	19 MB
`-acg`	73.0 ± 0.6 ms	5.3 MB
`--strict`	**71.5 ± 0.8 ms**	5.3 MB
`lalr`	67.4 ± 0.6 ms	35 MB
`rad`	68.3 ± 0.6 ms	32 MB
`--optims`	**66.9 ± 0.5 ms**	32 MB

symbol functions. Why is it that the RAD parser still produces significantly smaller compiled code? We can think of two reasons:

1. Many of the rule and symbol functions are simple (as they just link state and symbol functions together) and get optimized away
2. RAD states are generally smaller than LALR states:
   - A RAD state often has equally many or less actions than the LALR state it emerged from, because some of its actions may have been discarded.
   - Often, a RAD state's default action includes more actions/tokens than the default action of its associated LALR state (because many different shift and reduce actions of an LALR state may collapse into the single same announce action, which is then used as default a action).

We also notice another connection playing a role here: the *LL-ness* of a grammar (which will be introduced in section 5.2) states, intuitively, how close a grammar is to being LL. This then also correlates with the ratio of unambiguous nonterminals and how good states can be reused, as seen later.

This seems to play a great role in the size saving phenomenon:

1. the LL-ness of the Rust grammar is very low. Here, the size saving effect nearly vanishes (from 35MB to 32MB).
2. the LL-ness of the C grammar is much greater. The effect is strong (factor 4).

3. the LL-ness of the Haskell grammar (section 5.1.3) is even larger. The effect is even greater (factor 4.5).

This is a compelling advantage, besides speed, of a continuation-based RAD parser: it can produce really small compiled code compared to a continuation-based LALR parser.

5.1.3. GHC: Parsing Haskell

GHC uses a monadic parser-lexer combination to parse Haskell code. As one might already expect, GHC uses `happy -acg --strict` to generate the parser for their Haskell grammar (which is quite large having 834 rules and 479 symbols).

As GHC is written in Haskell itself, compilation of GHC proceeds in two phases: in the first phase, an existing GHC on the system is used to compile the GHC sources. This then produces the *stage-1* GHC. Then, this stage-1 GHC is used to again compile the sources to produce the *stage-2* GHC.

We performed our tests with both stage-1 and stage-2 GHCs. The results cannot be compared between stage-1 and stage-2 runs, however, as different underlying GHC versions were used to compile them: stage-1 was compiled with GHC 8.8.1, while stage-2 was compiled with stage-1 which is a GHC 8.10.1 clone.

GHC was compiled with `BuildFlavour = perf`, which is meant to produce a performant GHC ready for distribution. Before looking at performance results, we first compare the sizes of the produced *Parser.o* files.

Table 5.5.: Sizes of compiled *Parser.o* files. `--strict` is the status quo GHC parser.

Type	size (stage-1)	size (stage-2)
`happy`	12 MB	12 MB
`-acg`	2.4 MB	2.4 MB
`--strict`	2.4 MB	2.4 MB
`lalr`	15 MB	33 MB
`rad`	6.3 MB	7.1 MB
`--optims`	5.9 MB	7.6 MB

As already stated earlier, the RAD parser is significantly smaller than the continuation-based LALR parser, especially in stage-2. Also, while the RAD parser is not as small as the status quo `happy` parser, it is still in the same order of magnitude – it is only 3 times as large.

The Haskell files that we used to measure the parsing performance are large files either from GHC itself or generated by us.

We now show two performance comparisons. The first one is a stage-1 comparison of parsing a 1.1 MB Haskell file (*Parser-RAD.hs*), the second one a stage-2 comparison of parsing a 6.4 MB file (*ManyTokens.hs*). Tables 5.6 and 5.7 show the results.

Table 5.6.: *Parser-RAD.hs* (1.1 MB), parsed using stage-1.

Type	T(lex & parse)	Allocations
`happy`	814.4 ± 2.5 ms	1.31 G
`-acg`	826.4 ± 2.1 ms	1.36 G
`--strict`	**783.7 ± 2.2 ms**	1.31 G
`lalr`	693.0 ± 3.3 ms	1.24 G
`rad`	695.5 ± 2.3 ms	1.25 G
`--optims`	**680.2 ± 2.5 ms**	1.19 G

Table 5.7.: *ManyTokens.hs* (6.4 MB), parsed using stage-2.

