From Context-Free Grammars to Parsers

Eelco Visser

TU Delft

CS4200 | Compiler Construction | October 14, 2021

This Lecture

Derivations

- Generating sentences and trees from context-free grammars

Reductions

- From sentences to symbols

Parsing

- Shift/reduce parsing
- SLR parsing

Reading Material

These slides are the primary material for study of parsing in this course.

This provides a basic idea of parsing by studying derivations and one particular parsing algorithm.

It is by no means exhaustive.

There is a vast literature on parsing, and we encourage the interested student to read on.

Parsing

Eelco Visser

TU Delft

CS4200 | Compiler Construction | October 8, 2020

Classical compiler textbook

Chapter 4: Syntax Analysis

The material covered in this lecture corresponds to Sections 4.1, 4.2, 4.3, 4.5, 4.6

Pictures in these slides are copies from the book

Compilers: Principles, Techniques, and Tools, 2nd Edition

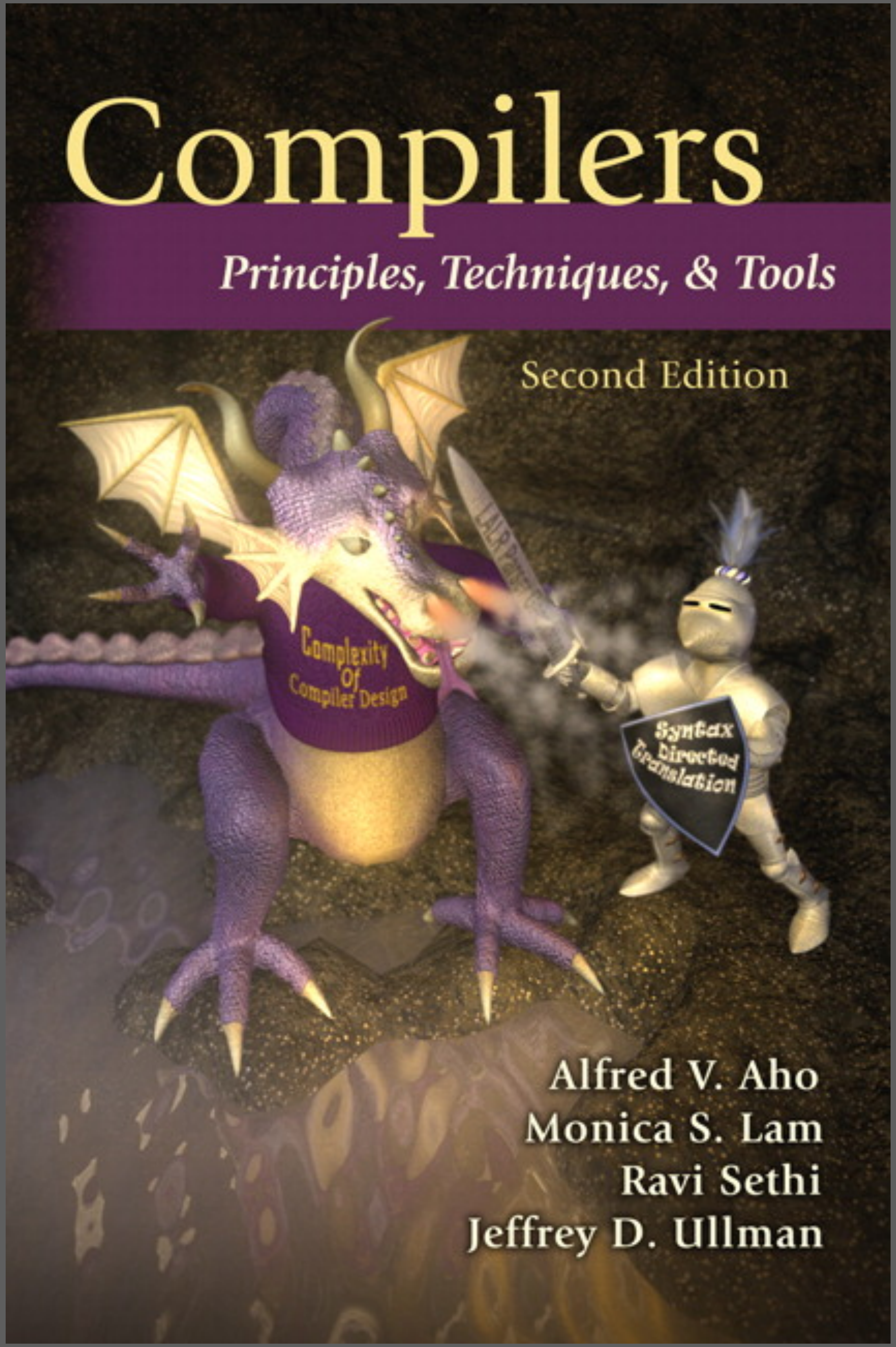
Alfred V. Aho, Columbia University

Monica S. Lam, Stanford University

Ravi Sethi, Avaya Labs

Jeffrey D. Ullman, Stanford University

2007 | *Pearson*


Compilers
Principles, Techniques, & Tools
Second Edition
Complexity Of Compiler Design
Syntax Directed Translation
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

The perspective of this lecture on declarative syntax definition is Explained more elaborately in this Onward! 2010 essay. It uses an on older version of SDF than used in these slides. Production rules have the form

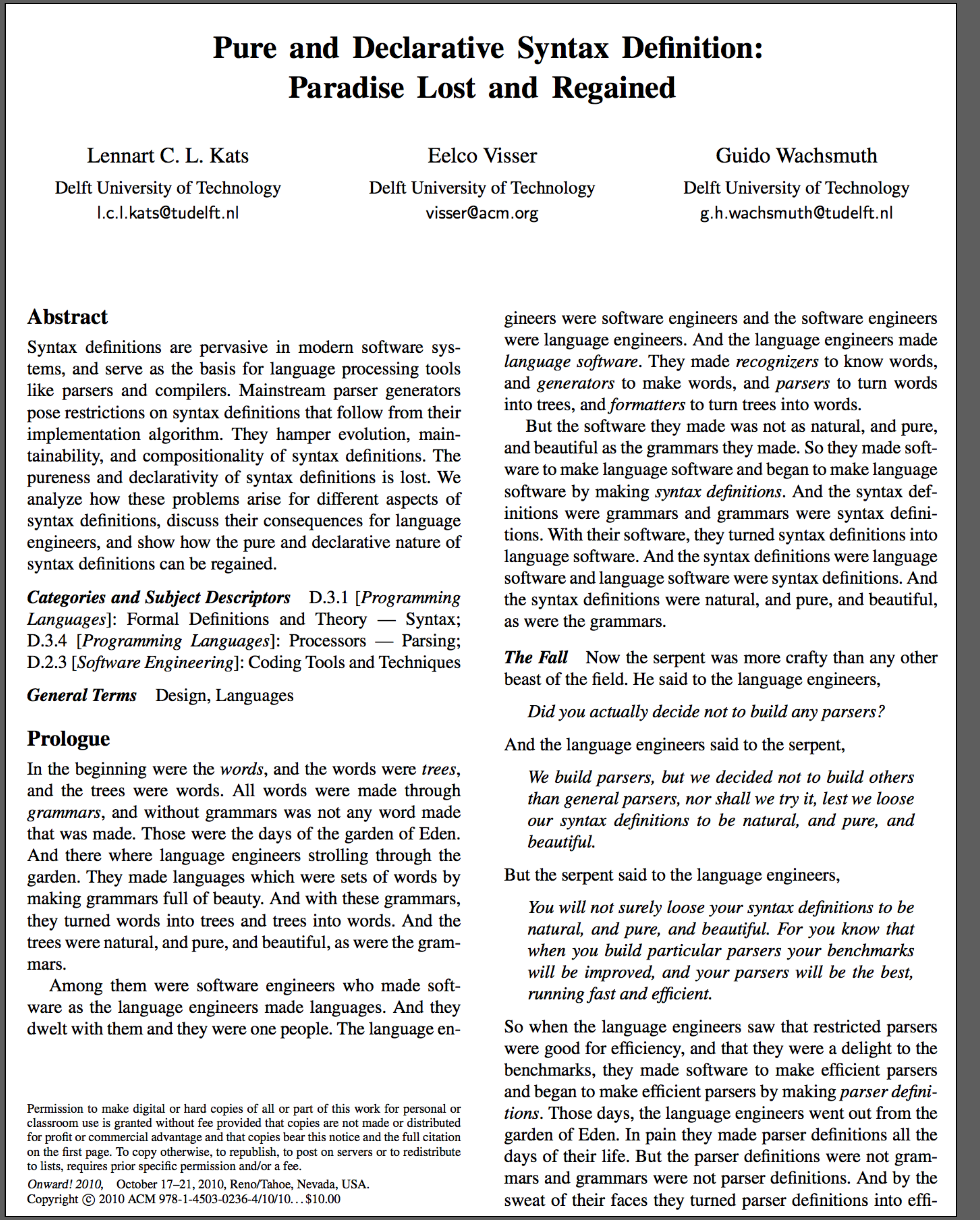
$$X_1 \ldots X_n \rightarrow N \ \{\texttt{cons("C")}\}$$

instead of

$$N.C = X_1 \ldots X_n$$

```
https://doi.org/10.1145/1932682.1869535
```

```
https://eelcovisser.org/publications/2010/KatsVW10.pdf
```



Pure and Declarative Syntax Definition: Paradise Lost and Regained

Lennart C. L. Kats
Delft University of Technology
l.c.l.kats@tudelft.nl

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
g.h.wachsmuth@tudelft.nl

Abstract

Syntax definitions are pervasive in modern software systems, and serve as the basis for language processing tools like parsers and compilers. Mainstream parser generators pose restrictions on syntax definitions that follow from their implementation algorithm. They hamper evolution, maintainability, and compositionality of syntax definitions. The pureness and declarativity of syntax definitions is lost. We analyze how these problems arise for different aspects of syntax definitions, discuss their consequences for language engineers, and show how the pure and declarative nature of syntax definitions can be regained.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory — Syntax; D.3.4 [*Programming Languages*]: Processors — Parsing; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Design, Languages

Prologue

In the beginning were the *words*, and the words were *trees*, and the trees were words. All words were made through *grammars*, and without grammars was not any word made that was made. Those were the days of the garden of Eden. And there where language engineers strolling through the garden. They made languages which were sets of words by making grammars full of beauty. And with these grammars, they turned words into trees and trees into words. And the trees were natural, and pure, and beautiful, as were the grammars.

Among them were software engineers who made software as the language engineers made languages. And they dwelt with them and they were one people. The language engineers were software engineers and the software engineers were language engineers. And the language engineers made *language software*. They made *recognizers* to know words, and *generators* to make words, and *parsers* to turn words into trees, and *formatters* to turn trees into words.

But the software they made was not as natural, and pure, and beautiful as the grammars they made. So they made software to make language software and began to make language software by making *syntax definitions*. And the syntax definitions were grammars and grammars were syntax definitions. With their software, they turned syntax definitions into language software. And the syntax definitions were language software and language software were syntax definitions. And the syntax definitions were natural, and pure, and beautiful, as were the grammars.

The Fall Now the serpent was more crafty than any other beast of the field. He said to the language engineers,

Did you actually decide not to build any parsers?

And the language engineers said to the serpent,

We build parsers, but we decided not to build others than general parsers, nor shall we try it, lest we loose our syntax definitions to be natural, and pure, and beautiful.

But the serpent said to the language engineers,

You will not surely loose your syntax definitions to be natural, and pure, and beautiful. For you know that when you build particular parsers your benchmarks will be improved, and your parsers will be the best, running fast and efficient.

So when the language engineers saw that restricted parsers were good for efficiency, and that they were a delight to the benchmarks, they made software to make efficient parsers and began to make efficient parsers by making *parser definitions*. Those days, the language engineers went out from the garden of Eden. In pain they made parser definitions all the days of their life. But the parser definitions were not grammars and grammars were not parser definitions. And by the sweat of their faces they turned parser definitions into effi-

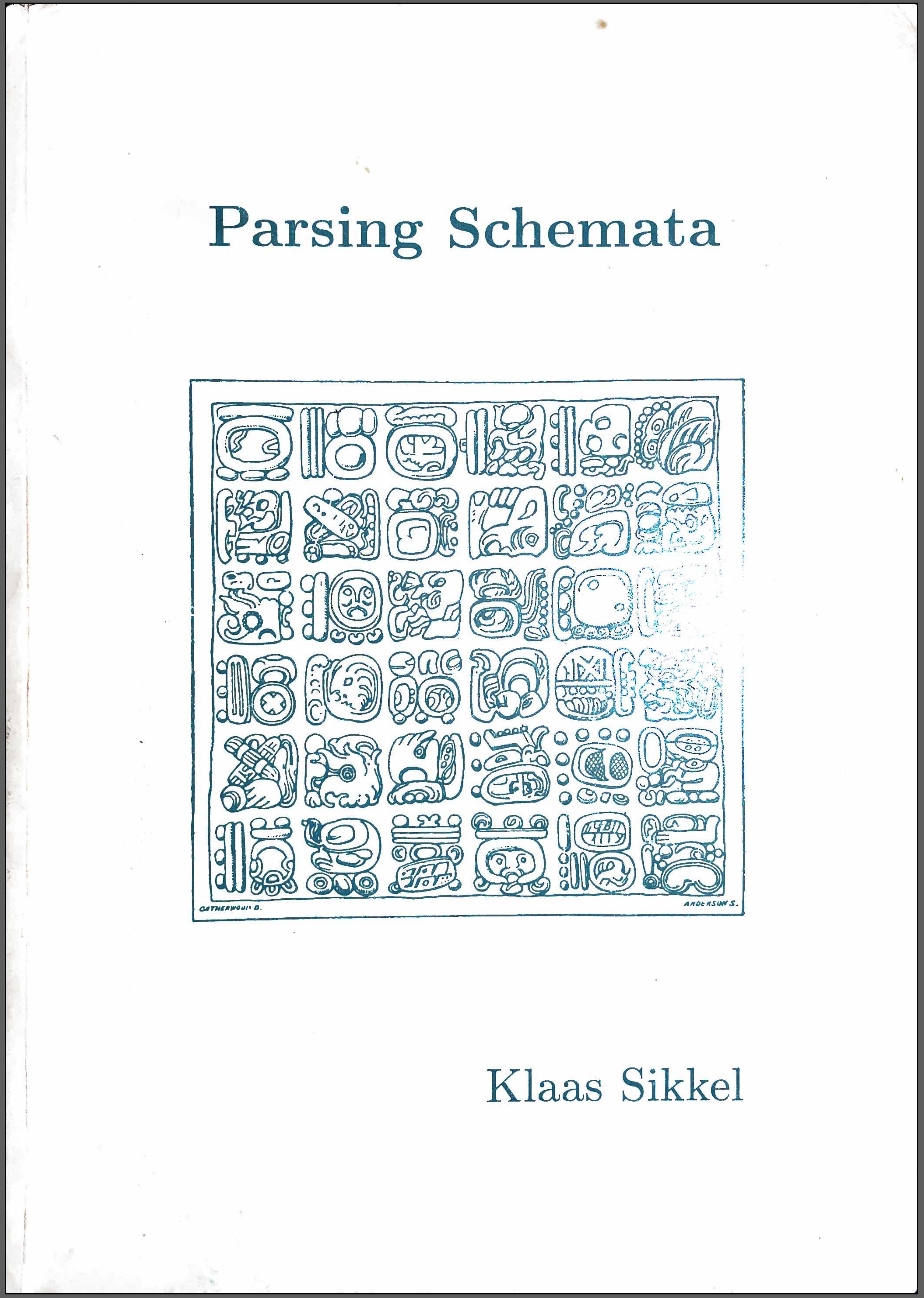
This PhD thesis presents a uniform framework for describing a wide range of parsing algorithms.

“Parsing schemata provide a general framework for specication, analysis and comparison of (sequential and/or parallel) parsing algorithms. A grammar specifies implicitly what the valid parses of a sentence are; a parsing algorithm specifies Elicitly how to compute these. Parsing schemata form a well-defined level of abstraction in between grammars and parsing algorithms. A parsing schema specifies the types of intermediate results that can be computed by a parser, and the rules that allow to Eand a given set of such results with new results. A parsing schema does not specify the data structures, control structures, and (in case of parallel processing) communication structures that are to be used by a parser.”

