Office - Room 2.4, 13 Westland Row (Upstairs from DUCSS)

Lecture 01

### Language Processors

1. **Interpreters** - Executes code directly.
2. **Translators**
   1. **Assemblers** - Low-level languages. One-to-one translation i.e. single assembly line statements make a single object statement.
   2. **Compilers** - High-level languages. One-to-many translation.

Use x := x + 1 instead of x = x + 1.

Compile time vs. Run time.

Compile time - Translating what you see from high-level language to low-level language. Leaves behind a set of instructions.

Top-down compiler design instead of bottom-up.

### Simple Model of a Compiler

Source language⟶Lexical Analyser⟶Syntax Analyser⟶Code Generator⟶Object code

Lex. Analyser / Syn. Analyser / Code Gen. can all put / take data from a Symbol Table.

### Symbol Table

An ST does **not** exist when the program is running.

Takes names of identifiers. Values do **not** belong in Symbol Table. Instead they will live somewhere in memory when the program is running.

EXCEPTION - Static constants do live in ST.

If a constant is used in an instruction, it is directly used in that instruction.

### Lexical Analysis

Splits input (statements in source language) into a list of **lexical tokens**. It recognises the lexemes (lexical elements).

Each token represents a lexime that it sees.

Classes of Tokens

1. (Class)
2. (Class, Value)
3. (Class, Pointer Value) - Pointer to Symbol Table.

IF x > y THEN A := B + C \* D

turns into

(IF) (ID, ↑x) (RELOP, GT) (ID, ↑Y) (THEN) (ID, ↑A) (ASSIGN) (ID, ↑B) (OP, PLUS) (ID, ↑C) (OP, TIMES) (ID, ↑D)

RELOP = Relation Operator

↑x = Pointer to x

Lecture 02

### Syntax Analysis

Checks that the input is **grammatically correct** and translates the sequence of lexical tokens into a sequence of atoms where the order of the atoms reflects the order in which the operations are to be performed at run-time.

A := B + C \* D

3 atoms are generated

1. :=
2. +
3. \*

Generated in BOMDAS order.

Right associative.

Output from Lexical Analyser (input to Syntax Analyser):

(ID, ↑A) (ASSIGN) (ID, ↑B) (OP, PLUS) (ID, ↑C) (OP, TIMES) (ID, ↑D)

Operation 1 → ANSWER1 = C \* D

Operation 2 → ANSWER2 = B + ANSWER1

Operation 3 → A = ANSWER2

Atoms:

{MULT ↑C, ↑D, ↑PR1} {ADD ↑B, ↑PR1, ↑PR2} {ASSIGN ↑PR2, ↑A}

PR1 (Partial Result 1) is an entry in the symbol table. Its value will be found at run-time in memory or a register.

### Code Generator

Expands the atoms into object code. (assembly code / machine code / other)

### Semantic Analyser (Not Needed)

Processes the meaning. Goes between the Syntax Analyser and the Code Generator. Can be done in Lexical Analyser and Code Generator instead.

### Static Semantic Processing

Putting the name and type of identifiers into the Symbol Table.

### Dynamic Semantic Analysis

Selecting the correct class of instruction for an operator. (i.e. If B and C are integers, it is an integer plus we need to add them together.) Can be performed in the Code Generator.

### Passes

Not units of design like Lexical Analyser / Code Generator. Invented to save memory. Lexical Analyser run first, then overwritten in memory with Syntax Analyser and so on…

### Optimisation

Code-improvement techniques. e.g. Check to see if a value is stored elsewhere before storing. Prior to Code Generation.

### Symbol Table Management

Run-time implementation describes how things such as recursion and the mapping of arrays are handled at run-time.

### Finite State Machines

Used to build Lexical Analysers.

Check to see if there’s an odd number of 1s in a sequence.

States: {EVEN, ODD}

Inputs: {0, 1}

Accept States: {ODD}

Start States: {EVEN}

δ(EVEN, 0) → EVEN

δ(EVEN, 1) → ODD

δ(ODD, 0) → ODD

δ(ODD, 1) → EVEN

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| learn | | Inputs | |  |
| 0 | 1 |
| States | EVEN | EVEN | ODD | 0 |
| ODD | ODD | EVEN | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | **⊣** |
| One 1 | No 1’s | ACCEPT | REJECT |
| Two 1’s | Two 1’s | Two 1’s | ACCEPT |
| No 1’s | No 1’s | One 1 | REJECT |

Lecture 03

|  |  |  |  |
| --- | --- | --- | --- |
|  | **0** | **1** | **⊣** |
| **S** | S | S | 0 |

Recognises nothing. {} or Ø is the empty set.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **0** | **1** | **⊣** |
| **S** | T | T | 1 |
| **T** | T | T | 0 |

Recognises the null sequence ∊

{∊}

### Using Finite State Machine for Lexical Analysis

Design a FSM to recognise any valid sequence that may follow the keyword INTEGER in Fortran.

INTEGER X(5, I, 7), Y X and Y are type integer. (x, y, z) is a 3D array.

INTEGER V(C, V, C), V

V to be made integer, C = constant, I = variable identifer, C = dimension, V = variable identifier.

INTEGER TABLE(2, 3)

Table => V

INTEGER X ( 5 , I , 7 ) , Y

INTEGER |V|(|C|,|V|,|C|)|,|V

Usage 1|2|3|4|5|6|5|4|7|8|2

X = variable id to be made integer

I = variable id to specify dimension

5 is separating dimensions

1 = Starting state

2-8 = States

1 extra state for E (Error)

INTEGER |V|(|C|,|V|,|C|)|,|V|,|V (You can add more variables to the sequence)

Usage 1|2|3|4|5|6|5|4|7|8|2|8|

**Transition Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **V** | **C** | **,** | **(** | **)** | **⊣** |
| **1** | 2 |  |  |  |  | 0 |
| **2** |  |  | 8 | 3 |  | 1 |
| **3** | 6 | 4 |  |  |  | 0 |
| **4** |  |  | 5 |  | 7 | 0 |
| **5** | 6 | 4 |  |  |  | 0 |
| **6** |  |  | 5 |  | 7 | 0 |
| **7** |  |  | 8 |  |  | 1 |
| **8** | 2 |  |  |  |  | 0 |
| **E** |  |  |  |  |  | 0 |

All **blank** table entries represent transitions to the error state E.

{1, 2, 8, 3, 6, 4, 5, 7, 8, E} **DON’T FORGET THE ERROR STATE**

Partition the states into sets of **accepting** and **rejecting** states

P0 - ({2, 7}, {1, 3, 4, 5, 6, 8, E})

Separate {1, 3, 4, 5, 6, 8, E} with respect to input V.

P1 - ({2, 7}, {1, 8}, {3, 4, 5, 6, E})

{1, 8} are equivalent, as are {4, 6} and {3, 5}.

Replace:

{1, 8} - A

{3, 5} - B

{4, 6} - C

Partition with respect to C

P2 - ({2, 7}, {1, 8}, {3, 5}, {4, 6, E})

Partition with respect to )

P3 - ({2, 7}, {1, 8}, {3, 5}, {4, 6}, {E})

Partition 2 and 7 as they are not the same

P3 - ({2}, {7}, {1, 8}, {3, 5}, {4, 6}, {E})

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **V** | **C** | **,** | **(** | **)** | **⊣** |
| **A** | 2 |  |  |  |  | 0 |
| **2** |  |  | 8 | 3 |  | 1 |
| **B** | 6 | 4 |  |  |  | 0 |
| **C** |  |  | 5 |  | 7 | 0 |
| **7** |  |  | 8 |  |  | 1 |
| **E** |  |  |  |  |  | 0 |

Lecture 04

### Design Lexical Analyser for Floating Point (Real) Constants.

0 is represented by 0.0, instead of 0. or .0

2 - digit before decimal point state.

3 - decimal point.

4 - digit after decimal point.

5 - exponent.

6 - sign of exponent

7 - digit after exponent.

8 - sign of digit.

no sign

| |d|d|.|d|d|E|-|d|d|

|1|2|2|3|4|4|5|6|7|7|

has sign

| |-|d|.|d|d|E|D

|1|8|2|3|4|4|5|6

no exponent

| |d|.|d|

|1|2|3|4|

| |d|E|d|

|1|2|5|7|

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Digit** | **Sign** | **E** | **.** | **⊣** |
| **1** | 2A | 8 |  |  |  |
| **2** | 2B |  | 5A | 3 |  |
| **3** | 4A |  |  |  |  |
| **4** | 4B |  | 5B |  | OK1 |
| **5** | 7A | 6 |  |  |  |
| **6** | 7B |  |  |  |  |
| **7** | 7C |  |  |  | OK2 |
| **8** | 2C |  |  |  |  |

Blank table entries represent exits to an error routine.