Type	T(lex & parse)	Allocations
`happy`	4941 ± 26.0 ms	6.60 G
`-acg`	4447 ± 22.9 ms	5.73 G
`--strict`	**3459 ± 10.2 ms**	5.39 G
`lalr`	3247 ± 14.1 ms	6.46 G
`rad`	3311 ± 7.0 ms	6.55 G
`--optims`	**3162 ± 6.4 ms**	5.97 G

We used `-ddump-timings` to extract the parsing time and the number of allocated bytes during parsing from GHC. Naturally, the number of allocated bytes is also an attribute one should strive to minimize, as a greater number of allocations corresponds to greater execution time. Note that the times and allocations **only** concern the plain parsing/lexing time, no subsequent steps in the compilation process.

In both cases, the optimized RAD parser beats the status quo `happy`-parser.
Table 5.8 shows the comparison results for 4 files, considering stage-1 and stage-2 separately each time. We only included the `--strict` and `--optims` parsers.

Table 5.8.: Comparison between the status-quo `--strict` parser and our optimized RAD parser.

		`--strict`		`--optims`		
Test	Stage	Time [ms]	Allocs	Time [ms]	Allocs	Gain
ManyTokens	1	3780 ± 10.3	6.63 G	3051 ± 11.7	6.01 G	19.4%
	2	3459 ± 10.2	5.39 G	3162 ± 6.4	5.97 G	8.6%
TcTyClsDecls	1	60.1 ± 0.32	95.5 M	55.5 ± 0.25	90.3 M	7.7%
	2	47.0 ± 0.20	87.0 M	45.8 ± 0.27	90.6 M	2.5%
Parser	1	432.2 ± 1.9	950 M	387.2 ± 1.2	913 M	11.4%
	2	404.1 ± 2.4	883 M	382.0 ± 1.0	914 M	5.5%
Parser-RAD	1	783.7 ± 2.2	1.31 G	680.2 ± 2.5	1.19 G	13.3%
	2	748.0 ± 2.2	1.11 G	712.8 ± 2.7	1.19 G	4.9%

The last column, *Gain*, shows the relative performance gain which would come from replacing the status quo parser with an optimized RAD parser.

Looking at the table, we see: with a directly-executable RAD parser just 3 times as large as the table-based LALR one, we get performance speedups from 5% to 10% with stage-2 parsers, and up to 20% with stage-1 parsers. As these stages are dependent on the exact underlying GHC version, these values can of course vary from version to version.

Also note that the times describe combined parsing and lexing. Would we somehow just extract the pure time that is spent in the parser, the relative performance gain would be even higher. On the other hand, the parsing/lexing process is only one of many steps of a full compilation process, where other steps such as type checking, optimizations and code generation often dominate the total compilation time.

Readers of Hinze and Paterson [2] may have noticed that our performance speedups on the Haskell grammar are not as high as those achieved in [2], using their `frown -cs`[5] LALR(1) parser generator which generates continuation-based code.

We can say two things about this: First, our `lalr` parsers are very similar to the parsers generated by `frown -cs` – we deliberately created this `lalr` option to generate code in Hinze-Paterson style. Second, Hinze and Paterson [2] didn't conduct tests using GHC – they used a much smaller Haskell 98 grammar, only having 277 rules and 482 states, and performed parsing tests using this grammar, independently from GHC.

Speedups that have been achieved by `frown -cs` can of course be reproduced using our `lalr` parsers, and in many cases even be surpassed with our `rad --optims` parsers.

5.2. LL-Ness

This section is dedicated to a key aspect of RAD parsing. As described throughout this thesis, a RAD parser can parse any LR grammar. Thereby it tries to make use of LL-like features as much as possible: rules are parsed in a top-down manner starting at their recognition points. During a top-down parse, states are "reused": parsing a nonterminal leads to entering its entry state, which is reused each time this nonterminal is encountered after a recognition point. This distinguishes RAD parsing from LALR parsing where a new state is required for every parsing situation where the nonterminal appears – states are not reused.

Also, it is clear that the top-down ratio of a RAD parser is higher the further left the recognition points are, as this allows for earlier transition into the top-down part of a rule. A grammar with all recognition points on the very left is already LL.

This leads us to two definitions which try to measure how good a RAD parser can make use of existing top-down aspects, or, in other words, how much LL a grammar is.

