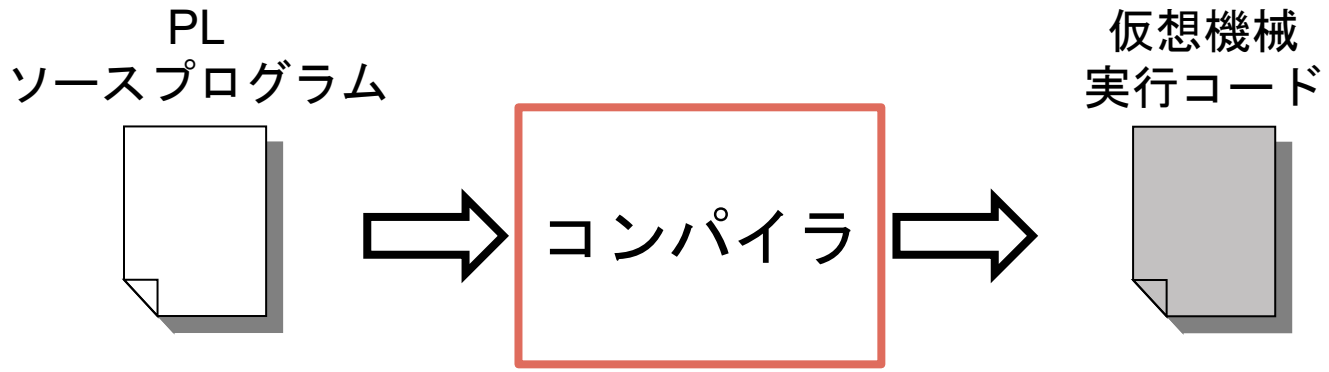
コンピュータ科学実験3
コンパイラの作成

2019年12月5日

演習の目的

- コンパイラの作成
  - Pascal に基づく簡単なプログラミング言語(PL)で書かれたプログラムを対象
  - 仮想機械（LLVM）上で動作するコードを生成



演習資料

- NUCT にアップロードしてあります
  - 「リソース」
    - 「サンプルプログラム」
      - コンパイラを作成するのに使うファイル
    - 「ソースプログラム例」
      - 作成したコンパイラの動作をテストするための ソースプログラム (PL0～3)
    - 「計算機演習資料」
      - 課題内容を説明した文書
    - 「計算機演習スライド」

評価方法

- プログラムおよびレポートに基づいて判断
  - プログラム
    - コンパイラの各段階（課題1～10）に対応するプログラムを作成
    - 提出：ディレクトリごとzip圧縮し，NUCTに提出
  - レポート （提出は3回；2週目，5周目，7周目の後）
    - 基本的には各課題でどのように実装したかを説明
    - NUCT上の各課題ページに提出

言語仕様と仮想機械

- 言語仕様
  - 実験指導書1.1節を参照
- 仮想機械
  - 目的プログラムの実行環境
  - LLVMを対象とする
  - 実験指導書1.2節を参照

ソースプログラムの言語仕様（PL-0）

- Pascalのサブセット
  - データ型は整数のみ
    - 変数型の宣言は不要
  - 手続きがあり，再帰呼出しが可能
  - 名前の有効範囲は標準Pascalと同様
    - 手続きの入れ子の深さは1まで
  - 制御文は4種類
    - if - then – else　（条件分岐）
    - while – do　（whileループ）
    - for – do　（forループ）
    - begin – end　（複合文）
  - 入出力のための命令文，read, writeが存在

ソースプログラムの言語仕様（PL-1以降)

- PL-1
  - 手続き呼び出しのときに引数の値渡しが存在
    - 引数は複数でも可
- PL-2
  - 1次元の配列が存在
    - 配列の開始番号，添え字の範囲は自由に設定
- PL-3
  - 関数が存在
    - 戻り値の存在する手続き

仮想機械

- 目的プログラム: LLVM IR（中間表現）
  - 多くの命令はアセンブリに似た3 番地コード
  - 変数はレジスタ（無限個の仮想レジスタ）に保存
    - 基本的にすべてのレジスタ変数はSSA 形式で表現
- レジスタマシン
  - 計算する値はすべてレジスタを介してやり取り
    - 値をレジスタにロード
    - レジスタを指定して演算を実行
    - 演算結果をレジスタに格納
- スタックマシンに対して
  - 最適化が容易
  - 命令が複雑

コンパイラ演習　課題１＆２

2019年12月5日

課題内容

- 課題1
  - lex を用いて 字句解析部 を作成（scanner）
    - 各トークンにつき、そのトークン列と種類を出力
      - 種類：予約語，数値，識別子
- 課題2
  - yaccを用いて 構文解析部 を作成（parser）
    - 入力が構文に従っていれば、何も出力しない
    - 構文に従っていなければ、誤りを見つけた時点の行番号とトークンを出力
  - scanner と parser をコンパイルするための Makefile を作成

課題1: 字句解析部

- 入力として与えられたプログラムを意味のある字句（トークン）に分割

program EX1 ; var a ; begin ....

program | EX1 | ; | var | a | ; | begin | ....