For the interested

Sikkel, N. (1993). *Parsing Schemata.*
PhD thesis. Enschede: Universiteit Twente

https://research.utwente.nl/en/publications/parsing-schemata



This paper applies parsing schemata to disambiguation filters for priority conflicts.

A Case Study in Optimizing Parsing Schemata by Disambiguation Filters

Eelco Visser

Abstract

Disambiguation methods for context-free grammars enable concise specification of programming languages by ambiguous grammars. A disambiguation filter is a function that selects a subset from a set of parse trees—the possible parse trees for an ambiguous sentence. The framework of filters provides a declarative description of disambiguation methods independent of parsing. Although filters can be implemented straightforwardly as functions that prune the parse forest produced by some generalized parser, this can be too inefficient for practical applications.

In this paper the optimization of parsing schemata, a framework for high-level description of parsing algorithms, by disambiguation filters is considered in order to find efficient parsing algorithms for declaratively specified disambiguation methods. As a case study the optimization of the parsing schema of Earley's parsing algorithm by two filters is investigated. The main result is a technique for generation of efficient LR-like parsers for ambiguous grammars modulo priorities.

1 Introduction

The syntax of programming languages is conventionally described by context-free grammars. Although programming languages should be unambiguous, they are often described by ambiguous grammars because these allow a more natural formulation and yield better abstract syntax. For instance, the grammar

$$E \to E + E; E \to E * E; E \to a$$

gives a clearer description of arithmetic expressions than the grammar

$$E \to E + T; E \to T; T \to T * a; T \to a$$

To obtain an unambiguous specification of a language described by an ambiguous grammar it has to be disambiguated. For example, the grammar above can be disambiguated by associativity and priority rules that express that $E \to E * E$ has higher priority than $E \to E + E$ and that both productions are left associative.

Technical Report P9507, Programming Research Group, University of Amsterdam, July 1995.
Available as: ftp://ftp.fwi.uva.nl/pub/programming-research/reports/1995/P9507.ps.Z
This is an extended version of [Vis95].

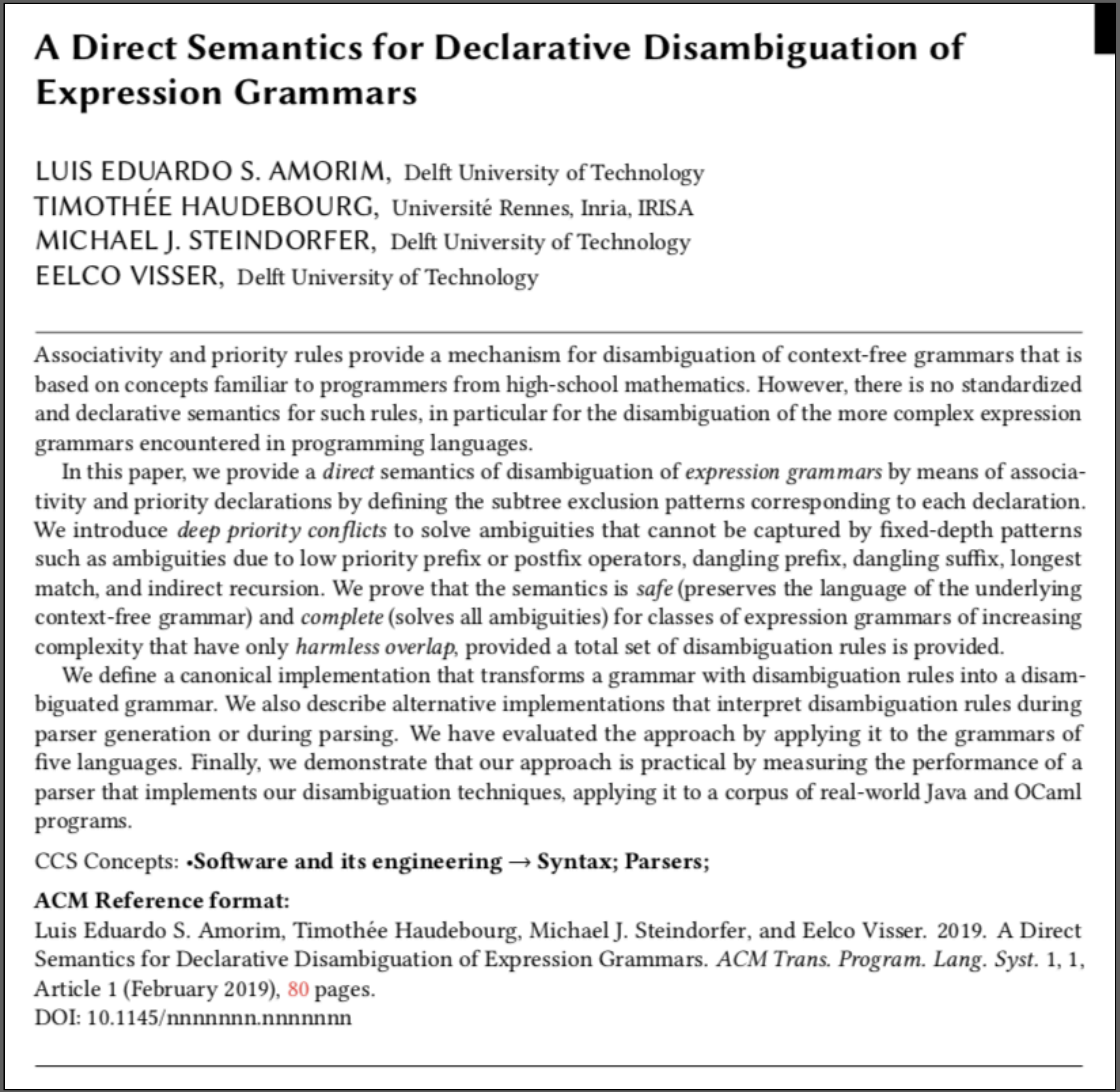
For the interested

https://eelcovisser.org/publications/1997/Visser97-IWPT.pdf

This paper defines a declarative semantics for associativity and priority declarations for disambiguation.

The paper provides a safe semantics and extends it to deep priority conflicts.

The result of disambiguation is a contextual grammar, which generalises the disambiguation for the dangling-else grammar.



A Direct Semantics for Declarative Disambiguation of Expression Grammars

LUIS EDUARDO S. AMORIM, Delft University of Technology
TIMOTHÉE HAUDEBOURG, Université Rennes, Inria, IRISA
MICHAEL J. STEINDORFER, Delft University of Technology
EELCO VISSER, Delft University of Technology

Associativity and priority rules provide a mechanism for disambiguation of context-free grammars that is based on concepts familiar to programmers from high-school mathematics. However, there is no standardized and declarative semantics for such rules, in particular for the disambiguation of the more complex expression grammars encountered in programming languages.

In this paper, we provide a *direct* semantics of disambiguation of *expression grammars* by means of associativity and priority declarations by defining the subtree exclusion patterns corresponding to each declaration. We introduce *deep priority conflicts* to solve ambiguities that cannot be captured by fixed-depth patterns such as ambiguities due to low priority prefix or postfix operators, dangling prefix, dangling suffix, longest match, and indirect recursion. We prove that the semantics is *safe* (preserves the language of the underlying context-free grammar) and *complete* (solves all ambiguities) for classes of expression grammars of increasing complexity that have only *harmless overlap*, provided a total set of disambiguation rules is provided.

We define a canonical implementation that transforms a grammar with disambiguation rules into a disambiguated grammar. We also describe alternative implementations that interpret disambiguation rules during parser generation or during parsing. We have evaluated the approach by applying it to the grammars of five languages. Finally, we demonstrate that our approach is practical by measuring the performance of a parser that implements our disambiguation techniques, applying it to a corpus of real-world Java and OCaml programs.

CCS Concepts: •**Software and its engineering → Syntax; Parsers;**

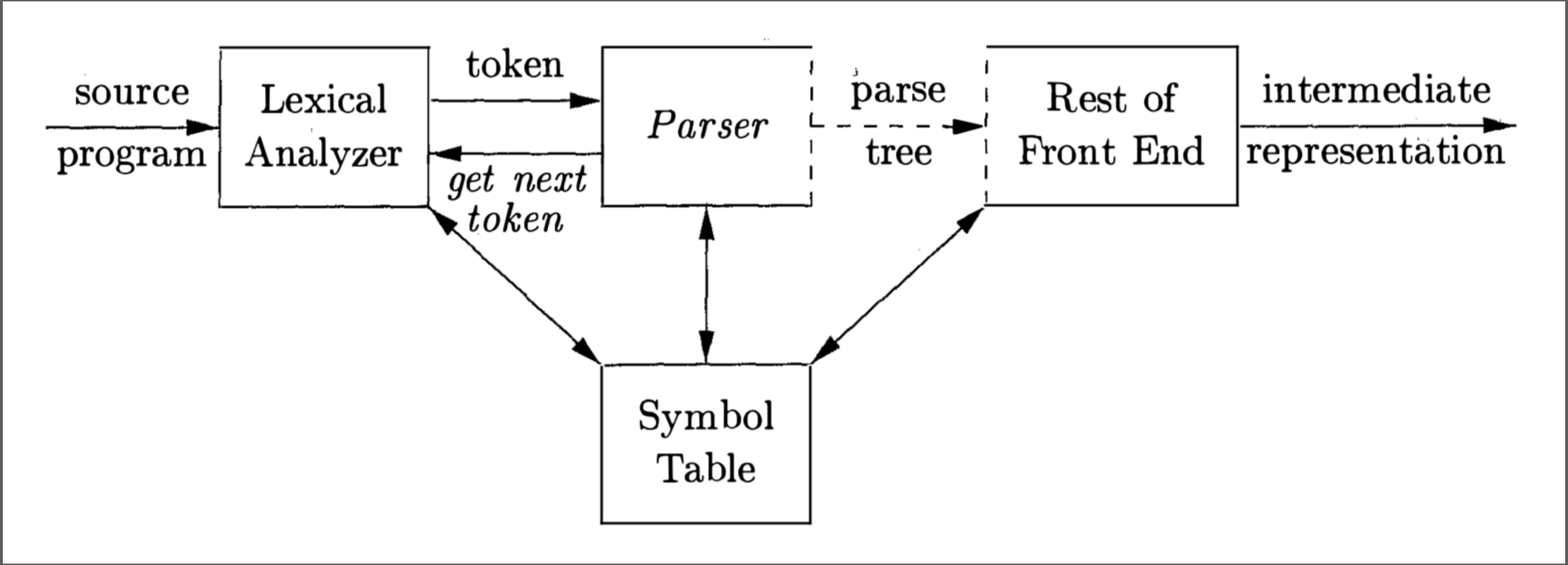
ACM Reference format:
Luis Eduardo S. Amorim, Timothée Haudebourg, Michael J. Steindorfer, and Eelco Visser. 2019. A Direct Semantics for Declarative Disambiguation of Expression Grammars. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (February 2019), 80 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

For the interested

The paper is still in production. Ask me for a copy of the draft.

Parser Architecture

Traditional Parser Architecture



Source: Compilers Principles, Techniques & Tools

Context-Free Grammars

Context-Free Grammars

Terminals

- Basic symbols from which strings are formed

Nonterminals

- Syntactic variables that denote sets of strings

Start Symbol

- Denotes the nonterminal that generates strings of the languages

Productions

- A = X … X
- Head/left side (A) is a nonterminal
- Body/right side (X … X) zero or more terminals and nonterminals

Example Context-Free Grammar

```
grammar
  start S
  non-terminals E T F
  terminals "+" "*" "(" ")" ID
  productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

Abbreviated Grammar

```
grammar
  start S
  non-terminals E T F
  terminals "+" "*" "(" ")" ID
  productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
grammar
  productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

Nonterminals, terminals can be derived from productions

First production defines start symbol

Notation

```
A,B,C : non-terminals
l     : terminals
α,β,γ : strings of non-terminals and terminals
w,v   : strings of terminal symbols
```

Meta: Syntax of Grammars

```
context-free syntax // grammars

  Grammar.Grammar = <
    grammar
      <Start?>
      <Sorts?>
      <Terminals?>
      <Productions>
  >
```

```
context-free syntax

  Production.Prod = <
    <Symbol><Constructor?> = <Symbol*>
  >

  Symbol.NT = <<ID>>
  Symbol.T  = <<STRING>>
  Symbol.L  = <<LCID>>

  Constructor.Con = <.<ID>>
```

```
context-free syntax

  Start.Start = <
    start <ID>
  >

  Sorts.Sorts = <
    sorts <ID*>
  >

  Sorts.NonTerminals = <
    non-terminals <ID*>
  >

  Terminals.Terminals = <
    terminals <Symbol*>
  >

  Productions.Productions = <
    productions
      <Production*>
  >
```

Derivations: Generating Sentences from Symbols

Meta: Syntax of Derivations

```
context-free syntax // derivations

  Derivation.Derivation = <
    derivation
      <Symbol> <Step*>
  >

  Step.Step   = [⇒ [Symbol*]]
  Step.Steps  = [⇛ [Symbol*]]
  Step.Steps1 = [⇒+ [Symbol*]]
```

Derivations

```
grammar
  productions
    E = E "+" E
    E = E "*" E
    E = "-" E
    E = "(" E ")"
    E = ID
```

```
Derivation step: replace symbol by rhs of production

Example: Given production E = E "+" E replace E by E "+" E

Derivation: repeatedly apply derivations
```

```
derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" ID ")"
```

```
derivation // derives in zero or more steps
  E ⟹ "-" "(" ID "+" ID ")"
```

Left-Most Derivation

```
grammar
  productions
    E = E "+" E
    E = E "*" E
    E = "-" E
    E = "(" E ")"
    E = ID
```

```
derivation // left-most derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" E "+" E ")"
  ⇒ "-" "(" ID "+" E ")"
  ⇒ "-" "(" ID "+" ID ")"
```

Left-most derivation: Expand left-most non-terminal at each step

Right-Most Derivation

```
grammar
  productions
    E = E "+" E
    E = E "*" E
    E = "-" E
    E = "(" E ")"
    E = ID
```

```
derivation // left-most derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" E "+" E ")"
  ⇒ "-" "(" ID "+" E ")"
  ⇒ "-" "(" ID "+" ID ")"
```

```
derivation // right-most derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" E "+" E ")"
  ⇒ "-" "(" E "+" ID ")"
  ⇒ "-" "(" ID "+" ID ")"
```

Right-most derivation: Expand right-most non-terminal at each step

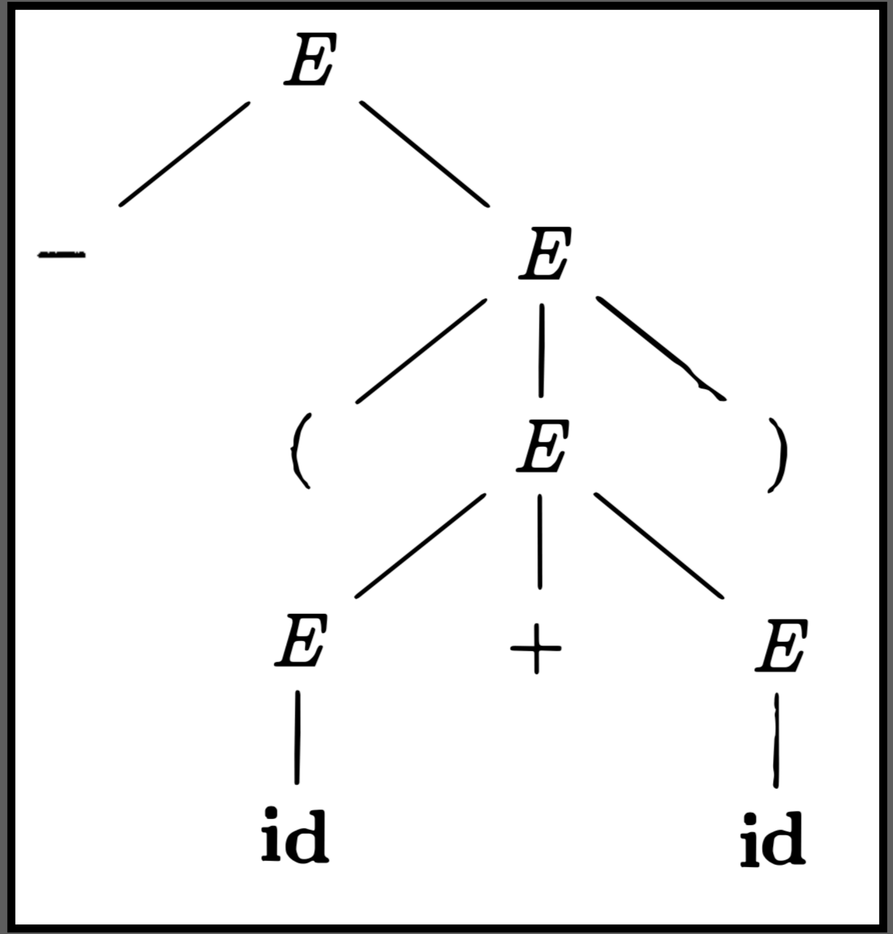
Meta: Tree Derivations

```
context-free syntax // tree derivations

  Derivation.TreeDerivation = <
    tree derivation
      <Symbol> <PStep*>
  >

  PStep.Step   = [⇒ [PT*]]
  PStep.Steps  = [⟹ [PT*]]
  PStep.Steps1 = [⇒+ [PT*]]

  PT.App = <<Symbol>[<PT*>]>
  PT.Str = <<STRING>>
  PT.Sym = <<Symbol>>
```