[5] https://hackage.haskell.org/package/frown

1. **LL-ness**. We define the *LL-ness* of a grammar as $1 - \frac{\#\text{Spaces}_{\text{left}}}{\#\text{Spaces}}$, where $\#\text{Spaces}_{\text{left}}$ is the number of spaces left to recognition points in all rules and #Spaces the number of total spaces in all rules, *excluding* the rightmost space. For example, the rule $A \rightarrow b \bullet C\ D$ has 3 total spaces (before the b, before the C, before the D – we do not count the rightmost space), one of which is left to the recognition point.

2. **State reuse**. We define the *state reuse* of a grammar as $\frac{\#\text{States}_{\text{entry/exit}}}{\#\text{States}}$, where $\#\text{States}_{\text{entry/exit}}$ denotes the number of entry- and exit states (or 2 times the number of unambiguous nonterminals) and #States denotes the total number of generated RAD states.

Why do we exclude the rightmost space when calculating the LL-ness? Consider the rule $A \rightarrow B\ C$. The recognition point could be in three different locations: $A \rightarrow \bullet B\ C$, or $A \rightarrow B \bullet C$, or $A \rightarrow B\ C \bullet$. If we would define the LL-ness for this single rule, we would want it to be 0 for the first variant, $\frac{1}{2}$ for the second, and 1 for the third variant. Therefore we need to divide by 2, as there cannot be more than 2 spaces left to the recognition point – this is why we do not count the rightmost space, as this wouldn't allow for an LL-ness of 1.

As an example, consider the expression grammar:

$0: E^\star \rightarrow E\ \$$	$E^\star \rightarrow \bullet E\ \$$
$1: E \rightarrow E * T$	$E \rightarrow E * \bullet T$
$2: E \rightarrow E * * T$	$E \rightarrow E * \bullet * T$
$3: E \rightarrow T$	$E \rightarrow \bullet T$
$4: T \rightarrow T + F$	$T \rightarrow T \bullet + F$
$5: T \rightarrow F$	$T \rightarrow \bullet F$
$6: F \rightarrow (\ E\ )$	$F \rightarrow \bullet (\ E\ )$
$7: F \rightarrow id$	$F \rightarrow \bullet id$

There is a total of 17 spaces inside rules and there are 5 spaces left to recognition points. Therefore, the LL-ness is $\frac{12}{17} \approx 71\%$. Considering the state reuse, there are six entry- / exit-states and one auxiliary state (see appendix), which leads to a state reuse of $\frac{6}{7} \approx 86\%$.

Of course, we want LL grammars to have 100% LL-ness and state reuse. This works:

1. In an LL grammar, all recognition points are on the left. Then, $\#\text{Spaces}_{\text{Left}}$ is 0 and the LL-ness is 100%.

2. In an LL grammar, there are no auxiliary states (as there are no shift actions but only announce actions) – the state reuse is 100%.

The following table shows the LL-ness and state reuse for the grammars that were used for the performance evaluation in section 5.1.

Table 5.9.: Properties of multiple grammars, including LL-ness and state reuse.

Grammar	#Rules	#NTs (unamb.)	#LALR-States	#RAD-S. (#E/E)	State reuse	LL-ness
Rust	1209	219 (98)	2404	1773 (196)	11.1%	24.2%
C	502	135 (80)	963	542 (160)	29.5%	48.0%
Haskell (GHC)	834	327 (247)	1435	907 (494)	54.5%	62.3%
JSON	42	20 (15)	67	48 (30)	62.5%	62.5%
Expression	8	4 (3)	16	7 (6)	85.7%	70.6%
Any LL	–	–	–	–	100%	100%

The third column shows the number of nonterminals and how much of them are unambiguous. Looking at the Rust grammar, there are only 98 unambiguous nonterminals, which results in 196 entry- and exit-states (which can be seen in the fifth column) – at a total of 1773 RAD states. A RAD parser therefore does not bring much benefit over an LALR parser in the case of the Rust grammar. This can also be seen when comparing the object file sizes (as seen in section 5.1.2): The compiled Rust RAD parser is nearly as large as a corresponding LALR parser (32 MB vs 35 MB) – RAD parsing simply cannot realize much of its potential on the Rust grammar.

The other grammars have a higher LL-ness and state reuse and perform accordingly better under these aspects. In the Haskell grammar, for example, more than half of all states are "reusable" entry- and exit-states. Consequently, a compiled RAD parser for Haskell is much smaller than an LALR parser.

