Lecture 4: Parsing

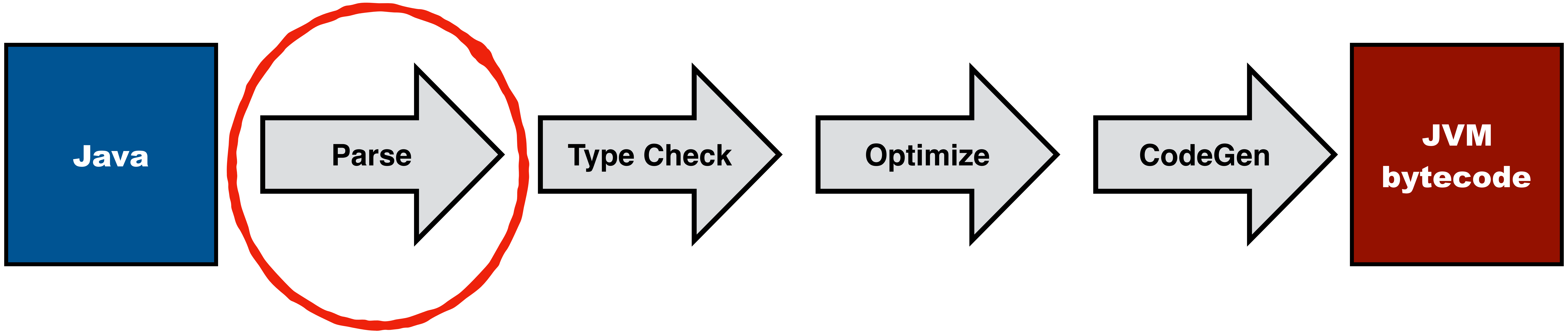
CS4200 Compiler Construction

Eelco Visser

TU Delft

September 2018

This Lecture



Turning syntax definitions into parsers

Reading Material

The perspective of this lecture on declarative syntax definition is Elained 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

$X_1 \ldots X_n$ -> N {cons("C")}

instead of

N.C = $X_1 \ldots X_n$

https://doi.org/10.1145/1932682.1869535

http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2010-019.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-

Classical compiler textbook

Chapter 4: Syntax Analysis

Read 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

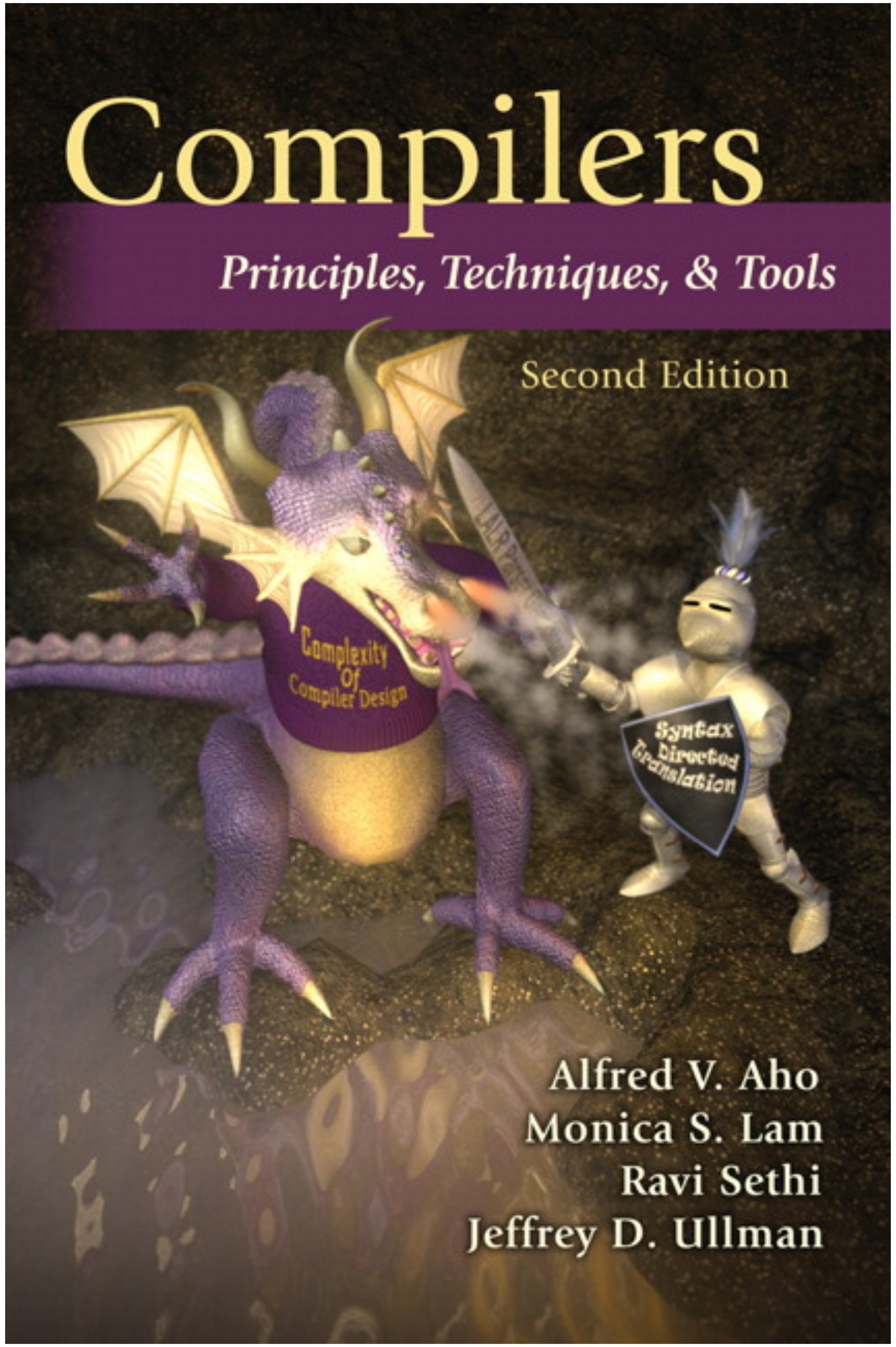
Alfred V. Aho, Columbia University

Monica S. Lam, Stanford University

Ravi Sethi, Avaya Labs

Jeffrey D. Ullman, Stanford University

2007 | Pearson



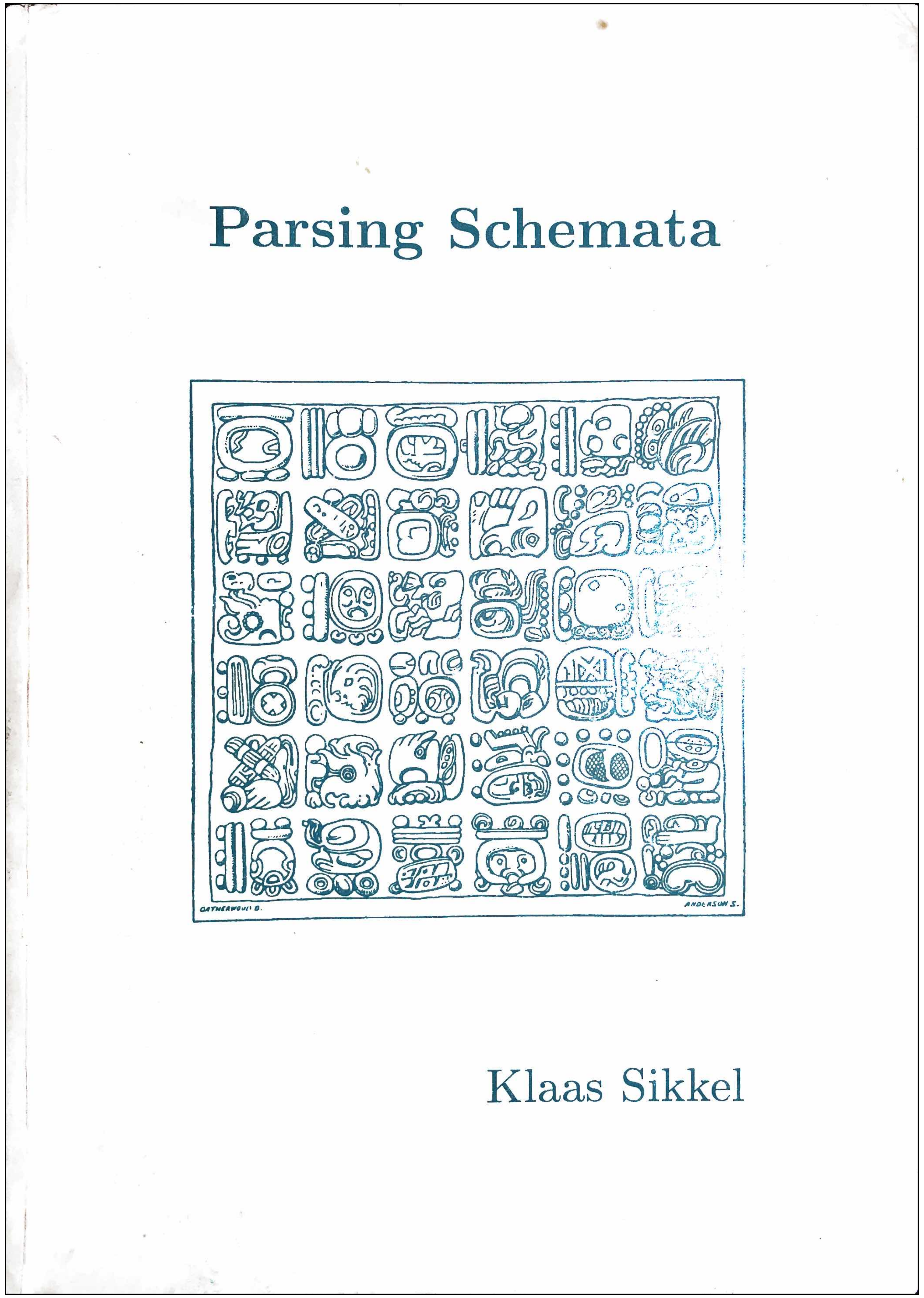
This PhD thesis presents a uniform framework for describing a wide range of parsing algorithms.