Left-Most Tree Derivation

```
grammar
  productions
    E.A = E "+" E
    E.T = E "*" E
    E.N = "-" E
    E.P = "(" E ")"
    E.V = ID
```

```
derivation // left-most derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" E "+" E ")"
  ⇒ "-" "(" ID "+" E ")"
  ⇒ "-" "(" ID "+" ID ")"
```



```
tree derivation // left-most
  E
  ⇒ E["-" E]
  ⇒ E["-" E["(" E ")"]]
  ⇒ E["-" E["(" E[E "+" E] ")"]]
  ⇒ E["-" E["(" E[E[ID] "+" E] ")"]]
  ⇒ E["-" E["(" E[E[ID] "+" E[ID]] ")"]]
```

Left-Most Tree Derivation

E
⇒
E
–
E
⇒
E
–
E
(
E
)
⇒
E
–
E
(
E
)
E
+
E
⇒
E
–
E
(
E
)
E
+
E
id
⇒
E
–
E
(
E
)
E
+
E
id
id

```
tree derivation // left-most
  E
  ⇒ E["-" E]
  ⇒ E["-" E["(" E ")"]]
  ⇒ E["-" E["(" E[E "+" E] ")"]]
  ⇒ E["-" E["(" E[E[ID] "+" E] ")"]]
  ⇒ E["-" E["(" E[E[ID] "+" E[ID]] ")"]]
```

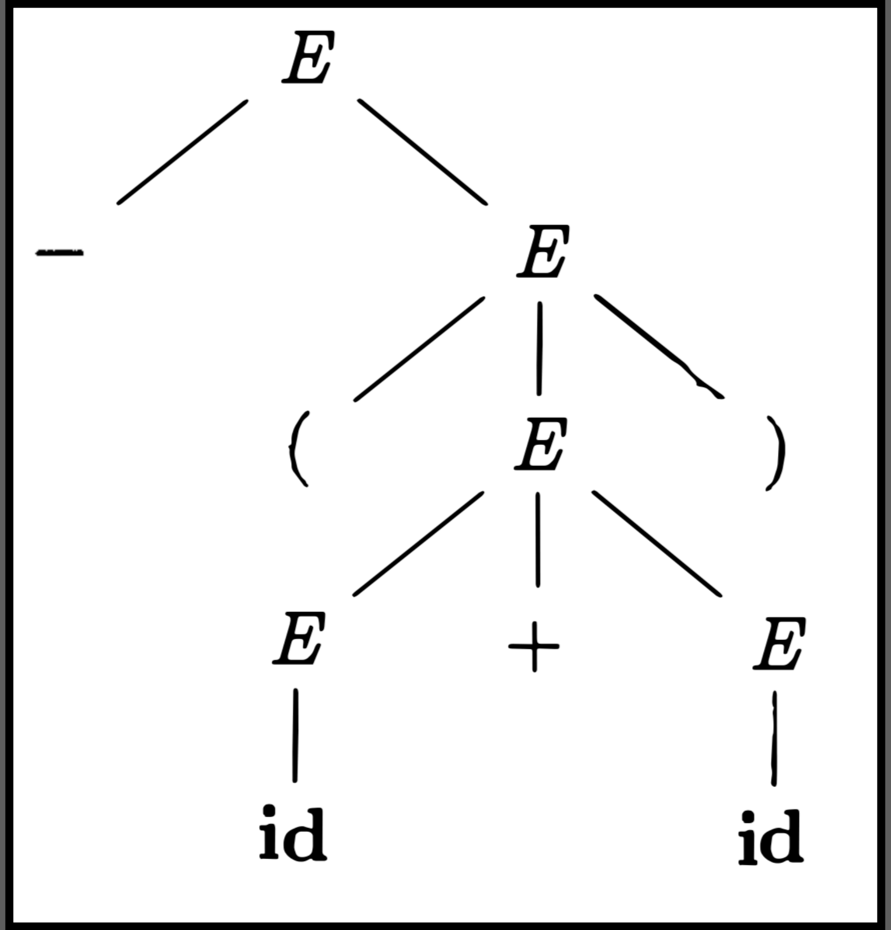
Meta: Term Derivations

```
context-free syntax // term derivations

  Derivation.TermDerivation = <
    term derivation
      <Symbol> <TStep*>
  >

  TStep.Step   = [⇒ [Term*]]
  TStep.Steps  = [⟹ [Term*]]
  TStep.Steps1 = [⇒+ [Term*]]

  Term.App = <<ID>(<{Term ","}*>)>
  Term.Str = <<STRING>>
  Term.Sym = <<Symbol>>
```

Left-Most Term Derivation

```
grammar
  productions
    E.A = E "+" E
    E.T = E "*" E
    E.N = "-" E
    E.P = "(" E ")"
    E.V = ID
```

```
derivation // left-most derivation
  E
  ⇒ "-" E
  ⇒ "-" "(" E ")"
  ⇒ "-" "(" E "+" E ")"
  ⇒ "-" "(" ID "+" E ")"
  ⇒ "-" "(" ID "+" ID ")"
```



```
term derivation // left-most
  E
  ⇒ N(E)
  ⇒ N(P(E))
  ⇒ N(P(A(E, E))
  ⇒ N(P(A(V(ID), E))
  ⇒ N(P(A(V(ID), V(ID)))
```

Parse Trees Represent Derivations

```
List<String> YIELD(T : Tree) {
  T match {
    [A = Ts] ⇒ YIELDs(Ts);
    Str ⇒ [Str];
  };
}

List<String> YIELDs(Ts : List<Tree>) {
  Ts match {
    [] ⇒ "";
    [T | Ts] ⇒ YIELD(T) ++ YIELDs(Ts);
  };
}
```

$$S \Longrightarrow\!\!\!\!\!\!\gg PT$$

iff

$$S \Longrightarrow\!\!\!\!\!\!\gg \mathrm{YIELD}(PT)$$

Language Defined by a Grammar

```
L(G) = { w | S ⟹> w }
```

Language: sentences

```
T(G) = { T | S ⟹> T }
```

Language: trees

```
L(G) = YIELD(T(G))
```

Reducing Sentences to Symbols

Meta: Reductions

```
context-free syntax

  Reduction.Reduction = <
    reduction
      <Symbol*> <RStep*>
  >

  RStep.Step    = [ ⟸ [Symbol*]]
  RStep.Steps   = [ <⟸ [Symbol*]]
  RStep.Steps1  = [ ⟸+ [Symbol*]]
```

```
context-free syntax

  Reduction.TreeReduction = <
    tree reduction
      <PT*> <PRStep*>
  >

  PRStep.Step    = [ ⟸ [PT*]]
  PRStep.Steps   = [ <⟸ [PT*]]
  PRStep.Steps1  = [ ⟸+ [PT*]]
```

```
context-free syntax

  Reduction.TermReduction = <
    term reduction
      <Term*> <TRStep*>
  >

  TRStep.Step    = [ ⟸ [Term*]]
  TRStep.Steps   = [ <⟸ [Term*]]
  TRStep.Steps1  = [ ⟸+ [Term*]]
```

A Reduction is an Inverse Derivation

```
grammar
  sorts A
  productions
    A = β
```

```
reduction
  α β γ ⟸ α A γ
```

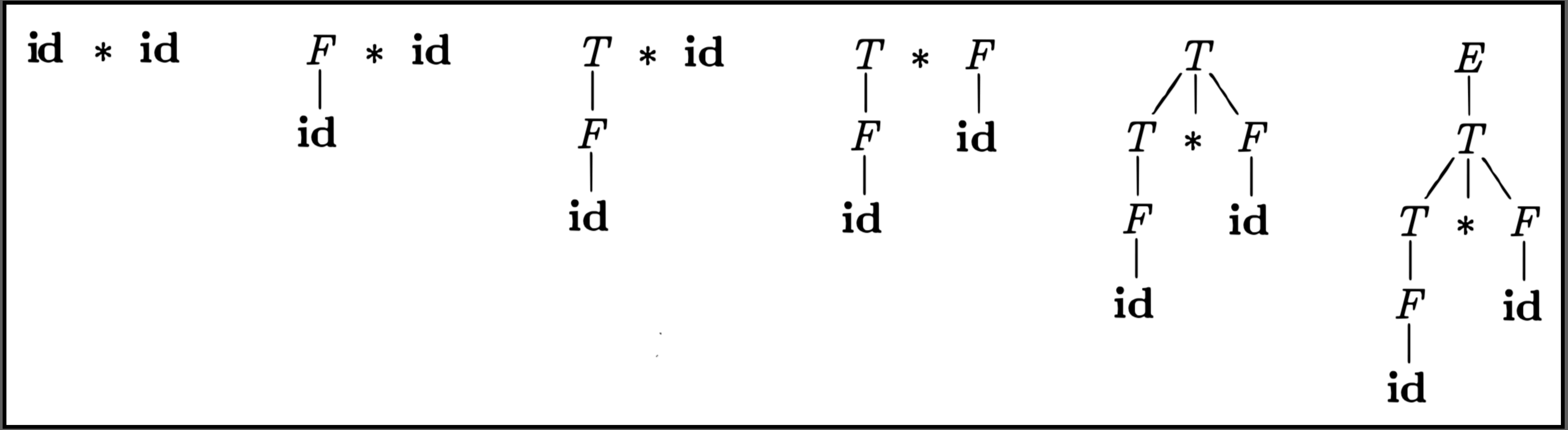
Reducing to Symbols

```
grammar
  sorts E T F ID
  productions
    E.P = E "+" T
    E.E = T
    T.M = T "*" F
    T.T = F
    F.B = "(" E ")"
    F.V = ID
```

```
reduction
  ID "*" ID
   ⟸ F "*" ID
   ⟸ T "*" ID
   ⟸ T "*" F
   ⟸ T
   ⟸ E
```

Reducing to Parse Trees

```
grammar
  sorts E T F ID
  productions
    E.P = E "+" T
    E.E = T
    T.M = T "*" F
    T.T = F
    F.B = "(" E ")"
    F.V = ID
```

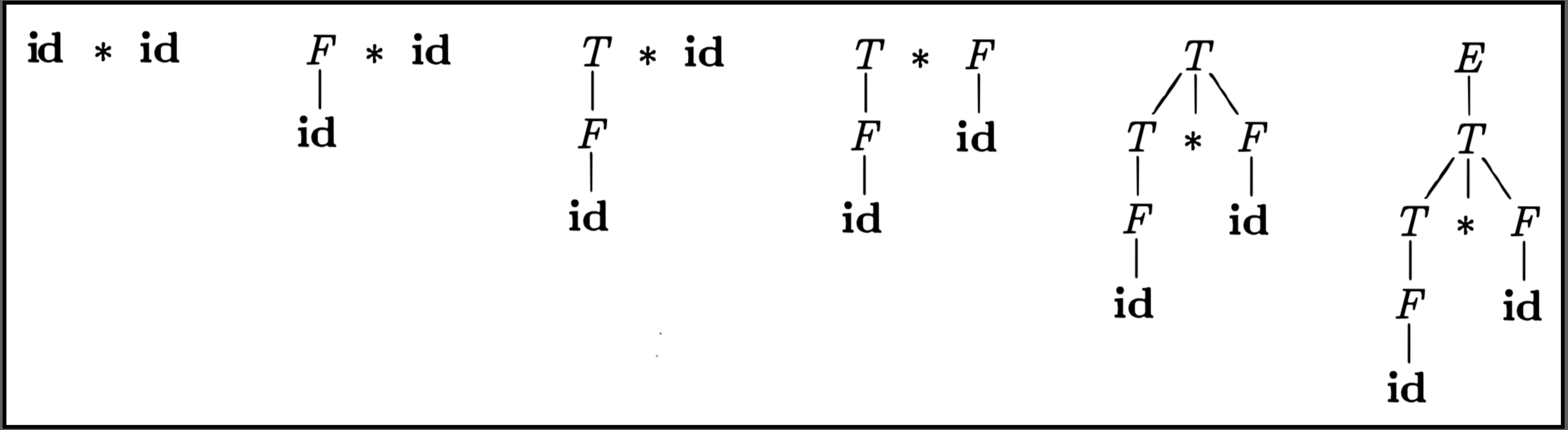


```
reduction
  ID "*" ID
   ⟸ F "*" ID
   ⟸ T "*" ID
   ⟸ T "*" F
   ⟸ T
   ⟸ E
```

```
tree reduction
  ID "*" ID
   ⟸ F[ID] "*" ID
   ⟸ T[F[ID]] "*" ID
   ⟸ T[F[ID]] "*" F[ID]
   ⟸ T[T[F[ID]] "*" F[ID]]
   ⟸ E[T[T[F[ID]] "*" F[ID]]]
```

Reducing to Abstract Syntax Terms

```
grammar
  sorts E T F ID
  productions
    E.P = E "+" T
    E.E = T
    T.M = T "*" F
    T.T = F
    F.B = "(" E ")"
    F.V = ID
```



```
reduction
  ID "*" ID
   ⟸ F "*" ID
   ⟸ T "*" ID
   ⟸ T "*" F
   ⟸ T
   ⟸ E
```

```
tree reduction
  ID "*" ID
   ⟸ F[ID] "*" ID
   ⟸ T[F[ID]] "*" ID
   ⟸ T[F[ID]] "*" F[ID]
   ⟸ T[T[F[ID]] "*" F[ID]]
   ⟸ E[T[T[F[ID]] "*" F[ID]]]
```

```
term reduction
  ID "*" ID
   ⟸ V(ID) "*" ID
   ⟸ T(V(ID)) "*" ID
   ⟸ T(V(ID)) "*" V(ID)
   ⟸ M(T(V(ID)), V(ID))
   ⟸ E(M(T(V(ID)), V(ID)))
```

Handle

```
grammar
  productions
    A = β
```

```
derivation // right-most derivation
  S ⟹» α A w ⟹ α b w
```

```
reduction
  α β w  ⟸ α A w «⟸ S
```

```
Sentential form: string of non-terminal and terminal symbols
that can be derived from start symbol

S ⟹» α

Right sentential form: a sentential form derived by a right-most derivation

Handle: the part of a right sentential form that if reduced
would produce the previous right-sentential form in a right-most derivation
```

Shift-Reduce Parsing

Shift-Reduce Parsing Machine

```
shift-reduce parse
  $ stack | input $ action
```

```
     $ a   | l w $ shift
  ⇒ $ a l |   w $
  // shift input symbol on the stack
```

```
    $ a b | w $ reduce by A = b
 ⇒ $ a A | w $
  // reduce n symbols of the stack to symbol A
```

```
$ S | $ accept
// reduced to start symbol; accept
```

```
 $ a | w $ error
  // no action possible in this state
```

Shift-Reduce Parsing

```
grammar
  productions
    E.P = E "+" T
    E.E = T
    T.M = T "*" F
    T.T = F
    F.B = "(" E ")"
    F.V = ID
```

```
shift-reduce parse
     $           | ID "*" ID $ shift
  ⇒ $ ID         |    "*" ID $ reduce by F = ID
  ⇒ $ F          |    "*" ID $ reduce by T = F
  ⇒ $ T          |    "*" ID $ shift
  ⇒ $ T "*"      |        ID $ shift
  ⇒ $ T "*" ID   |           $ reduce by F = ID
  ⇒ $ T "*" F    |           $ reduce by T = T "*" F
  ⇒ $ T          |           $ reduce by E = T
  ⇒ $ E          |           $ accept
```