There are further conceivable properties which could be used as similar LL-ness indicators for a grammar: for example, the ratio between shift and announce actions, or the ratio of states that are saved by transitioning from LALR to RAD.

Note that LL-ness and state reuse are grammar-specific definitions, as they can vary between multiple grammars producing the same language. We concludingly remark that the notions of LL-ness and state reuse can naturally be extended from grammars to languages by defining the LL-ness (state reuse) of a language to be the greatest LL-ness (state reuse) of any LR grammar generating this language, but this would exceed the scope of this thesis.

6. Conclusion

In this thesis we combined the two rather unknown concepts of well-typed continuation-based parsing and recursive ascent-descent parsing. We described the inner workings of recursive ascent-descent parsers and how they can reuse a single state in completely different parsing situations, which is what distinguishes them from LALR parsers. We introduced the notions of *LL-ness* and *state reuse* which try to capture how strong a RAD parser can make use of top-down parsing and said state reuse.

We implemented a parser generator backend producing continuation-based parsing code, both in LALR or RAD form. We compared the performance of classical table-based LALR parsers, as generated by `happy`, with our continuation-based LALR and RAD parsers and came to the result that continuation-based RAD parsers actually outperform all other types of parsers that we considered. Both in small grammars with few rules and in large grammars which describe programming languages, RAD parsing brings good advantages over LALR parsing regarding the number of states, while the continuation-based, directly-executable (i.e. function-based) form improves the speed compared to a table-based approach. We employed some optimizations, which were ultimately suggestions of and come from analyses by our advisor Sebastian Graf, to improve the parser performance even more. Without these optimizations, the performance advantage over GHC's status-quo parser would have vanished in many cases.

As a result emerged a parser for Haskell which is faster than the current Haskell parser. Considering stage-2 compilers, our parser brings a performance gain of 5% to 10% for parsing Haskell code. This comes at the cost of extra 5 megabytes of object code, which is something we think GHC users and programmers are willing to swallow.

6.1. Future Work

As mentioned, even a tiny bit of optimization (rule function inlining and massive eta-expanding) brought a significant performance gain to our RAD parsers. But why stop there?

While we didn't have the time to profoundly look into further optimization ideas, we believe that there is a lot of unexploited optimization potential. We could think of the following ideas, each of which may or may not be helpful for further performance tuning:

- By looking at the tables in section 5.1, one sees that parsers generated via `happy -acg --strict` always outperform corresponding parsers generated via

`happy -acg` significantly. The `--strict` flag causes the parser to evaluate intermediate results such as partial ASTs which emerged through semantic actions immediately instead of lazily at the end. Lazy evaluation doesn't make sense here – the full parsing result is always required for further processing, which is why a strict parser is very sensible. In addition to the execution time, allocations went also down considerably when using a strict parser inside GHC. We didn't make use of any strictness features in our generated code, but ultimately we could expect similar results when using them. Evaluating semantic actions early could bring performance gains comparable to those introduced by `happy`'s `--strict` flag.

- Another interesting phenomenon, which we couldn't explain, is the occasional performance regression between stage-1 and stage-2 parsers generated by our backend. For example, for the file *Parser-RAD.hs*, as visible in table 5.8, the stage-1 RAD parser performs better than the stage-2 RAD parser. This is peculiar as this does never happen to `happy`-generated parsers, but only to our RAD parsers. In the future, this phenomenon could maybe be explained and then resolved.

- When going from stage-1 to stage-2, our RAD parsers hardly change in their number of allocations, while the `happy -acg --strict`-parser does reduce its allocations significantly between stage-1 and stage-2. We could also try to leverage this behavior to generate stage-2 parsers which are even faster than stage-1 ones.

- Our parser generator could take the LL-ness and state reuse of the input grammar into account and perform specific optimizations based on these values. For example, inlining of specific functions could be decided based on whether the parser behaves a lot like an LL parser or not.

- Our way to repeat tokens seems inefficient: a call to `repeatTok` is always immediately followed by a call to `lexerWrapper` to retrieve the token which was just repeated. As the compiler does not know this (due to the myriad of state functions and their announce or accept actions, each of which is accompanied by a `repeatTok` call), it probably doesn't optimize this phenomenon as well as it could. *Short cut fusion*[1] is an optimization technique which tries to merge two matching function calls into one. Thereby, intermediate results and data structures can be removed in some cases. `repeatTok` and `lexerWrapper` match these requirements – `repeatTok` produces a value which is immediately consumed by `lexerWrapper`. If this behavior can be made evident to the compiler, it could probably generate code which performs much fewer allocations and is consequently faster.

