4190.409 Compilers

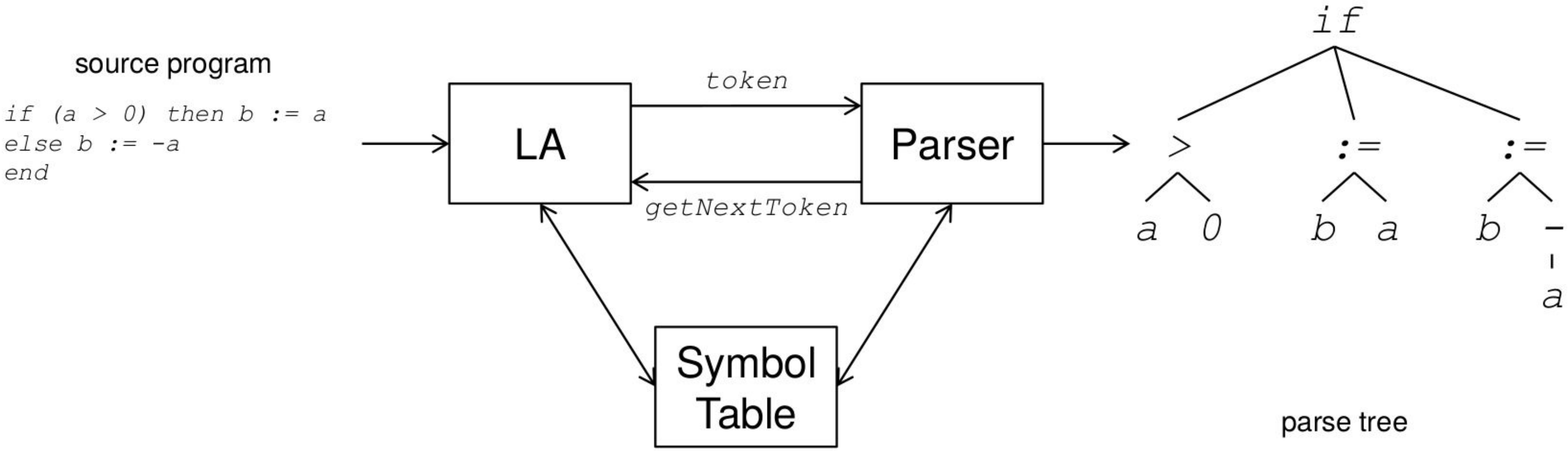
Building a Compiler for SnuPL/1

Computer Systems and Platforms Laboratory
http://csap.snu.ac.kr

Parsing: writing a syntax analyzer

The parser

- Input: tokenized input stream from the lexer
- Output: parse tree of program

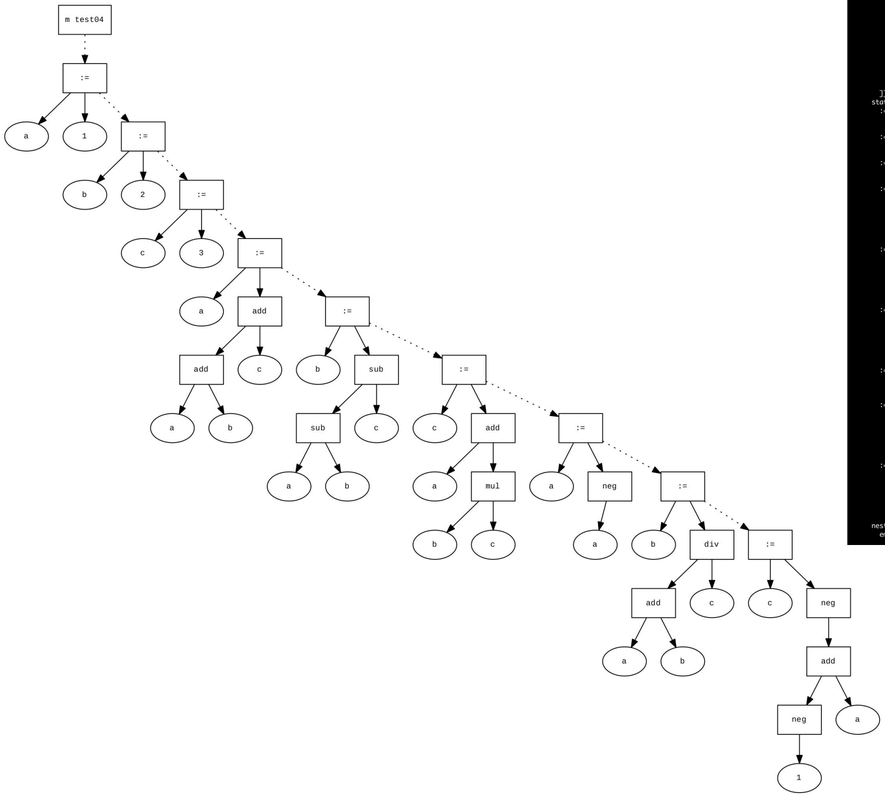


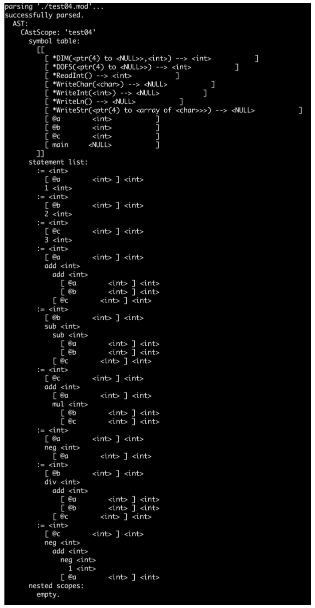
Phase2: SnuPL/1 parser

- Input: tokenized input stream from the scanner
  - you can use your implementation / reference
- Output: the abstract syntax tree (AST) and the symbol table
  - textual or graphical form

```
module test04;

var a,b,c : integer;

begin
  a := 1;
  b := 2;
  c := 3;

  a := a + b + c;
  b := a - b - c;
  c := a + b * c;
  a := -a;
  b := (a + b) / c;
  c := -(-1 + a)
end test04.
```

Input





```
parsing './test04.mod'...
successfully parsed.
  AST:
    CAstScope: 'test04'
      symbol table:
        [[
          [ *DIM(<ptr(4) to <NULL>>,<int>) --> <int>           ]
          [ *DOFS(<ptr(4) to <NULL>>) --> <int>           ]
          [ *ReadInt() --> <int>           ]
          [ *WriteChar(<char>) --> <NULL>           ]
          [ *WriteInt(<int>) --> <NULL>           ]
          [ *WriteLn() --> <NULL>           ]
          [ *WriteStr(<ptr(4) to <array of <char>>>) --> <NULL>           ]
          [ @a       <int>           ]
          [ @b       <int>           ]
          [ @c       <int>           ]
          [ main     <NULL>          ]
        ]]
      statement list:
        := <int>
          [ @a       <int> ] <int>