Shift-reduce parsing constructs a right-most derivation

Shift-Reduce Conflicts

```
grammar
  sorts S E
  productions
    S.If  = if E then S
    S.IfE = if E then S else S
    S     = other
```

```
shift-reduce parse
      $                 | if E then S else S $ shift
  ⇒   $ if              |    E then S else S $ shift
  ⇒>> $ if E then S     |             else S $ shift
  ⇒   $ if E then S else |                  S $ shift
  ⇒   $ if E then S else S |                  $ reduce by S = if E then S else S
  ⇒   $ S               |                    $ accept
```

```
shift-reduce parse
      $                 | if E then S else S $ shift
  ⇒   $ if              |    E then S else S $ shift
  ⇒>> $ if E then S     |             else S $ reduce by S = if E then S
  ⇒   $ S               |             else S $ error
```

Shift-Reduce Conflicts

```
grammar
  sorts S E
  productions
    S.If  = if E then S
    S.IfE = if E then S else S
    S     = other
```

```
shift-reduce parse
     $                            | if E then if E then S else S $ ...
  ⟹» $ if E then if E then S      |                     else S $ shift
  ⇒  $ if E then if E then S else |                          S $ shift
  ⇒  $ if E then if E then S else S |                          $ reduce by S = if E then S else S
  ⇒  $ if E then S                |                            $ reduce by S = if E then S
  ⇒  $ S                          |                            $ accept
```

```
shift-reduce parse
     $                            | if E then if E then S else S $ ...
  ⟹» $ if E then if E then S      |                     else S $ reduce by S = if E then S
  ⇒  $ if E then S                |                     else S $ shift
  ⇒  $ if E then S else           |                          S $ shift
  ⇒  $ if E then S else S         |                            $ reduce by S = if E then S else S
  ⇒  $ S
```

Shift-Reduce Conflicts (with Trees)

```
grammar
  sorts S E
  productions
    S.If  = if E then S
    S.IfE = if E then S else S
    S     = other
```

```
shift-reduce tree parse
     $                              | if E then if E then S else S $ ...
  ⟹» $ if E then if E then S        |                      else S $ shift
  ⇒  $ if E then if E then S else   |                           S $ shift
  ⇒  $ if E then if E then S else S |                             $ reduce by S = if E then S else S
  ⇒  $ if E then S[if E then S else S]  |                         $ reduce by S = if E then S
  ⇒  $ S[if E then S[if E then S else S]] |                       $ accept
```

```
shift-reduce tree parse
     $                              | if E then if E then S else S $ ...
  ⟹» $ if E then if E then S        |                      else S $ reduce by S = if E then S
  ⇒  $ if E then S[if E then S]     |                      else S $ shift
  ⇒  $ if E then S[if E then S] else  |                         S $ shift
  ⇒  $ if E then S[if E then S] else S |                          $ reduce by S = if E then S else S
  ⇒  $ S[if E then S[if E then S] else S] |                       $ accept
```

Simple LR Parsing

How can we make shift-reduce parsing deterministic?

```
grammar
  productions
    E.P = E "+" T
    E.E = T
    T.M = T "*" F
    T.T = F
    F.B = "(" E ")"
    F.V = ID
```

```
shift-reduce parse
     $           | ID "*" ID $ shift
  ⇒ $ ID        |    "*" ID $ reduce by F = ID
  ⇒ $ F         |    "*" ID $ reduce by T = F
  ⇒ $ T         |    "*" ID $ shift
  ⇒ $ T "*"     |        ID $ shift
  ⇒ $ T "*" ID  |           $ reduce by F = ID
  ⇒ $ T "*" F   |           $ reduce by T = T "*" F
  ⇒ $ T         |           $ reduce by E = T
  ⇒ $ E         |           $ accept
```

Is there a production in the grammar that matches the top of the stack?

How to choose between possible shift and reduce actions?

LR Parsing

LR(k) Parsing

- L: left-to-right scanning
- R: constructing a right-most derivation
- k: the number of lookahead symbols

Motivation for LR

- LR grammars for many programming languages
- Most general non-backtracking shift-reduce parsing method
- Early detection of parse errors
- Class of grammars superset of predictive/LL methods

SLR: Simple LR

Item

```
E = E "+" T
```

Production

```
[E = . E "+" T]
[E = E . "+" T]
[E = E "+" . T]
[E = E "+" T .]
```

Items

We expect to see an expression

We have seen an expression and may see a “+”

Item indicates how far we have progressed in parsing a production

Item Set

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
{
  [F = "(" . E ")"]
  [E = . E "+" T]
  [E = . T]
  [T = . T "*" F]
  [T = . F]
  [F = . "(" E ")"]
  [F = . ID]
}
```

Item set used to keep track where we are in a parse

Closure of an Item Set

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
{
  [F = "(" . E ")"]
}
```

```
{
  [F = "(" . E ")"]
  [E = . E "+" T]
  [E = . T]
  [T = . T "*" F]
  [T = . F]
  [F = . "(" E ")"]
  [F = . ID]
}
```

```
Set<Item> Closure(I) {
  J := I;
  for(each [A = a . B b] in J)
    for(each [B = c] in G)
      if(not [B = . c] in J)
        J := J + [B = .c];
  return J;
}
```

Goto

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
{
  [S = . E]
  [E = . E "+" T]
  [E = . T]
  [T = . T "*" F]
  [T = . F]
  [F = . "(" E ")"]
  [F = . ID]
}
```

```
Set<Item> Goto(I, X) {
  J := {};
  for(each [A = a . X b] in I)
    J := J + [A = a X . b];
  return Closure(J);
}
```

```
{
  [F = "(" . E ")"]
}
```

```
{
  [F = "(" . E ")"]
  [E = . E "+" T]
  [E = . T]
  [T = . T "*" F]
  [T = . F]
  [F = . "(" E ")"]
  [F = . ID]
}
```

Computing LR(0) Automaton

```
Set<Set<Item>> Items(G) {
  C := {Closure({[Sp = . S]})};
  W := C;
  while(W = {I | W'}) {
    W := W';
    for(each X in G) {
      J := Goto(I, X);
      if(not J = {} and not J in C)
        C := C + J;
        W := W + J;
      }
  }
  return C;
}
```

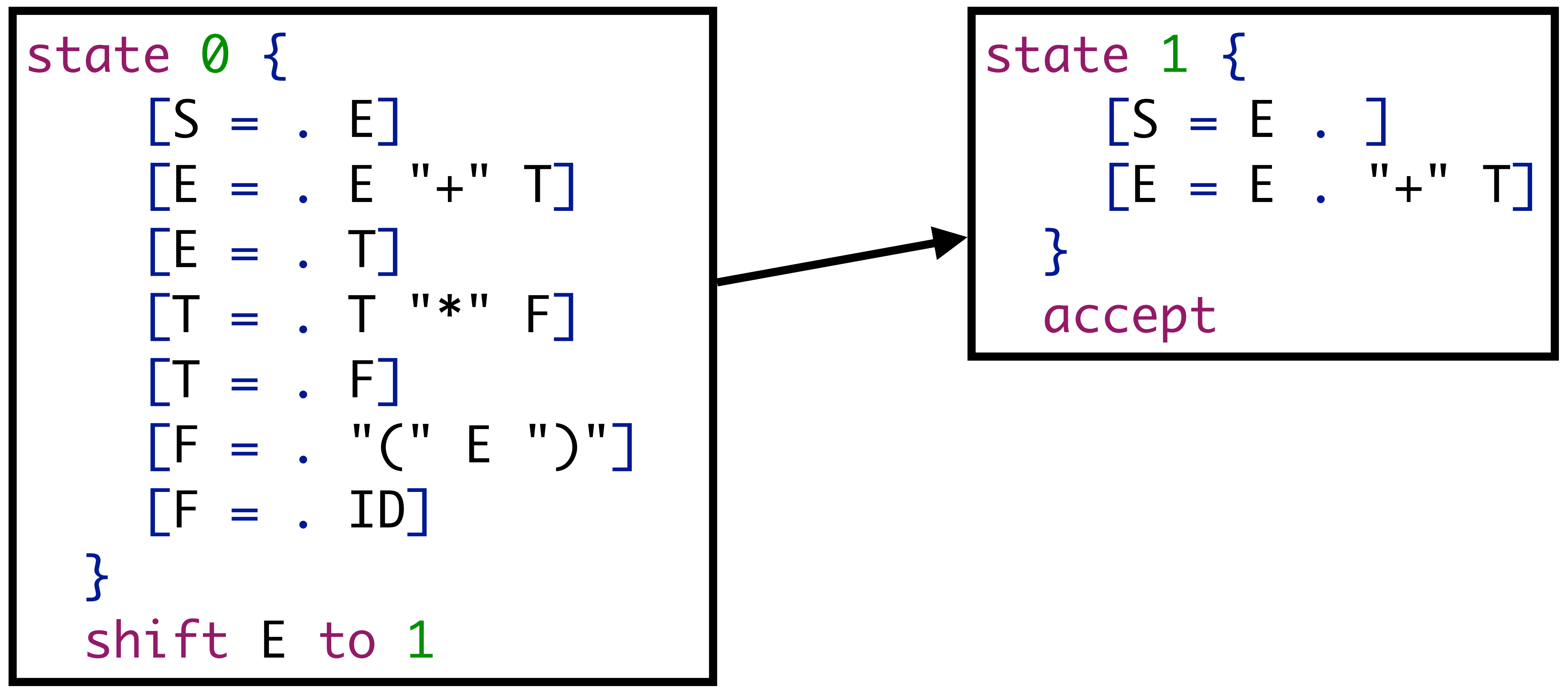
Computing LR(0) Automaton

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
state 0 {
    [S = . E]
  }
```

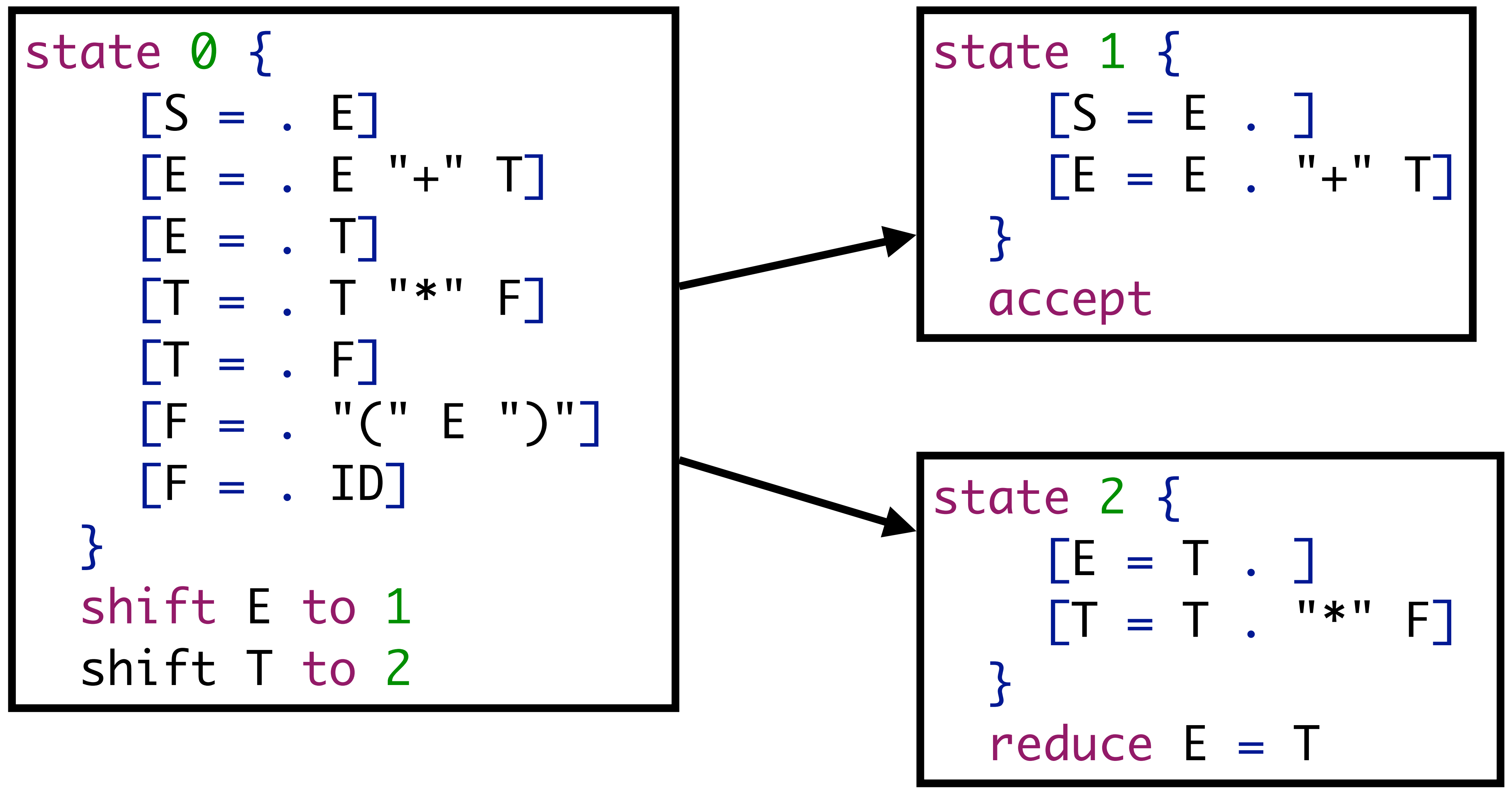
```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
```

```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
```

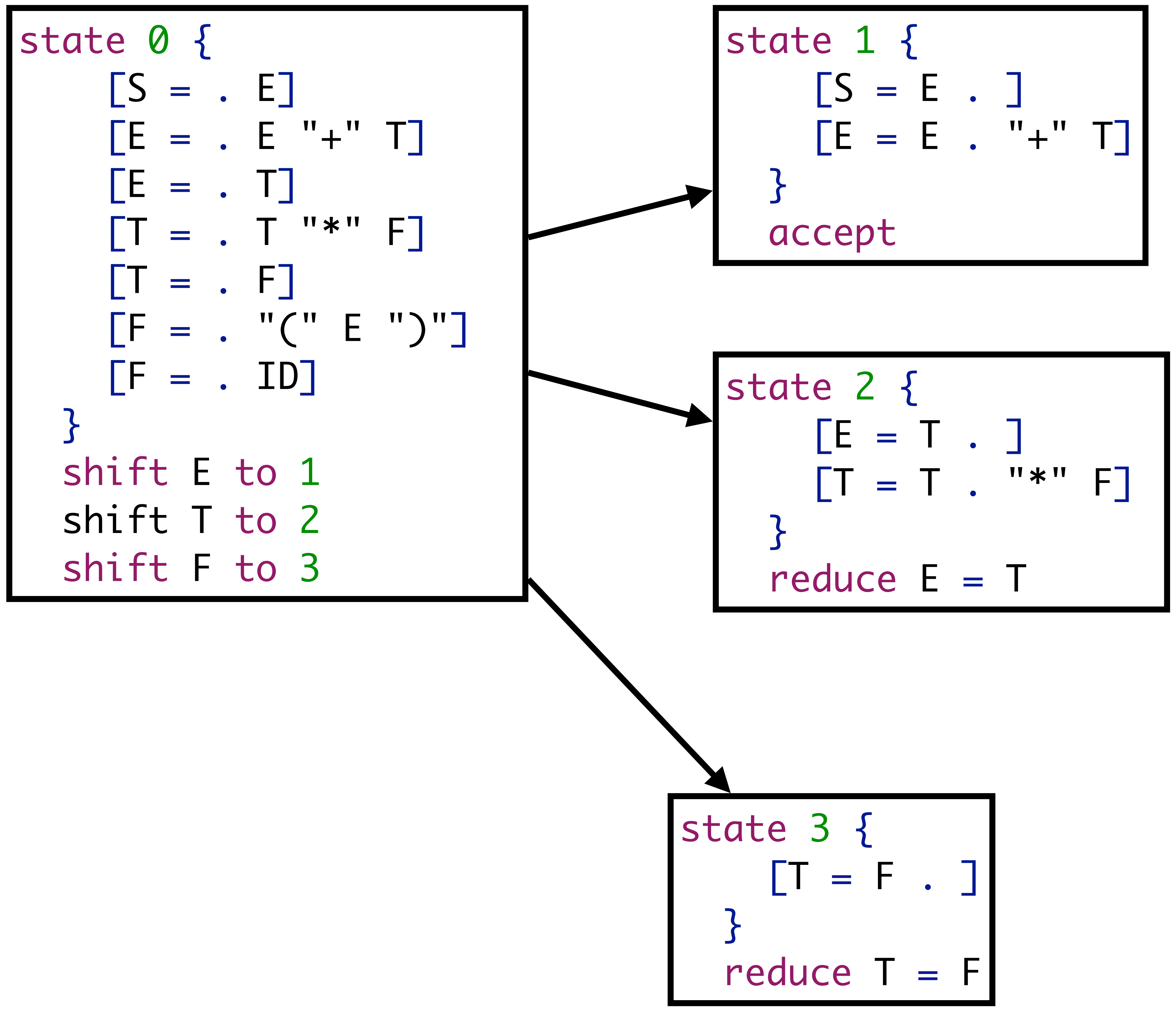
```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
```

