## Syntax Analysis

- structural relationship among tokens
- detection of incorrectly formed programs
- compiler organization
  - multi-pass
    - input file contains tokens
    - only valid tokens (maybe special error token)
    - scanner guarantees eof token
  - single pass
    - parser calls scanner when needs next token
    - scanner handles eof and returns eof token

## Recursive Descent Parsing

- recursive
- top-down
- single symbol (token) lookahead
  - must be able to look ahead to determine if at end of sequence
- construction rules for parsing based on syntax
  - derived from Brinch Hansen
- classes match syntax rules of language
- SyntacticUnit
  - superclass of all rule (syntactic unit) classes
    
```java
public abstract class SyntacticUnit {
    public abstract void parse ( ) ;
}
```

## Rule 1

- for every EBNF rule of the form:
  
  \[ N \rightarrow E \]
  
  there is a parser class of the same name with the form:

```java
public class N extends SyntacticUnit { 
    public void parse ( ) { 
        a(E); 
    } // parse
}
```

where \( a(E) \) is an algorithm which parses (recognizes) the sequence of tokens which satisfies the expression \( E \).

- \( a(E) \) looks at tokens in order
- once \( a(E) \) has found a string satisfying \( E \), it has input all the tokens satisfying \( E \) plus one more (single symbol lookahead)
- if sequence does not satisfy \( E \), \( a(E) \) issues an error message after inputting some number of tokens
Rule 2

- an expression of the form:
  \[ E_1 \; E_2 \; \ldots \; E_n \]
is recognized by the algorithm \( a(E_1 \; E_2 \; \ldots \; E_n) \) through recognition of the individual subexpressions in turn, i.e.:
\[
a(E_1 \; E_2 \; \ldots \; E_n) = a(E_1) \cdot a(E_2) \cdot \ldots \cdot a(E_n)
\]

Rule 3

- the expression \( t \) (where \( t \) is a token) is recognized by a call to the
  procedure \( \text{expect}(t) \), i.e.
\[
a(t) = \text{expect}(t)
\]
where \( \text{expect} \) (defined in \( \text{SyntaxicUnit} \)) is:

```java
protected void expect (TokenKind t) {
    if (current token is t) {
        next token
    } else {
        syntax error
    }
} // expect
```

- note that \( \text{expect} \) satisfies the requirements for a parsing algorithm
  (i.e. inputs a string of tokens matching \( E \) (plus one more) or emits a
  syntax error after inputting some number of tokens)

Rule 4

- the expression \( N \) (where \( N \) is a non-terminal symbol) is recognized
  by a call to the \( \text{parse} \) method of a new object \( n \) of class \( N \), i.e.
\[
a(n) = n = new N();
n.parse();
\]
Option and Repetition

- consider the syntax rule for `class-dcl`:
  ```java
  class identifier [ extends identifier ] is body end
  ```
  The extends clause only occurs when `extends` occurs, so it can be handled by:
  ```java
  if (current token is extends) {
    parse extends clause
  }
  ```
- consider the syntax rule for `while_stmt`:
  ```java
  while expression do [ statement ] end ;
  ```
  Which indicates 0 or more occurrences of `statement`, and can be parsed by:
  ```java
  while (current token starts a statement) {
    stmt = new Statement();
    stmt.parse();
  }
  ```
- Note that `{ statement } `⇒ `[ statement [ statement { … } ] ]`

First Symbols

- From Brinch Hansen
- Define `first(E)` to be the set of all possible symbols starting strings derived from `E`
- E.g.
  ```plaintext
  first(class-dcl) = { class }
  first(var-dcl) = { identifier }
  first(method-dcl) = { identifier, method }
  ```
Rule 5

- an expression of the form:
  
  \[ E \]

  is recognized by the algorithm:

  \[
  a( \[ E \] ) = \text{if} ( \text{current token in first}(E) ) \{ \\
  a(E) ; \\
  \}
  \]

Rule 6

- an expression of the form:
  
  \{ E \}

  is recognized by the algorithm:

  \[
  a( \{ E \} ) = \text{while} ( \text{current token in first}(E) ) \{ \\
  a(E) ; \\
  \}
  \]
Rule 7

* if all of the expressions \( E_i \) are non-empty, an expression of the form:
  \[ E_1 | E_2 | \ldots | E_n \]
  is recognized by the algorithm:
  
  \[
  a( E_1 | E_2 | \ldots | E_n ) =
  \begin{align*}
  &\text{if ( current token in first}(E_1)\text{ ) } \{
  &\quad a(E_1);
  &\}\text{;}
  &\text{else if ( current token in first}(E_2)\text{ ) } \{
  &\quad a(E_2);
  &\}\text{;}
  &\text{\ldots}
  &\text{else if ( current token in first}(E_n)\text{ ) } \{
  &\quad a(E_n);
  &\}\text{;}
  &\text{else }
  &\quad \text{syntax error}
  &\}\text{;}
  \end{align*}
\]