“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.

For the interested

https://ivi.fnwi.uva.nl/tcs/pub/reports/1995/P9507.ps.Z

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

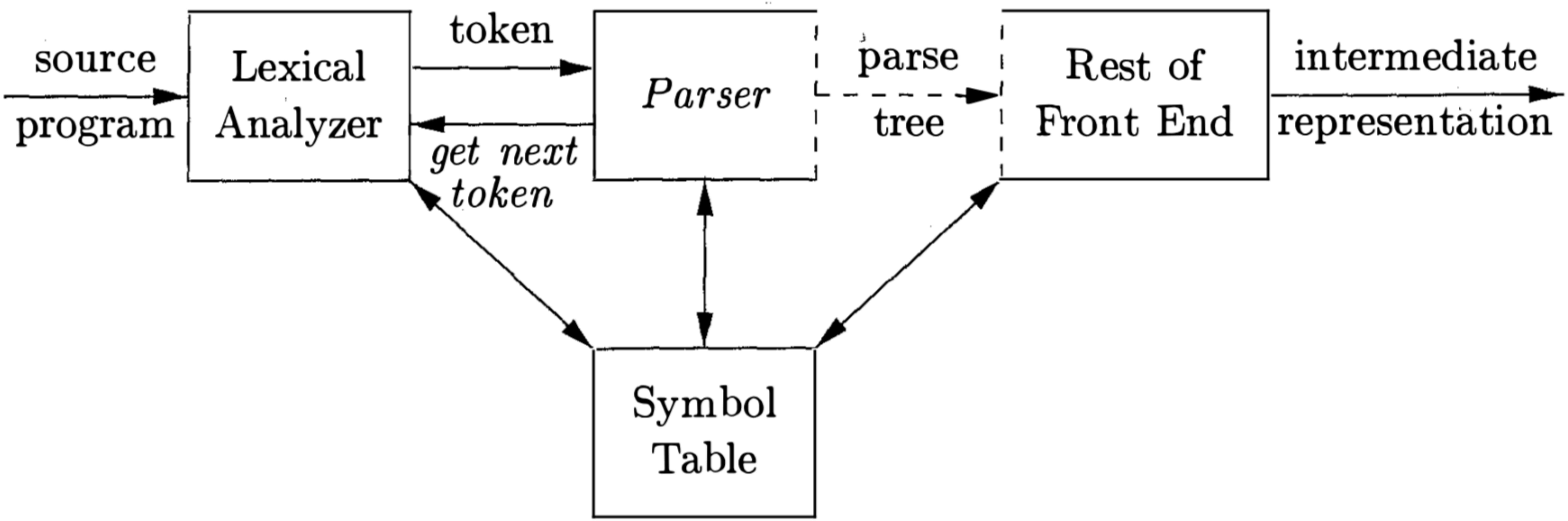
$$E \to E + T; E \to T; T \to T * a; T \to a$$

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].

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
a, b, c: strings of non-terminals and terminals
    (alpha, beta, gamma in math)
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

Derivations

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

```
// derivation step: replace symbol by rhs of 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 ")"
```

Meta: Syntax of Derivations

```
context-free syntax // derivations

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

  Step.Step   = [=> [Symbol*]]
  Step.Steps  = [=>* [Symbol*]]
  Step.Steps1 = [=>+ [Symbol*]]
```

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

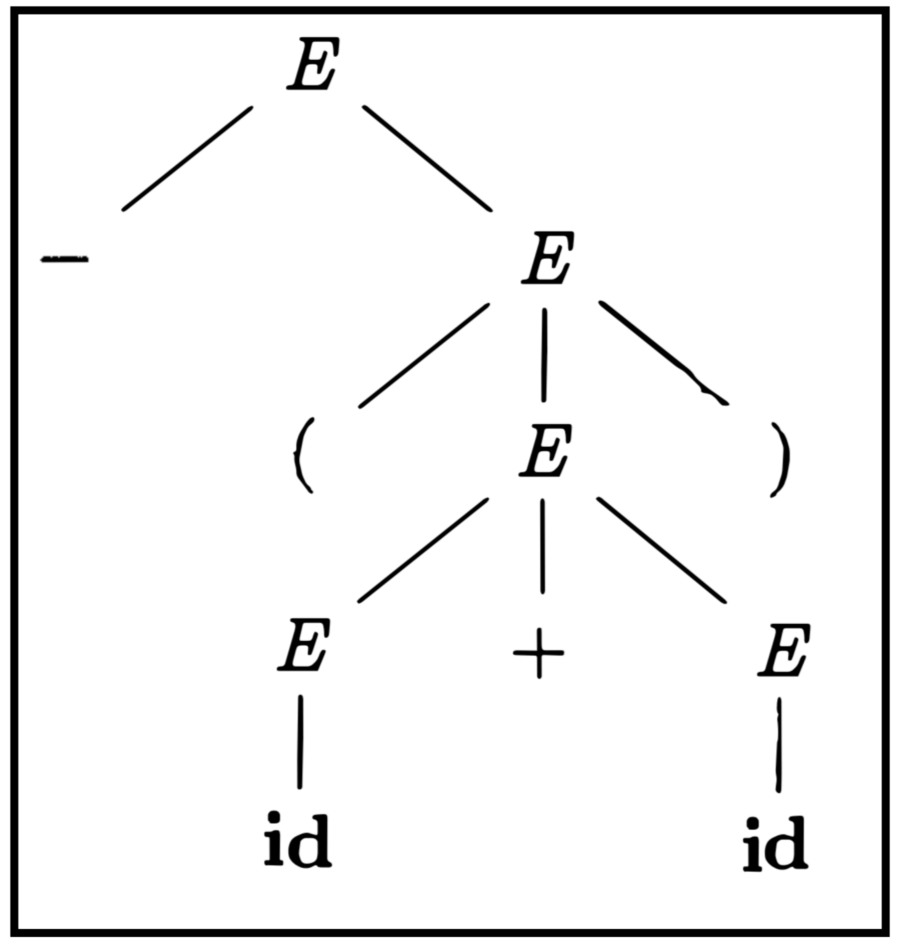
Meta: Tree Derivations

```
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


```
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]] ")"]]
```

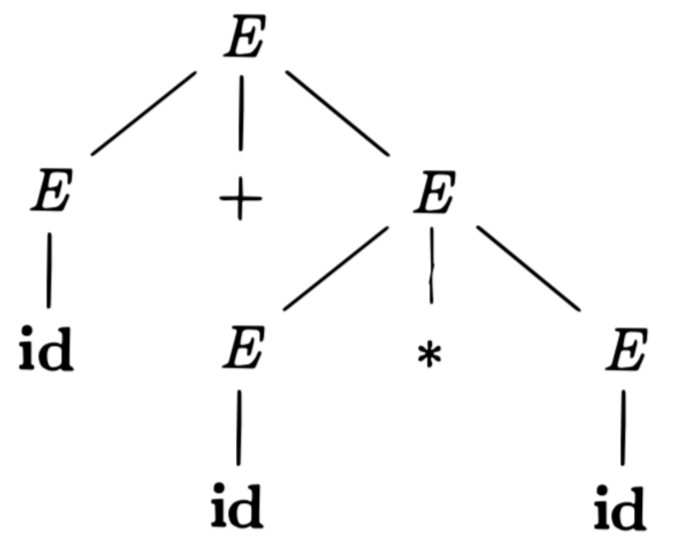
Ambiguity: Deriving Multiple Parse Trees

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

```
derivation
  E =>* ID "+" ID "*" ID
```

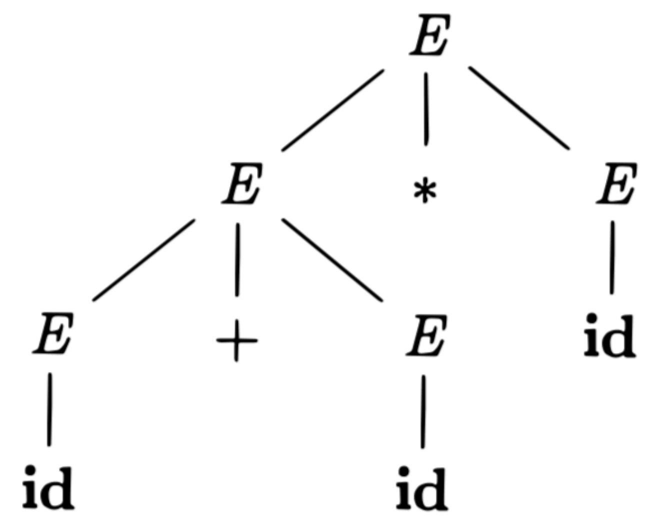
```
derivation
  E
  => E "+" E
  => ID "+" E
  => ID "+" E "*" E
  => ID "+" ID "*" E
  => ID "+" ID "*" ID
```

```
tree derivation
  E =>* E[E[ID] "+" E[E[ID] "*" E[ID]]]
```




```
derivation
  E
  => E "*" E
  => E "+" E "*" E
  => ID "+" E "*" E
  => ID "+" ID "*" E
  => ID "+" ID "*" ID
```

```
tree derivation
  E =>* E[E[E[ID] "+" E[ID]] "*" E[ID]]
```



Ambiguous grammar: produces >1 parse tree for a sentence

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>>
```

Ambiguity: Deriving Abstract Syntax Terms

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

```
derivation
  E =>* ID "+" ID "*" ID
```

```
derivation
  E
  => E "+" E
  => ID "+" E
  => ID "+" E "*" E
  => ID "+" ID "*" E
  => ID "+" ID "*" ID
```

```
term derivation
  E
  => A(E, E)
  => A(V(ID), E)
  => A(V(ID), T(E, E))
  => A(V(ID), T(V(ID), E))
  => A(V(ID), T(V(ID), V(ID)))
```

```
derivation
  E
  => E "*" E
  => E "+" E "*" E
  => ID "+" E "*" E
  => ID "+" ID "*" E
  => ID "+" ID "*" ID
```

```
term derivation
  E
  => T(E, E)
  => T(A(E, E), E)
  => T(A(V(ID), E), E)
  => T(A(V(ID), V(ID)), E)
  => T(A(V(ID), V(ID)), V(ID))
```

Grammar Transformations

Grammar Transformations

Why?