- 構文解析部より後の処理は，字句解析部で得られた字句を用いて行う

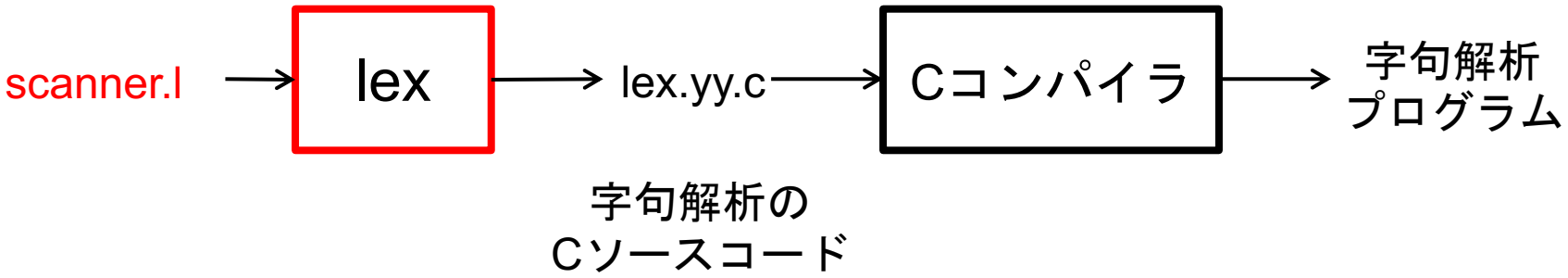
字句解析部での仕事

- プログラムを単語に分割
- 単語を予約語，数値，識別子に分類
- 数値の場合は属性値として数値を計算
  - “**1**”, “**0**”, “**0**” という文字の並び
  - ➔ 数字の 100 として認識
- 識別子の場合は属性値としてその文字列

課題1: PL用の字句解析部の作成（1/2）

- lex の利用

字句解析部の生成の流れ

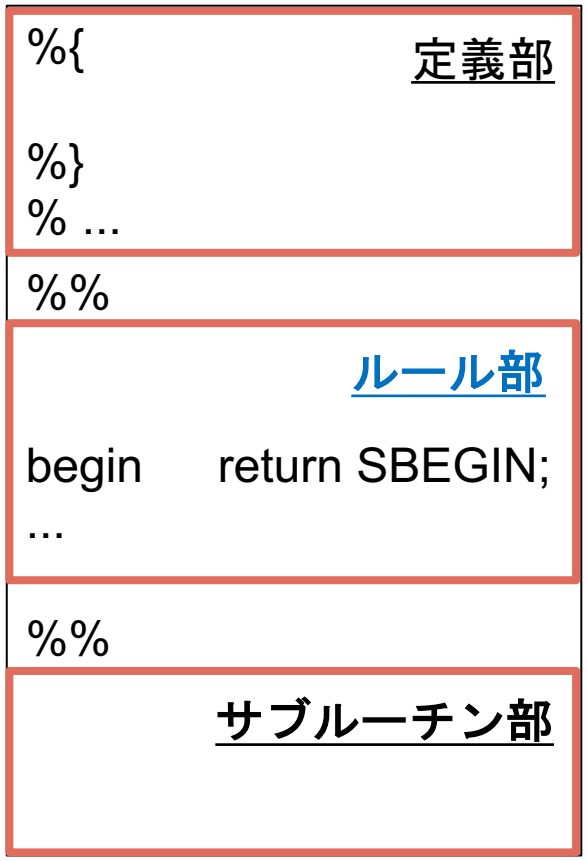


- サンプルプログラムの scanner.l を編集
  - 「･･･」の個所を埋めて完成させる
  - 何をトークンとみなすかについては、PL-0 の構文規則を参照すること（ 資料「計算機演習概要」）
  - 各トークンに対してどのような値を返すかは、シンボルテーブル（symbols.h）を参照

scanner.l

- ３つの部分から構成
  - 定義部
  - ルール部　　←課題1ではルールの追記のみ

    「パターン　　アクション」
  - サブルーチン部



- lexで作成される関数と変数
  - yylex() : トークンを１つ切り出す
    - 戻り値は、マッチしたパターンのアクションで定義した値
  - yyin: 入力ファイルのファイルポインタを格納するための変数
  - yytext: 切り出されたトークンが格納される変数

