Compiler Design
Syntax Analysis
Introduction to LR Parsing
Simple LR

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Why LR parsers?
- Items and the LR(0) Automaton
- The LR Parsing Algorithm
- Constructing SLR Parsing Tables
- Viable Prefixes

Introduction

- LR(k) – the most prevalent type of bottom-up parser
- L – from left to right
- R – constructing the rightmost derivation in reverse
- k - the number of lookahead input symbols
- k=0, k=1cases are of practical interest
- we consider k<=1
- when k is omitted then k=1

Introduction

- basic concepts of LR parsing
- the easiest method to construct shift-reduce parsers – simple LR (SLR)
- usually LR parsers are built by automatic generators
- “items”
- “parser states”
- next section presents
  - canonical LR
  - LALR
  - complex methods used in the majority of LR parsers

Why LR Parsers ?

- table driven
  - like non-recursive LL parsers
- LR grammar
  - a grammar for which we can use the methods in this section
- is named intuitively
  - a left to right shift-reduce parser
  - to recognize handles of right sentential forms
  - when they appear on the top of the stack

Why LR Parsing ?

- LR parsers can recognize all language constructs written in context free grammars
- the most general non-backtracking shift-reduce parsing method
- can be implemented as efficiently as other more primitive shift-reduce methods
- can detect syntactic error as soon as possible
- the class of LR grammars is a superset of LL or predictive grammars
- too much work to write a LR parser by hand for a typical programming language grammar

Items and LR(0) Automaton

- LR(0) item – production of G with a dot as some point at some position of the body
- production A->XYZ yields four items
  - A->•XYZ
  - A->X•YZ
  - A->XY•Z
  - A->XYZ•
- production A->ε
  - A->•

Items

- indicates how much of a production we have seen at a given point in the parsing process
- item A->•XYZ
  - we hope to see a string derivable from XYZ next on input
- item A->X•YZ
  - we have just seen a string derivable from X
  - we hope to see a string derivable from YZ next on input
- item A->XYZ•
  - we have just seen a string derivable from XYZ
  - it may be time to reduce XYZ to A

Representing Item Sets

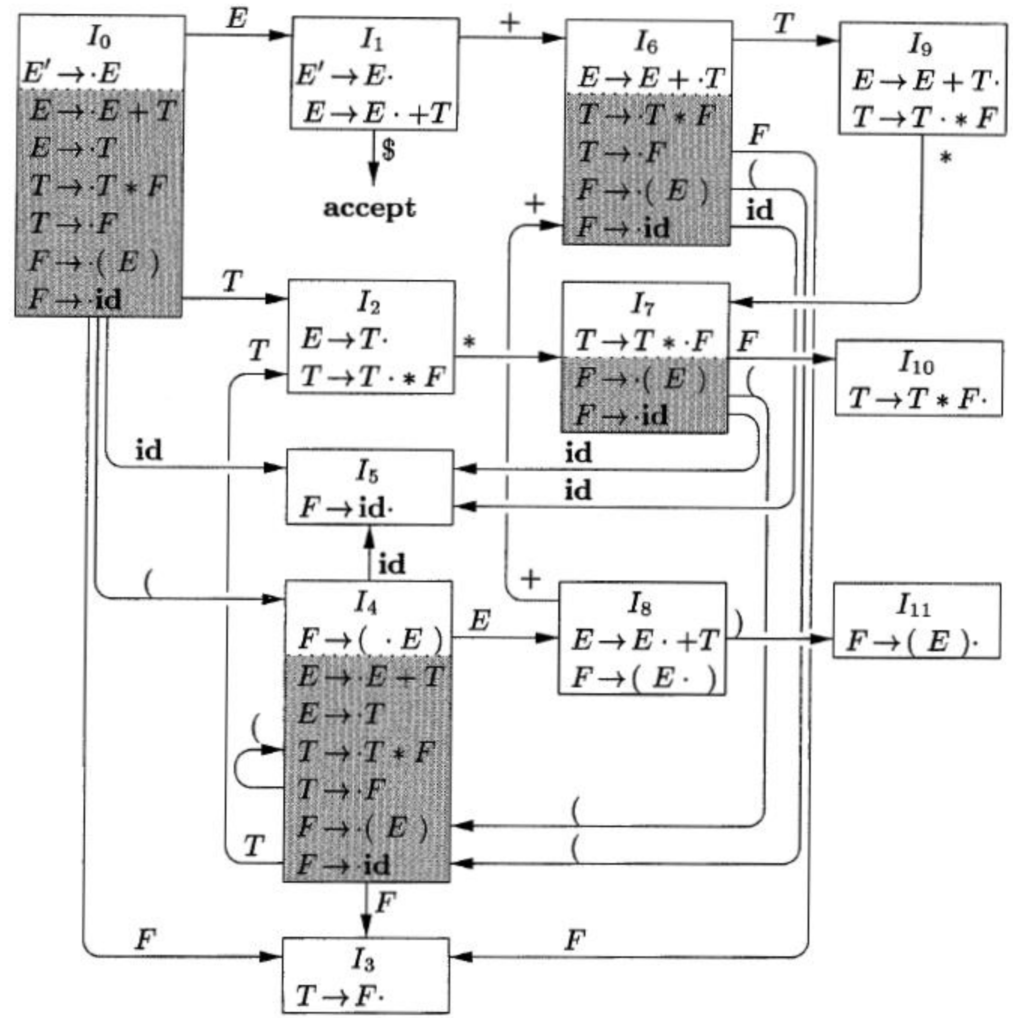
- pair of integers
  - the number of the production
  - the position of the dot
- sets of items
  - lists of such pairs
- closure items
  - the dot is at the beginning of the body
  - can be reconstructed from other items in the set
  - we do not have to include them in the list