- Disambiguation
- For use by a particular parsing algorithm

Transformations

- Eliminating ambiguities
- Eliminating left recursion
- Left factoring

Properties

- Does transformation preserve the language (set of strings, trees)?
- Does transformation preserve the structure of trees?

Ambiguous Expression Grammar

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

```
derivation
  E =>* ID "*" ID "+" ID
```

```
term derivation
  E
  => A(E, E)
  => A(T(E, E), E)
  => A(T(E, E), E)
  => A(T(V(ID), E), E)
  => A(T(V(ID), V(ID)), E)
  => A(T(V(ID), V(ID)), V(ID))
```

```
term derivation
  E
  => T(E, E)
  => T(E, E)
  => T(V(ID), E)
  => T(V(ID), A(E, E))
  => T(V(ID), A(V(ID), E))
  => T(V(ID), A(V(ID), V(ID)))
```

Associativity and Priority Filter Ambiguities

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

```
grammar
  productions
    E.A = E "+" E {left}
    E.T = E "*" E {left}
    E.M = "-" E
    E.B = "(" E ")"
    E.V = ID
  priorities
    E.M > E. T > E.A
```

```
derivation
  E =>* ID "*" ID "+" ID
```

```
term derivation
  E
  => A(E, E)
  => A(T(E, E), E)
  => A(T(E, E), E)
  => A(T(V(ID), E), E)
  => A(T(V(ID), V(ID)), E)
  => A(T(V(ID), V(ID)), V(ID))
```

```
term derivation
  E
  => T(E, E)
  => T(E, E)
  => T(V(ID), E)
  => T(V(ID), A(E, E))
  => T(V(ID), A(V(ID), E))
  => T(V(ID), A(V(ID), V(ID)))
```

Define Associativity and Priority by Transformation

```
grammar
  productions
    E.A = E "+" E {left}
    E.T = E "*" E {left}
    E.M = "-" E
    E.B = "(" E ")"
    E.V = ID
  priorities
    E.M > E. T > E.A
```

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

```
derivation
  E =>* ID "*" ID "+" ID
```

```
term derivation
  E
  => A(E, E)
  => A(T(E, E), E)
  => A(T(E, E), E)
  => A(T(V(ID), E), E)
  => A(T(V(ID), V(ID)), E)
  => A(T(V(ID), V(ID)), V(ID))
```

```
term derivation
  E
  => T(E, E)
  => T(E, E)
  => T(V(ID), E)
  => T(V(ID), A(E, E))
  => T(V(ID), A(V(ID), E))
  => T(V(ID), A(V(ID), V(ID)))
```

Define Associativity and Priority by Transformation

```
grammar
  productions
    E.A = E "+" E {left}
    E.T = E "*" E {left}
    E.M = "-" E
    E.B = "(" E ")"
    E.V = ID
  priorities
    E.M > E. T > E.A
```

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

Define new non-terminal for each priority level:
E, T, F

Add ‘injection’ productions to include priority level n+1 in n:
E = T
T = F

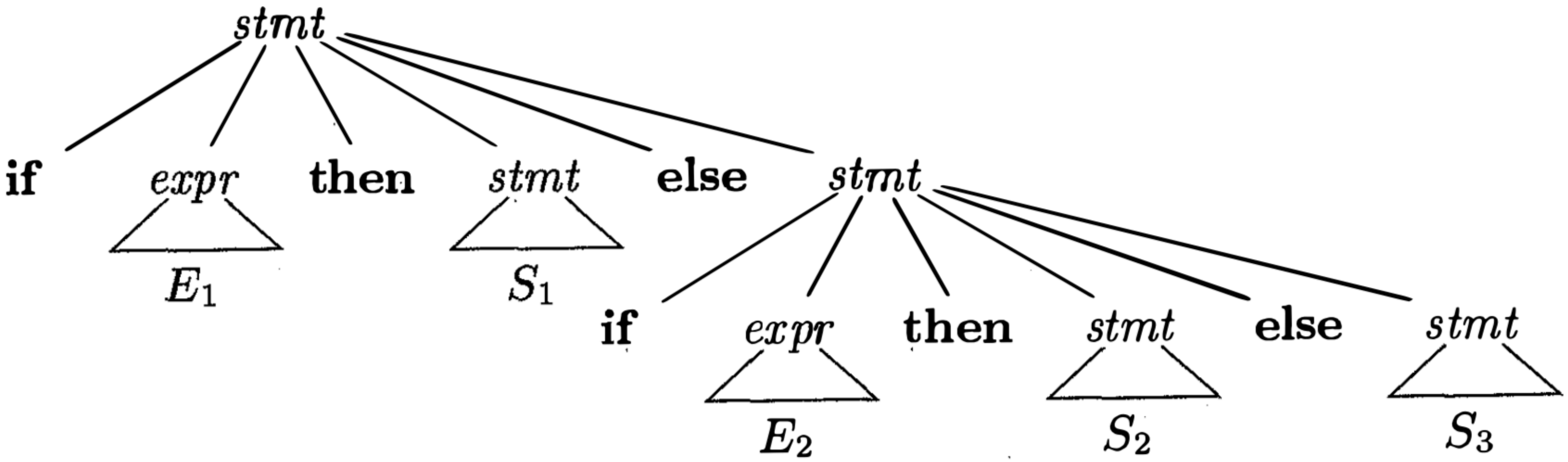
Change head of production to reflect priority level
T = T “*” F

Transform productions
Left: E = E “+” T
Right: E = T “+” E

Dangling Else Grammar

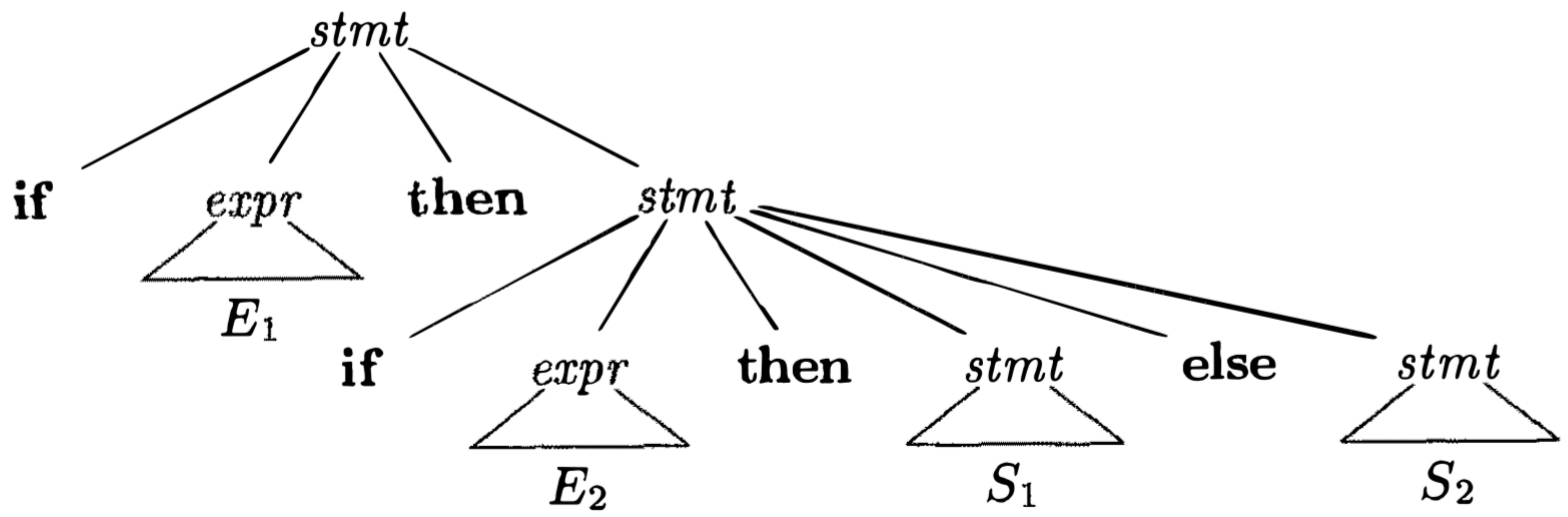
```
grammar
  sorts S E
  productions
    S.If  = if E then S
    S.IfE = if E then S else S
    S     = other
```

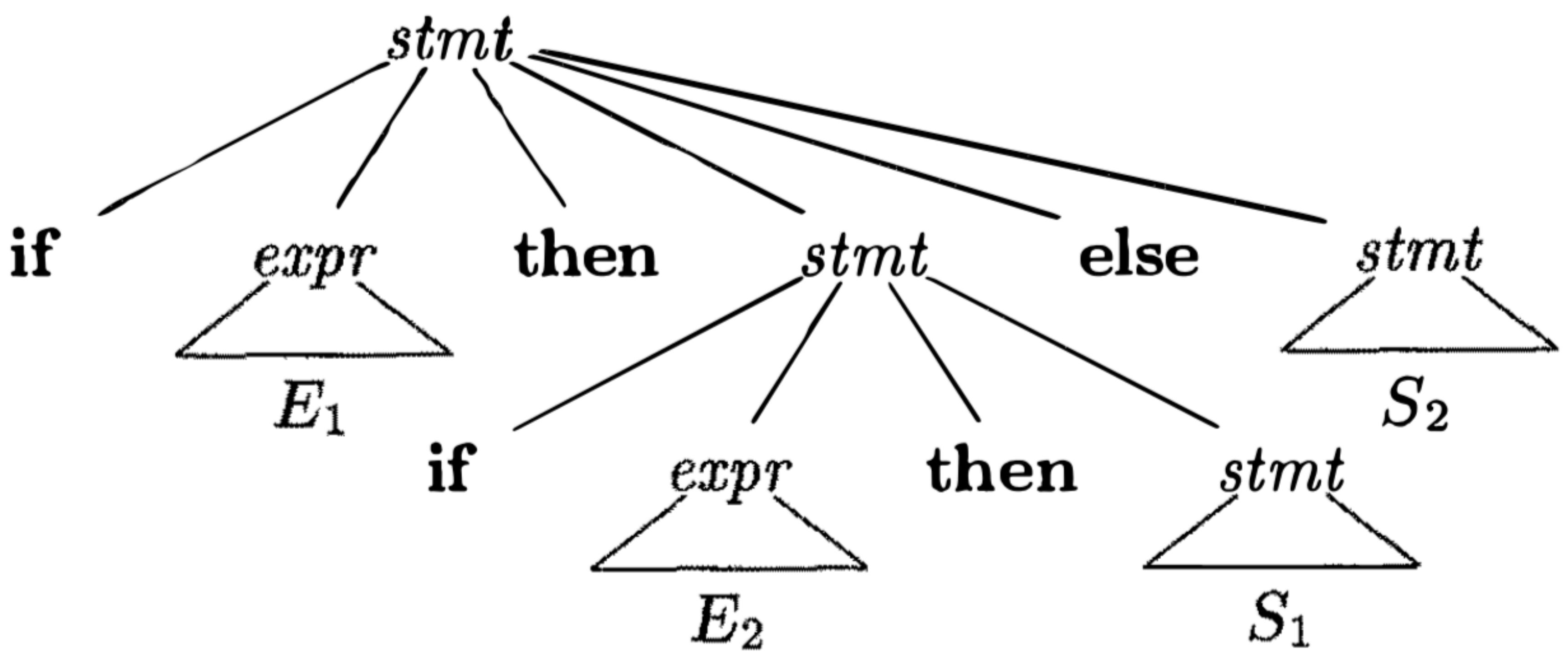
```
derivation
  S =>* if E1 then S1 else if E2 then S2 else S3

term derivation
  S =>* IfE(E1, S1, IfE(E2, S2, S3))

term derivation
  S
  => IfE(E1, S1, S)
  => IfE(E1, S1, IfE(E2, S2, S3))
```