2A - Advance the input and move to state 2.

Sign

Number

Count

Exponent

Exponent Sign

\_\_\_\_\_\_\_\_\_\_\_\_

Number

Exponent

2A: Number <- Value(Digit), Sign <- 1

2B: Number <- Number\*10 + Value(Digit)

2C: Number <- Value(Digit)

3 : Count <- 0

5A/

4A: Number <- Number\*10 + Value(Digit), Count <- Count+1

4B:

6 : If sign = “+” then the exponent sign <- +1

else the exponent sign <- -1

7A: Exponent <- Value(Digit), Exponent sign <- 1

7B: Exponent <- Value(Digit)

7C: Exponent <- Exponent\*10 + Value(Digit)

OK1: Number <- Number \* sign, Exponent <- Count \* -1

OK2: Number <- Number \* sign, Exponent <- Exponent \* ExponentSign - Count

Count = how many times we have multiplied by 10, so we can adjust the exponent afterwards.

-123.345 exponent = -3

45E-23

Lecture 05

How do we prevent overflow in the previous example? Incomplete.

### Pushdown Machines

Design a pushdown machine to recognise any sequence of left and right parentheses where they match

(()())⊣

Inputs = { (, ), ⊣ }

Stack Symbols = { L, ▽ }

States = { S }

Starting State = ▽

|  |  |  |  |
| --- | --- | --- | --- |
|  | **(** | **)** | **⊣** |
| **L** | Push (L)  States (S)  Advance | Pop  State (S)  Advance | Unmatched Left Parenthesis |
| **▽** | Push (L)  State (S)  Advance | Extra Right Parenthesis | ACCEPT |

Stack Input

▽ (()())⊣

▽L ()())⊣

▽LL )())⊣

▽L ())⊣

▽LL ))⊣

▽L )⊣

▽ ⊣

ACCEPT

Design a pushdown machine to recognise any sequence of 1s & 0s of the form 1n0n | n > 0

e.g. 10, 1100, 11110000

Inputs = { 1, 0, ⊣ }

Stack Symbols = { I, ▽ }

States = { S1, S2 }

S1, Starting State = ▽

|  |  |  |  |
| --- | --- | --- | --- |
|  | **1** | **0** | **⊣** |
| **I** | Push (I)  State (S1)  Advance | Pop  State (S2)  Advance | Unmatched 1  (No 0s) |
| **▽** | Push (I)  State (S1)  Advance | REJECT  Expected 1  (No 1s) | No Input  (n=0) |

S2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **1** | **0** | **⊣** |
| **I** | REJECT  Expected 0 | Pop  State (S2)  Advance | Unmatched 1 |
| **▽** | REJECT  Not of form 1n0n | Extra 0 | ACCEPT |

Lecture 06

### Extended Stack Operation

Replace (ABC) = Pop, Push A, Push B, Push C.

Input = 1100

**Stack** => ▽Y

I see a 1 => ▽XY

I see a 1 => ▽XXY

I see a 0 => ▽XX

I see a 0 => ▽X

I see a ⊣ => ▽ - ACCEPT

Starting Stack = ▽Y

|  |  |  |  |
| --- | --- | --- | --- |
|  | **0** | **1** | **⊣** |
| **X** | Pop  Advance |  |  |
| **Y** | Pop  Retain | Replace Y w/ (X Y)  Advance |  |
| **▽** |  |  | ACCEPT |

We’ll design our parsers with single-state pushdown machines.

### Pushdown Translators

A pushdown translator is simply a pushdown recogniser that produces an output.

Out(X)

Output X

Design a pushdown translator to convert an arbitrary string of 0s and 1s into a string of the form 1n0m where n and m are the number of 1s and 0s respectively.

01011 => 11100

0 - push Z & advance, 1 - output 1 and advance.

pop 0, output 0 and advance.

Starting Stack = ▽

|  |  |  |  |
| --- | --- | --- | --- |
|  | **0** | **1** | **⊣** |
| **Z** | Push Z  Advance | Output 1  Advance | Pop  Output 0  Retain |
| **▽** | Push Z  Advance | Output 1  Advance | ACCEPT |

### Context-Free Grammars

1. <A> ➝ a <A> <B> c
2. <A> ➝ <A> b
3. <A> ➝ ε
4. <B> ➝ b
5. <B> ➝ ε

a b c

Rightmost derivation

<A> ➝1 a <A> <B> c

➝2 a <A> b <B> c

➝3 a b <B> c

➝5 a b c

Leftmost derivation

<A> ➝1 a <A> <B> c

➝3 a <B> c

➝4 a b c

Because the string abc has two leftmost derivations the grammar is said to be **ambiguous**.

**Note**: Ambiguity is a property of a **grammar** not a language.

<s> <s>

/ / \ \ / / \ \

a <A> <B> c a <A> <B> c

/ \ \ / \

<A> b ε ε b

/

c

A language is defined by a context-free grammar plus a set of semantic restriction.

<E> ➝ <E> <OP> <T>

<E> ➝ <T>

<OP> ➝ +

<OP> ➝ OR

<T> ➝ IDENT

a + b OR c

### A grammar for arithmetic expressions

E - Expression

T - Term

P - Primary

F - Factor

1. <E> ➝ <E> + <T>
2. <E> ➝ <T>
3. <T> ➝ <T> \* <P>
4. <T> ➝ <P>
5. <P> ➝ (<E>)
6. <P> ➝ CONST

1 + 2 \* 3 + 4

<E> ➝1 <E> + <T>

➝1 <E> + <T> + <T>

➝2 <T> + <T> + <T>

➝4 <P> + <T> + <T>

➝6 CONST1 + <T> + <T>

➝3 CONST1 + <T>\*<P> + <T>

➝\* CONST1 + CONST2 \* CONST3 + CONST4

\* = 0 or more steps

<E> +

/ \ \ / \

<E> T <T> +

/ | \ \ / \

<E> T <T> <P> \*

/ / | \ \ / \

<T> <T> \* <P> CONST4

/ / \

<P> <P> CONST3

/ /

CONST1 CONST2

Lecture 07 - CouRse ReALLY STArts

### Translation Grammars

Design a translation grammar to convert an arithmetic expression from infix to postfix form.

A + B \* C + D => A B C \* + D +

**Activity Sequence**:

A {A} + B {B} \* C {C} {\*} {+} + D {D} {+}

**Action Symbols**: Print out what’s inside { }.

**Translation Grammar**:

1. <E> ➝ <E> + <T> { + }
2. <E> ➝ <T>
3. <T> ➝ <T> \* <P> { \* }
4. <T> ➝ <P>
5. <P> ➝ (<E>)
6. <P> ➝ A { A } |
7. <P> ➝ B { B } | <P> ➝ IDENTP { IDENT } P = ptr. to symbol table
8. <P> ➝ C { C } |
9. <P> ➝ D { D } |

This is an example of syntax directed translation.

<E>

\_\_\_|\_\_\_\_\_\_\_

/ \ \ \

<E> + <T> {+}

\_\_\_|\_\_\_\_\_ \

/ | \ \ \

<E> + <T> {+} \

/ \_\\_\_\_\_ \

/ / | \ \ \

<T> <T> \* <P> {\*} <P>

/ / / \ / \

<P> <P> IDENT↑C\ IDENT↑D\

/ \ / \ {IDENT↑C} {IDENT↑D}

IDENT↑A \ IDENT↑B\

{IDENT↑A} {IDENT↑B}

Add 2 productions to the grammar to cater for exponentiation ↑ where the ↑ operator is right-associative i.e. AB^C = A(B^C)

<F> - Factor

A B C ↑ ↑

Write down the modified grammar and the derivation tree.