[1]As described in `https://wiki.haskell.org/Short_cut_fusion`

In addition to performance tuning, there are further things that can be incorporated into our parser generator. While Horspool [1] described how an LAXLC parser can be converted into directly-executable form, calling it in this case a RAD parser, he also discussed different ways of code realization. When dividing the parser into the *rule* (top-down) and *control* (bottom-up) components, he showed that it is possible to realize the control component either as an explicit LALR automaton using table-based form, or in recursive ascent style using mutually recursive functions. A user which is more concerned about code size could then choose to generate the control part in table-based form, while the rule part is still in recursive descent form.

Even HAPPY itself has multiple ways of generating its LALR parsers: normal HAPPY uses explicit shift and action functions and stores the full automaton state on a stack. Calling HAPPY with `-acg` causes it to use actual array-encoded action tables.

We could integrate similar functionalities into our new HAPPY backend: the user could decide on the form of the generated control component, choosing either table-based or recursive-ascent style. Further, instead of using the function call stack as an implicit parser state storage, a recursive-ascent implementation could also use an explicit stack as HAPPY does. Whether and how this would however interfere with the strongly-typed continuation-based code style would have to be seen.

In addition to pure implementation aspects, future work could also be concerned with more theoretical aspects of recursive ascent-descent parsing. For example, explicit relations between, or explicit bounds for indicators such as the LL-ness or state reuse could be formulated. Further characteristics of LR grammars could be examined and put in relation to LL-ness and state reuse.

Bibliography

[1] R. N. Horspool, "Recursive ascent-descent parsing," *Computer Languages*, vol. 18, pp. 1–15, july 1991.

[2] R. Hinze and R. Paterson, "Derivation of a typed functional LR parser," 2005.

[3] R. Wilhelm, H. Seidl, and S. Hack, *Compiler design. Syntactic and semantic analysis.* Springer, 2013.

[4] D. J. Rosenkrantz and P. M. Lewis, "Deterministic left corner parsing," in *11th Annual Symposium on Switching and Automata Theory (swat 1970)*, pp. 139–152, IEEE, 1970.

[5] A. J. Demers, "Generalized left corner parsing," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, United States), pp. 170–182, Association for Computing Machinery, jan 1977.

[6] J. Boyland and D. Spiewak, "Tool paper: ScalaBison recursive ascent-descent parser generator," *Electronic Notes in Theoretical Computer Science*, vol. 253, pp. 65–74, sept. 2010.

[7] "scala-bison." `https://github.com/djspiewak/scala-bison`, 2020. ScalaBison's GitHub page. Accessed 25. Sep. 2020.

[8] A. Gill and S. Marlow, "happy: Happy is a parser generator for haskell." `https://hackage.haskell.org/package/happy`, 2020.

[9] haskell.org, "The glasgow haskell compiler." `https://www.haskell.org/ghc/`, 2020.

[10] C. Dornan and S. Marlow, "alex: Alex is a tool for generating lexical analysers in haskell." `https://hackage.haskell.org/package/happy`, 2020.

[11] P. Purdom and C. A. Brown, "Semantic routines and LR(k) parsers," *Acta Informatica*, vol. 14, pp. 299–315, oct. 1980.

[12] A. Gill and S. Marlow, "Happy user guide." `https://www.haskell.org/happy/doc/html/index.html`, 2009.

[13] A. J. Kfoury and J. B. Wells, "A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus," in *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, (New York, NY, USA), pp. 196–207, Association for Computing Machinery, 1994.

Erklärung

Hiermit erkläre ich, David Knothe, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift

A. Generated Code Examples

The following two sections contain generated parsing code in the above described well-typed, continuation-based style. One section showcases a classical LALR(1) parser, having 16 states, while the other section presents a RAD parser, having seven states.

