COSC 4P75

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

Compiler Construction 3.1

COSC 4P75

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

```
public abstract class SyntacticUnit {
          :
    public abstract void parse ( ) ;
          :
}  // SyntacticUnit
```

Compiler Construction 3.2

COSC 4P75

Rule 1

- for every EBNF rule of the form:

  ```
  N = E
  ```

  there is a parser class of the same name with the form:

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

  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

Compiler Construction 3.3

Rule 2

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

Rule 3

- the expression `t` (where `t` is a token) is recognized by a call to the procedure `expect`, i.e.
  a(t) = expect(t)
  where `expect` (defined in `SyntacticUnit`) is:

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

- note that `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 `parse` method of a new object `n` of class `N`, i.e.

```
a(n) = n = new N();
       n.parse();
```

```
return-stmt = return expression ;
```


Option and Repetition

- consider the syntax rule for *class-dcl*:
  class *identifier* [**extends** *identifier*] **is** *body* **end**
  the extends clause only occurs when `extends` occurs so it can be handled by:

```
if ( current token is extends ) {
    parse extends clause
};
```

- consider the syntax rule for *while-stmt*:
  while *expression* **do** { *statement* } **end ;**
  `which` indicates 0 or more occurrences of *statement*, and can be parsed by:

```
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.
  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 ] ) = if ( current token in first(E) ) {
                     a(E);
                };
```


Rule 6

- an expression of the form:
  `{ E }`
  is recognized by the algorithm:

```
a( { E } ) = while ( current token in first(E) ) {
                     a(E);
                };
```


if *expression* **then** { *statement* } [**else** { *statement* }] **end ;**

Rule 7

- if all of the expressions E_i are non-empty, an expression of the form:
  E_1 | E_2 | … | E_n
  is recognized by the algorithm:

```
a( E1 | E2 | … | En ) =
    if      ( current token in first(E1) ) {
        a(E1);
    }
    else if ( current token in first(E2) ) {
        a(E2);
    }
      :
    else if ( current token in first(En) ) {
        a(En);
    }
    else {
        syntax error
    };
```



```
… | if-stmt | while-stmt | for-stmt | return-stmt
```


First Symbols

- definition: *first(E)*
  - the set of all tokens which begin any string derivable from E
- 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.
    `first( ) = { }`
- rule 2
  - the first symbol of an expression consisting of a terminal symbol `t` is the set containing that symbol, i.e.
    `first(t) = { t }`
- rule 3
  - if all derivations from `E` are non-empty then:
    `first(E F) = first(E)`
  - e.g. `first(class-dcl)`
- rule 4
  - if any of the derivations from `E` can be empty then:
    `first(E F) = first(E)` ∪ `first(F)`
  - e.g. `first(method-dcl)`

`first(class-dcl)`

`first(method-dcl)`

- rule 5
  - an expression of the form:
    $E_1 \mid E_2 \mid \ldots \mid E_n$
    has the first symbol set
    $first(E_1) \cup first\ (E_2) \cup \ldots \cup first(E_n)$
  - e.g. `first(statement)`
- rule 6
  - since
    `N = [ E ]` and `N = { E }`
    can be rewritten as
    `N = E | empty` and `N = E N | empty`
    respectively:

```
first([ E ])   = first(E) ∪ first( )    = first(E)
first({ E })   = first(E N) ∪ first ( ) = first(E)
first([ E ] F) = first(E) ∪ first(F)
first({ E } F) = first(E) ∪ first(F)
```

  - e.g. `first(method-dcl)` and `first(method-body)` *first(statement)*

first(method-dcl)

first(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
  ```
  where *m* and *o* are (possibly empty) sequences of terminals and non-terminals
- four rules for derivation

- rule 1
  - if all strings derivable from *o* are non-empty then
    `follow(n) includes first(o)`
- rule 2
  - if some of the strings derivable from *o* can be empty then
    `follow(n) includes first(o)` ∪ `follow(N)`
- rule 3
  - if *o* is the empty sequence then
    `follow(n) includes follow(N)`
- rule 4
  - if n occurs as `{n}` then
    `follow(n) includes first(n)`
- e.g. `follow(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.:
  first(E) ∩ *first*(F) ≡ { }
- e.g. in `statement`
  - check all pair-wise intersections for null set, e.g.
    first(method-call-stmt) ∩ *first(assign-stmt)*
    = {**super**,*identifier*} ∩ {**super**,*identifier*}
    = {**super**,*identifier*} ≠ { }
  - single symbol lookahead cannot be used!
  - rewrite grammar to remove problem