          1 <int>
        := <int>
          [ @b       <int> ] <int>
          2 <int>
        := <int>
          [ @c       <int> ] <int>
          3 <int>
        := <int>
          [ @a       <int> ] <int>
          add <int>
            add <int>
              [ @a       <int> ] <int>
              [ @b       <int> ] <int>
            [ @c       <int> ] <int>
        := <int>
          [ @b       <int> ] <int>
          sub <int>
            sub <int>
              [ @a       <int> ] <int>
              [ @b       <int> ] <int>
            [ @c       <int> ] <int>
        := <int>
          [ @c       <int> ] <int>
          add <int>
            [ @a       <int> ] <int>
            mul <int>
              [ @b       <int> ] <int>
              [ @c       <int> ] <int>
        := <int>
          [ @a       <int> ] <int>
          neg <int>
            [ @a       <int> ] <int>
        := <int>
          [ @b       <int> ] <int>
          div <int>
            add <int>
              [ @a       <int> ] <int>
              [ @b       <int> ] <int>
            [ @c       <int> ] <int>
        := <int>
          [ @c       <int> ] <int>
          neg <int>
            add <int>
              neg <int>
                1 <int>
              [ @a       <int> ] <int>
      nested scopes:
        empty.
```

Output

Reference implementation

- Use the skeleton code
  - Predictive parser (basically an LL(1) parser)
  - Most helper functions are implemented so that you can focus on interesting parts.