First Symbols

* definition: \textit{first} \( (E) \)
  - the set of all tokens which begin any string derivable from \( E \)
  
  \textit{uses}
  - develop parsing procedures
  - determine viability of single-symbol lookahead without backtracking
  - six rules for derivation
- rule 1
  - the empty expression has no first set, i.e.
  \[ \text{first}(\{\}) = \{\} \]

- rule 2
  - the first symbol of an expression consisting of a terminal symbol \( t \) is the set containing that symbol, i.e.
  \[ \text{first}(t) = \{ t \} \]

- rule 3
  - if all derivations from \( E \) are non-empty then:
  \[ \text{first}(E F) = \text{first}(E) \]
  - e.g. \( \text{first}(\text{class-dcl}) \)

- rule 4
  - if any of the derivations from \( E \) can be empty then:
  \[ \text{first}(E F) = \text{first}(E) \cup \text{first}(F) \]
  - e.g. \( \text{first}(\text{method-dcl}) \)

- rule 5
  - an expression of the form:
  \[ E_1 | E_2 | \ldots | E_n \]
  has the first symbol set
  \[ \text{first}(E_1) \cup \text{first}(E_2) \cup \ldots \cup \text{first}(E_n) \]
  - e.g. \( \text{first}(\text{statement}) \)

- rule 6
  - since
  \[ N = \{ E \} \text{ and } N = \{ E \} \]
  can be rewritten as
  \[ N = E | \text{empty} \text{ and } N = E N | \text{empty} \]
  respectively:
  \[ \text{first}(\{ E \}) = \text{first}(E) \cup \text{first}(\{} = \text{first}(E) \]
  \[ \text{first}(\{ E \}) = \text{first}(E N) \cup \text{first}(\{} = \text{first}(E) \]
  \[ \text{first}(\{ E \} F) = \text{first}(E) \cup \text{first}(F) \]
  \[ \text{first}(\{ E \} F) = \text{first}(E) \cup \text{first}(F) \]
  - e.g. \( \text{first}(\text{method-dcl}) \) and \( \text{first}(\text{method-body}) \)
Follow Symbols

- from Brinch Hansen
- definition: follow(n)
  - the set of tokens which can follow strings generated from n in strings generated by the grammar
- uses
  - determine viability of single-symbol lookahead without backtracking
- look at each occurrence of n on the right-hand side of a rule in the grammar
- rules have the forms:
  \[ N = m n o \]
  \[ N = m \{ n \} o \]
  \[ N = m [ n ] o \]
- four rules for derivation
• rule 1
  - if all strings derivable from $o$ are non-empty then
    $\text{follow}(n)$ includes $\text{first}(o)$
• rule 2
  - if some of the strings derivable from $o$ can be empty then
    $\text{follow}(n)$ includes $\text{first}(o) \cup \text{follow}(N)$
• rule 3
  - if $o$ is the empty sequence then
    $\text{follow}(n)$ includes $\text{follow}(N)$
• rule 4
  - if $n$ occurs as $\{n\}$ then
    $\text{follow}(n)$ includes $\text{first}(n)$
  
  e.g. $\text{follow}(\text{var-dcl})$

follow(var-dcl)

---

**Grammatical Restrictions**

- choice ( [], [], [] ) in grammar implies parser must be able to decide which alternative to follow
- if no backtracking, must decide looking at only a fixed number of tokens (one, for single symbol lookahead) ahead of the point where the decision must be made
- poses restrictions on grammar
  - if each alternative begins with an unique token the problem is trivial
    - can design language to allow this e.g.
      - "let a := b instead of a := b"
      - "call p(x) instead of p(x)"
  - parser can know (at any point) which rules can apply and thus only those which apply at this point must begin with unique symbols
Restriction 1

- in each expression of the form $E | F$
  - the alternatives ($E$ & $F$) must begin with disjoint sets of symbols, i.e.:
    \[ \text{first}(E) \cap \text{first}(F) = \{ \} \]
  - e.g. in statement
    - check all pair-wise intersections for null set, e.g.
      \[ \text{first}(\text{method-call-stmt}) \cap \text{first}(\text{assign-stmt}) = \{ \text{super, identifier} \} \cap \{ \text{super, identifier} \} = \{ \} \]
    - single symbol lookahead cannot be used!
    - rewrite grammar to remove problem