Dangling Else Grammar is Ambiguous

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






```
derivation
  S =>* if E1 then if E2 then S1 else S2
```

```
term derivation
  S
  => If(E1, S)
  => If(E1, IfE(E2, S1, S2))

derivation
  S
  => if E1 then S
  => if E1 then if E2 then S1 else S2
```

```
term derivation
  S
  => IfE(E1, S, S2)
  => IfE(E1, If(E2, S1), S2)

derivation
  S
  => if E1 then S else S2
  => if E1 then if E2 then S1 else S2
```

Eliminating Dangling Else Ambiguity

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

```
grammar
  productions
    S  = MS
    S  = OS
    MS = if E then MS else MS
    MS = other
    OS = if E then S
    OS = if E then MS else OS
```

Generalization of this transformation: contextual grammars

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.

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

Declarative Disambiguation of Deep Priority Conflicts

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

Disambiguation of context-free grammars for operator expressions by means of priority and associativity declarations enables a direct correspondence between grammar and abstract syntax trees, more concise grammars, and a better expression of intent than encoding associativity and priority in the grammar. However, there is no standardized, declarative semantics of disambiguation with associativity and priority declarations. Indirect approaches to the semantics use a translation into another formalism (such as parse tables or tree automata), inhibiting generalization and/or understanding. The direct approach defines the semantics of priority and associativity declarations by means of subtree exclusion patterns. However, existing definitions of this direct approach are not safe and/or not complete.

In this paper, we provide the first *direct* semantics of disambiguation by means of associativity and priority declarations that is safe and complete, and not limited to one-level tree patterns. We define a semantics for *safe* disambiguation with operator priority and associativity in terms of one-level subtree exclusion patterns, i.e., one that only filters trees when the input is ambiguous. We extend the semantics with a formal definition of *deep priority conflicts* that are not covered by fixed-depth patterns. We show how this semantics can be used to resolve ambiguities such as dangling else and longest match. Furthermore, we extend the approach to productions with indirect recursion. We have implemented the semantics in a parser generator for SDF3 and we have evaluated the approach by applying it to the grammars of seven languages.

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

ACM Reference format:
Luis Eduardo S. Amorim, Timothée Haudebourg, Michael J. Steindorfer, and Eelco Visser. 2018. Declarative Disambiguation of Deep Priority Conflicts. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (February 2018), 47 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context-free grammars provide a concise, high-level, and well-understood formalism for the specification and documentation of the syntax of programming languages. However, grammars play a dual role in such descriptions. On the one hand, a grammar describes the *structure* of programs in a language, i.e. the set of *trees* that represent its well-formed programs. On the other hand, grammars also specify *parsers*, i.e. the mapping from sentences to trees.

These roles pose conflicting requirements on grammars. For the purpose of describing structure, the (abstract) syntax definitions in reference manuals and academic papers are often *ambiguous*, providing concise descriptions and a direct correspondence between abstract syntax trees and grammar rules. For the purpose of parsing, a grammar should *unambiguously* identify the structure of a program text.

Eliminating Left Recursion

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

```
grammar
  productions
    E  = T E'
    E' = "+" T E'
    E' =
    T  = F T'
    T' = "*" F T'
    T' =
    F  = "(" E ")"
    F  = ID
```

```
grammar
  productions
    A = A a
    A = b
```

```
grammar
  productions
    A  = b A'
    A' = a A'
    A' = // empty
```

```
// b followed by a list of as
```

Left Factoring

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

```
grammar
  sorts S E
  productions
    S.If       = if E then S S'
    S'.Else    = else S
    S'.NoElse  = // empty
    S          = other
```

```
grammar
  productions
    A = a b1
    A = a b2
    A = c
```

```
grammar
  productions
    A  = a A'
    A' = b1
    A' = b2
    A  = c
```

Properties of Grammar Transformations

Preservation

- Preserves set of sentences
- Preserves set of trees
- Preserves tree structure

Systematic

- Algorithmic
- Heuristic

Top-Down Parsing

Top-Down Parse

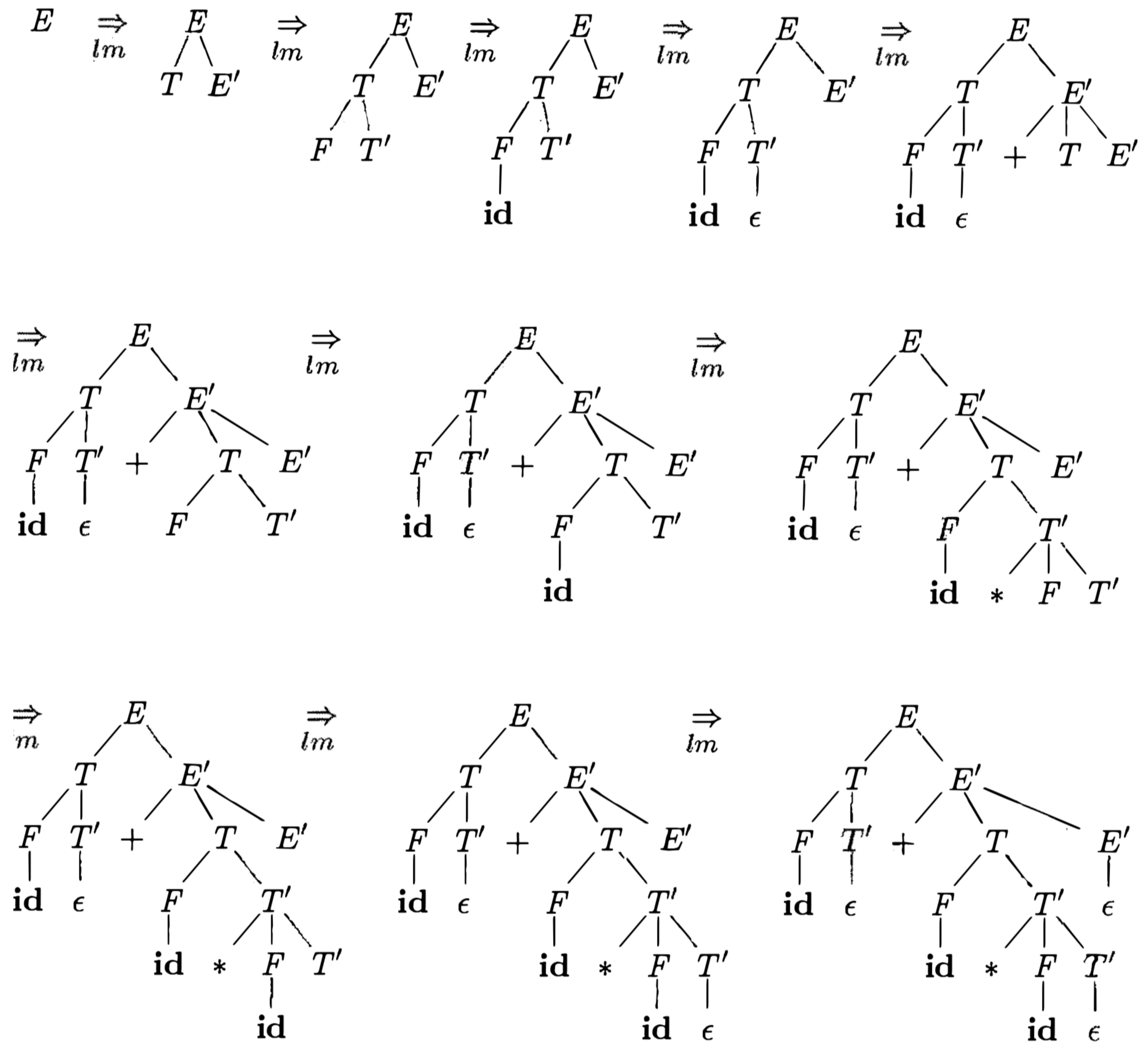
```
grammar
  sorts E T E' F T'
  productions
    E  = T E'
    E' = "+" T E'
    E' =
    T  = F T'
    T' = "*" F T'
    T' =
    F  = "(" E ")"
    F  = ID
```

```
derivation
  E =>* ID "+" ID "*" ID
```

```
tree derivation // top-down parse
  E
  => E[T E']
  => E[T[F T'] E']
  => E[T[F[ID] T'] E']
  => E[T[F[ID] T'[]] E']
  => E[T[F[ID] T'[]] E'["+" T E']]
  => E[T[F[ID] T'[]] E'["+" T[F T'] E']]
  => E[T[F[ID] T'[]] E'["+" T[F[ID] T'] E']]
  => E[T[F[ID] T'[]] E'["+" T[F[ID] T'["*" F T']] E']]
  => E[T[F[ID] T'[]] E'["+" T[F[ID] T'["*" F[ID] T']] E']]
  => E[T[F[ID] T'[]] E'["+" T[F[ID] T'["*" F[ID] T'[]]] E']]
  => E[T[F[ID] T'[]] E'["+" T[F[ID] T'["*" F[ID] T'[]]] E'[]]]
```