The grammar is a simple expression grammar extended by an exponentiation operation, denoted by “$**$”. These are the productions with their respective recognition points:

$$\begin{array}{rlll@{\qquad\qquad}lll}
0: & E^{\star} & \rightarrow & E\ \$ & E^{\star} & \rightarrow & \bullet\ E\ \$ \\
1: & E & \rightarrow & E\ *\ T & E & \rightarrow & E\ *\ \bullet\ T \\
2: & E & \rightarrow & E\ *\ *\ T & E & \rightarrow & E\ *\ \bullet\ *\ T \\
3: & E & \rightarrow & T & E & \rightarrow & \bullet\ T \\
4: & T & \rightarrow & T\ +\ F & T & \rightarrow & T\ \bullet\ +\ F \\
5: & T & \rightarrow & F & T & \rightarrow & \bullet\ F \\
6: & F & \rightarrow & (\ E\ ) & F & \rightarrow & \bullet\ (\ E\ ) \\
7: & F & \rightarrow & id & F & \rightarrow & \bullet\ id
\end{array}$$

Both parsers were generated with our extended version of HAPPY, using the new continuation-based LALR and RAD backends.

A.1. LALR

```haskell
type Parser r = [Token] -> r

-- User-supplied definitions
data Token = TokenInt Int | TokenTimes | TokenPlus | TokenOB | TokenCB

data Expr = Times Expr Term | Pow Expr Term | Term Term
data Term = Plus Term Factor | Factor Factor
data Factor = Num Int | Expr Expr

happyError :: [Token] -> a
happyError _ = error ("Parse error!")


-- Main parsing entry point.
entry :: Parser Expr
entry = state0 (parseEof . const) where
  parseEof k [] = k []
  parseEof k ts = happyError ts


-- core: [E' -> · E]
state0 :: (Expr -> Parser r) -> Parser r
state0 k ts = case ts of
  t@(TokenInt v):tr -> state5 (action7 g6 v) tr
  t@(TokenOB):tr -> state6 (action6 g6 t) tr
  _ -> happyError ts
```

```
27    where
28      g4 x = state7 (k x) (action1 g4 x) (action2 g4 x)
29      g5 x = state3 (action3 g4 x) (action4 g5 x)
30      g6 x = state4 (action5 g5 x)
31
32  -- core: [E -> · E * T]
33  state1 :: (Expr -> Token -> Term -> Parser r) -> Parser r
34  state1 k ts = case ts of
35    t@(TokenInt v):tr -> state5 (action7 g6 v) tr
36    t@(TokenOB):tr -> state6 (action6 g6 t) tr
37    _ -> happyError ts
38    where
39      g4 x = state2 (k x) (action2 g4 x)
40      g5 x = state3 (action3 g4 x) (action4 g5 x)
41      g6 x = state4 (action5 g5 x)
42
43  -- core: [E -> E · * T], [E -> E · * * T]
44  state2 :: (Token -> Term -> Parser r) -> (Token -> Token -> Term -> Parser r)
    ↪  -> Parser r
45  state2 k1 k2 ts = case ts of
46    t@(TokenTimes):tr -> state8 (k1 t) (k2 t) tr
47    _ -> happyError ts
48
49  -- core: [E -> T ·], [T -> T · + F]
50  state3 :: Parser r -> (Token -> Factor -> Parser r) -> Parser r
51  state3 k1 k2 ts = case ts of
52    t@(TokenPlus):tr -> state10 (k2 t) tr
53    _ -> k1 ts
54
55  -- core: [T -> F ·]
56  state4 = id
57
58  -- core: [F -> NUM ·]
59  state5 = id
60
61  -- core: [F -> ( · E )]
62  state6 :: (Expr -> Token -> Parser r) -> Parser r
63  state6 k ts = case ts of
64    t@(TokenInt v):tr -> state5 (action7 g6 v) tr
65    t@(TokenOB):tr -> state6 (action6 g6 t) tr
66    _ -> happyError ts
67    where
68      g4 x = state9 (action1 g4 x) (action2 g4 x) (k x)
69      g5 x = state3 (action3 g4 x) (action4 g5 x)
70      g6 x = state4 (action5 g5 x)
71