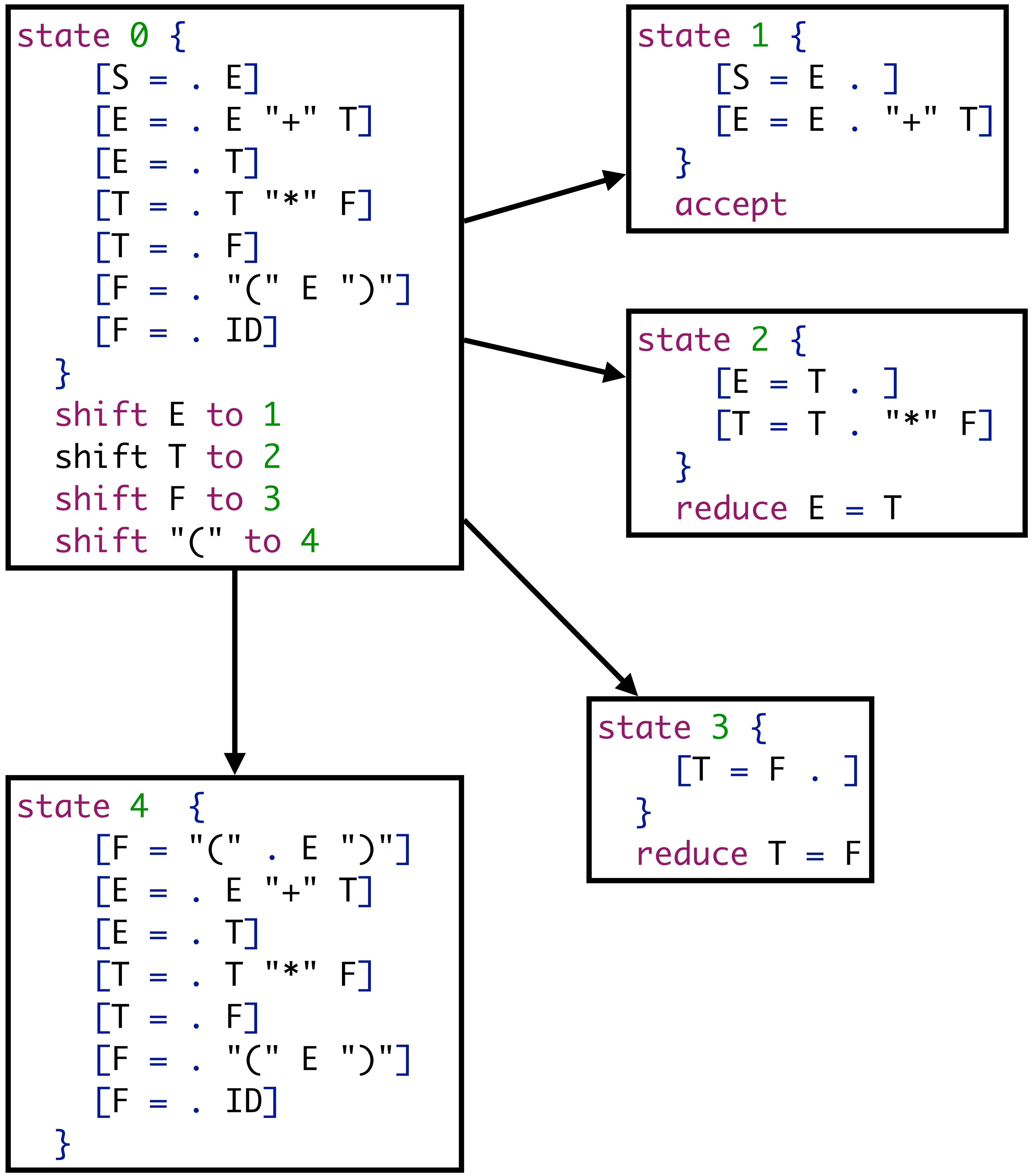


```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```



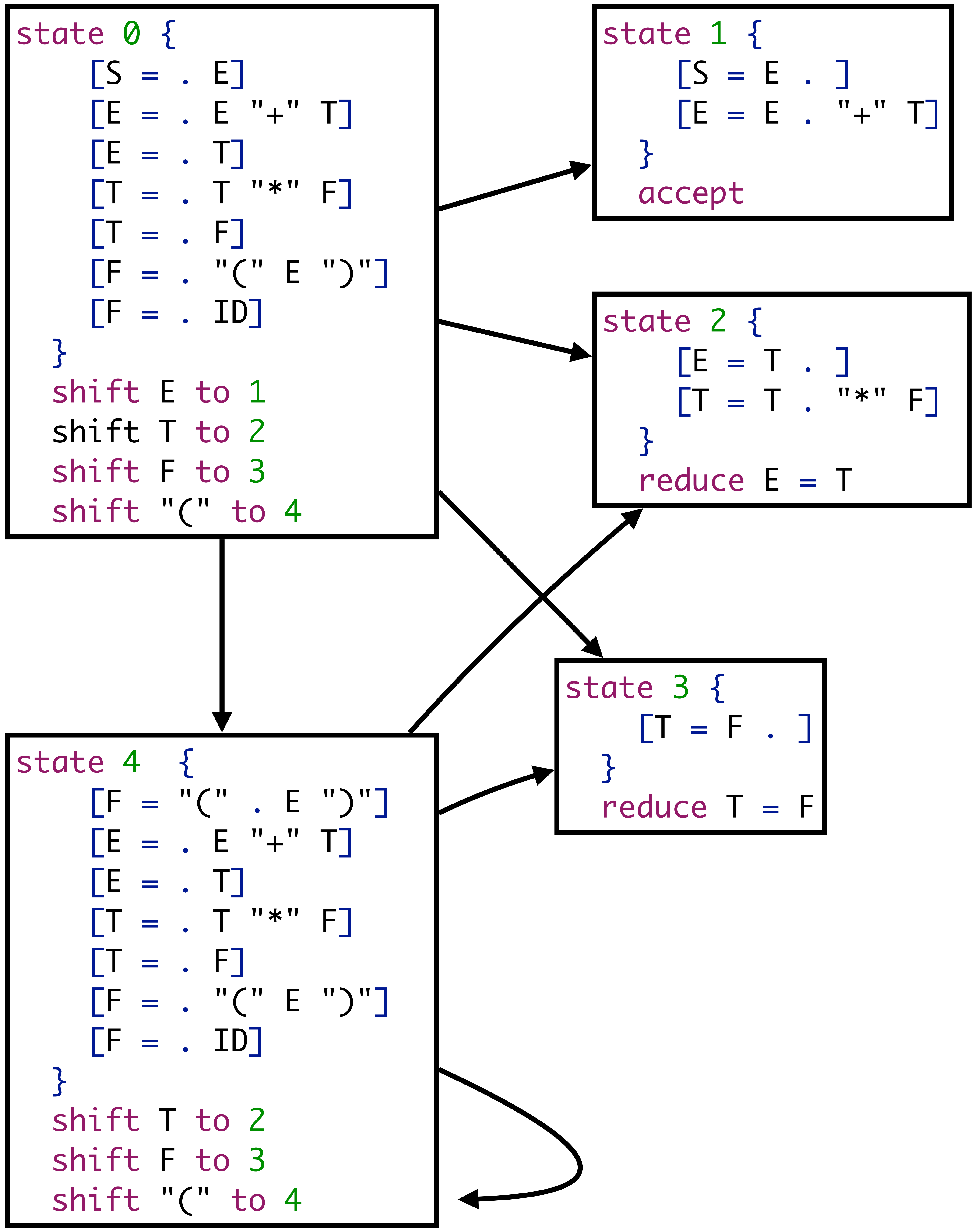
```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
```



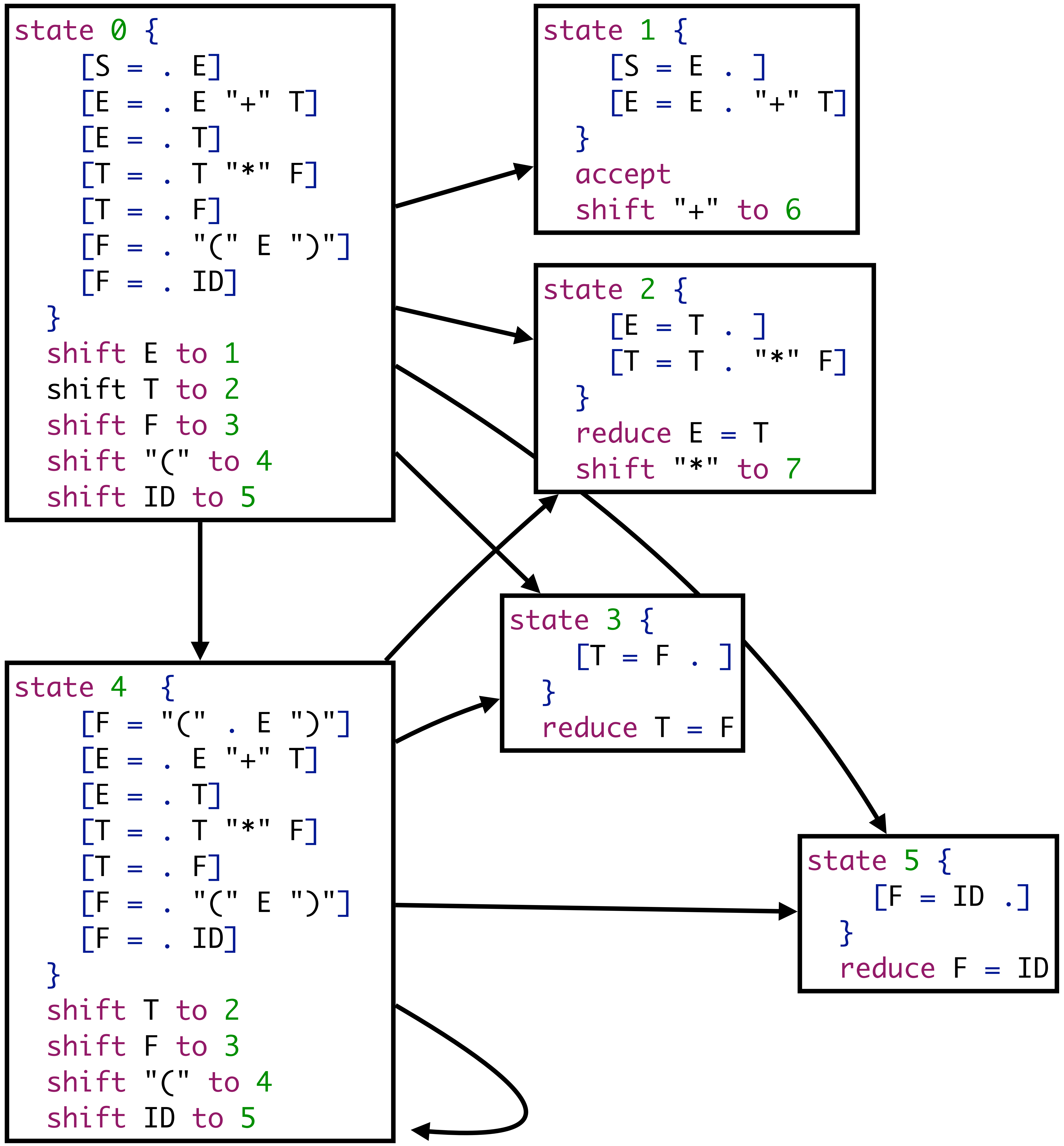
```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 2
  shift F to 3
  shift "(" to 4
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