- Source code

- asp.cpp/h	used for generating AST / type system
- ir.cpp/h	you may refer to make an AST node
- parser.cpp/h	you mostly work on this file
- scanner.cpp/h	your implementation/reference
- symtab.cpp/h	help to build nested symbol tables
- type.cpp/h	help to construct symbol types
- test_scanner/parser.cpp	

Nested symbol tables

- You need to make nested symbol tables
  - use functions in symtab.cpp/h and type.cpp/h

```
module test01;

var a, b, c: integer;

procedure foo (a: integer);
var b: integer;
begin
    b := c;
end foo;

begin
end test01.
```

Input

```
parsing test01
CAstScope: “test01”
symbol table:
[[      [ @a        <int>              ]
        [ @b        <int>              ]
        [ @c        <int>              ]
        [ *foo(<int>      →    <NULL>  ]
        [ main      <NULL>             ]      ]]

Nested scopes:
CAstScope: “foo”
[[      [ %a        <int>              ]
        [ @b        <int>              ]      ]]
```

symbol table

Tips for this project

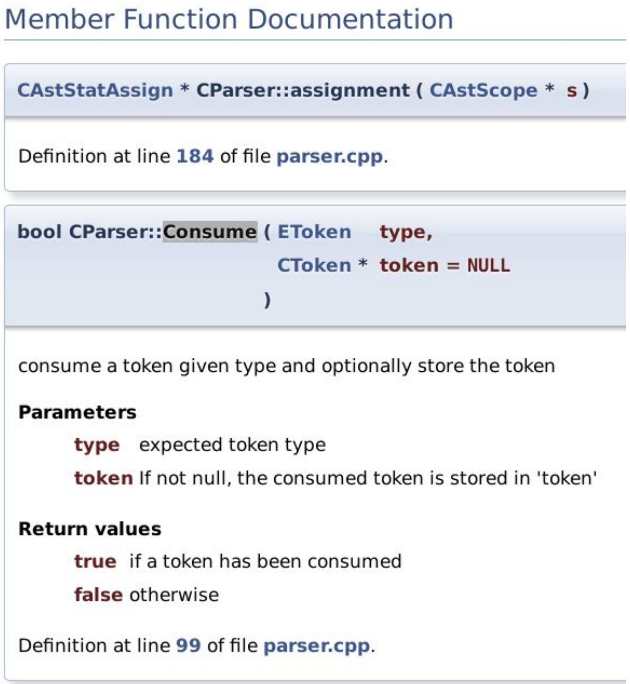
Using Doxygen

- Parser.h

```
/// @brief consume a token given type and optimally share the token
/// @param type expected the error
/// @param token If not null, the consumed token is stored in ‘token’
/// @retval true if a token has been consumed
/// @retval false otherwise
bool Consume(EToken type, CToken *token = NULL);
```

- Generating documentation

```
$ make doc
$ firefox doc/html/index.html
```



Member Function Documentation

```
CAstStatAssign * CParser::assignment ( CAstScope * s )
```

Definition at line 184 of file parser.cpp.

```
bool CParser::Consume ( EToken    type,
                        CToken *  token = NULL
                      )
```

consume a token given type and optionally store the token

Parameters

type expected token type

token If not null, the consumed token is stored in 'token'