課題1: PL用の字句解析部の作成（2/2）

- コンパイル方法
  - lex scanner.l
  - cc lex.yy.c –ll –o scanner
- 実行方法
  - ./scanner ソースファイル名

＊ ソースプログラム例に対して scanner を実行

- プログラム例 pl0a.p, ..., pl3b.p

  ./scanner pl0a.p
- 正しくトークンが認識されることを確かめること!

scanner の実行例

```
program EX1;
var a;
begin
    a:= 100;
end.
```

ソースプログラム
ex1.p

./scanner ex1.p

```
"program":   11   RESERVE
"EX1":       38   EX1
";":         32   RESERVE
"var":       15   RESERVE
"a":         38   a
";":         32   RESERVE
"begin":     1    RESERVE
"a":         38   a
":=":        36   RESERVE
"100":       37   100
";":         32   RESERVE
"end":       5    RESERVE
".":         35   RESERVE
```

課題2: 構文解析部

- 字句解析部で出力された字句の並びを解析
  - 入力ファイルがプログラミング言語の文法に従っているかを判定
  - 実現したい文法を yacc に記述することで，構文解析処理を実現
    - yacc は LALR(1) 解析

構文規則（BNF記法）

- コンピュータ言語の文法を記述するために使用されるポピュラーな言語
  - 文脈自由文法に対応

例:

<program> ::= ‘program’ ‘IDENT’ ‘;’ <outblock> ‘.’

<outblock> ::= <ver_decl_part> <subprog_decl_part> <statement>

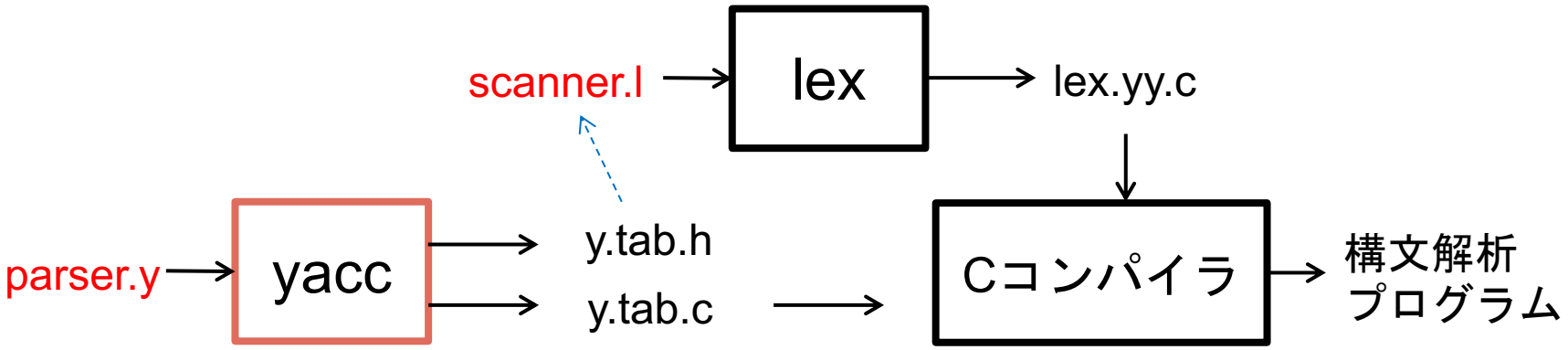
<program> → program IDENT SEMICOLON <outblock> PERIOD

<outblock> → <ver_decl_part> <subprog_decl_part> <statement>

課題2: 構文解析部の作成

- yacc を利用

構文解析プログラムの生成の流れ



- サンプルプログラムの parser.y を編集
  - 「･･･」をPL-0の構文規則を参考にして埋めること
- lexとyaccを連携させるために scanner.l を編集

yaccとlexの連携についての注意点（1）

- 生成されるプログラムについて
  - lex.yy.c：scanner.l から lex によって生成
  - y.tab.c ： parser.y から yacc によって生成
  - y.tab.h ： トークンや共用体を出力したファイル
    - yacc の実行時に –d オプションを付けると生成される.

