COMPILER DESIGN

Error recovery in top-down predictive parsing

Syntax error recovery

- A syntax error happens when the stream of tokens coming from the lexical analyzer does not comply with the grammatical rules defining the programming language.
- A syntax error is found when the next token in input is not expected according to the syntactic definition of the language.
- One of the main roles of a compiler is to **identify** all programming errors and give **meaningful indications** about the **location** and **nature** of errors in the input program.

Goals of error recovery

- Detect all compile-time errors
- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to be able to detect subsequent errors
- Should not slow down the processing of correct programs
- Avoid spurious errors that are just a consequence of an earlier error

Reporting errors

- Give the position of the error in the source file, maybe print the offending line and point at the error location.

```
doy.cpp: In function `int main()': doy.cpp:25: `DayOfYear' undeclared (first use in this function)
doy.cpp:25: DayOfYear birthday;
            ^
```

- If the nature of the error is easily identifiable, give a **meaningful** error message.
- The compiler should not provide erroneous information about the nature of errors.

Error recovery

- Good error recovery highly depends on how quickly the error is *detected*.
- Often, an error will be detected only after the faulty token has passed.
- It will then be more difficult to achieve good error reporting, as well as good error recovery.
- Bottom-up parsers generally detect errors quicker than top-down parsers.

- Should recover from each error quickly enough to be able to detect subsequent errors. Error recovery should skip as less tokens as possible.

- Should not identify more errors than there really is. Cascades of errors that result from token skipping should be avoided.

- Should induce processing overhead only when errors are encountered.

- Should avoid to report other errors that are consequences of the application of error recovery, e.g. semantic errors.

Error recovery strategies

- There are many different strategies that a parser can employ to recover from syntactic errors.
- Although some are better than others, none of these methods provide a universal solution.
  - Panic mode, or *don’t panic* (Nicklaus Wirth)
  - Error productions
  - Phrase level correction
  - Global correction

Error Recovery Strategies

- Panic Mode
  - On discovering an error, the parser discards input tokens until an element of a designated set of synchronizing tokens is found. Synchronizing tokens are typically delimiters such as semicolons or end of block delimiters.
  - A systematic and general approach is to use the FIRST and FOLLOW sets as synchronizing tokens.
  - Skipping tokens often has a side-effect of skipping other errors. Choosing the right set of synchronizing tokens is of prime importance.
  - Simplest method to implement.
  - Can be integrated in most parsing methods.
  - Cannot enter an infinite loop.

Error Recovery Strategies

- Error Productions
  - The grammar is augmented with “error productions”. For each possible error, an error production is added. An error is trapped when an error production is used.
  - Assumes that all specific errors are known in advance.
  - One error production is needed for each possible error.
  - Error productions are specific to the rules in the grammar. A change in the grammar implies a change of the corresponding error productions.
  - Extremely hard to maintain.

Error Recovery Strategies

- Phrase-Level Correction
  - On discovering an error, the parser performs a local correction on the remaining input, e.g. replace a comma by a semicolon, delete an extraneous semicolon, insert a missing semicolon, etc.
  - Corrections are done in specific contexts. There are myriads of different such contexts.
  - Cannot cope with errors that occurred before the point of detection.
  - Can enter an infinite loop, e.g. insertion of an expected token.

Error Recovery Strategies

- <u>Global Correction</u>
  - Ideally, a compiler should make as few changes as possible in processing an incorrect token stream.
  - Global correction is about choosing the minimal sequence of changes to obtain a least-cost correction.
  - Given an incorrect input token stream *x*, global correction will find a parse tree for a related token stream *y*, such that the number of insertions, deletions, and changes of tokens required to transform *x* into *y* is as reduced as possible.
  - Too costly to implement.
  - The closest correct program does not carry the meaning intended by the programmer anyway.
  - Can be used as a benchmark for other error correction techniques.

Different variations of “panic mode” error recovery

Panic mode error recovery: variations

- Variation 1:
  - Given a non-terminal A on top of the stack, skip input tokens until an element of FOLLOW(A) appears in the token stream.
  - Pop A from the stack and resume parsing.
  - Report on the error found and where the parsing was resumed.