A + BC^D \* E

**Translation Grammar**:

1. <E> ➝ <E> + <T> { + }
2. <E> ➝ <T>
3. <T> ➝ <T> \* <F> { \* }
4. <T> ➝ <F>
5. <F> ➝ <P> ↑ <F> { ↑ }
6. <F> ➝ <P>
7. <P> ➝ (<E>)
8. <P> ➝ A { A } |
9. <P> ➝ B { B } | <P> ➝ IDENTP { IDENT } P = ptr. to symbol table
10. <P> ➝ C { C } |
11. <P> ➝ D { D } |

<E>

\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_

/ | \ \

<E> + <T> {+}

/ \_\_|\_\_\_\_\_+

<T> / | \ \

/ <T> \* <F> {\*}

<F> / \

/ / <P>

<P> <F> / \

/ \ \_\_\_|\_\_\_\_\_\_\_\_\_ IDENT↑E{IDENT↑E}

IDENT↑A{IDENT↑A} / | \ \

<P> ↑ <F> {↑}

/ \ \_\_\_|\_\_\_\_\_

IDENT↑B {IDENT↑B} / | \ \

<P> ↑ <F> {↑}

/ \ \

IDENT↑C {IDENT↑C} <P>

/ \

IDENT↑D {IDENT↑D}

Lecture 08 - InteReSTinG MAterIaL

### Synthesised Attributes

For **interpreter**

Production Rules:

1. <S> ➝ <E> {VALUE} <S> = Starting non-terminal
2. <E>p ➝ <E>q + <T>r p ← q + r
3. <E>p ➝ <T>q p ← q
4. <T>p ➝ <T>q \* <P>r p ← q \* r
5. <T>p ➝ <P>q p ← q
6. <P>p ➝ (<E>q) p ← q
7. <P>p ➝ CONSTq p ← q

(1 + 2) \* (3 + 4)

where <E>p, <T>p and <P>p synthesised p and { VALUEp } inherited p.

<S>

/ \

<E>21 {VALUE}21 INHERITENCE

|

<T>21

\_\_\_|\_\_\_

/ | \

<T>3 \* <P2>7

/ / | \

<P1>3 ( <E2>7 )

/ | \ / | \

( <E1>3 ) <E>3 + <T>4

/ | \ | |

<E>1 + <T>2 <T>3 <P>4

| | | |

<T>1 <P>2 <P>3 CONST4 SYNTHESIS

| | |

<P>1 CONST2 CONST3

|

CONST1

For a compiler???

A \* (B + C)

{ ADD↑B ↑C ↑PR1 } { MULT↑A ↑PR1 ↑PR2 } PR = Partial Result

Processing in postfix form.

Activity Sequence

A \* (B + C { ADD }) { MULT }

Production Rules:

1. <E>p ➝ <E>q + <T>r { ADD s, t, u } s ← q, t ← r, (p, u) ← NEWT (new t)
2. <E>p ➝ <T>q p ← q
3. <T>p ➝ <T>q \* <P>r { MULT s, t, u } s ← q, t ← r, (p, u) ← NEWT
4. <T>p ➝ <P>q p ← q
5. <P>p ➝ (<E>q) p ← q
6. <P>p ➝ IDENTq p ← q

where <E>p, <T>p, and <P>p synthesised p. All action symbol attributes are inherited.

<E>PR2

|

<T>PR2

\_|\_\_\_\_\_\_\_\_\_

/ | \ \

<T>A \* <P>PR1 {MUL A, PR1, PR2}

/ / | \

<P>A ( <E>PR1 )

/ \_\_|\_\_\_\_\_\_\_

IDENTA / | \ \

<E>B + <T>C {ADD B, C, PR1}

| |

<T>B <P>C

| |

<P>B IDENTC

|

IDENTB

### Inherited Attributes

|  |
| --- |
| int a, b, c |

<DECL> ➝ TYPE IDENT <IDENT LIST>

<IDENT LIST> ➝ , IDENT <IDENT LIST>

<IDENT LIST> ➝ ε

<DECL>

/ | \

TYPEINT IDENT↑A <IDENT LIST>

/ | \

, IDENT↑B <IDENT LIST>

\

ε

<DECL> ➝ TYPEp IDENTq {SET TYPE}r, s <IDENT LIST>t r ← q, (s, t) ← p

<IDENT LIST>p ➝ , IDENTq {SET TYPE}r, s <IDENT LIST>t r ← q, (s, t) ← p

<IDENT LIST>p ➝ ε

<DECL>

\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/ | \ \

TYPEINT IDENT↑A {SET TYPE} <IDENT LIST>

\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/ | \ \

, IDENT↑A {SET TYPE} <IDENT LIST>

\

ε

### Exercise 5 Hints

Object:

* Kind - Represents the kind of object - variable / procedure - need to add new kind for construct (constant) = vs :=.
* Array is still a variable - elements have a type. differentiate between scalar variable and 2D array variable - add other field
* Type
* Sort - add for different sorts of arrays - scalar[single value] vs array

Lecture 09

<E> → CONSTp <R>q q ← p R = Remainder of expression

<R>p → + CONSTq <R>r r ← p + q

<R>p → \* CONSTq <R>r

<R>p → ε

1 + 2 \* 3

<E>

/ \

CONST1 <R>1 CONST1 => <R>1

/ | \

+ CONST2<R>3 <R>1 + CONST2 => <R>3

/ | \

\* CONST3<R>9 <R>3 \* CONST4 => <R>9

\

ε

Tree shows right-associativity.

Activity Sequence:

1 { 1 } + 2 { 2 } { + } \* 3 { 3 } { \* }

A + B \* C

{ ADD } { MULT }

<E> → IDENTp <R>q q ← p

<R>p → IDENTq { ADD s, t, u } <R>v s ← p, t ← q, (u, v) ← NEWT

<R>p → \* IDENTq { MULT s, t, u } <R>v s ← p, t ← q, (u, v) ← NEWT

<R>p → ε

A := B + C

{ ADD } { ASSIGN } ADD has higher precedence.

<S> → IDENTp := <E>q { ASSIGNr, s } r ← q, s ← p

<E>p → IDENTq <R>r, s r ← q, p ← s

<R>p → IDENTq { ADD s, t, u } <R>v, w s ← p, t ← q, (u, v) ← NEWT, q ← w

<R>p → \* IDENTq { MULT s, t, u } <R>v, w s ← p, t ← q, (u, v) ← NEWT, q ← w

<R>p, q → ε q ← p

<S>

\_\_|\_\_\_\_\_

/ | \ \

IDENT := <E> {ASSIGN}

\_|\_

/ \

IDENT <R>

\_\_|\_\_\_\_\_\_\_\_\_

/ | \ \

+ IDENT {ADD} <R>↑PR

Lecture 10 - QuITE a LoT TO CoVEr

### Removal of left recursion

<IDLIST> → <IDLIST> , IDENT

<IDLIST> → IDENT

<IDLIST> => <IDLIST> , IDENT

=> <IDLIST> , IDENT , IDENT

=> IDENT, IDENT, IDENT

<IDLIST>

/ | \

<IDLIST> , IDENT↑C

/ | \

<IDLIST> , IDENT↑B

/

IDENT↑A

Tree is going left - Bottom-Up Parsable

<IDLIST> → IDENT <REST OF LIST>

<REST OF LIST> → , IDENT <REST OF LIST>

<REST OF LIST> → ε

<IDLIST> => IDENT <REST OF LIST>

=> IDENT , IDENT <REST OF LIST>

=> IDENT , IDENT , IDENT <REST OF LIST>

=> IDENT , IDENT , IDENT

<IDLIST>

/ \

<IDENT> <REST OF LIST>

/ | \

, IDENT <REST OF LIST>

/ | \

, IDENT <REST OF LIST>

\

ε

Tree is going right - Top-Down Parsable

LR(1) - Bottom-Up Parsable

L = Left (Input is scanned from L→R)

R = while its parsing, it’s producing a rightmost derivation tree.

1 = Can look ahead 1 symbol. Number of symbols in needs to look ahead for to work.

LL(1) - Top-Down Parsable

L = Left (Input is scanned from L→R)

L = while its parsing, it’s producing a leftmost derivation tree.

1 = Can look ahead 1 symbol.

### Left Factoring

<F> → <P> ↑ <F>

<F> → <P>

<P> ↑ <P> ↑ <P>

<F> → <P> <POWER PART>

<POWER PART> → ↑ <F>

<POWER PART> → ε

<P> ↑ <P> ↑ <P>

Coco warning:

If statement:

<IF STAT> → IF <C> THEN <STAT>

<IF STAT> → IF <C> THEN <STAT> ELSE <STAT>

Can’t have 2 productions of the same name that start with the same symbol.

LR(1)

This is the **E + T** Grammar. **Know it**

<E> → <E> + <T> Left recursion - Replace them

<E> → <T>

<T> → <T> \* <F>

<T> → <F>

<F> → <P> ↑ <F>

<F> → <P>

<P> → (<E>)

<P> → IDENT

LL(1)

This is the **E-LIST** grammar. **Know it**

<E> → <T> <E-LIST>

<E-LIST> → + <T> <E-LIST>

<E-LIST> → ε

<T> → <F> <T-LIST>

<T-LIST> → \* <F> <T-LIST>

<T-LIST> → ε

<F> → <P> <POWER PART>

<POWER PART> → ↑ <F>

<POWER PART> → ε

<P> → (<E>)

<P> → IDENT

A + B + C

<E> ⇒ <T> <E-LIST>

⇒ <F> <T-LIST> <E-LIST>

⇒ <P> <POWER PART> <T-LIST> <E-LIST>

⇒ IDENT <POWER PART> <T-LIST> <E-LIST>

⇒ ...