Canonical LR(0)

- one collection of sets of LR(0) items
- provides the basis for constructing a DFA
- DFA is used to make parsing decisions
- LR(0) automaton
  - each state of LR(0) – set of items in the canonical LR(0) collection
  - the dead state is not represented !!!
- to build canonical LR(0) means to define
  - an augmented grammar
    - G - grammar with starting symbol S
    - G’ – augmented grammar for G, S’->S
    - acceptance when S’ -> S
  - two functions
    - CLOSURE
    - GOTO

LR(0) DFA Example

- E->E+T|T
- T->T*F|F
- F->id | (E)



Closure of Item Sets

- I is the set of items for grammar G
- CLOSURE(I) – set of items built from I
  - (1) initially add every item I to closure(I)
  - (2) if A->α•Bβ is in CLOSURE(I) and B->γ
    - then add B->•γ to CLOSURE(I) if not already there
  - apply this rule until no more new items can be added to CLOSURE(I)

Explanations

- A->α•Bβ is in CLOSURE(I)
- we might see a substring derivable from Bβ as input
- the string derivable from Bβ
  - will have a prefix derivable from B applying one of the B-productions
- if B->γ
  - then we include B->•γ in CLOSURE(I)

Augmented Expression Grammar

- E'->E
- E->E+T|T
- T->T*F|F
- F->id | (E)
- if I is the set of one item {[E'->•E]}
  - then CLOSURE{I} contains the set items of I_0
    - E'->•E
  - since E->E+T and E->T we also add
    - E->•E+T and E->•T
  - since T->T*F and T->F we also add
    - T->•T*F and T->•F
  - since F->(E) and F->id we also add
    - F->•(E) and F->•id

Computation of CLOSURE

```
SetOfItems CLOSURE(I)
{
   J=I;
   repeat
         for(each item A->α•Bβ in J)
               for(each production B->γ of G)
                     if(B->•γ is not in J)
                           add B->•γ to J;
   until no more items are added to J on one round;
   return J;
}
```

Kernel and Non-kernel Items

- Kernel Items
  - S'->•S
  - all items whose dots are not at the left end
- Non-kernel Items
  - all items with their dots at the left end
  - Except S'->•S
- each set is formed by
  - taking the closure of a set of kernel items
  - the items added to the closure can never be kernel items
    - they must not be stored since they can be regenerated

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007