72  -- core: [E' -> E ·], [E -> E · * T], [E -> E · * * T]
73  state7 :: Parser r -> (Token -> Term -> Parser r) -> (Token -> Token -> Term
    ↪  -> Parser r) -> Parser r
74  state7 k1 k2 k3 ts = case ts of
75    t@(TokenTimes):tr -> state8 (k2 t) (k3 t) tr
76    [] -> k1 ts -- %eof
77    _ -> happyError ts
78
79  -- core: [E -> E * · T], [E -> E * · * T]
80  state8 :: (Term -> Parser r) -> (Token -> Term -> Parser r) -> Parser r
81  state8 k1 k2 ts = case ts of
82    t@(TokenInt v):tr -> state5 (action7 g6 v) tr
83    t@(TokenTimes):tr -> state14 (k2 t) tr
84    t@(TokenOB):tr -> state6 (action6 g6 t) tr
85    _ -> happyError ts
```

```
86      where
87        g5 x = state13 (k1 x) (action4 g5 x)
88        g6 x = state4 (action5 g5 x)
89
90    -- core: [E -> E · * T], [E -> E · * * T], [F -> ( E · )]
91    state9 :: (Token -> Term -> Parser r) -> (Token -> Token -> Term -> Parser r)
      ↪ -> (Token -> Parser r) -> Parser r
92    state9 k1 k2 k3 ts = case ts of
93      t@(TokenTimes):tr -> state8 (k1 t) (k2 t) tr
94      t@(TokenCB):tr -> state12 (k3 t) tr
95      _ -> happyError ts
96
97    -- core: [T -> T + · F]
98    state10 :: (Factor -> Parser r) -> Parser r
99    state10 k ts = case ts of
100     t@(TokenInt v):tr -> state5 (action7 g6 v) tr
101     t@(TokenOB):tr -> state6 (action6 g6 t) tr
102     _ -> happyError ts
103     where
104       g6 x = state11 (k x)
105
106   -- core: [T -> T + F ·]
107   state11 = id
108
109   -- core: [F -> ( E ) ·]
110   state12 = id
111
112   -- core: [E -> E * T ·], [T -> T · + F]
113   state13 :: Parser r -> (Token -> Factor -> Parser r) -> Parser r
114   state13 k1 k2 ts = case ts of
115     t@(TokenPlus):tr -> state10 (k2 t) tr
116     _ -> k1 ts
117
118   -- core: [E -> E * * · T]
119   state14 :: (Term -> Parser r) -> Parser r
120   state14 k ts = case ts of
121     t@(TokenInt v):tr -> state5 (action7 g6 v) tr
122     t@(TokenOB):tr -> state6 (action6 g6 t) tr
123     _ -> happyError ts
124     where
125       g5 x = state15 (k x) (action4 g5 x)
126       g6 x = state4 (action5 g5 x)
127
128   -- core: [E -> E * * T ·], [T -> T · + F]
129   state15 :: Parser r -> (Token -> Factor -> Parser r) -> Parser r
130   state15 k1 k2 ts = case ts of
131     t@(TokenPlus):tr -> state10 (k2 t) tr
132     _ -> k1 ts
133
134
135   -- Semantic actions:
136
137   -- E -> E * T
138   action1 :: (Expr -> Parser r) -> Expr -> Token -> Term -> Parser r
139   action1 k v1 v2 v3 = k (Times v1 v3)
140
141   -- E -> E * * T
142   action2 :: (Expr -> Parser r) -> Expr -> Token -> Token -> Term -> Parser r
143   action2 k v1 v2 v3 v4 = k (Pow v1 v4)
144
145   -- E -> T
```

```
146 action3 :: (Expr -> Parser r) -> Term -> Parser r
147 action3 k v1 = k (Term v1)
148
149 -- T -> T + F
150 action4 :: (Term -> Parser r) -> Term -> Token -> Factor -> Parser r
151 action4 k v1 v2 v3 = k (Plus v1 v3)
152
153 -- T -> F
154 action5 :: (Term -> Parser r) -> Factor -> Parser r
155 action5 k v1 = k (Factor v1)
156
157 -- F -> ( E )
158 action6 :: (Factor -> Parser r) -> Token -> Expr -> Token -> Parser r
159 action6 k v1 v2 v3 = k (Expr v2)
160
161 -- F -> NUM
162 action7 k v1 = k (Num v1)
```