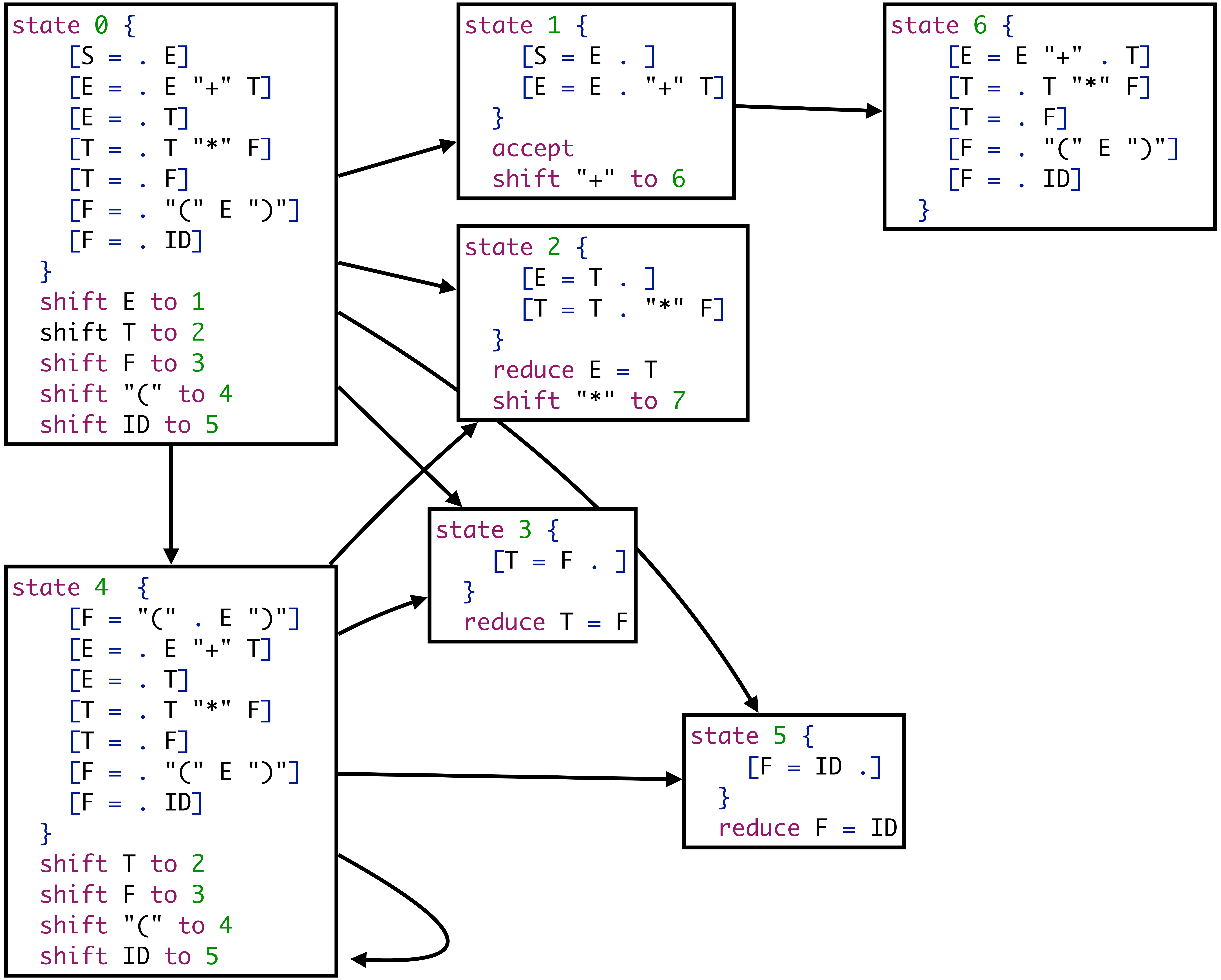
```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

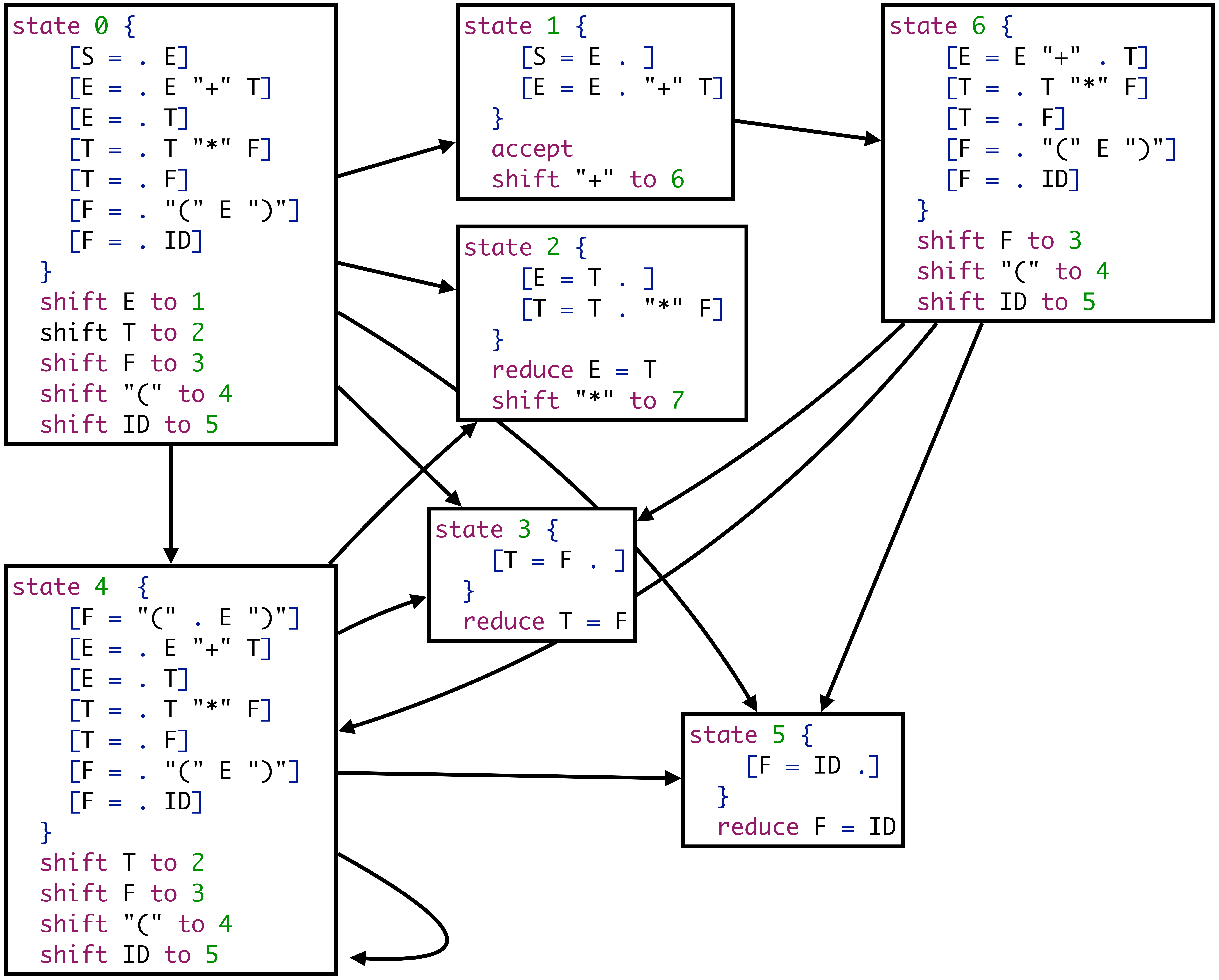
```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

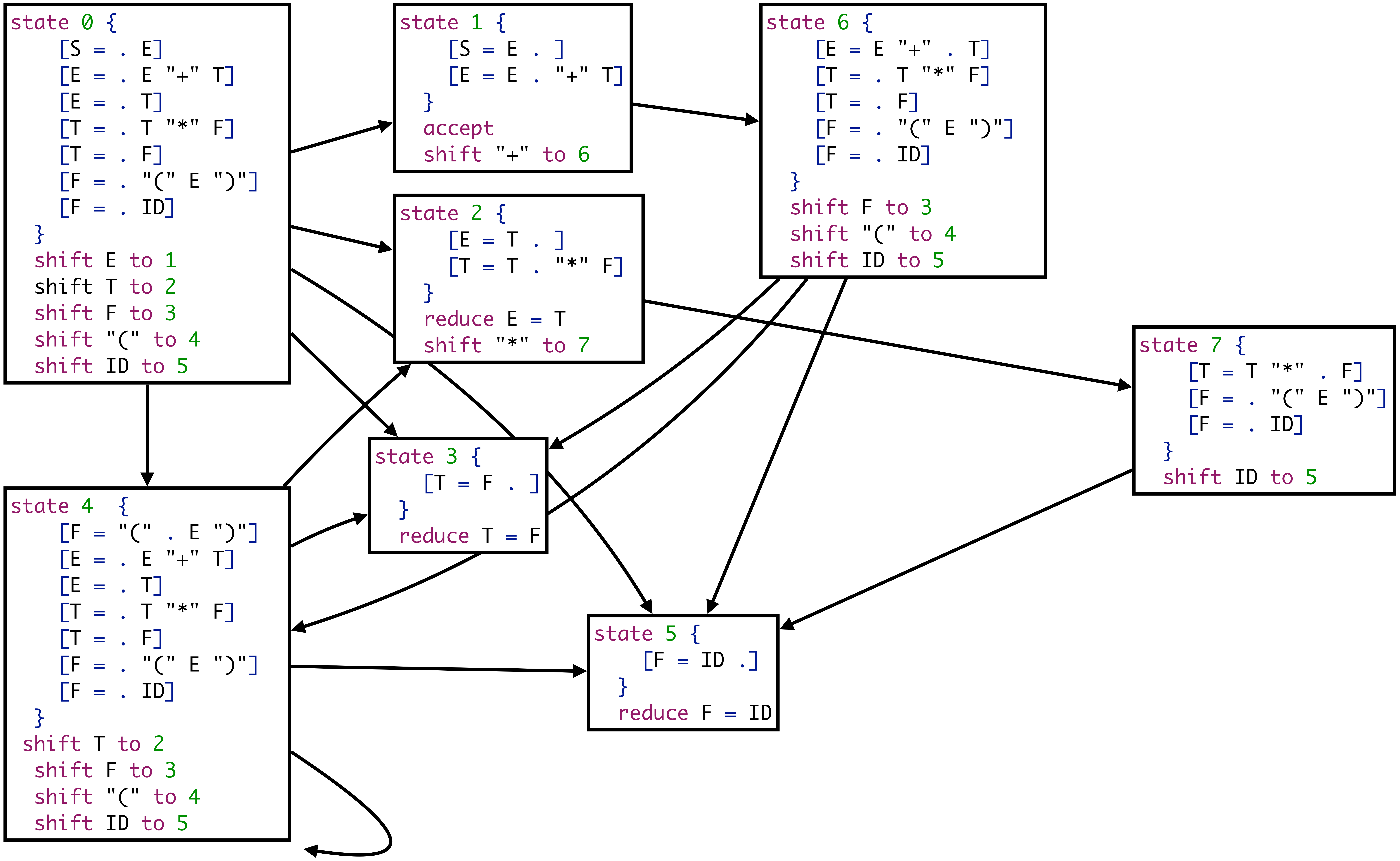
```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

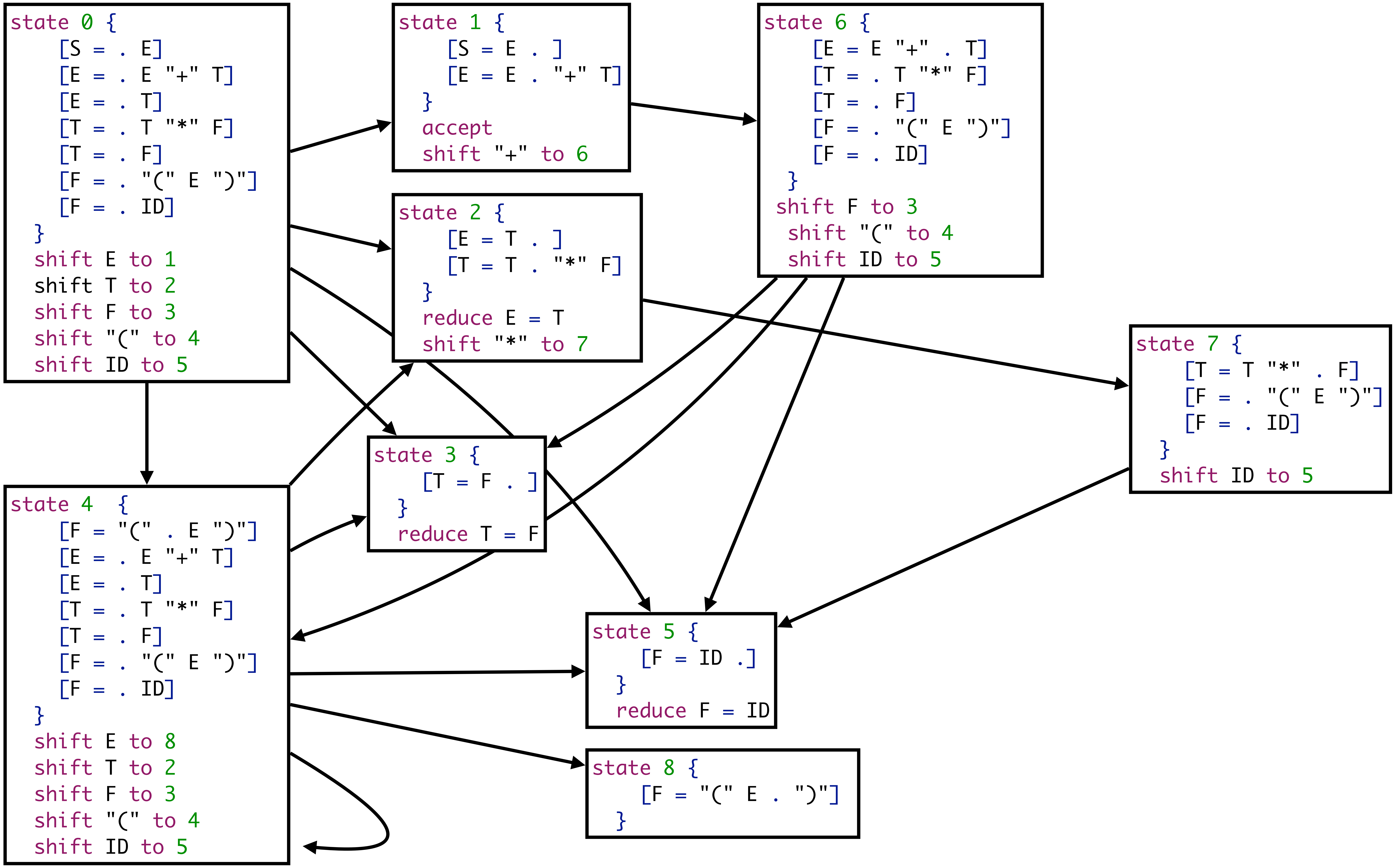
```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
 shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift ID to 5
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
 shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

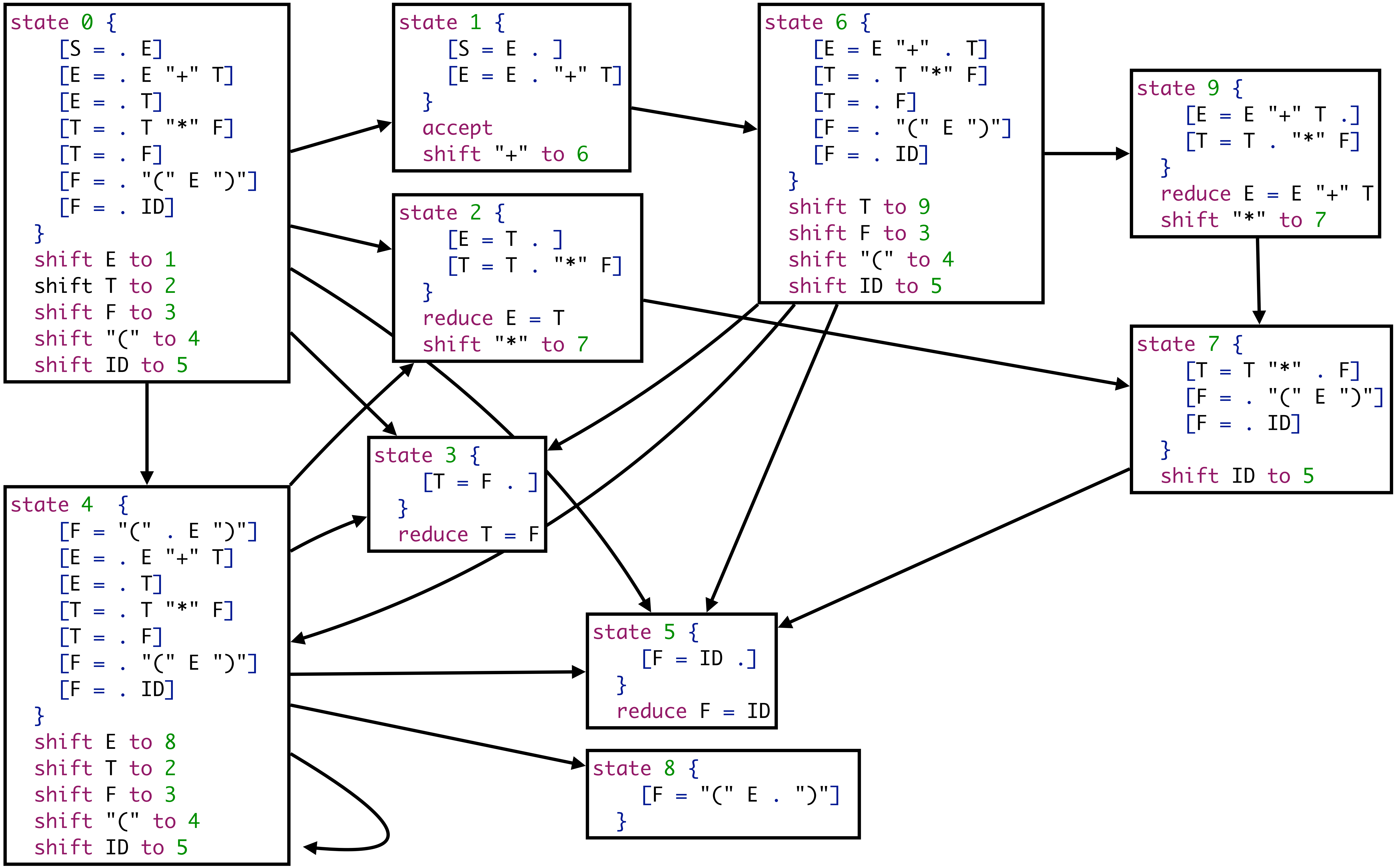
```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift ID to 5
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 8
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 8 {
    [F = "(" E . ")"]
  }
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 8
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

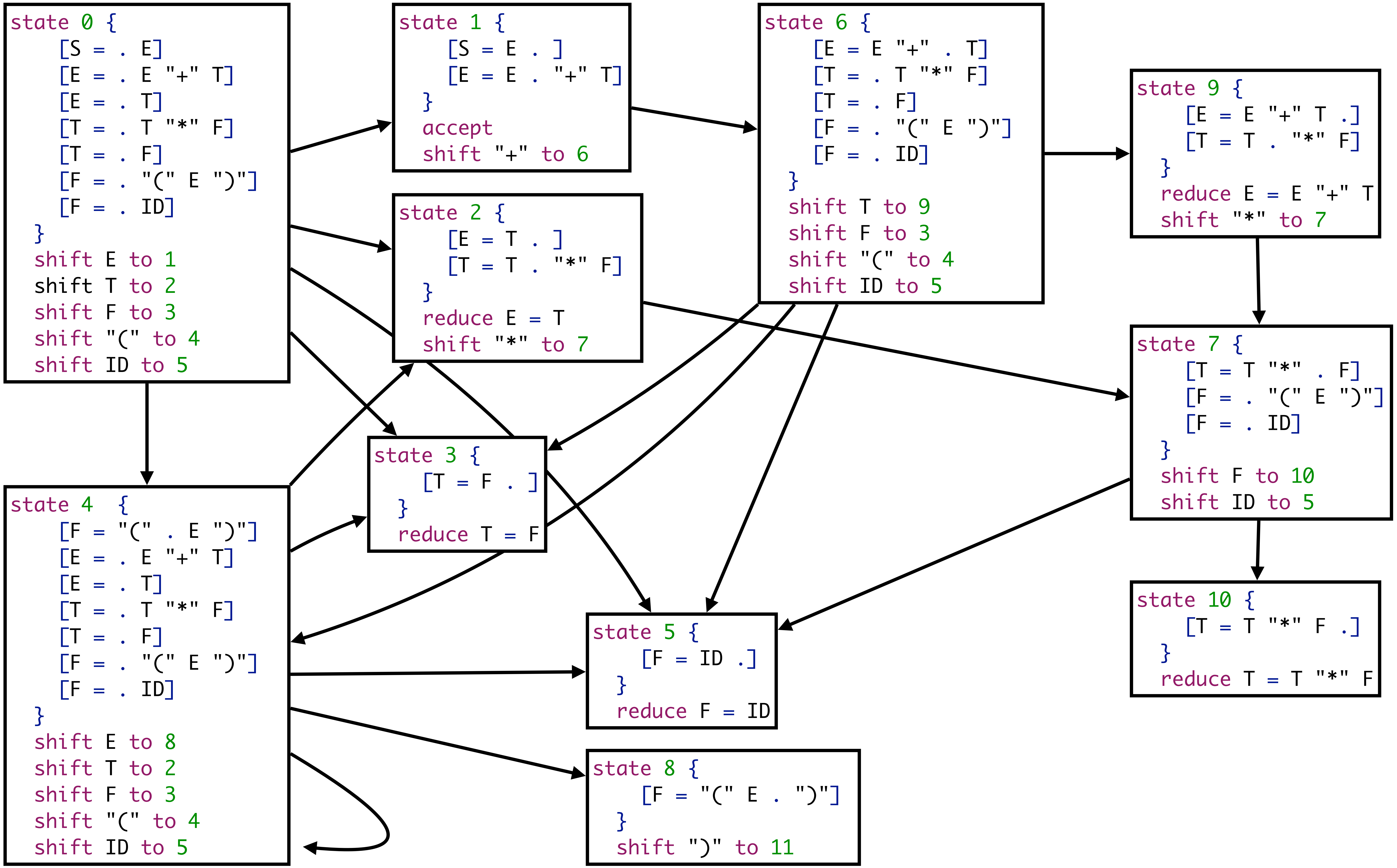
```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift ID to 5
```