Top-Down Parse

```
grammar
  sorts E T E' F T'
  productions
    E  = T E'
    E' = "+" T E'
    E' =
    T  = F T'
    T' = "*" F T'
    T' =
    F  = "(" E ")"
    F  = ID
```



Non-Deterministic Recursive Descent Parsing

void $A()$ {
    Choose an A-production, $A \rightarrow X_1 X_2 \cdots X_k$;
    for ($i = 1$ to k) {
        if (X_i is a nonterminal)
            call procedure $X_i()$;
        else if (X_i equals the current input symbol a)
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}

LL Parsing

Use LL(1) grammar

- Not left recursive
- Left factored

Top-down back-track parsing

- Predict symbol
- If terminal: corresponds to next input symbol?
- Try productions for non-terminal in turn

Predictive parsing

- Predict symbol to parse
- Use lookahead to deterministically chose production for non-terminal

Variants

- Parser combinators, PEG, packrat, …

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 = b
```

```
reduction
  a b c <= a A c
```

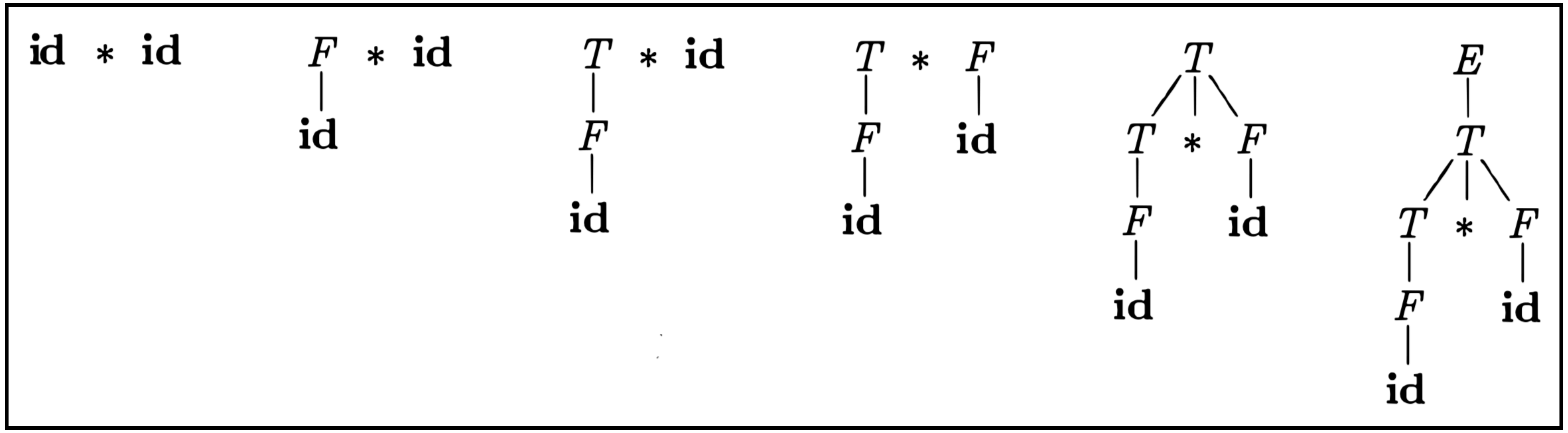
Reducing to Symbols

```
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
```

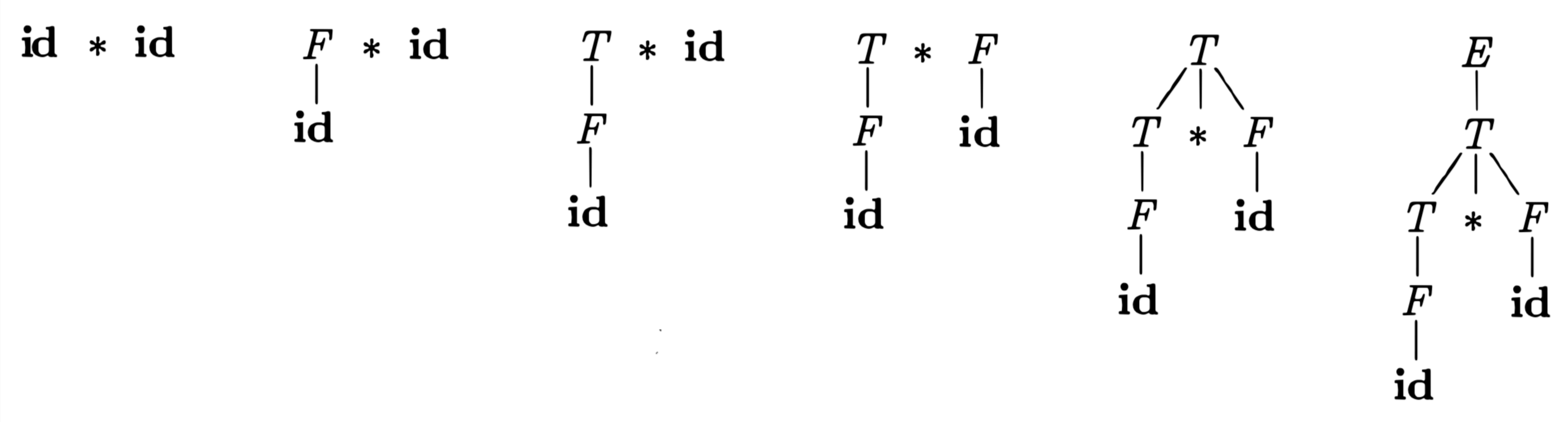



```
reduction
  ID "*" ID
  <= F "*" ID
  <= T "*" ID
  <= T "*" F
  <= T
  <= E
```

```
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)))
```

Handles

```
grammar
  productions
    A = b
```

```
derivation // right-most derivation
  S =>* a A w =>  a b w
```

```
reduction
  a b w <= a A w <=* S
```

```
// Handle

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

// S =>* a

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

// handle: the part of a right sentential form that if reduced
// would produced the previous right-sential 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
```

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 chose between possible shift and reduce actions?

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

Items

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

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

We expect to see an expression

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

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

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

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

Production

Items

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]
}
```

```
SetOfItems Closure(I) {
  J := I;
  repeat
    for(each [A = a . B b] in J)
      for(each [B = c] in G)
        if([B = . c] is not in J)
          J := Add([B = .c], J);
  until Not(Changed(J));
  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]
}
```

```
SetOfItems Goto(I, X) {
  J := {};
  for(each [A = a . X b] in I)
    J := Add([A = a X . b], J);
  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

```
Items(G) {
  C := {Closure({[Sp = . S]})};
  repeat
    for(each I in C)
      for(each X in G) {
        if(NotEmpty(Goto(I, X)) and Not(In(J, C)))
          C := Add(Goto(I, X), C);
      }
  until Not(Changed(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]
  }
```

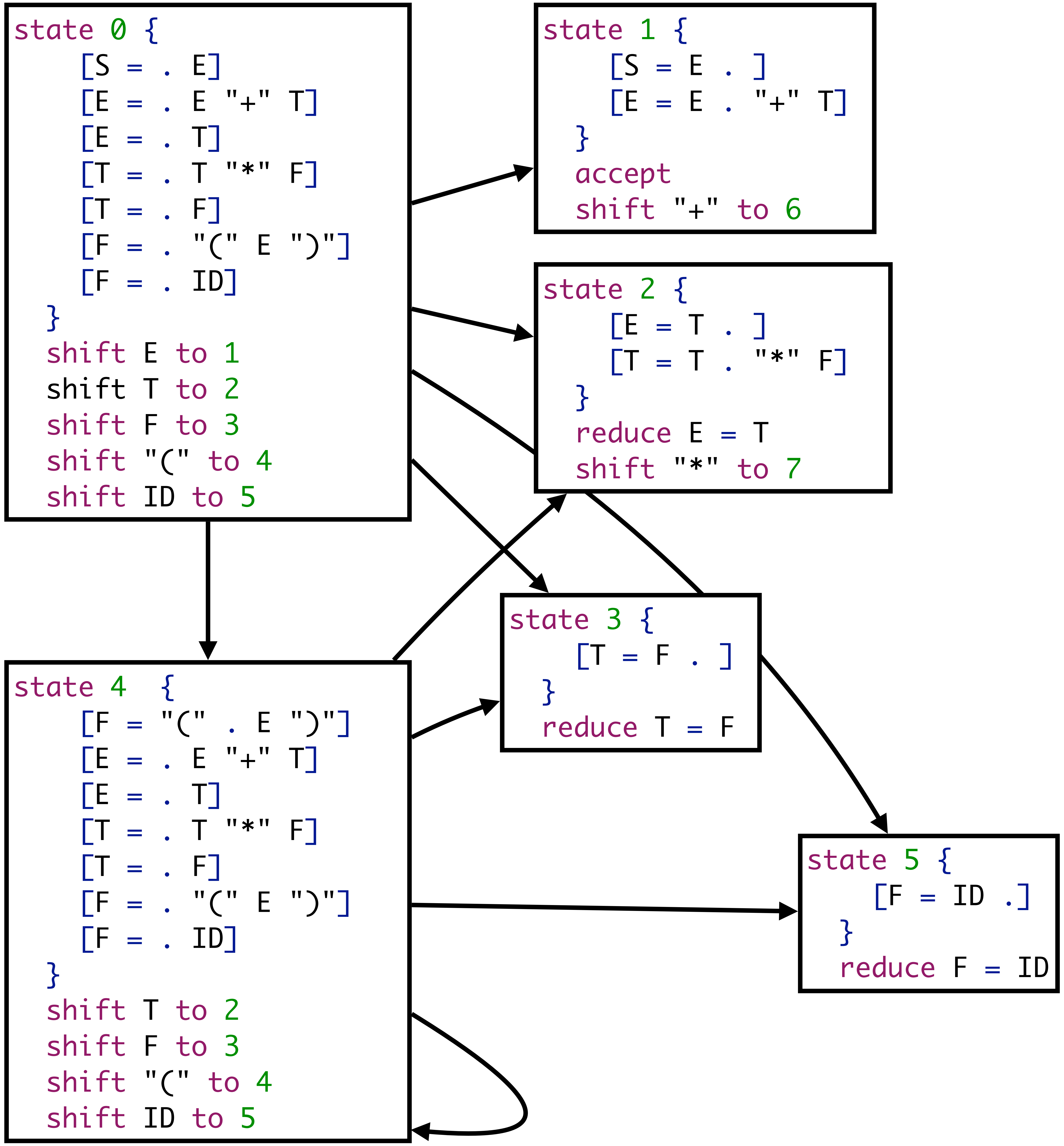
```
state 0 {
    [S = . E]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift E to 1
  shift T to 2
  shift F to 3
  shift "(" to 4
```