<E>

\_\_|\_\_

/ \

<T> <E-LIST>

\_\_\_\_/ \_\_\\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/ \ / \ \

<F> <T-LIST> + <T> <E-LIST>

\_\_|\_ \ / \ \_\_\_\\_\_\_\_\_\_\_

/ \ ε <F> <T-LIST> / \ \

<P> <POWER PART> / \ \ + <T> <E-LIST>

/ \ <P> \ ε / \ \

IDENT↑A ε / <POWER PART> <F> <T-LIST> ε

IDENT↑B \ / \ \

ε / \ ε

<P> <POWER PART>

/ \

IDENT↑C ε

A \* (B + C) ⇒ A B C + \* Operands must stay in the same order

A { A } \* (B { B } + C { C } { + } ) { \* }

Output the operator once you have seen the left operand and right operand.

<E> → <T> <E-LIST>

<E-LIST> → + <T> { + } <E-LIST>

<E-LIST> → ε

<T> → <F> <T-LIST>

<T-LIST> → \* <F> { \* } <T-LIST>

<T-LIST> → ε

<F> → <P> <POWER PART>

<POWER PART> → ↑ <F>

<POWER PART> → ε

<P> → (<E>)

<P> → IDENT

<E>

\_\_|\_\_\_

/ \

<T> <E-LIST>

\_\_\_\_|\_\_ \

/ \ \

<P> <T-LIST> ε

\_\_\_/ \_\_\_\_\\_\_\_\_\_\_\_

/ \ / / | \

IDENT {IDENT↑A} \* <P> {\*} <T-LIST>

\ \

/ | \ \

( <E> ) ε

\_\_|\_\_

/ \

<T> <E-LIST>

\_\_\_\_\_\_\_\_/ \_\_\_\_|\_\_\_\_\_\_\_\_

/ / / | \ \

<P> <T-LIST> + <T> {+} <E-LIST>

\_|\_\_ \ \_\\_\_ \

/ \ \ / \ \

IDENT {IDENT↑B} ε <P> <T-LIST> ε

/ \ \

<IDENT> {IDENT↑C} ε

<E> → <T> <E-LIST>

<E-LIST> → + <T> <E-LIST>

<E-LIST> → ε

<T> → <P> <T-LIST>

<T-LIST> → \* <P> <T-LIST>

<T-LIST> → ε

<P> → (<E>)

<P> → CONST

(1+2) \* (3+4)

<E>

\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_

/ \

<T> <E-LIST>

\_\_\_\_\_\_\_\_\_/\_\_\_\_\_ \

/ \ \

<P>3 <T-LIST>3 ε

\_/\_ \_\_\\_\_\_\_\_\_\_\_\_\_\_\_\_

/ | \ / \ \

( <E>3) \* <P> <T-LIST>

\_\_\_\_/\_\_\_\_ \\_\_\_\_ \

/ \ / | \ \

<T>1 <E-LIST>1 3 ( <E> ) ε

\_\_/\_\_\_ \_\\_\_\_\_\_\_\_ \_\\_\_\_

/ \ / | \ / \

<P>1 <T-LIST>1 + <T>2 <E-LIST>3 <T>3<E-LIST>3

/ | \_/\_\_ \ \_\_\_\_/\_ \\_\_\_\_\_\_\_\_

/ | / \ ε3 / \ \ \ \

CONST1 ε1 <P>2 <T-LIST>2 <P>3 <T-LIST>3 + <T>4<E-LIST>

| \ | | \_\_|\_ \

| \ | | / \ \

CONST2 ε2 CONST3 ε3 <P>4<T-LIST>4 ε

/ |

/ |

CONST4 ε4

Lecture 11

### Control Flow Statements

<Repeat Statement> →

REPEAT <Statement> UNTIL <Cond>

<If Statement> →

IF <Cond> THEN <Statement>

Flow of Control

↓

REPEAT

↓

EXECUTE <Statement>

↓

UNTIL

↓

EVALUATE <Cond>

↓

False (return to repeat) ?

↓ True

Translation:

↓

REPEAT

↓

{Label}

↓

<Statement>

↓

UNTIL

↓

<Cond>

↓

{JUMPF} (jump cond false to label)

This gives:

<Repeat Statement> → REPEAT {LABELp} <Statement> UNTIL <Cond>q {JUMPFr,s}

(JUMPFr,s = Jump if false, on condition r, to label s.)

r ← q (r inherits q)

(p, s) ← NEWL (p and s get assigned a new label)

WHERE <Cond>p synthesizes p, all action symbol attributes are inherited, and NEWL allocates a new symbol table entry for a label.

Coco/R Pseudocode:

|  |
| --- |
| “Repeat” (. int l;  L = Gen.Label  .)  Stats  “Until”  Cond (.  if (type == boolean)  Gen.BranchFalse(L)  else  SemErr  .) |

For tomorrow:

<WHILE STAT> → WHILE <COND> DO <STATEMENTS>

FLOW OF CONTROL

↓

WHILE

↓

EVALUATE <Cond>

↓

Jump out on false ?

↓ true

DO

↓

EXECUTE <Statement>

↓

jump to WHILE

WHILE

↓

{LABEL}

↓

<CONDITION>

↓

<JUMPF> Jump to bottom label on condition false

↓

DO

↓

<STATEMENTS>

↓

{JUMP}

{LABEL}

This gives:

<While Statement> → WHILE {LABELp} <Cond>q <JUMPF>r, s DO <Stats> <JUMP>t {LABEL}u

r ← q

(u, s) ← NEWL

(t, p) ← NEWL

Lecture 12 - whEre iS The spICy naMe

<WHILE STAT> → WHILE <EXPR> DO <STAT>

FLOW

↓

EVALUATE<EXPR>

↓

? → FALSE → (to end)

↓ TRUE

EXECUTE

↓

(to start)

TRANSLATION

↓

WHILE

↓

{LABEL}

↓

<EXPR>

↓

{JUMPF} → (to end)

↓

DO

↓

<STAT>

↓

{JUMP} → (to start)

↓

{LABEL}

↓

Translation Grammar:

<WHILE STAT> → WHILE {LABEL}<EXPR> {JUMPF} DO <STAT> {JUMP} {LABEL}

Add Attributes:

<WHILE STAT> → WHILE {LABELp}<EXPR>q {JUMPFr, s} DO <STAT> {JUMPt} {LABELu}

r ← q (r is assigned q)

(s, u) ← NEWL (s and u need to have the same value)

(p, t) ← NEWL

Where <EXPR>p synthesized p.

All attributes are inherited and NEWL allocates a new symbol table entry for a label.

Coco/R ATG:

|  |
| --- |
| “WHILE” (. int startLabel = gen.NewLabel();  int endLabel = gen.NewLabel();  gen.Label(startLabel);  .)  EXPR<out reg,  out type> (. if (type == boolean)  gen.BranchFalse(endLabel);  else SemErr(“boolean type expected”);  .)  “DO”  {STAT} (. gen.Branch(startLabel);  gen.Label(endLabel);  .) |

Pascal:

|  |
| --- |
| PROGRAM TEST (INPUT, OUTPUT);  VAR CH : CHAR;  X, Y: REAL;  I, J, K: INTEGER  BEGIN:  ...  END. |

<PROGRAM>p → PROGRAM IDENTq (<ID-LIST>); <DECLARATIONS>; <COMPOUND STAT>

p ← q

<ID-LIST>p → IDENTq, <ID-LIST>r {LINK IDs}s, t where {LINK IDs}p, q links p to the front of

the linked list pointed to by q. (→ q → p)

<ID-LIST>p → IDENTq

p ← q s ← r t ← q

… (<ID-LIST> {SET FILE}) …

q ← p

… <DECLARATIONS> {ALLOCATE}q

q ← p

Symbol Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **NAME** | **TYPE** | **ADDR** | **...** | **ID-LINK** | **DEC-LINK** |
| **1** | TEST |  |  |  | / |  |
| **2** | INPUT | FILE |  |  | / |  |
| **3** | OUTPUT | FILE |  |  | INPUT |  |
| **4** | CH |  |  |  | / |  |
| **5** | X |  |  |  | / |  |
| **6** | Y |  |  |  | / |  |
| **7** | I |  |  |  | / |  |
| **8** | J |  |  |  | / |  |
| **9** | K |  |  |  | / |  |

<ID-LIST>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/ / \ \

<IDENT>2 , <ID-LIST>3 {LINK IDs}3, 2

/

/

IDENT3

<DECLS>p → VAR <DEC-LIST>q p ← q

<DECLS>p → ε p ← nil

<DECLIST>p → <ID-LIST>q ? <TYPE>r {SET TYPE}s, t <MORE DECLS>u {LINK DECS}v, w

<MORE DECLS>p → ; <IDENT LIST>q : <TYPE>r {SET TYPE}s, t <MORE DECLS>u {LINK DECS}v, w

<MORE DECLS>p → ε p ← nil

; is a separator in pascal, not needed after every line.