```
state 8 {
    [F = "(" E . ")"]
  }
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 8
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

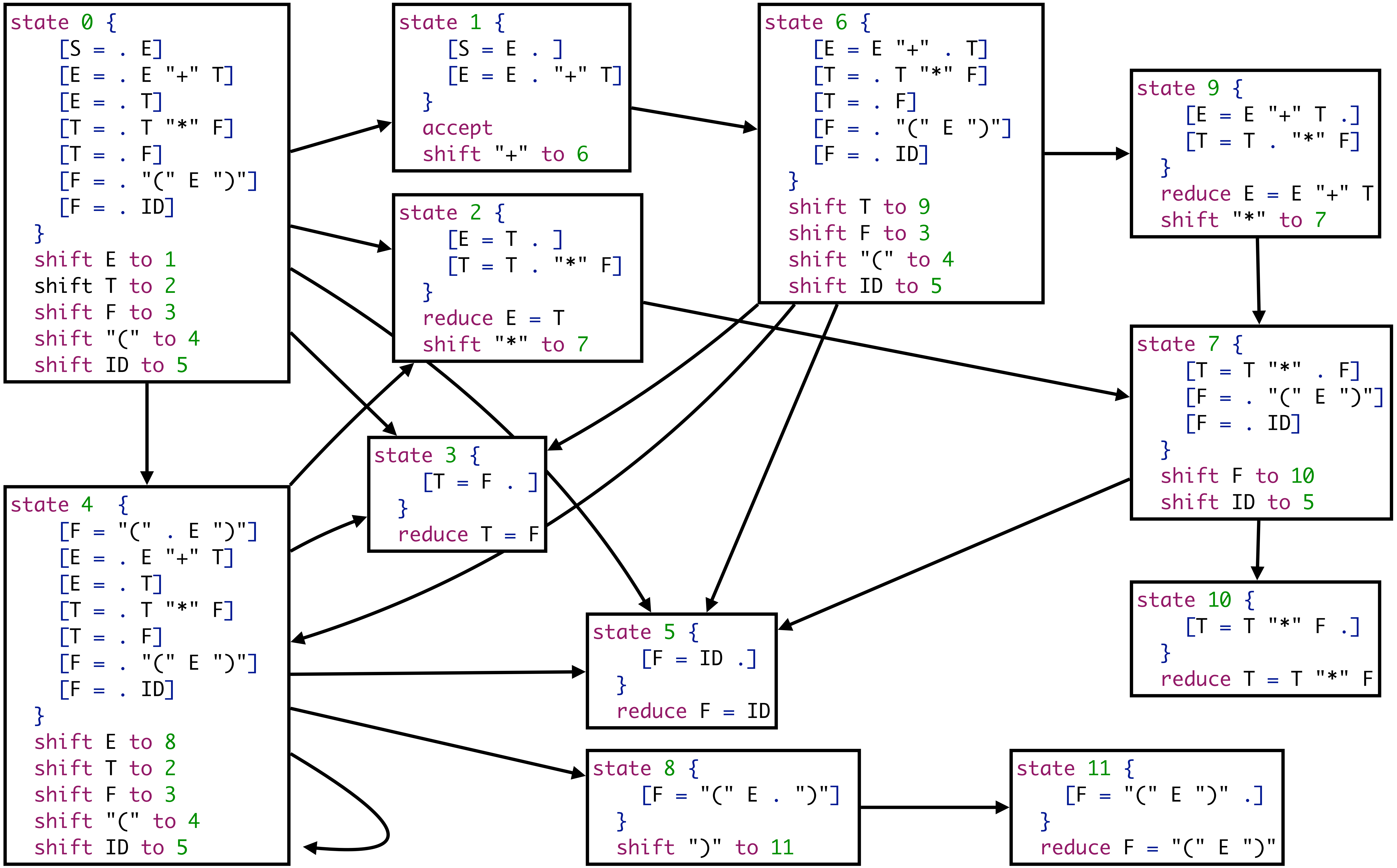
```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
```

```
state 8 {
    [F = "(" E . ")"]
  }
  shift ")" to 11
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 8
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

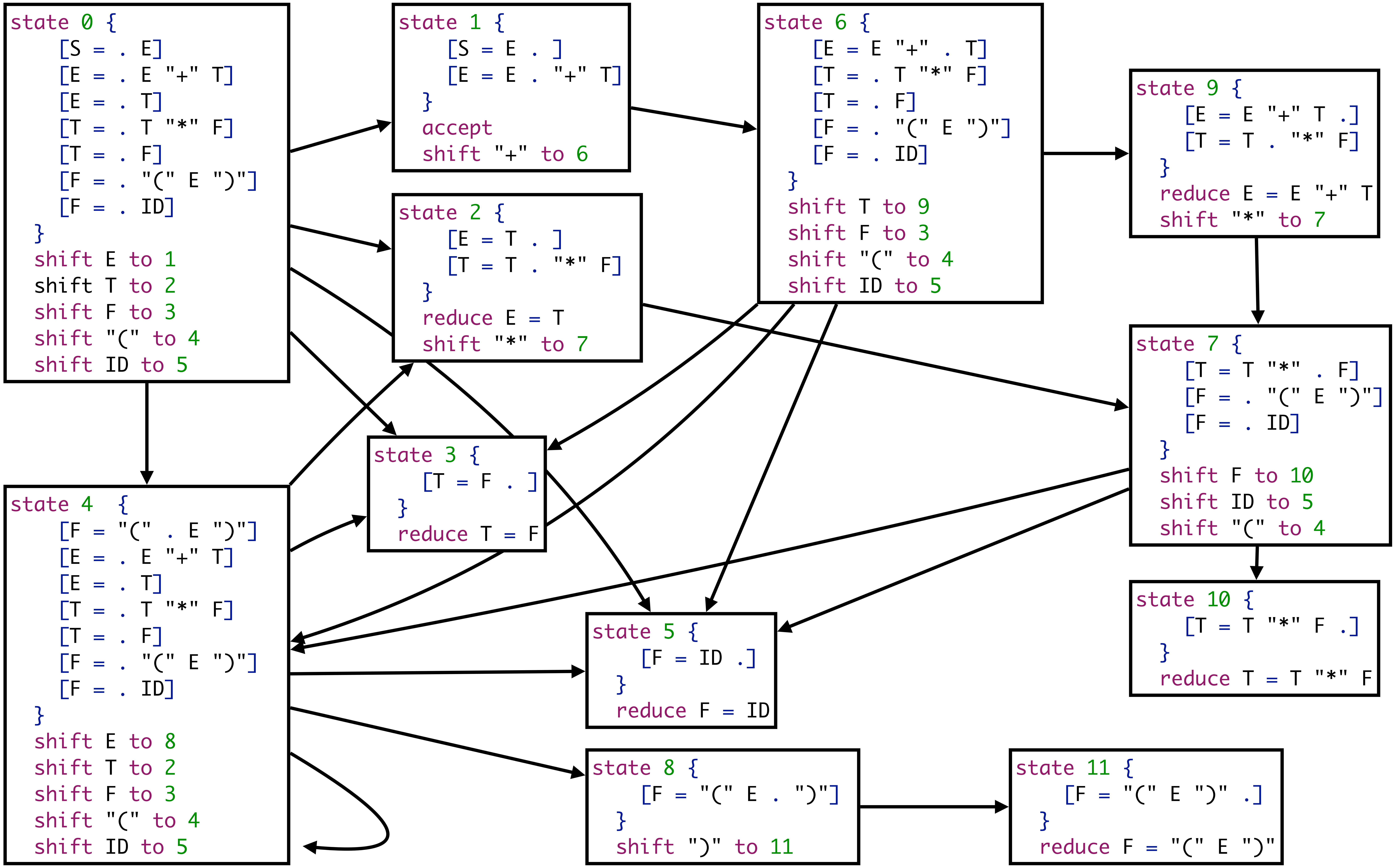
```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
```

```
state 8 {
    [F = "(" E . ")"]
  }
  shift ")" to 11
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

```
state 11 {
    [F = "(" E ")" .]
  }
  reduce F = "(" E ")"
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 8
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
  shift "(" to 4
```

```
state 8 {
    [F = "(" E . ")"]
  }
  shift ")" to 11
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

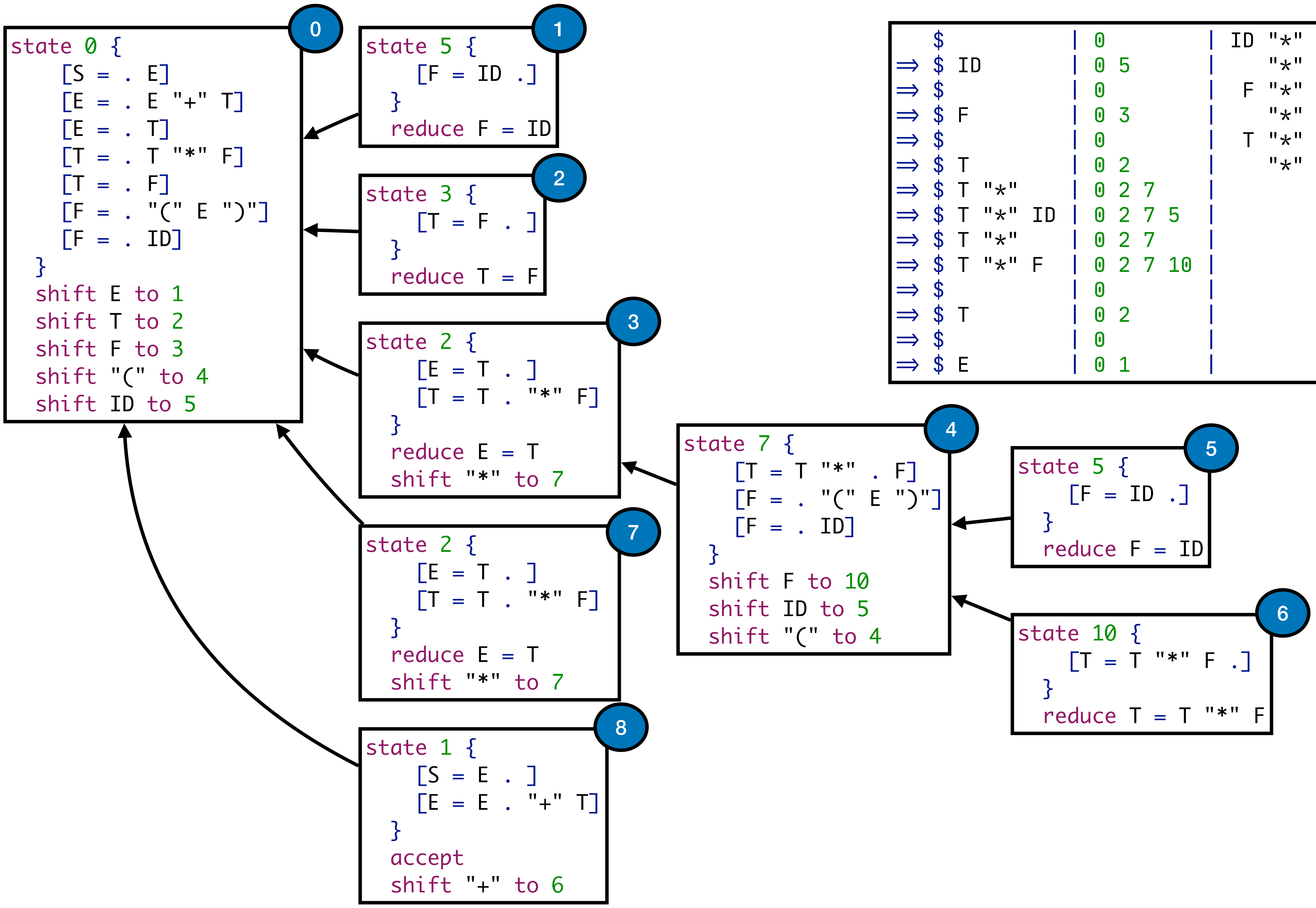
```
state 11 {
    [F = "(" E ")" .]
  }
  reduce F = "(" E ")"
```

SLR Parse

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
   $            | 0         | ID "*" ID $ shift ID to 5
⇒ $ ID         | 0 5       |    "*" ID $ reduce F = ID
⇒ $            | 0         |  F "*" ID $ shift F to 3
⇒ $ F          | 0 3       |    "*" ID $ reduce T = F
⇒ $            | 0         |  T "*" ID $ shift T to 2
⇒ $ T          | 0 2       |    "*" ID $ shift "*" to 7
⇒ $ T "*"      | 0 2 7     |        ID $ shift ID to 5
⇒ $ T "*" ID   | 0 2 7 5   |           $ reduce F = ID
⇒ $ T "*"      | 0 2 7     |         F $ shift F to 10
⇒ $ T "*" F    | 0 2 7 10  |           $ reduce T = T "*" F
⇒ $            | 0         |         T $ shift T to 2
⇒ $ T          | 0 2       |           $ reduce E = T
⇒ $            | 0         |         E $ shift E to 1
⇒ $ E          | 0 1       |           $ accept
```

```
    $           | 0          | ID "*" ID $ shift ID to 5
⇒ $ ID          | 0 5        |    "*" ID $ reduce F = ID
⇒ $             | 0          |  F "*" ID $ shift F to 3
⇒ $ F           | 0 3        |    "*" ID $ reduce T = F
⇒ $             | 0          |  T "*" ID $ shift T to 2
⇒ $ T           | 0 2        |    "*" ID $ shift "*" to 7
⇒ $ T "*"       | 0 2 7      |        ID $ shift ID to 5
⇒ $ T "*" ID    | 0 2 7 5    |           $ reduce F = ID
⇒ $ T "*"       | 0 2 7      |         F $ shift F to 10
⇒ $ T "*" F     | 0 2 7 10   |           $ reduce T = T "*" F
⇒ $             | 0          |         T $ shift T to 2
⇒ $ T           | 0 2        |           $ reduce E = T
⇒ $             | 0          |         E $ shift E to 1
⇒ $ E           | 0 1        |           $ accept
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

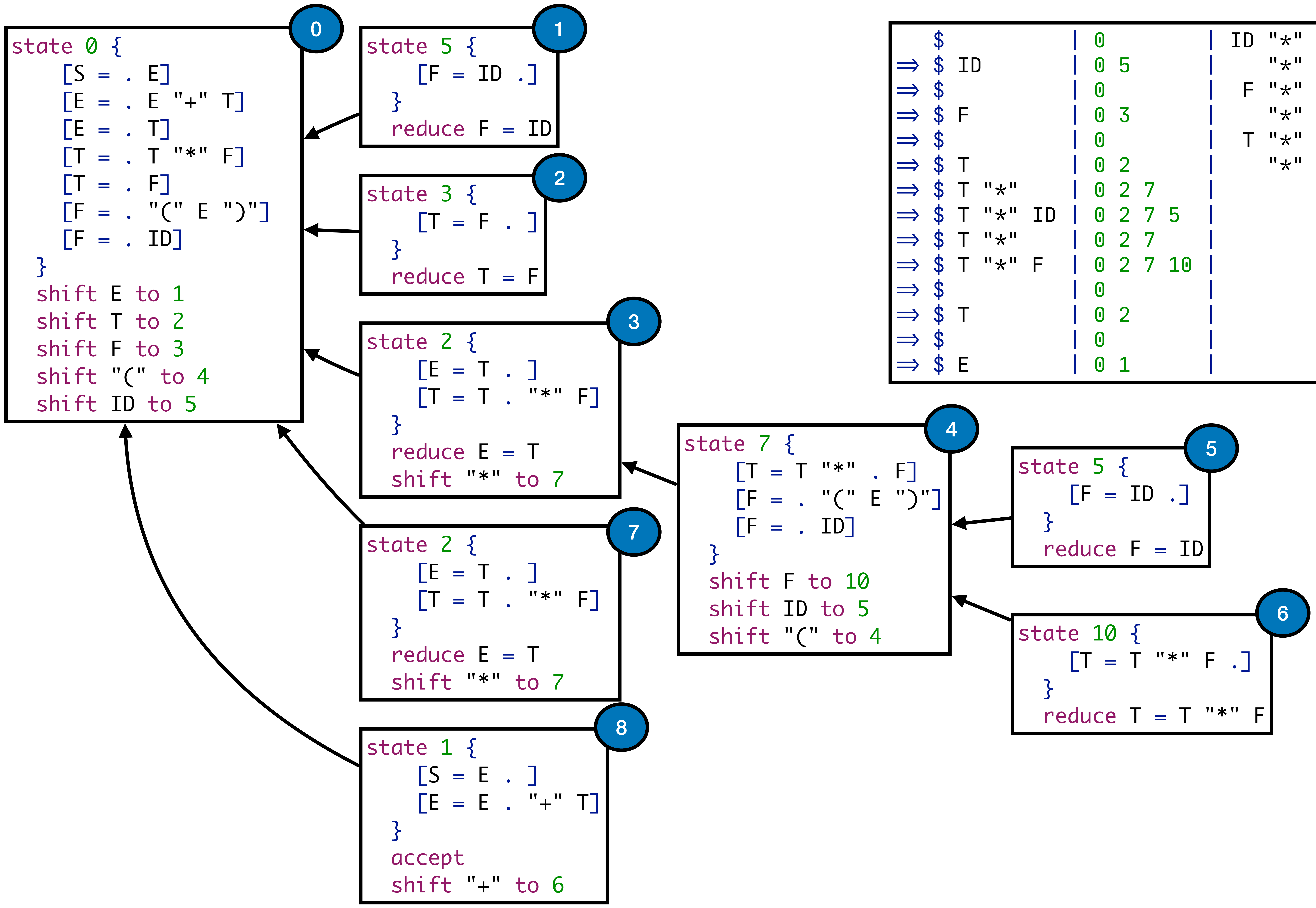
```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
  shift "(" to 4
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```



```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
  shift "(" to 4
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
  $            | 0        | ID "*" ID $ shift ID to 5
⇒ $ ID         | 0 5      |    "*" ID $ reduce F = ID
⇒ $            | 0        |  F "*" ID $ shift F to 3
⇒ $ F          | 0 3      |    "*" ID $ reduce T = F
⇒ $            | 0        |  T "*" ID $ shift T to 2
⇒ $ T          | 0 2      |    "*" ID $ shift "*" to 7
⇒ $ T "*"      | 0 2 7    |        ID $ shift ID to 5
⇒ $ T "*" ID   | 0 2 7 5  |           $ reduce F = ID
⇒ $ T "*"      | 0 2 7    |         F $ shift F to 10
⇒ $ T "*" F    | 0 2 7 10 |           $ reduce T = T "*" F
⇒ $            | 0        |         T $ shift T to 2
⇒ $ T          | 0 2      |           $ reduce E = T
⇒ $            | 0        |         E $ shift E to 1
⇒ $ E          | 0 1      |           $ accept
```