- yacc と lexでトークンを共有させる
  - scanner.l の定義部では、「#include “symbols.h”」 の代わりに「#include “y.tab.h” 」を書く.
    - ただし、

      y.tab.h の中で MAXLENGTH が使用されているので、scanner.l の中で 「#include “y.tab.h”」を書く位置に注意
    - scanner.l において union や yylval の宣言は不要

yaccとlexの連携についての注意点（2）

- yacc が生成する関数
  - yyparse() : 構文解析を行う関数
    - トークンを切り出す関数 yylex() を随時呼び出す (y.tab.c 参照)
    - 構文に合わない部分を発見した際に、yyerror() を呼ぶ。
- main 関数でやること
  - yyparse() を呼び出す
    - 課題2 では、（scanner.l の）main関数は、yyin に入力ファイルのポインタを格納して、yyparse()を呼び出すだけでよい。

エラー時の処理を記述するためのヒント

- yyerror()：エラーが生じたら呼ばれる関数
- yylineno：行番号をカウントするlexの変数
  - lex.yy.cでの宣言　　int yylineno;
- yytext：その時点のトークンを保持するlexの変数
  - lex.yy.cでの宣言　　char *yytext;

yyerrorの中に行番号とトークンを出力する命令を書く

parser.y にある yyerror 関数内で yylineno や yytext を利用するために、parser.y の宣言部（%{と%}の中）で yylineno, yytext を外部変数として宣言する

```
extern int  yylineno;
extern char *yytext;
```

実行方法

- コンパイル方法
  - yacc -d parser.y
  - lex scanner.l
  - cc y.tab.c lex.yy.c -ll -o parser
- 実行方法
  - ./parser ソースファイル名

＊PL-0 の構文規則に対応できていることを確認

- pl0a.p, pl0b.p, pl0c.p, pl0d.p に対しては正しく受理
- これら以外のプログラムについてはエラーを返す

構文解析部のコンパイル結果

- 正しい実行結果
  - サンプルをそのまま入力した場合

    2 rules never reduced

    1 shift/reduce conflict
  - arg_listを無効にした場合
    （コメント文にした場合）

    1 shift/reduce conflict

※ arg_list はどの構文規則からも呼ばれていない

※ arg_list は今後の課題で使用する

makeを用いたコンパイル

- makeとは
  - プログラムのコンパイル時の作業を自動化するツール
  - ファイルの更新時間をチェックし，更新されているもののみを再コンパイルする
  - ソースファイルが複数のファイルに分割されている場合に特に便利
- ターゲットファイルを生成するためのコンパイル情報を **Makefile** に記述
- コンパイル方法

  % make

Makefileの書き方

- マクロ定義部
- 生成規則

マクロ定義部

- マクロ名＝値
- 例
  - CC = gcc
  - YACC = yacc -d
- マクロの展開
  - $(マクロ名) でマクロを展開
- 例
  - $(CC)

生成規則

- ファイル間の依存関係とターゲットファイルの生成手順を記述

  ターゲットファイル：依存ファイル群

  　　＜タブ＞生成規則

- 例
  - a.out: sub1.o sub2.o

    　　$(CC) -o a.out sub1.o sub2.o

clean ルール

- 不必要なファイルを削除するための記述

```
clean :
        削除規則
```

- 実行方法
  - %make clean
- 例

```
clean:
        rm parser *.o
```

課題の提出（1/2）

- 課題1
  - 提出物: scanner.l 、symbols.h
  - 上記ファイルを kadai1 というディレクトリに保存
  - 以下のコマンドで圧縮し，kadai1.tar.gzを提出

    (kadai1 のディレクトリがある階層で)
    tar zcvf kadai1.tar.gz kadai1

- 課題2
  - 提出物: scanner.l 、parser.y、Makefile
  - 上記ファイルを kadai2 というディレクトリに保存
  - 以下のコマンドで圧縮し，kadai2.tar.gzを提出

    (kadai2 のディレクトリがある階層で)
    tar zcvf kadai2.tar.gz kadai2

課題の提出（2/2）

- 注意点（以下のいずれかを満たすものは再提出）
  - ディレクトリの構成，階層が間違っている
  - コンパイルに必要なファイルがそろっていない
  - （課題2以降）make で実行ファイルが生成されない
  - 生成される実行ファイル名が違う
    - 課題1は scanner、課題2以降は parser で統一