```
statement = name-stmt | if-stmt | while-stmt |
            for-stmt | return-stmt
name-stmt = name ( assign-rest | call-rest ) ;
assign-rest = [ var-rest ] := expression
call-rest = ( [ argument-list ] )
var-rest = [ expression ]
```


Restriction 2

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

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	same symbol
`int x,`	`int x,`
`int y;`	`int y;`
`int ;`	`int ;`
`int z;`	`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 `;`), abandon parse of `;` and skip to start of whatever can follow in this context (`var-dcl` or `constr-dcl`) to continue (i.e. skip `,` and continue with `int` as start of `var-dcl`).
  - `;` is error (expecting `identifier`), abandon parse of `var-dcl` and skip to start of whatever can follow `var-dcl` (here `;`) to continue (i.e. skip 0 symbols and continue with `;`)

```
int x,
int y;
int  ;
int z;
```

COSC 4P75

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, Gavrilla, Gheorghe, Nicolescu & Sofonea (1986)

Compiler Construction 3.32

COSC 4P75

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`

Compiler Construction 3.33

Rule 1

- for every EBNF rule of the form:
  `N = E`
  there is a parser class of the same name with the form:

```
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:
  $E_1\ E_2\ \dots\ E_n$
  is recognized by an algorithm $a(E_1\ E_2\ \dots\ E_n, stopSet)$ by recognition of the individual subexpressions in turn, i.e.:
  $a(E_1\ E_2\ \dots\ E_n, stopSet) =$
  $a(E_1, first(E_2) \cup first(E_3) \cup \dots \cup first(E_n) \cup stopSet)$
  $a(E_2, first(E_3) \cup first(E_4) \cup \dots \cup first(E_n) \cup stopSet)$
  $\vdots$
  $a(E_n, stopSet)$

Rule 3

- the expression `t` (where `t` is a token) is recognized by a call to the procedure `expect`, i.e.
  a(`t`,`stopSet`) = `expect(t,stopSet)`
  where `expect` (defined in `SyntacticUnit`) is:

```
protected void expect ( TokenKind expected, String errMsg,
                        TokenSet context ) {
    if ( tokenIs(expected) ) {
        accept();
    }
    else {
        listing.writeError(…);
        skipTo(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,stopSet) = n = new N(stopSet);
                   n.parse();
```


Rule 5

- an expression of the form:

```
    [ E ] F
```

  is recognized by the algorithm:

```
    a([E]F,stopSet) =
        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,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);
```

COSC 4P75

Rule 7

- if all of the expressions E_i are non-empty, an expression of the form:
  E_1 | E_2 | … | E_n
  is recognized by the algorithm:

```
a(E1|E2|…|En,stopSet) =
 check(oneOf(first(E1),first(E2),…,first(En)),…,stopSet);
 if      ( tokenIn(first(E1)) ) {
     a(E1,stopSet);
 }
 else if ( tokenIn(first(E2)) ) {
     a(E2,stopSet);
 }
       :
 else if ( tokenIn(first(En)) ) {
     a(En,stopSet);
 };
```

Compiler Construction 3.40

COSC 4P75

Special Cases

```
• a(E{E}F,stopSet) =
    do {
        a(E,oneOf(stopSet,first(E),first(F));
        check(oneOf(first(E),first(F)),…,stopSet);
    while (!tokenIn(oneOf(first(F),stopSet).except(first(E))) );
    a(F,stopSet);
• a({E t}F,stopSet) =
    while ( true ) {
        check(oneOf(first(E),first(F)),…,oneOf(stopSet,t));
    if ( tokenIn(oneOf(first(F),stopSet).except(first(E),t)) )
                                                          break;
        a(E,oneOf(stopSet,first(E),first(F),t));
        expect(t,…,oneOf(stopSet,first(E),first(F)));
    };
    a(F,stopSet);
```

Compiler Construction 3.41

COSC 4P75

```
• a(E{t E}F,stopSet) =
    while ( true ) {
        a(E,oneOf(stopSet,first(E),first(F),t));
        check(oneOf(first(F),t),…,oneOf(stopSet,first(E)));
      if ( tokenIn(oneOf(first(F),stopSet).except(t)) ) break;
          if ( tokenIs(t) ) {
              accept();
          };
      };
      a(F,stopSet);
```

- if `first(F)`∪`stopSet` includes any of `first(E)`, this will prematurely exit on missing `t`, can correct by

```
• a(E{t E}F,stopSet) =
    while ( true ) {
        a(E,oneOf(stopSet,first(E),first(F),t));
        check(oneOf(first(F),t),…,oneOf(stopSet,first(E)));
      if (tokenIn(oneOf(first(F),stopSet).except(first(E),t))
                                                          break;
          expect(t,…,oneOf(stopSet,first(E));
      };
      a(F,stopSet);
```

- however, this favors missing `t` over `stopSet` and generates an extra error message

Compiler Construction 3.42

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
  - initializes `stopSet`
- convenience (helper) methods
  - `tokenIs` and `tokenIn`
- parsing helper methods
  - `accept`
  - `expect`
  - `check`
  - `skipTo`
  - `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