<MORE DECS> is needed as you need at least one declaration after the keyword VAR.

Lecture 13

<E>p → <T>q <E-LIST>r, s r ← q, p ← s

<E-LIST>p, q → + <T>r {ADDs, t, u} <E-LIST>v, w \*

<E-LIST>p, q → ε q ← p

<T>p → <P>q <T-LIST>r, s r ← q, q ← s

<T-LIST>p, q → \* <P>r {MULTs, t, u} T-LISTv, w \*

<T-LIST>p, q → ε q ← p (q synthesizes p)

<P>p → (<ε>q) p ← q

<P>p → IDENTq p ← q

\* s ← p, t ← r

(u, v) ← NEWT

q ← w

where…

(NEWT = new symbol table entry describing partial result.)

Derivation Tree: (grows in left-most derivation form. tree hides the derivation)

<E>

\_\_|\_\_\_

/ \

<T> <E-LIST>

/ \

<P>↑A → <T-LIST>↑A

/ ↑ \_\\_\_\_\_\_\_\_\_\_\_\_

/ / | \ \

IDENT↑A \* <P> {MULT} <T-LIST>

\_|\_\_\_\_\_

/ | \

( <E>↑PR1 )

\_\_\_|\_\_\_\_\_\_\_\_\_\_\_\_\_ ↑

/ \

<T>↑B → <E-LIST>↑B, ↑PR1

/ \ ↑ \_\_\_\_\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ↑

/ \ / / \ \

<P>↑B →<T-LIST>↑B + <T>↑C {ADD↑B,↑C,↑PR1} <E-LIST>↑PR1,↑PR1

/ | / \ ↑ |

IDENT↑B ε <P>↑C<T-LIST>↑C ε

| \

IDENT↑C ε

↑ = synthesized

### Top-Down Parsing

Production Rules:

1. <S> → D <S> <A>
2. <S> → B <A> C
3. <A> → D <A>
4. <A> → C

Example: DBCCDC

Deriving the string:

<S> 1⇒ D <S> <A>

2⇒ D B <A> C <A>

4⇒ D B C C <A>

3⇒ D B C C D <A>

4⇒ D B C C D C

<S>

\_\_|\_\_\_

/ | \

D <S> <A>

\_\_\_/\_ \

/ / \ / \

B <A> C D <A>

| |

C C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | B | C | D | ⊣ |
| <S> | #2 |  | #1 |  |
| <A> |  | #4 | #3 |  |
| C |  | POP  ADVANCE |  |  |
| ▽ |  |  |  | ACCEPT |

#1 ⇒ REPLACE (<A><S>) All blank symbol table entries represent REJECT.

ADVANCE

#2 ⇒ REPLACE (C<A>) Starting stack: ▽<S>

ADVANCE

#3 ⇒ [REPLACE (<A>)]

ADVANCE

#4 ⇒ POP

ADVANCE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Stack | Input | Past Inputs | Stack | Intermediate String |
| ▽<S>  ▽<A><S>  ▽<A>C<S>  ▽<A>C  ▽<A>  ▽<A>  ▽ | DBCCDC⊣  BCCDC⊣  CCDC⊣  CDC⊣  DC⊣  C⊣  ⊣ | D DB  DBC  DBCC  DBCCD  DBCCDC | <S>▽  <S><A>▽  <A>C<A>▽  C<A>▽  <A>▽  <A>▽  ▽ | <S>  D<S><A>  DB<A>C<A>  DBCC<A>  DBCC<A>  DBCCD<A>  DBCCDC |

⇒ACCEPT

We are doing a leftmost derivation here because of top-down parsing.

Lecture 14 - LeSs THaN 50 pErCEnt

During top-down processing, there is an assertion that the entire input is correct iff the string of remaining terminals can be derived from the sequence of symbols on the stack.

1. <S> → a <A> <S>
2. <S> → b
3. <A> → c <A> <S>
4. <A> → ε

aacbb

<S> 1⇒ a <A> <S>

4⇒ a <S>

1⇒ a a <A> <S>

3⇒ a a c <A> <S> <A> <S>

4⇒ a a c <S> <A> <S>

2⇒ a a c b <A> <S>

4⇒ a a c b <S>

2⇒ a a c b b

### Q-Grammars

S-grammar = Simple

Q-grammar = Quasi-simple

Follow

<S>⊣

If x is the starting non-terminal s, then s can be followed by an end marker.

Follow (<A>) = {a, b}

Select set for prod 1 = {a}

Select set for prod 2 = {b}

Select set for prod 3 = {c}

Select set for prod 4 = Follow (<A>) = {a, b}

{a} and {b} are disjoint, i.e. {a} ∩ {b} = Ø

<A> → ε POP, RETAIN

1. <S> → a <A> <S>
2. <S> → b
3. <A> → c <A> <S>
4. <A> → ε

Control Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | ⊣ |
| <S> | #1 | #2 |  |  |
| <A> | #4 | #4 | #3 |  |
| ▽ |  |  |  | ACCEPT |

Starting stack = ▽<S>

#1 REPLACE WITH (<S><A>), ADVANCE

#2 POP, ADVANCE

#3 REPLACE WITH (<S><A>), ADVANCE

#4 POP, RETAIN

**Stack: Input:**

▽<S> aacbb⊣

▽<S><A> acbb⊣

▽<S> acbb⊣

▽<S><A> cbb⊣

▽<S><S><A> bb⊣

▽<S><S> bb⊣

▽<S> b⊣

▽ ⊣

The only thing under an <A> is an <S>

Null production removes <A> from the top of the stack.

### Tutorial

Show that the following grammar is a Q grammar and construct a pushdown control to parse it.

1. <S> → a <A> <B>
2. <S> → b <B> <S>
3. <S> → ε
4. <A> → c <B> <S>
5. <A> → ε
6. <B> → d <B>
7. <B> → e

select 1 = {a}

select 2 = {b}

select 3 = follow <S> = {⊣} + follow <A> = {⊣, d, e}

select 4 = {c}

select 5 = follow <A> = {d, e}

select 6 = {d}

select 7 = {e}

1. <B> → c <S> <A> e

Control Table: (Ignoring rule 8)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e | ⊣ |
| <S> | #1 | #2 |  | #3 | #3 | #3 |
| <A> |  |  | #4 | #5 | #5 |  |
| <B> |  |  |  | #6 | #7 |  |
| ▽ |  |  |  |  |  | ACCEPT |

Starting Stack = ▽<S>

#1 REPLACE WITH (<B><A>), ADVANCE switch order because of stack

#2 REPLACE WITH (<S><B>), ADVANCE

#3 POP, RETAIN

#4 REPLACE WITH (<S><B>), ADVANCE

#5 POP, RETAIN

#6 ADVANCE

#7 POP, ADVANCE

Lecture 15 - oNE mOrE ForM

1. <S> → <A> b <B>
2. <S> → d
3. <A> → <C> <A> b
4. <A> → <B>
5. <B> → c <S> d
6. <B> → ε
7. <C> → a
8. <C> → e d

S-Grammar

Property - right hand side starts with a terminal symbol ⇒ Not an s-grammar

Q-Grammar

Property - right hand side is empty, or starts with a terminal symbol ⇒ Not a q-grammar

For all top-down parsers, the grammar is deterministic

bcdd

Starting with the starting non-terminal, and then performing the **left-most** derivation.