Restriction 2

- if an empty sentence can be derived from rule $N$ then
  \[ \text{first}(N) \cap \text{follow}(N) = \{ \} \]
- since
  \[ N = [E] \text{ and } N = \{E\} \]
  are abbreviations for
  \[ N = E | \text{empty} \text{ and } N = E N | \text{empty} \]
  respectively
  \[ \text{first}(E) \cap \text{follow}([E]) = \{ \} \]
  and
  \[ \text{first}(E) \cap \text{follow}({E}) = \{ \} \]
- e.g. in method-dcl and body
  - this restriction prohibits left (infinitely) recursive rules
    \[ \text{expr} = \text{expr} \{ \text{op expr} \} \]
    by restriction 2
    \[ \text{first}([\text{op expr})] \cap \text{follow}([\text{op expr})] = \text{first}([\text{op}) \cap \text{follow}([\text{expr})] = \text{first}([\text{op}) \cap \{ \} \]

Syntax Errors

- parser detects an error, then what?
  - quit
  - continue with same symbol
    * i.e. assume missing symbol
  - ignore the symbol
    * i.e. assume inserted symbol
  - skip some number of symbols
    * i.e. 0 or more
  
  - e.g.
ignoring symbol
int x,
int y;
int z;
same symbol
int x,
int y;
int z;

Recovery

- error could be
  - omitted symbol(s)
  - inserted symbol(s)
  - replaced symbol(s)
- goal
  - correct context to continue parse
- solution
  - at any point there is a set of symbols which can legitimately follow the current expression in current context (not follow set) — the stop set
  - abandon parse of current expression and skip 0 or more symbols until get a symbol which can legitimately follow the expression and continue parse at that point

- e.g.
  - ; is error (expecting identifier), abandon parse of ; and skip to start of whatever can follow in context (var-dcl or constr-dcl) to continue (i.e. skip , and continue with int as start of var-dcl).
  - ; is error (expecting identifer) abandon parse of var-dcl and skip to start of whatever can follow var-dcl (here semicolon) to continue (i.e. skip 0 symbols and continue with ;).
Stop Sets

- context of rule is defined by parent rule
- cannot proceed past the set of symbols which may occur after it in the parent rule
- parent rule also has a stop set
- stop set for child is union of the local context and the stop set for the parent
- must revise the 7 parser construction rules
  - original technique by Hartmann (1977) with improvements by Pemberton (1980) and Balanescu, Gavrila, Gheorghe, Nicolescu & Sofonea (1986)

Symbol Sets

- need sets of symbols (TokenKind)
- class TokenSet
  - should be immutable (treat sets as values)
  - private constructor(s)
  - operations
    - factory methods oneOf
      - var-args
      - overloading
    - contains
    - except
    - toString
Rule 1

- for every EBNF rule of the form:
  
  \[ N = E \]
  
  there is a parser class of the same name with the form:
  
  ```java
  public class N extends SyntacticUnit {
    public static final TokenSet STARTS = ...;
    public N ( TokenSet s ) { super(s); }
    // constructor
    public void parse ( ) {
      a(E,stopSet);
      // parse
    }  // N
  }
  ```

  where \( a(E,stopSet) \) is an algorithm which parses (recognizes) the sequence of tokens which satisfies the expression \( E \) and \( stopSet \) is a set of tokens which could follow \( E \) in the current context (not necessarily follow(\( E \))).

- \( a(E,stopSet) \) looks at tokens in order and either:
  - recognizes a sentence derivable from \( E \) and inputs all the of the sentence plus one more symbol
  - fails to recognize a sentence derivable from \( E \), generates an error message and inputs some number of symbols until it has input one symbol from \( stopSet \).

Rule 2

- an expression of the form:
  
  \[ E_1 E_2 \ldots E_n \]
  
  is recognized by an algorithm \( a(E_1 E_2 \ldots E_n,stopSet) \) by recognition of the individual subexpressions in turn, i.e.:
  
  ```java
  a(E_1 E_2 \ldots E_n,stopSet) =
  a(E_1,first(E_2) \cup \ldots \cup first(E_n) \cup stopSet)
  a(E_2,first(E_3) \cup \ldots \cup first(E_n) \cup stopSet)
  \ldots
  a(E_n,stopSet)
  ```

Rule 3

- the expression \( t \) (where \( t \) is a token) is recognized by a call to the procedure \( expect(t) \) where \( expect \) (defined in \( SyntacticUnit \)) is:
  ```java
  protected void expect ( TokenKind expected, String errMsg, TokenSet context ) {
    if ( tokenIs(expected) ) { accept(); }
    else { listing.writeError(errMsg, context.except(expected)); }
  }  // expect
  ```
  
  \( skipTo \) discards tokens until it encounters one in the specified set.
Rule 4

- the expression $N$ (where $N$ is a non-terminal symbol) is recognized by a call to the `parse` method of a new object $n$ of class $N$, i.e.
  