Return values

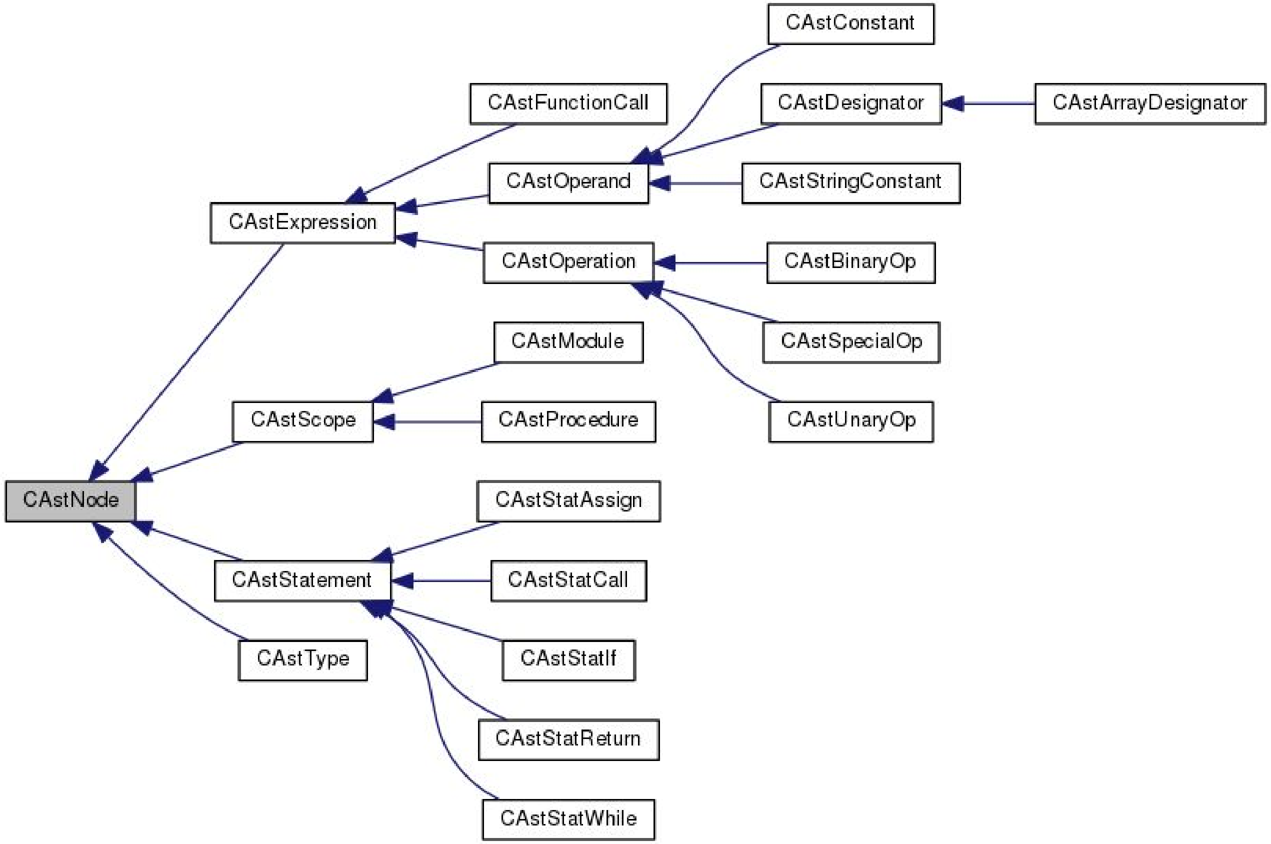
true if a token has been consumed

false otherwise

Definition at line 99 of file parser.cpp.

Using Doxygen

- Use the documentation to understand the skeleton code structure



Hierarchy of the CAstNode

Frequently asked questions in this phase

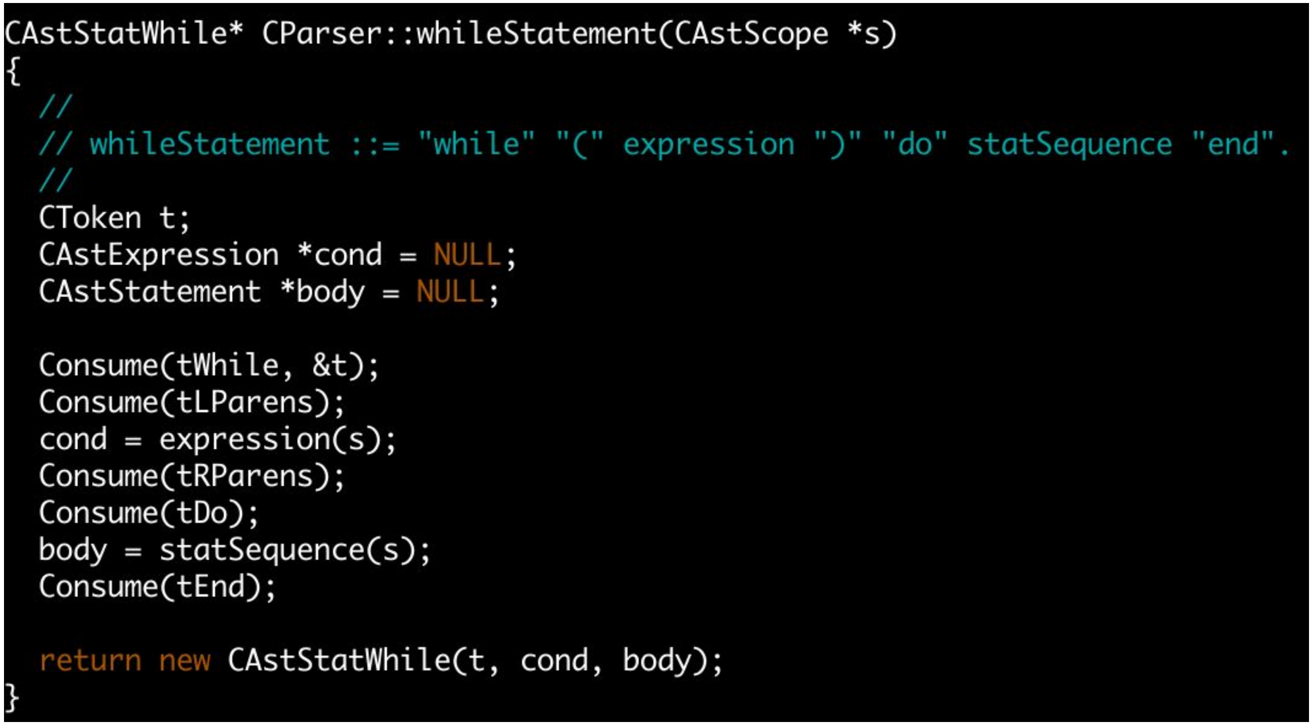
- What to do or what not to do in the parser?
  - Refer to the supplement document