<S> 1⇒ <A> b <B>

4⇒ <B> b <B>

6⇒ b <B>

5⇒ b c <S> d

2⇒ b c d d

Derivation Tree:

<S>

/ | \

<A> b <B>

/ / | \

<B> c <S> d

/ /

ε d

Determine if this grammar is LL1 - compute select sets

FIRST - can appear at the leftmost pos of this string

FOLLOWS - for nullable productions

FIRST FOLLOW SELECT

1. <S> → <A> b <B> {a, b, c, e} {a, b, c, e}
2. <S> → d {d} {d}
3. <A> → <C> <A> b {a, e} {a, e}
4. <A> → <B> {c} \* {c} + FOLLOW(<A>) = {b, c}
5. <B> → c <S> d {c} {c}
6. <B> → ε {} = {b, d, ⊣} {b, d, ⊣}
7. <C> → a {a} {a}
8. <C> → e d {e} {e}

\* = FOLLOW(<B>) = FOLLOW(<A>) + FOLLOW(<S>) = {b} + {d, ⊣}

SELECT is disjoint for each non-terminal

This is an LL1 grammar

Build Pushdown Control: (Using SELECT)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** | **e** | **⊣** |
| **<S>** | #1 | #1 | #1 | #2 | #1 |  |
| **<A>** | #3 | #4 | #4 |  | #3 |  |
| **<B>** |  | #6 | #5 | #6 |  | #6 |
| **<C>** | #7 |  |  |  | #8 |  |
| **b** |  | #9 |  |  |  |  |
| **d** |  |  |  | #10 |  |  |
| **▽** |  |  |  |  |  | ACCEPT |

Blank table entries are REJECT

#1 REPLACE w/ (<B> b <A>), RETAIN Push last non-terminal

#2 POP, ADVANCE

#3 REPLACE w/ (b <A> <C>), RETAIN

#4 REPLACE w/ (<B>), RETAIN

#5 REPLACE w/ (d <S>), ADVANCE

#6 POP, RETAIN

#7 POP, ADVANCE

#8 REPLACE w/ (d), ADVANCE

#9 POP, ADVANCE

#10 POP, ADVANCE

Stack: Input:

▽<S> bcdd⊣

▽<B>b<A> bcdd⊣

▽<B>b<B> bcdd⊣

▽<B>b cdd⊣

▽d<S> dd⊣

▽d d⊣

▽ ⊣

⇒ ACCEPT

Q-Grammars parse faster than LL1

LL1 keeps control tables smaller

### Task

Show that the <E-LIST> grammar is LL(1) and construct a pushdown machine for it.

**PRODUCTION RULES FIRST**

1. <E> → <T> <E-LIST> { (, IDENT }
2. <E-LIST> → + <T> <E-LIST> { + }
3. <E-LIST> → ε
4. <T> → <F> <T-LIST> { (, IDENT }
5. <T-LIST> → \* <F> <T-LIST> { \* }
6. <T-LIST> → ε
7. <F> → <P> <POWER PART> { (, IDENT }
8. <POWER PART> → ↑ <F> { ↑ }
9. <POWER PART> → ε
10. <P> → (<E>) { ( }
11. <P> → IDENT { IDENT }

**FOLLOW** of nullables

<E> ⇒ { ), ⊣ }

<E-LIST> ⇒ FOLLOW(<E>) = { ), ⊣ }

<T> ⇒ FIRST(<E-LIST>) + FOLLOW(<E-LIST>) = { +, ), ⊣ }

<T-LIST> ⇒ FOLLOW(<T>) = { +, ), ⊣ }

<F> ⇒ FIRST(<T-LIST>) + FOLLOW(<T-LIST>) = { \*, +, ), ⊣ }

<POWER-PART> ⇒ FOLLOW(<F>) = { \*, +, ), ⊣ }

**SELECT**

1. <E> ⇒ FIRST = { (, IDENT }
2. <E-LIST> ⇒ FIRST = { + }
3. <E-LIST> ⇒ FOLLOW(<E-LIST>) = { ), ⊣ }
4. <T> ⇒ FIRST = { (, IDENT }
5. <T-LIST> ⇒ FIRST = { \* }
6. <T-LIST> ⇒ FOLLOW(<T-LIST>) = { +, ), ⊣ }
7. <F> ⇒ FIRST = { (, IDENT }
8. <POWER-PART> ⇒ FIRST = { ↑ }
9. <POWER-PART> ⇒ FOLLOW(<POWER-PART>) = { +, \*, ), ⊣ }
10. <P> ⇒ FIRST = { ( }
11. <P> ⇒ FIRST = { IDENT }

Pushdown Control:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **IDENT** | **+** | **\*** | **↑** | **(** | **)** | **⊣** |
| **<E>** | #1 |  |  |  | #1 |  |  |
| **<E-LIST>** |  | #2 |  |  |  | #3 | #3 |
| **<T>** | #4 |  |  |  | #4 |  |  |
| **<T-LIST>** |  | #6 | #5 |  |  | #6 | #6 |
| **<F>** | #7 |  |  |  | #7 |  |  |
| **<POWER-PART>** |  | #9 | #9 | #8 |  | #9 | #9 |
| **<P>** | #11 |  |  |  | #10 |  |  |
| **)** |  |  |  |  |  | #12 |  |
| **▽** |  |  |  |  |  |  | ACCEPT |

Blank table entries are REJECT

Terminal stack symbols are non-starting symbols.

Starting Stack: ▽<E>

#1 REPLACE w/ (<E-LIST> <T>), RETAIN

#2 REPLACE w/ (<E-LIST> <T>), ADVANCE

#3 POP, ADVANCE

#4 REPLACE w/ (<T-LIST> <F>), RETAIN

#5 REPLACE w/ (<T-LIST> <F>), ADVANCE

#6 POP, ADVANCE

#7 REPLACE w/ (<POWER-PART> <P>), RETAIN

#8 REPLACE w/ (<F>), ADVANCE

#9 POP, ADVANCE

#10 REPLACE w/ (‘)’ <E>), ADVANCE

#11 POP, ADVANCE

#12 POP, ADVANCE

How do

1. Generate columns of table (input symbols) from symbols in language
2. Generate rows of table (stack symbols) from non-terminals and non-first symbols.
3. Fill in table using production rule numbers. Place the number in the select symbols for that non-terminal.
4. Define actions:
   1. If only non-terminals, REPLACE w/ (backwards non-terminals), RETAIN.
   2. If begins with terminal, REPLACE w/ (backwards non-terminals), ADVANCE.

### Worked Example

**PRODUCTION RULES: FIRST: FOLLOW: SELECT:**

1. <E> → <T> <E-LIST> { IDENT, ( } { IDENT, ( }
2. <E-LIST> → + <T> { ADD } <E-LIST> { + } { + }
3. <E-LIST> → ε ø { ), ⊣ } { ), ⊣ }
4. <T> → <F> <T-LIST> { IDENT, ( } { IDENT, ( }
5. <T-LIST> → \* <F> { MULT } <T-LIST> { \* } { \* }
6. <T-LIST> → ε ø { +, ), ⊣ } { +, ), ⊣ }
7. <F> → <P> <F-PART> { IDENT, ( } { IDENT, ( }
8. <F-PART> → ↑ <F> { POWER } { ↑ } { ↑ }
9. <F-PART> → ε ø { \*, +, ), ⊣ } { \*, +, ), ⊣ }
10. <P> → (<E>) { ( } { ( }
11. <P> → IDENT { IDENT } { IDENT }

**FOLLOW:**

<E-LIST> FOLLOW(<E>) = { ), ⊣ }

<T-LIST> FOLLOW(<T>) = { + } ∪ FOLLOW(<E-LIST>) = { +, ), ⊣ }

<F-PART> FOLLOW(<F>) = { \* } ∪ FOLLOW(<T-LIST>} = { \*, +, ), ⊣ }

Sets are disjoint ⇒ Grammar is an LL(1) grammar.

Pushdown Control:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **IDENT** | **+** | **\*** | **↑** | **(** | **)** | **⊣** |
| **<E>** | #1 |  |  |  | #1 |  |  |
| **<E-LIST>** |  | #2 |  |  |  | #3 | #3 |
| **<T>** | #4 |  |  |  | #4 |  |  |
| **<T-LIST>** |  | #6 | #5 |  |  | #6 | #6 |
| **<F>** | #7 |  |  |  | #7 |  |  |
| **<F-PART>** |  | #9 | #9 | #8 |  | #9 | #9 |
| **<P>** | #11 |  |  |  | #10 |  |  |
| **)** |  |  |  |  |  | POP ADVANCE |  |
| **▽** |  |  |  |  |  |  | ACCEPT |
| **{ ADD }** | OUT(ADD), POP, RETAIN | | | | | | |
| **{ MULT }** | OUT(MULT), POP, RETAIN | | | | | | |
| **{ POWER }** | OUT(POWER), POP, RETAIN | | | | | | |

All blank table entries represent REJECT.