  $a(N, \text{stopSet}) = n = \text{new } N(\text{stopSet});$
  
  $n.parse();$

Rule 5

- an expression of the form:
  
  $\{E\}F$
  
  is recognized by the algorithm:
  
  $a(\{E\}F, \text{stopSet}) =$
  
  $\text{check(oneOf(first(E), first(F)),..., stopSet); if ( tokenIn(first(E)) ) { a(E, oneOf(first(F), stopSet); }); a(F, stopSet);}$

  where `check` (defined in `SyntacticUnit`) is
  
  protected void check (TokenSet expect, String errMsg, TokenSet context) {
    if ( ! tokenIn(expect) ) {
      listing.writeError(…);
      skipTo(oneOf(expect, context));
    });
  // check

Rule 6

- an expression of the form:
  
  $\{E\}F$
  
  is recognized by the algorithm:
  
  $a(\{E\}F, \text{stopSet}) =$
  
  while ( true ) {
    check(oneOf(first(E), first(F),..., stopSet);
    if ( tokenIn(oneOf(first(F), stopSet).except(first(E)))) break;
    if ( tokenIn(first(E)) ) {
      a(E, oneOf(stopSet, first(E), first(F)));
    };
  a(F, stopSet);
Rule 7

- if all of the expressions $E_i$ are non-empty, an expression of the form:
  \[ E_1 \mid E_2 \mid \ldots \mid E_n \]
  is recognized by the algorithm:
  
  ```
  a(E_1 \ldots E_n, \text{stopSet}) =
  check(oneOf(first(E_1), first(E_2), \ldots, first(E_n)), \text{stopSet});
  if ( \text{tokenIn(first(E_1))}) {
  a(E_1, \text{stopSet});
  }
  else if ( \text{tokenIn(first(E_2))}) {
  a(E_2, \text{stopSet});
  }
  \ldots
  else if ( \text{tokenIn(first(E_n))}) {
  a(E_n, \text{stopSet});
  }
  ```

Special Cases

- $a(E \{E\} F, \text{stopSet}) =$
  
  ```
  do {
  a(E, oneOf(\text{stopSet}, first(E), first(F)));
  check(oneOf(first(F), \text{stopSet}), first(E), first(F));
  while (\text{tokenIn(oneOf(first(F), \text{stopSet}), except(first(E)))})
  a(F, \text{stopSet});
  ```

- $a(E \{t \} F, \text{stopSet}) =$
  
  ```
  while (true) {
  check(oneOf(first(F), t), \text{stopSet}, first(E), first(F));
  if ( \text{tokenIn(oneOf(first(F), \text{stopSet}), except(t))})
  break;
  a(E, oneOf(\text{stopSet}, first(E), first(F)));
  expect(t, oneOf(\text{stopSet}, first(E), first(F)));
  }
  a(F, \text{stopSet});
  ```

- if first(F)\text{stopSet} includes any of first(E), this will prematurely exit on missing $t$, can correct by
  
  ```
  a(E \{t \} F, \text{stopSet}) =
  while (true) {
  a(E, oneOf(\text{stopSet}, first(E), first(F), t));
  check(oneOf(first(F), t), oneOf(\text{stopSet}, first(E), first(F)));
  if ( \text{tokenIn(oneOf(first(F), \text{stopSet}), except(t))})
  break;
  expect(t, oneOf(\text{stopSet}, first(E), first(F)));
  }
  a(F, \text{stopSet});
  ```

- however, this favors missing $t$ over $\text{stopSet}$ and generates an extra error message
### SyntacticUnit Class

- Abstract superclass of all syntactic unit classes
- Maintains a set of tokens (`stopSet`) beyond which the `parse` procedure is not to continue
- All subclasses define a set of tokens (`STARTS`) as `first(N)`
- Subclasses define parsing method by implementing `parse`
- Constructor
  - Initializer `stopSet`
- Convenience (helper) methods
  - `tokenIs` and `tokenIn`
- Parsing helper methods
  - `accept`
  - `expect`
  - `check`
  - `skipIn`
  - `recognized`

### Examples

- `return-stmt`
- `class-dcl`
- `statement`
- `if-stmt`

### Testing Syntactic Analysis

- Input
  - Class
- Output
  - Listing with error messages
  - To trace execution, can display message whenever a syntactic unit is recognized (`recognized`)
- Tests
  - Every construct (syntactic unit)
  - For alternatives (|), each part
  - For options ([]), with and without
  - For repetitions ({}), 0 and 1
- Error recovery
  - Test omitted, inserted and replaced symbols
- Many short tests rather than one large test