This is a predictive parser

- Use top-down approach when you write the parser
- Construct the nested symbol tables
- Construct (and return) the AST node

```
CAstModule* CParser::module(void)
{
    //
    // module ::= “module” ident “;” [varDeclartion] {funcDeclaration}
    //                  “begin” statSequence “end” ident “.”.
    //

    1. Consume terminals for the module according to the definition.
       e.g.,) Consume module, identifier, and semicolon.

    2. Initialize the module with the consumed terminals.
       e.g.,) Construct the module, and make a global symbol table.

    3. You may deal with non-terminals using functions
}
```

Example



```
CAstStatWhile* CParser::whileStatement(CAstScope *s)
{
  //
  // whileStatement ::= "while" "(" expression ")" "do" statSequence "end".
  //
  CToken t;
  CAstExpression *cond = NULL;
  CAstStatement *body = NULL;

  Consume(tWhile, &t);
  Consume(tLParens);
  cond = expression(s);
  Consume(tRParens);
  Consume(tDo);
  body = statSequence(s);
  Consume(tEnd);

  return new CAstStatWhile(t, cond, body);
}
```

Global symbol table

- Register predefined open arrays / IO functions
- Use the type manager (type.cpp/h)
- Use symbol table functions (symtab.cpp/h)

```
void CParser::InitSymbolTable(CSymtab *st)
{
    //
    // reserved identifiers
    //   such identifiers cannot be used as function/procedure/global variable names
    //   ‘main’ is used to denote the module body in the generated assembly file
    //
}
```

Compute FIRST and FOLLOW sets

- FIRST provides the information to consume the first token

```
CAstExpression* CParser::factor(CAstScope *s)
{
  // factor ::= qualident | number | boolean | char | string |
  //            “(“ expression ”)” | subroutineCall | “!” factor.
  //
  // FIRST(factor) = {tIdent, tNumber, tBoolean, tCharConst, tString,
  //                  tLParens, tNot}
}
```

- FOLLOW provides the information to quit the routine

```
CAstStatement* CParser::statSequence(CAstScope *s)
{
  // statSequence ::= [ statement { “;” statement} ].
  // statement ::= assignment | subroutineCall | ifStatement | whileStatement |
  //               returnStatement.
  //
  // FIRST(statSequence) = { tIdent, tIf, tWhile, tReturn }
  // FOLLOW(statSequence) = { tElse, tEnd }
}
```

Construct nested symbol tables

- Subroutines has own symbol table

```
void CParser::subroutineDecl(CAstScope *s)
{
  //
  // subroutineDecl ::= (procedureDecl|functionDecl) subroutineBody ident “;”.
  // proc/funcDecl  ::= (“procedure”|“function”) ident [formalParam] “;”.
  // formalParam    ::= “(“ [ ident { “,” ident } ] “)”.
  //
}
```

In fact, the grammar is not fully LL(1)

- You can allow LL(2) for certain cases

- Variable declaration is called from different sources
  - module (varDeclaration)
  - function/parameters (formalParam)

Generating graphical form

- This is not mandatory, but it provides a good visualization
  - for grading, we basically consider the textual form
  - please try to use the built-in functions for textual outputs

- Generate a pdf file with the skeleton implementation
  - install graphviz

```
$ dot -Tpdf -o./test01.mod.ast.pdf test01.mod.ast.dot
```

How to submit

- Materials to submit
  - source code of the scanner (use Doxygen-style comments)
  - a report describing your implementation of the scanner (a pdf file)
  - compress your implementation and the report
    example) 2016-12345_NAME.tgz
    example - team) 2016-12345_NAME_2017-12345_NAME.tgz

- Email us your submission (.tgz)
  - compiler@csap.snu.ac.kr

- The deadline is **20th (Friday), October, 2017 at 14:00**. The arrival time of your email counts as the submission time.