**ACTIONS:**

#1 REPLACE (<E-LIST> <T>), RETAIN

#2 REPLACE (<E-LIST> { ADD } <T>), ADVANCE

#3, #6, #9 POP, RETAIN

#4 REPLACE (<T-LIST> <F>), RETAIN

#7 REPLACE (<F-PART> <P>), RETAIN

#8 REPLACE ({ POWER } <P>), ADVANCE

#10 REPLACE (‘)’ <E>), ADVANCE

#11 POP, ADVANCE

S-Grammar

1. <S> → a
2. <S> → ( <S> <R>
3. <R> → , <S> <R>
4. <R> → )

Recognises (a,(a,a))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **a** | **,** | **(** | **)** | **⊣** |
| **<S>** | #1 | RA | #2 | RB | RC |
| **<R>** | RD | #3 | RE | #4 | RF |
| **▽** | RG | RH | RI | RJ | ACCEPT |

Starting Stack: ▽<S>

#1 POP, ADVANCE

#2 REPLACE (<R> <S>), ADVANCE

#3 PUSH (<S>)

#4 POP, ADVANCE

RA → , occurs when S-EXPR expected IGNORED, ADVANCE

RB → ) occurs when S-EXPR expected IGNORED

RC → Incomplete S-EXPR EXIT

RD, RE → Missing comma ASSUMED, PUSH (<S>)

RF → Incomplete EXIT

RG → G appears after S-EXPR IGNORED, ADVANCE

RH → , appears after S-EXPR Assume remainder of S-EXPR is to follow, and

RI → ( appears after S-EXPR … replace (▽ <S> <R>), RETAIN

RJ → ) appears after S-EXPR IGNORED, ADVANCE

If stmt is not LL(1)

<STMT> -> IF <cond> THEN <STMT>

<STMT> -> IF <cond> THEN <STMNT> ELSE <STMT>

IF (A<B) THEN

IF(C > D THEN X <-Y

ELSE Q <- Q

<STMT>

/ | | \

IF <cond> then <STMT>

|

/ | | \ \ \

IF <cond> then <STMT> ELSE <STMT>

|

|

/ | | \

IF <cond> THEN <STMT>

OR

<STMT>

/ | | | | \

IF <COND> then <STMT> ELSE <STMT>

|

|

/ | | \

If cond then stmt

1. <STMT> -> IF <cond> THEN <STMT> <ELSEPART>
2. <ELSEPART> -> ELSE <STMT>
3. <ELSEPART> -> e

This hasn’t made anything better - the problem is with the select sets

(pause - Dave Abe stares confused at the board)

SELECT:

1. {IF}
2. {ELSE}
3. FOLLOW(<ELSEPART>) = FOLLOW(<STMT>) + {-|} = {ELSE, -|}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | IF | THEN | ELSE | -| |
| <STMT> | #1 |  |  |  |
| <ELSEPART> |  |  | #2 | #3 |

(whenever production rule 2 or 3 can be used, apply 3)

Lists

L stands for List, R is remainder of list

5 ways of lists:

<L> -> a<R>

<R> -> a<R>

<R> -> e

----------------------

<L> -> a<R>

<L> -> e

<R> -> a<R>

<R> -> e

----------------------

<L> -> a<L>

<L> -> e

----------------------

<L> -> a<R>

<R> -> , a<R>

<R> -> e

----------------------

<L> -> a<R>

<L> -> e

<R> -> , a<R>

<R> -> e

All the list structures needed when parsing

Insert {INIT}, {LINK} and {TERM} (and optionally {NULL})

Attributes are needed to link things together

Dave Abe decides not to tell us the answer, switches to {PROCESS} from {LINK}

P is the count of the number of “a”s

<L>p -> a{INIT}q <R>r,s

{INIT} - set q to 1

r ← q

p ← s (synthesizes)

<L>p → e {NULL}

{NULL} → set p to 0

<R>p,q → , a {PROCESS}r <R>s,t

{PROCESS} → increment r

r ← p

s ← r

q ← t (synthesizes)

<R>p,q → e {TERM}r

{TERM} - print value of r

(q,r) ← p

Lecture ???

**PRODUCTION RULES:** **SELECT:**

1. <S> → IDENT := <E> { ASSIGN } { IDENT }
2. <E> → <T> <E-LIST> { IDENT, CONST, ( }
3. <E-LIST> → + <T> { ADD } { + }
4. <E-LIST> → ε { ⊣, ) }
5. <T> → <F> <T-LIST> { IDENT, CONST, ( }
6. <T-LIST> → \* <F> { MULT } <T-LIST> { \* }
7. <T-LIST> → ε { +, ), ⊣ }
8. <F> → <P> <POWER-P> { IDENT, CONST, ( }
9. <POWER-P> → ↑ <F> { POWER } { ↑ }
10. <POWER-P> → ε { +, \*, ), ⊣ }
11. <P> → ( <E> ) { ( }
12. <P> → IDENT { IDENT }
13. <P> → CONST { CONST }

### Recursive Descent

For each non-terminal symbol, design a recursive procedure to parse the productions for that non-terminal symbol.

Work out the select sets, verify the grammar is LL(1), then write recursive descent for each non-terminal.

### Pseudo-code

Assume that NEXTSYMBOL is the call to the lexical analyser which returns symbol.

Used as a global for efficiency instead of passing through functions.

BEGIN // MAINLINE OF PARSER

NEXTSYMBOL; // GET FIRST LEXICAL TOKEN

S;

IF SYMBOL = END\_MARKER

THEN ACCEPT

ELSE ERROR(99)

END // MAINLINE OF PARSER

// PARSE <S>

PROC S;

BEGIN

IF SYMBOL = IDENT THEN

BEGIN

NEXTSYMBOL // ADVANCE

IF NOT (SYMBOL = ASSIGNOP) THEN

IF SYMBOL = EQUALOP THEN

BEGIN ERROR(1); NEXTSYMBOL END

ELSE ERROR(2)

ELSE NEXTSYMBOL;

E; GEN.ASSIGN // CALL PROC E TO PARSE AN <E>

END

ELSE ERROR(3)

END;

// PARSE <E>

PROC E;

BEGIN

T; E-LIST

END;

// PARSE <E-LIST>

PROC E-LIST;

BEGIN

IF SYMBOL = PLUSOP THEN

BEGIN

NEXTSYMBOL;

T; GEN.ADDOP;

E-LIST;

END

ELSE

IF NOT (SYMBOL IN [), ENDMARKER] THEN

ERROR(6)

END

// PARSE <P>

PROC P;

BEGIN

CASE SYMBOL OF

LEFTPAREN:

BEGIN

NEXTSYMBOL;

E;

IF SYMBOL = RIGHTPAREN THEN

NEXTSYMBOL

ELSE ERROR(4)

END

IDENT, CONST: NEXTSYMBOL

DEFAULT: ERROR(5)

END CASE

END

TYPE SYMBOLSET = SET OF SYMBOLS | VAR SYMBOL: SYMBOLS

// Procedure to skip to next synchronising symbol

PROC SKIPTO(VALIDSYMBOLS, OTHERSYMBOLS: SYMBOLSET);

BEGIN

IF NOT (SYMBOL IN VALIDSYMBOLS) THEN

BEGIN

ERROR(0);

VALIDSYMBOLS := VALIDSYMBOLS + OTHERSYMBOLS;

// + is set union if the operands are sets

WHILE NOT (SYMBOL IN VALIDSYMBOLS)

DO NEXTSYMBOL // Advance

END

END

BEGIN // MAINLINE

NEXTSYMBOL;

S([ENDMARKER]); // [] indicates a set

IF SYMBOL = ENDMARKER THEN // Do not advance beyond ⊣

ACCEPT

ELSE ERROR(99)

END;

// PARSE <S>

PROC S(OTHERVALIDSYMBOLS: SYMBOLSET); // Symbols seen outside of <S>

BEGIN

SKIPTO([IDENT], OTHERVALIDSYMBOLS);

IF SYMBOL = IDENT THEN

BEGIN

NEXTSYMBOL;

...

E(OTHERVALIDSYMBOLS); // Symbols <E> should not skip over

GEN.ASSIGN

END

END

// PARSE <T-LIST>

PROC T-LIST(OTHERVALIDSYMBOLS: SYMBOLSET);

BEGIN

SKIPTO([+, \*, ), ENDMARKER], OTHERVALIDSYMBOLS);

IF SYMBOL = MULOP THEN

BEGIN

NEXTSYMBOL;

// Don’t skip over anything that follows F

F([+, \*, ), ENDMARKER] + OTHERVALIDSYMBOLS);

GEN.MULT;

T-LIST(OTHERVALIDSYMBOLS)

END

END

Lecture ?