- Variation 2:
  - Given a non-terminal A on top of the stack, skip input tokens until an element of FIRST(A) appears in the token stream.
  - Report on the error found and where the parsing was resumed.

- Variation 3
  - If we combine variation 1 and 2, when there is a parse error and a variable A on top of the stack, we skip input tokens until we see either
    - a token in FIRST(A), in which case we simply continue,
    - a token in FOLLOW(A), in which case we pop A off the stack and continue.
  - Report on the error found and where the parsing was resumed.

Error Recovery in Recursive Descent Predictive Parsers

Error Recovery in Recursive Descent Predictive Parsers

- Three possible cases:
  - The lookahead symbol is not in FIRST(LHS).
  - If ε is in FIRST(LHS) and the lookahead symbol is not in FOLLOW(LHS).
  - The **`match()`** function is called in a no match situation.
- Solution:
  - Create a **`skipErrors()`** function that skips tokens until an element of FIRST(LHS) or FOLLOW(LHS) is encountered.
  - Upon entering any parsing function, call **`skipErrors()`**.

Error Recovery in Recursive Descent Predictive Parsers

```
skipErrors([FIRST],[FOLLOW])
  if (
       lookahead is in [FIRST]
       or
       ε is in [FIRST] and lookahead is in [FOLLOW]
     )
     return true              // no error detected, parse continues in this parsing function
  else
     write (“syntax error at “ lookahead.location)
     while (lookahead not in [FIRST ∪ FOLLOW] )
        lookahead = nextToken()
        if (ε is in [FIRST] and lookahead is in [FOLLOW])
           return false       // error detected and parsing function should be aborted
     return true              // error detected and parse continues in this parsing function
```

```
match(token)
  if ( lookahead == token )
    lookahead = nextToken()
    return true
  else
    write (“syntax error at” lookahead.location. “expected” token)
    lookahead = nextToken()
    return false
```

Error Recovery in Recursive Descent Predictive Parsers

```
LHS(){ // LHS→RHS1 | RHS2 | … | ε
  if ( !skipErrors( FIRST(LHS),FOLLOW(LHS) ) ) return false;
  if (lookahead ∈ FIRST(RHS1) )
    if (non-terminals() ∧ match(terminals) )
      write("LHS→RHS1")
    else success = false
  else if (lookahead ∈ FIRST(RHS2) )
    if (non-terminals() ∧ match(terminals) )
      write("LHS→RHS2")
    else success = false
  else if …                                    // other right hand sides
  else if (lookahead ∈ FOLLOW(LHS) )           // only if LHS→ε exists
    write("LHS→ε")
  else success = false
  return (success)
```

Example

```
E(){
  if ( !skipErrors([0,1,(],[),$]) ) return false;
  if (lookahead is in [0,1,(])
    if (T();E'();) write(E->TE')
    else success = false
  else success = false
  return (success)
}
```

```
E'(){
  if ( !skipErrors([+],[),$]) ) return false;
  if (lookahead is in [+])
    if (match('+');T();E'()) write(E'->TE')
    else success = false
  else if (lookahead is in [$,)]
    write(E'->epsilon);
  else success = false
  return (success)
}
```

```
T(){
  if ( !skipErrors([0,1,(],[+,),$]) ) return false;
  if (lookahead is in [0,1,(])
    if (F();T'();) write(T->FT')
    else success = false
  else success = false
  return (success)
}
```

```
T'(){
  if ( !skipErrors([*],[+,),$]) ) return false;
  if (lookahead is in [*])
    if (match('*');F();T'()) write(T'->*FT')
    else success = false
  else if (lookahead is in [+,),$]
    write(T'->epsilon)
  else success = false
  return (success)
}
```

```
F(){
  if ( !skipErrors([0,1,(],[*,+,$,)]) ) return false;
  if (lookahead is in [0])
    match('0') write(F->0)
  else if (lookahead is in [1])
    match('1') write(F->1)
  else if (lookahead is in [(])
    if (match('(');E();match(')')) write(F->1);
    else success = false
  else success = false
  return (success)
}
```