```
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
```

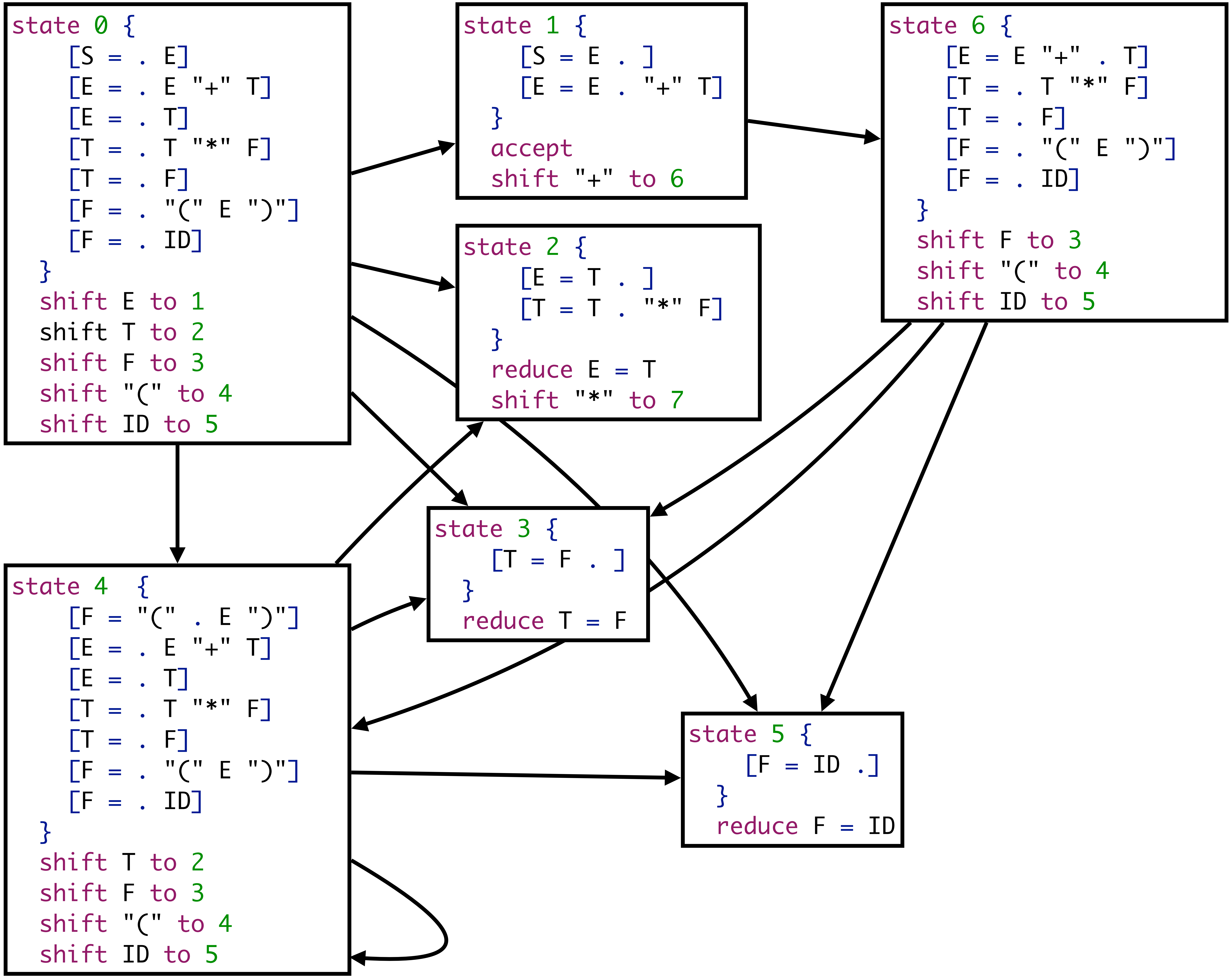
```
state 6 {
    [E = E "+" . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
```

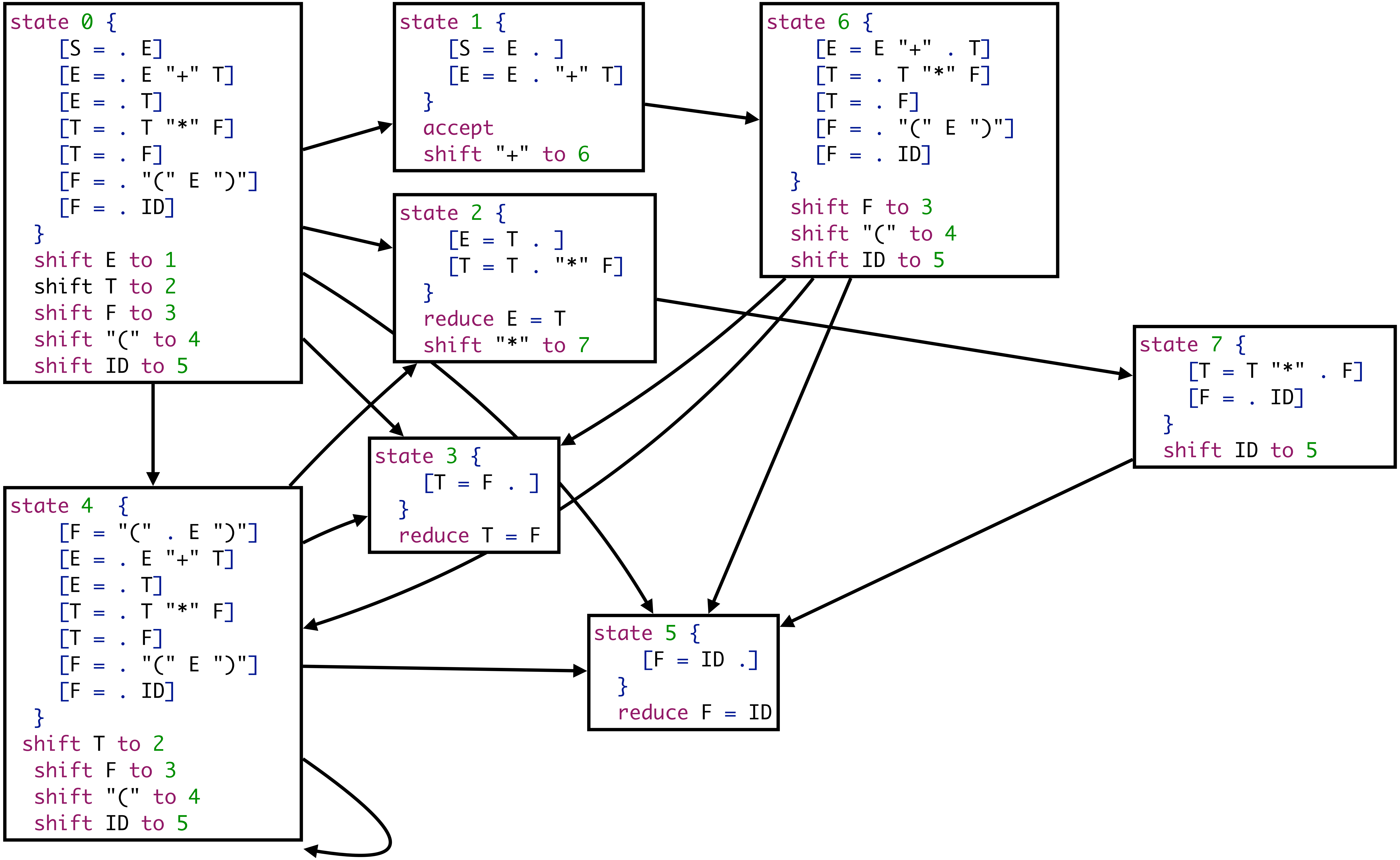
```
state 2 {
    [E = T . ]
    [T = T . "*" F]
  }
  reduce E = T
  shift "*" to 7
```

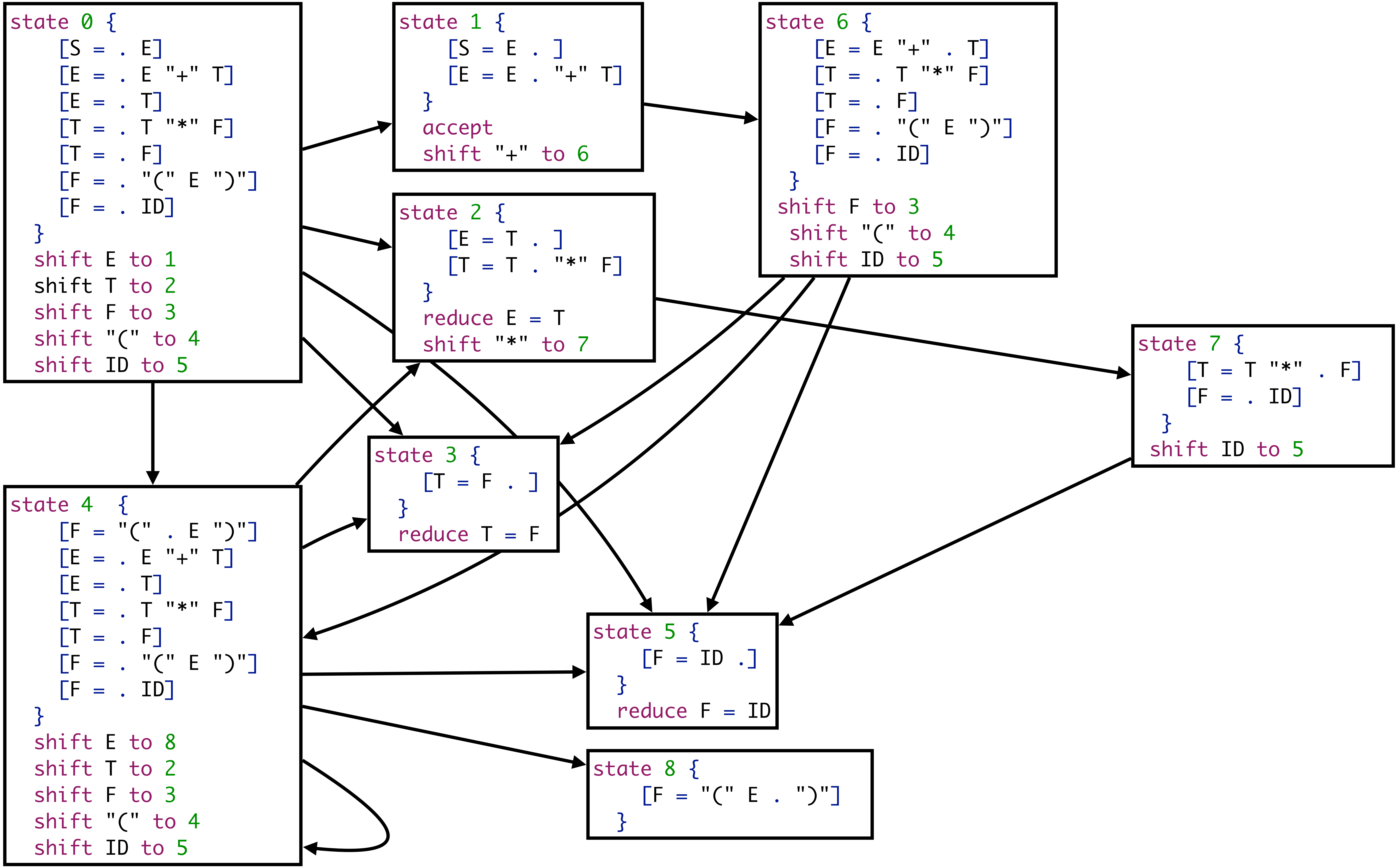
```
state 3 {
    [T = F . ]
  }
  reduce T = F
```

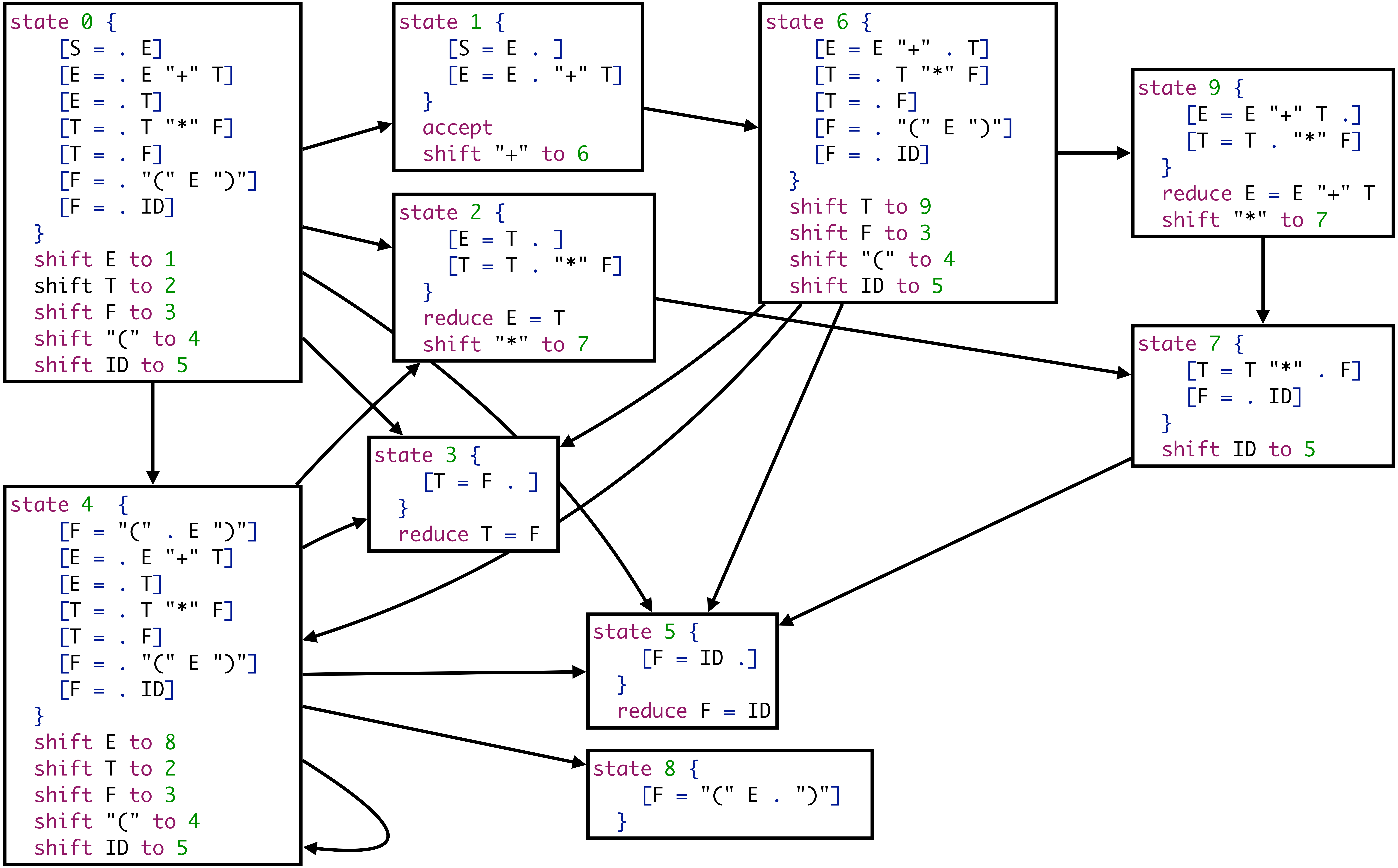
```
state 4  {
    [F = "(" . E ")"]
    [E = . E "+" T]
    [E = . T]
    [T = . T "*" F]
    [T = . F]
    [F = . "(" E ")"]
    [F = . ID]
  }
  shift T to 2
  shift F to 3
  shift "(" to 4
  shift ID to 5
```

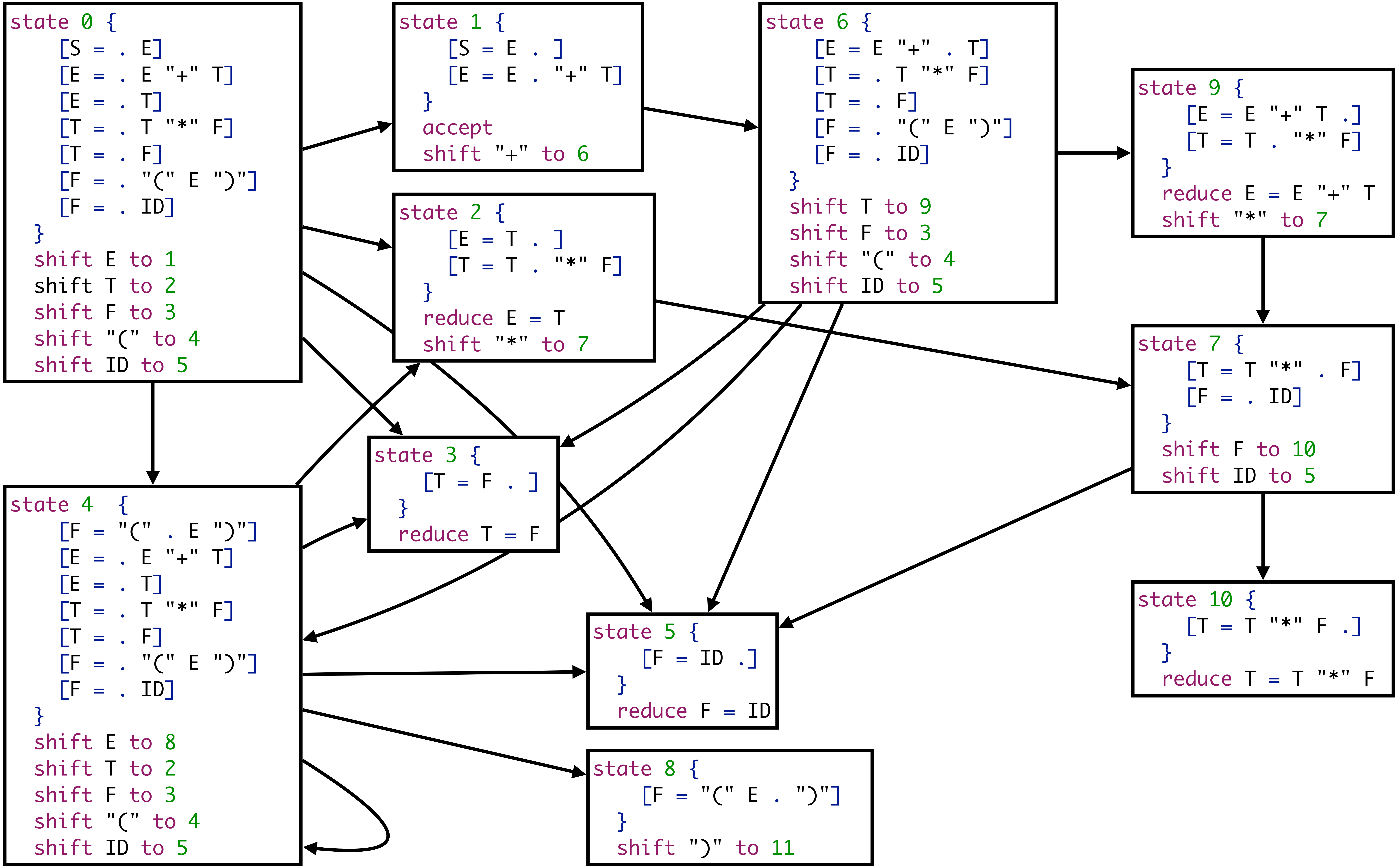
```
state 5 {
    [F = ID .]
  }
  reduce F = ID
```

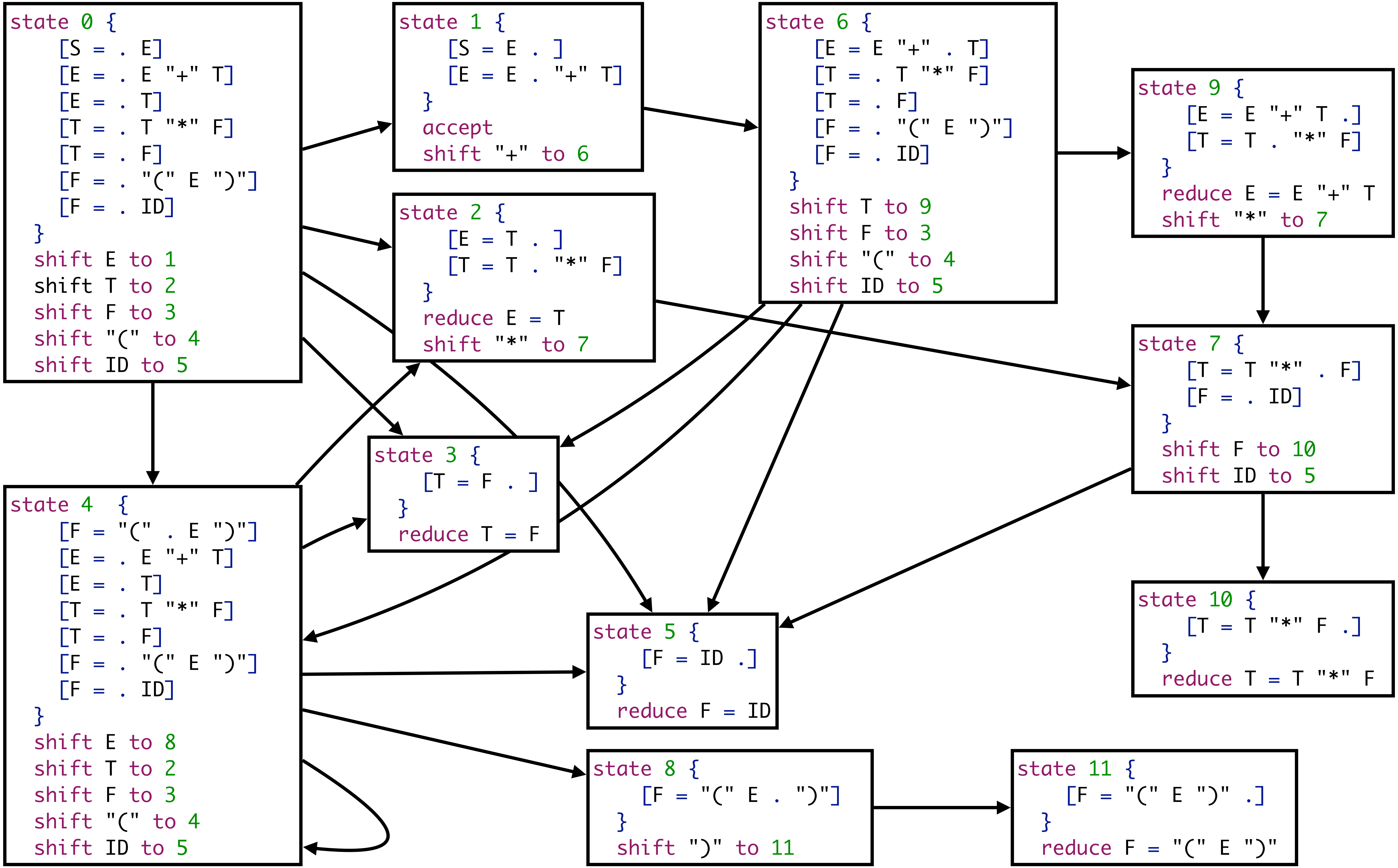

state 0 {
[S = . E]
[E = . E "+" T]
[E = . T]
[T = . T "*" F]
[T = . F]
[F = . "(" E ")"]
[F = . ID]
}
shift E to 1
shift T to 2
shift F to 3
shift "(" to 4