Parsing: ID "+" ID "*" ID .

E
"+"
ID
F
T
T
"*"
ID
F
E
1
2
3
4
5
6
7
8
9
10

```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
  shift "(" to 4
```

```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

Parsing: ID "+" ID "*" ID .

```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

1
```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

2
```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 9
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

3
```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

4
```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

5
```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

6
```
state 7 {
    [T = T "*" . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift F to 10
  shift ID to 5
  shift "(" to 4
```

7
```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

8
```
state 10 {
    [T = T "*" F .]
  }
  reduce T = T "*" F
```

9
```
state 9 {
    [E = E "+" T .]
    [T = T . "*" F]
  }
  reduce E = E "+" T
  shift "*" to 7
```

10
```
state 1 {
    [S = E . ]
    [E = E . "+" T]
  }
  accept
  shift "+" to 6
```

Solving Shift/Reduce Conflicts

SLR Parse

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
   $            | 0          | ID "*" ID $ shift ID to 5
⇒  $ ID         | 0 5        |    "*" ID $ reduce F = ID
⇒  $            | 0          |  F "*" ID $ shift F to 3
⇒  $ F          | 0 3        |    "*" ID $ reduce T = F
⇒  $            | 0          |  T "*" ID $ shift T to 2
⇒  $ T          | 0 2        |    "*" ID $ shift "*" to 7
⇒  $ T "*"      | 0 2 7      |        ID $ shift ID to 5
⇒  $ T "*" ID   | 0 2 7 5    |           $ reduce F = ID
⇒  $ T "*"      | 0 2 7      |         F $ shift F to 10
⇒  $ T "*" F    | 0 2 7 10   |           $ reduce T = T "*" F
⇒  $            | 0          |         T $ shift T to 2
⇒  $ T          | 0 2        |           $ reduce E = T
⇒  $            | 0          |         E $ shift E to 1
⇒  $ E          | 0 1        |           $ accept
```

Why did we choose shift?

SLR Parse

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
   $           | 0          | ID "*" ID $ shift ID to 5
⇒ $ ID         | 0 5        |    "*" ID $ reduce F = ID
⇒ $            | 0          |  F "*" ID $ shift F to 3
⇒ $ F          | 0 3        |    "*" ID $ reduce T = F
⇒ $            | 0          |  T "*" ID $ shift T to 2
⇒ $ T          | 0 2        |    "*" ID $ reduce E = T
⇒ $ E          | 0          |    "*" ID $ shift E to 1
⇒ $ E          | 0 1        |    "*" ID $ error
```

Reduce action is also possible, but leads to error

How can we avoid that?

SLR Parse

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
   $            | 0          | ID "*" ID $ shift ID to 5
⇒ $ ID          | 0 5        |    "*" ID $ reduce F = ID
⇒ $             | 0          |  F "*" ID $ shift F to 3
⇒ $ F           | 0 3        |    "*" ID $ reduce T = F
⇒ $             | 0          |  T "*" ID $ shift T to 2
⇒ $ T           | 0 2        |    "*" ID $ reduce E = T
⇒ $ E           | 0 2        |    "*" ID $ shift E to 1
⇒ $ E           | 0 1        |    "*" ID $ error
```

E can not be followed by a "*"!

Rule: only reduce with [A = Bs] if next token can follow A

First and Follow

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
first sets
  S    : {"(", ID}
  E    : {"(", ID}
  T    : {"(", ID}
  F    : {"(", ID}
  ID   : {ID}
  "+"  : {"+"}
  "*"  : {"*"}
  "("  : {"("}
  ")"  : {")"}
```

```
follow sets
  S  : {$}
  E  : {$,"+",")"}
  T  : {$,"+",")","*"}
  F  : {$,"+",")","*"}
  ID : {$,"+",")","*"}
```

First: the tokens that a phrase for a non-terminal can start with

Follow: the tokens that can follow a non-terminal

First

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
first sets
  S   : {"(", ID}
  E   : {"(", ID}
  T   : {"(", ID}
  F   : {"(", ID}
  ID  : {ID}
  "+" : {"+"}
  "*" : {"*"}
  "(" : {"("}
  ")" : {")"}
```

First: the tokens that a phrase for a symbol can start with

If A = Bs is a production,
then First[A] includes First[Bs]

A terminal starts with itself

First (Attempt 1)

```
grammar
 productions
   S = E
   E = E "+" T
   E = T
   T = T "*" F
   T = F
   F = "(" E ")"
   F = ID
```

```
first sets
  S   : {"(", ID}
  E   : {"(", ID}
  T   : {"(", ID}
  F   : {"(", ID}
  ID  : {ID}
  "+" : {"+"}
  "*" : {"*"}
  "(" : {"("}
  ")" : {")"}
```

```
var First : Map<Symbol, Set<Symbol>>

FIRST(G) {
  First := {};
  repeat {
    First' := First;
    for(each [A = As] in G) {
      First[A] := First[A] + Firsts(As);
    }
  } until First = First';
}

Set<Symbol> Firsts(As) {
  As match {
    [] ⇒ {};
    [A Bs] ⇒
      if(Terminal(A))
        {A}
      else
        First[A];
  };
}
```

First (Attempt 1)

```
grammar A
  start S
  non-terminals Decl Mod Args
  terminals ID "(" ")" "static" ""
  productions
    S    = Decl
    Decl = Mod ID "(" Args ")"
    Mod  = "static"
    Mod  =
    Args = Args ID
    Args =
```

```
first sets of A
  S        : {"static"}
  Decl     : {"static"}
  Mod      : {"static"}
  Args     : {}
  ID       : { ID }
  "("      : {"("}
  ")"      : {")"}
  "static" : {"static"}
```

What is wrong?

```
var First : Map<Symbol, Set<Symbol>>

FIRST(G) {
  First := {};
  repeat {
    First' := First;
    for(each [A = As] in G) {
      First[A] := First[A] + Firsts(As);
    }
  } until First = First';
}

Set<Symbol> Firsts(As) {
  As match {
    [] ⇒ {};
    [A Bs] ⇒
      if(Terminal(A))
        {A}
      else
        First[A];
  };
}
```

Is Non-Terminal Nullable?

```
grammar A
  start S
  non-terminals Decl Mod Args
  terminals ID "(" ")" "static" ""
  productions
    S    = Decl
    Decl = Mod ID "(" Args ")"
    Mod  = "static"
    Mod  =
    Args = Args ID
    Args =
```

```
nullable in A
  S        : false
  Decl     : false
  Mod      : true
  Args     : true
  ID       : false
  "("      : false
  ")"      : false
  "static" : false
```

```
var Nullable : Map<Symbol, Bool>

NULLABLE(G) {
  for(each A in G) {
    Nullable[A] := False;
  }
  repeat {
    Nullable' := Nullable;
    for(each [A = As] in G) {
      Nullable[A] := Nullable[A] || Nullables(As);
    }
  } until Nullable = Nullable';
}

Bool Nullables(As) {
  As match {
    [] ⇒ True;
    [B Bs] ⇒ not Terminal(B)
           and (Nullable[B] and Nullables(Bs));
  };
}
```

What are the First Terminals of a Non-Terminal?

```
grammar A
  start S
  non-terminals Decl Mod Args
  terminals ID "(" ")" "static" ""
  productions
    S    = Decl
    Decl = Mod ID "(" Args ")"
    Mod  = "static"
    Mod  =
    Args = Args ID
    Args =
```

```
first sets of A
  S        : {"static",ID}
  Decl     : {"static",ID}
  Mod      : {"static"}
  Args     : {ID}
  ID       : {ID}
  "("      : {"("}
  ")"      : {")"}
  "static" : {"static"}
```

```
var First : Map<Symbol, Set<Symbol>>

FIRST(G) {
  First := {};
  repeat {
    First' := First;
    for(each [A = As] in G) {
      First[A] := First[A] + Firsts(As);
    }
  } until First = First';
}
Set<Symbol> Firsts(As) {
  As match {
    [] ⇒ {};
    [A Bs] ⇒
      if(Terminal(A))
        {A}
      else
        First[A] + if(Nullable[A]) Firsts(Bs) else {};
  };
}
```

What Terminal can Follow a Non-Terminal Start with?

```
grammar
 productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

```
follow sets of A
  S  : {$}
  E  : {$,"+",")"}
  T  : {$,"+",")","*"}
  F  : {$,"+",")","*"}
  ID : {$,"+",")","*"}
```

```
var Follow : Map<Symbol, Set<Symbol>>

FOLLOW(G) {
  Follow := {};
  repeat {
    Follow' := Follow;
    for(each [A = As B Cs] in G) {
      Follow[B] := Follow[B]
                 + Firsts(Cs)
                 + if(Nullables(Cs)) Follow[A]
                   else {};
    }
  } until Follow = Follow';
}
```

What Terminal can Follow a Non-Terminal Start with?

```
grammar A
  start S
  non-terminals Decl Mod Args
  terminals ID "(" ")" "static" ""
  productions
    S    = Decl
    Decl = Mod ID "(" Args ")"
    Mod  = "static"
    Mod  =
    Args = Args ID
    Args =
```

```
follow sets of A
  S    : {$}
  Decl : {$}
  Mod  : {ID}
  Args : {")" ID}
  ID   : {"(",")"}
```

```
var Follow : Map<Symbol, Set<Symbol>>

FOLLOW(G) {
  Follow := {};
  repeat {
    Follow' := Follow;
    for(each [A = As B Cs] in G) {
      Follow[B] := Follow[B]
                 + Firsts(Cs)
                 + if(Nullables(Cs)) Follow[A]
                   else {};
    }
  } until Follow = Follow';
}
```

What Terminal can Follow a Non-Terminal Start with?

```
grammar G
  start S
  non-terminals Decl Mod Args
  terminals ID "(" ")" "static" ""
  productions
    F = … A “a”
    A = … B C D
    C = “c”
    C =
    D = “d”
    D =
```

```
follow sets of G
  A : {“a”}
  B : {“c”, “d”, “a”}
```

```
var Follow : Map<Symbol, Set<Symbol>>

FOLLOW(G) {
  Follow := {};
  repeat {
    Follow' := Follow;
    for(each [A = As B Cs] in G) {
      Follow[B] := Follow[B]
                 + Firsts(Cs)
                 + if(Nullables(Cs)) Follow[A]
                   else {};
    }
  } until Follow = Follow';
}
```

Parsing: Summary

Parsing

Context-free grammars

- Productions define how to generate sentences of language
- Derivation: generate sentence from (start) symbol
- Reduction: reduce sentence to (start) symbol

Parse tree

- Represents structure of derivation
- Abstracts from derivation order

Parser

- Algorithm to reconstruct derivation

First/Follow

- Selecting between actions in LR parse table

More Topics in Syntax and Parsing

Other algorithms

- Top-down: LL(k) table
- Generalized parsing: Earley, Generalized-LR
- Scannerless parsing: characters as tokens

Disambiguation

- Semantics of declarative disambiguation
- Deep priority conflicts