Error Recovery in Table-Driven Predictive Parsers

Error Recovery in Table-Driven Predictive Parsers

- All empty cells in the table represent the occurrence of a syntax error
- Each case represents a specific kind of error
- Task when an empty (error) cell is read:
  - Recover from the error
    - Either pop the stack, or skip tokens (often called “scan”)
  - Output an error message

Building the table with error cases

- Two possible cases:
  - pop the stack if the next token is in the FOLLOW set of our current non-terminal on top of the stack.
  - scan tokens until we get one with which we can resume the parse.

```
skipError(){                                // A is top()
  write ("syntax error at " lookahead.location)
  if ( lookahead is $ or in FOLLOW( top() ) )
     pop()                                  // pop - equivalent to A → ε
  else
     while ( lookahead ∉ FIRST( top() )
             or
             ε ∈ FIRST( top() ) and lookahead ∉ FOLLOW( top() )
           )
        lookahead = nextToken()             // scan
}
```

Original table, grammar and sets

```
r1: E  → TE′
r2: E′ → +TE′
r3: E′ → ε
r4: T  → FT′
r5: T′ → *FT′
r6: T′ → ε
r7: F  → 0
r8: F  → 1
r9: F  → ( E )
```

```
FST(E)   : { 0,1,( }
FST(E’)  : { ε,+ }
FST(T)   : { 0,1,( }
FST(T’)  : { ε,* }
FST(F)   : { 0,1,( }
```

```
FLW(E)   : { $,) }
FLW(E’)  : { $,) }
FLW(T)   : { +,$,) }
FLW(T’)  : { +,$,) }
FLW(F)   : { *,+,$,) }
```

	0	1	(	)	+	*	$
E	r1	r1	r1				
E’				r3	r2		r3
T	r4	r4	r4				
T’				r6	r6	r5	r6
F	r7	r8	r9				

Parsing table with error actions

	0	1	(	)	+	*	$
E	r1	r1	R1	pop	scan	scan	pop
E’	scan	scan	scan	R3	r2	scan	r3
T	r4	R4	R4	pop	pop	scan	pop
T’	scan	scan	scan	r6	r6	r5	r6
F	r7	R8	R9	pop	pop	pop	pop

- **pop**: if the next token in input is in FOLLOW(LHS), `pop()` RHS from the stack.
- **scan**: else, `repeat ( nextToken() )`

```
          until (FIRST(LHS) is found or
                 if FIRST(LHS) constains ε,  FOLLOW(RHS) is found)
```

Parsing algorithm

```
parse(){
    push($)
    push(S)
    a = nextToken()
    while ( stack ≠ $ ) do
        x = top()
        if ( x ∈ T )
            if  ( x == a )
                pop(x) ; a = nextToken()
            else
                skipError() ; success = false
        else
            if ( TT[x,a] ≠ ‘error’ )
                pop(x) ; inverseRHSPush(TT[x,a])
            else
                skipError() ; success = false
    if ( (a ≠ $) ∨ (success == false ) )
        return(false)
    else
        return(true)}
```

Parsing example with error recovery

	Stack	Input	Production	Derivation
1	\$E	0(*1)\$		E
2	\$E	0(*1)\$	r1: E→TE'	⇒ TE'
3	\$E'T	0(*1)\$	R4: T→FT'	⇒ FT'E'
4	\$E'T'F	0(*1)\$	R7: F→0	⇒ 0T'E'
5	\$E'T'0	0(*1)\$		
6	\$E'T'	(*1)\$	error - scan	
7	\$E'T'	*1)\$	r5: T' → *FT'	⇒ 0*FT'E'
8	\$E'T'F*	*1)\$		
9	\$E'T'F	1)\$	r8: F → 1	⇒ 0*1T'E'
10	\$E'T'1	1)\$		
11	\$E'T'	)\$	r6: T' → ε	⇒ 0*1E'
12	\$E'	)\$	r3: E' → ε	⇒ 0*1
13	\$	)\$	error - end	