shift ID to 5
state 1 {
[S = E .]
[E = E . "+" T]
}
accept
shift "+" to 6
state 2 {
[E = T .]
[T = T . "*" F]
}
reduce E = T
shift "*" to 7
state 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 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 7 {
[T = T "*" . F]
[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 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 7 {
[T = T "*" . F]
[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 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 9 {
[E = E "+" T .]
[T = T . "*" F]
}
reduce E = E "+" T
shift "*" to 7
state 7 {
[T = T "*" . F]
[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 . ")"]
}

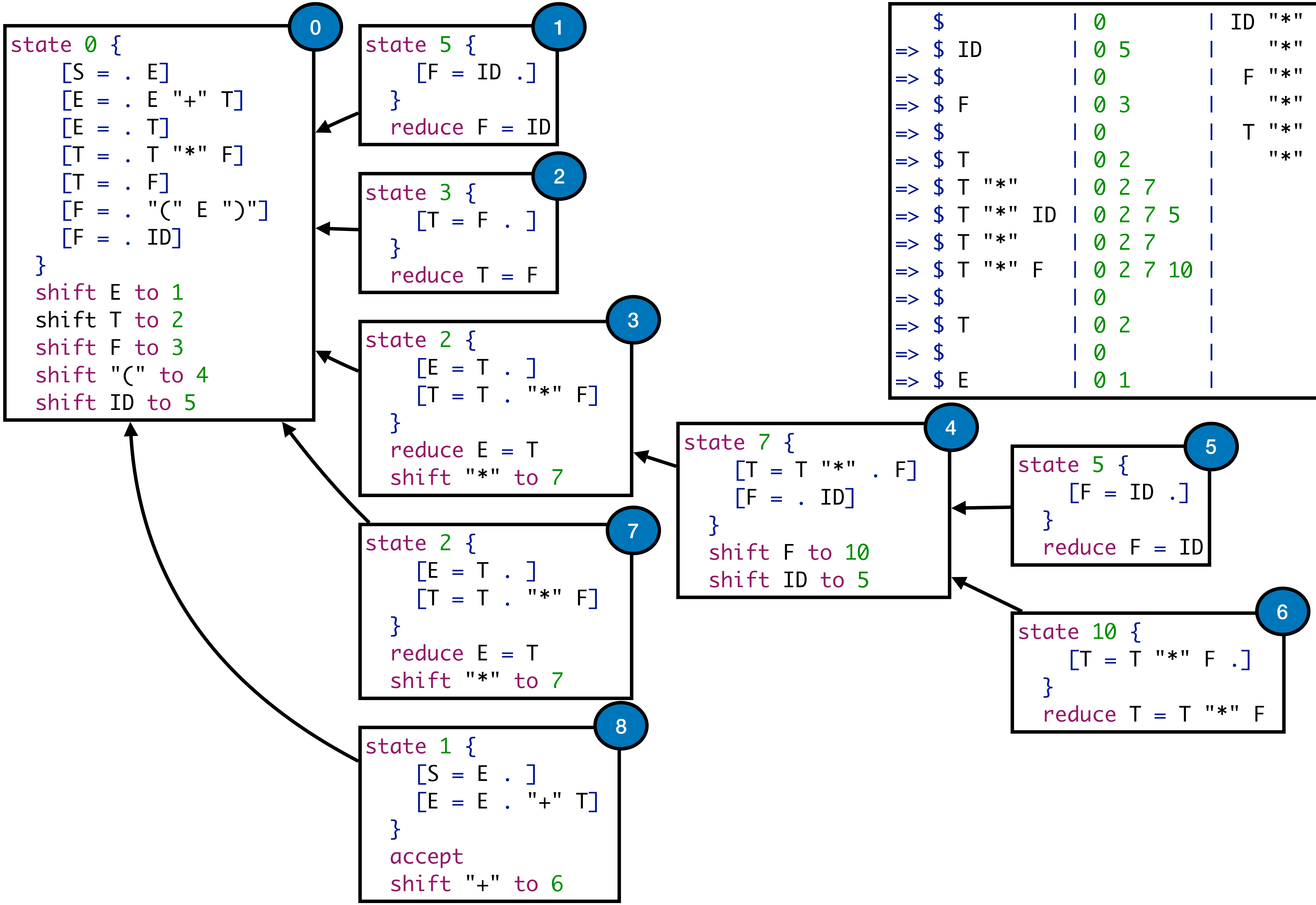

state 0 {
[S = . E]
[E = . E "+" T]
[E = . T]
[T = . T "*" F]
[T = . F]
[F = . "(" E ")"]
[F = . ID]
}
shift E to 1
shift T to 2
shift F to 3
shift "(" to 4
shift ID to 5
state 1 {
[S = E .]
[E = E . "+" T]
}
accept
shift "+" to 6
state 2 {
[E = T .]
[T = T . "*" F]
}
reduce E = T
shift "*" to 7
state 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 = . 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 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 9 {
[E = E "+" T .]
[T = T . "*" F]
}
reduce E = E "+" T
shift "*" to 7
state 7 {
[T = T "*" . F]
[F = . ID]
}
shift F to 10
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 10 {
[T = T "*" F .]
}
reduce T = T "*" F
state 8 {
[F = "(" E . ")"]
}
shift ")" to 11
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
```

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

LR(0) Parse Table

See book

Solving Shift/Reduce Conflicts

Solving Shift/Reduce Conflicts

First and Follow: see book

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

More Topics in Syntax and Parsing

First/Follow

- Selecting between actions in LR parse table

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

Next: Transformation