### L-Attributed Grammars

<A> → <B> <C>

1. Evaluate the inherited attributes of <A>
2. Evaluate the inherited attributes of <B>
3. Evaluate the synthesized attributes of <B>
4. Evaluate the inherited attributes of <C>
5. Evaluate the synthesized attributes of <C>

<A>

/ \

<B> <C>

<A>i1, s2 → <B>i3 <C>s4 <D>i5, s6 <E>s7 inherited, synthesized

i3 ← f ( i1 )

s4 ← f ( i1, i3 )

s6 ← f ( i1, i3, s4, i5 )

s2 ← f ( i1, i3, s4, i5, s6, s7 )

In order to evaluate s4, we need to evaluate everything before it

<S> → IDENTp := <E>q { ASSIGNr, s } r ← q, q ← p

<E>p → <T>q <E-LIST>r, s r ← q, p ← s

<E-LIST>p, q → <T>r { ADDs, t, u } <E-LIST>v, w s ← p, t ← r, (u, v) ← NEWT, q ← w

<E-LIST>p, q → ε q ← p

...

<P>p → ( <E>q ) p ← q

<P>p → IDENTq p ← q

<P>p → CONSTq p ← q

WHERE <E>p<T>p<F>p & <P>p SYNTHESIZED p

### Recursive Descent

<S> → IDENTp := <E>q { ASSIGNq, p }

<E>p → <T>q <E-LIST>q, p

<E-LIST>p, q → + <T>r { ADDp, r, s } <E-LIST>s, q s ← NEWT

<E-LIST>p, q → ε q ← q

...

<P>p → ( <E>p )

<P>p → IDENTp

<P>p → CONSTp

WHERE <E>p, <T>p, <F>p & <P>p synthesized p etc

// MAINLINE

NEXTSYMBOL;

S([ENDMARKER]);

IF SYMBOL = ENDMARKER

THEN ACCEPT

ELSE ERROR(99)

// PARSE S

PROC S (NEXTVALIDSYMBOLS: SYMBOLSET),

LOCAL p, q

SKIPTO([IDENT], NEXTVALIDSYMBOLS);

IF SYMBOL = IDENT THEN

p := SYMBOLVALUE;

NEXTSYMBOL;

IF SYMBOL = ASSIGNOP // :=

THEN NEXTSYMBOL

ELSE IF SYMBOL = EQUALSOP { // =

THEN ERROR(=);

NEXTSYMBOL

}

ELSE ERROR(2)

// PARSE E

PROC E(REF P, OTHERVALIDSYMBOLS: SYMBOLSET);

REF P: ATTR,

LOCAL q: ATTR

SKIPTO([IDENT, CONST, L\_PAREN], OTHERVALIDSYMBOLS); // skip to select<E>

<S> → REPEAT <STATS> UNTIL <COND>

Take CFG and convert to ATG

<S> → REPEAT { LABELp } <STATS> UNTIL <COND>q { JUMPFq, p } p ← NEWL

(JUMPFq, p} = Jump on cond p to label q.

WHERE …

PROC S (OTHERVALIDSYMBOLS: SYMBOLSET);

LOCAL p, q, ATTR {

SKIPTO([REPEAT], OTHERVALIDSYMBOLS);

IF SYMBOL = REPEAT THEN {

NEXTSYMBOL, p ← NEWL

OUTPUT(LABEL, p),

STATS([UNTIL] + OTHERVALIDSYMBOLS);

IF SYMBOL = UNTIL THEN {

NEXTSYMBOL;

COND (OTHERVALIDSYMBOLS)

OUTPUT (JUMPF, q, p)

}

}

}

Final Lecture :\_( doOdLE pOlL

Top-down processing with a pushdown machine

|  |
| --- |
| (A, B) ← C  (X, Y) ← C  ≡ (A, B, X, Y) ← C  (A, B) ← C  (X, Y) ← A  ≡ (A, B, X, Y) ← C  <A> ← bp <C>q <D>r r ← f(p, q)  <A> ← bp <C>q {f}r,s,t <D>u  WHERE <c>; - SYNTHESIZED p  {f}p,q,r INHERITED p, q, SYNTHESIZED r  and <D>u inherited p |

|  |
| --- |
| <S> → <E>  <E> → + <E> <E>  <E> → \* <E> <E>  <E> → CONST  input ⇒ \*+372 ((3+7)\*2)  <S> → <E>p {VALUEq} q ← p  <E>p → + <E>q <E>r p ← q + r  <E>p → \* <E>q <E>r  p ← q \* r  <E>p → CONSTq p ← q  WHERE <E>p SYNTHESIZED p  {ADD}p, q, r r ← q + p  {MULT}p, q, r r ← q \* p  <E>p → + <E>q <E>r {ADD}s,t,u s ← q, t ← r, p ← u  <E>p → \* <E>q <E>r {MUL}s,t,u s ← q, t ← r, p ← u |

3 \* 6 table

**Pushdown Machine**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **+** | **\*** | **CONST** | **⊣** |
| **<S>** | #1 | #1 | #1 | REJECT |
| **<E>** | #2 | #3 | #4 | REJECT |
| **▽** | REJECT | REJECT | REJECT | ACCEPT |
| **{VALUE}** | OUT(VALUE) | | | |
| **{ADD}** | OUT(ADD) | | | |
| **{MULT}** | OUT(MULT) | | | |

Starting Stack: ▽<S>

#1: REPLACE ({VALUE}, <E>), RETAIN

#2: REPLACE (<E>, <E>), ADVANCE

#3: REPLACE (<E>, <E>), ADVANCE

#4: POP, ADVANCE

### Design Stack Symbols

▽

<S>

<E>

Pointer to place synthesized value of <E> is stored

<VALUE>

Place to store inherited value of {VALUE}

**For Prod #1**:

|  |  |  |
| --- | --- | --- |
| <S> |  | <E> |
| ▽ | ⇒ | {VALUE} |
|  |  | ▽ |

#4: Store value of constant in attribute field of <E>, POP, ADVANCE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <S> |  | <E>  Pointer to value |  |  |  |  |
| ▽ | ⇒ | {VALUE}  3 | ⇒ | {VALUE}  3 | 3 |  |
|  |  | ▽ |  | ▽ |  | ▽ |

|  |
| --- |
| {ADD} {MULT} |
| Place where left operand is to be stored. |
| Place where right operand is to be stored. |
| Pointer to place where value of operation is to be stored. |

**Pushdown Machine**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **+** | **\*** | **CONST** | **⊣** |
| **<S>** | #1 | #1 | #1 | REJECT |
| **<E>** | #2 | #3 | #4 | REJECT |
| **▽** | REJECT | REJECT | REJECT | ACCEPT |
| **{VALUE}** | OUT(VALUE), POP, RETAIN | | | |
| **{ADD}** | ADD L + R OPERANDS, STORE RESULT, POP, RETAIN | | | |
| **{MULT}** | MULTIPLY L + R OPERANDS, STORE RESULT, POP, RETAIN <==3 | | | |

#1: REPLACE ({VALUE}, <E>), RETAIN

#2: REPLACE ({ADD}, <E>, <E>), ADVANCE

#3: REPLACE ({MULT}, <E>, <E>), ADVANCE

#4: POP, ADVANCE

|  |  |  |
| --- | --- | --- |
|  |  | <E> |
| <E> |  | <E> |
|  |  | {ADD} | {MULT} |
| ▽ |  | ▽ |

Input ⇒ \*37

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  | <E>  Points to value of MULT |
|  |  |  |  | <E>  Points to value of MULT |
|  | ⇒ | <E>  Points to {VALUE} | ⇒ | {MULT}  ... |
| <S> |  | {VALUE}  ... |  | {VALUE}  ... |
| ▽ |  | ▽ |  | ▽ |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  | <E>  Points to value of MULT |  |  |  |  |
|  | {MULT}  3  Points to {VALUE} |  | {MULT}  3  7  Points to {VALUE} |  |  |
| ⇒ | {VALUE}  ... | ⇒ | {VALUE}  ... | ⇒ | {VALUE}  21 |
|  | ▽ |  | ▽ |  | ▽ |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| ⇒ |  |
|  | ▽ |

<S> → REPEAT <S> UNTIL <C>

<S> → REPEAT {LABELp} <S> UNTIL <C>q {JUMPr, s} r ← q, (p, s) ← NEWL

WHERE …………..

REPLACE({JUMPF} <C> UNTIL <S>}, ADVANCE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | {LABEL}  points to p |  |  |  |
|  | <S> |  |  |  |
|  | UNTIL |  |  |  |
|  | <C>  points to {JUMPF} |  |  |  |
| <S> | {JUMPF}  …  p |  |  |  |
| ▽ | ▽ | ▽ | ▽ | ▽ |