A.2. RAD

The user-supplied data types (`Token`, `Expr` etc.) and the semantic action functions are identical to the LALR case, which is why we omitted them.

```
1  type Parser r = [Token] -> r
2
3  -- Main parsing entry point.
4  entry :: Parser Expr
5  entry = rule0 (parse12 . const)
6
7
8  -- RECURSIVE DESCENT PART: RULE FUNCTIONS AND SINGLE SYMBOLS
9
10 -- E' -> • E
11 rule0 :: (Expr -> Parser r) -> Parser r
12 rule0 = parse4 -- E
13
14 -- E -> E * • T
15 rule1 :: (Term -> Parser r) -> Parser r
16 rule1 = parse5 -- T
17
18 -- E -> E * • * T
19 rule2 :: (Token -> Term -> Parser r) -> Parser r
20 rule2 k ts = parse9 cont1 ts where -- *
21   cont1 v1 ts = parse5 (k v1) ts -- T
22
23 -- E -> • T
24 rule3 :: (Term -> Parser r) -> Parser r
25 rule3 = parse5 -- T
26
27 -- T -> T • + F
28 rule4 :: (Token -> Factor -> Parser r) -> Parser r
29 rule4 k ts = parse8 cont1 ts where -- +
30   cont1 v1 ts = parse6 (k v1) ts -- F
31
32 -- T -> • F
33 rule5 :: (Factor -> Parser r) -> Parser r
```

```
34 rule5 = parse6 -- F
35
36 -- F -> • ( E )
37 rule6 :: (Token -> Expr -> Token -> Parser r) -> Parser r
38 rule6 k ts = parse10 cont1 ts where -- (
39   cont1 v1 ts = parse4 (cont2 v1) ts -- E
40   cont2 v1 v2 ts = parse11 (k v1 v2) ts -- )
41
42 -- F -> • NUM
43 rule7 = parse7 -- NUM
44
45
46 -- E
47 parse4 = state0
48
49 -- T
50 parse5 = state1
51
52 -- F
53 parse6 = state2
54
55 -- NUM
56 parse7 k (t@(TokenInt v):tr) = k v tr
57 parse7 k ts = happyError ts
58
59 -- +
60 parse8 :: (Token -> Parser r) -> Parser r
61 parse8 k (t@(TokenPlus):tr) = k t tr
62 parse8 k ts = happyError ts
63
64 -- *
65 parse9 :: (Token -> Parser r) -> Parser r
66 parse9 k (t@(TokenTimes):tr) = k t tr
67 parse9 k ts = happyError ts
68
69 -- (
70 parse10 :: (Token -> Parser r) -> Parser r
71 parse10 k (t@(TokenOB):tr) = k t tr
72 parse10 k ts = happyError ts
73
74 -- )
75 parse11 :: (Token -> Parser r) -> Parser r
76 parse11 k (t@(TokenCB):tr) = k t tr
77 parse11 k ts = happyError ts
78
79 -- %eof
80 parse12 :: Parser r -> Parser r
81 parse12 k [] = k []
82 parse12 k ts = happyError ts
83
84
85 -- RECURSIVE ASCENT PART: STATE FUNCTIONS
86
87 -- core: [_ -> · E]
88 state0 :: (Expr -> Parser r) -> Parser r
89 state0 k ts = case ts of
90   _ -> rule3 (action3 g4) ts
91   where
92     g4 x = state3 (k x) (action1 g4 x) (action2 g4 x)
93
94 -- core: [_ -> · T]
```

```
95  state1 :: (Term -> Parser r) -> Parser r
96  state1 k ts = case ts of
97    _ -> rule5 (action5 g5) ts
98    where
99      g5 x = state4 (k x) (action4 g5 x)
100
101 -- core: [_ -> · F]
102 state2 :: (Factor -> Parser r) -> Parser r
103 state2 k ts = case ts of
104   (TokenInt _):tr -> rule7 (action7 g6) ts
105   _ -> rule6 (action6 g6) ts
106   where
107     g6 x = state5 (k x)
108
109 -- core: [_ -> E ·], [E -> E · * T], [E -> E · * * T]
110 state3 :: Parser r -> (Token -> Term -> Parser r) -> (Token -> Token -> Term
    ↪ -> Parser r) -> Parser r
111 state3 k1 k2 k3 ts = case ts of
112   t@(TokenTimes):tr -> state6 (k2 t) (k3 t) tr
113   _ -> k1 ts
114
115 -- core: [_ -> T ·], [T -> T · + F]
116 state4 :: Parser r -> (Token -> Factor -> Parser r) -> Parser r
117 state4 k1 k2 ts = case ts of
118   (TokenPlus):tr -> rule4 (k2) ts
119   _ -> k1 ts
120
121 -- core: [_ -> F ·]
122 state5 = id
123
124 -- core: [E -> E * · T], [E -> E * · * T]
125 state6 :: (Term -> Parser r) -> (Token -> Term -> Parser r) -> Parser r
126 state6 k1 k2 ts = case ts of
127   (TokenTimes):tr -> rule2 (k2) ts
128   _ -> rule1 (k1) ts
```